Chapter 11: Network Programming

CSCI3240: Lecture 14 and 15

Dr. Arpan Man Sainju

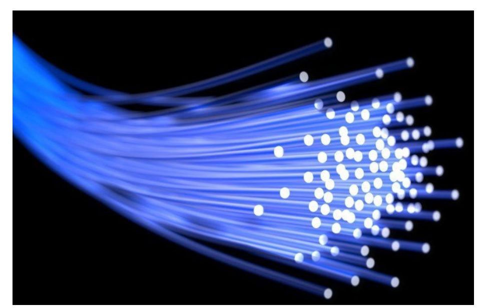
Middle Tennessee State University

Basic Terminology

- Computer Network: A computer network is composed of multiple computers connected using a telecommunication system.

- Host: machines
  - PCs, workstation
  - Network components: routers

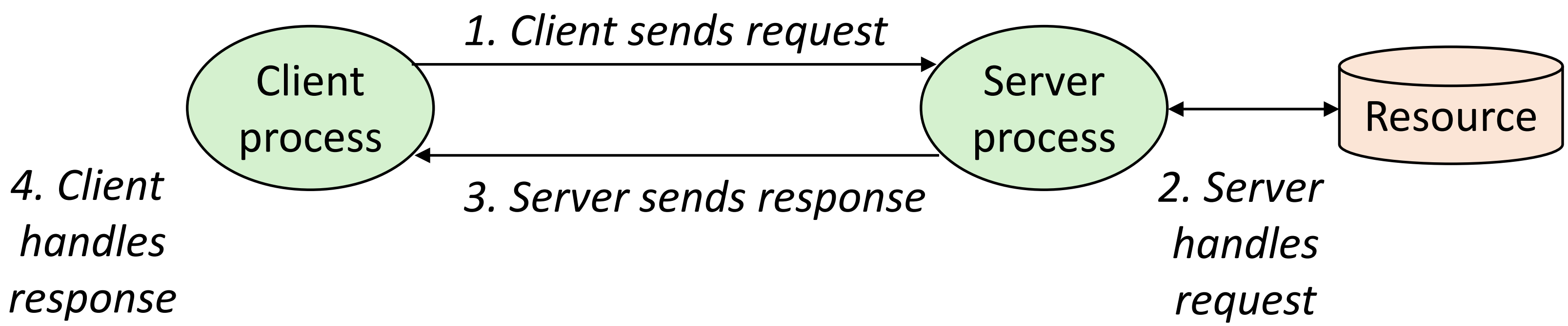
- Interconnection maybe any medium capable of communicating information:
  - Cooper wire (Ethernet)
  - Lasers (Optical fiber)
  - Radio/Satellite link
  - Cable (coax)



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)
- **A server is a process – not a machine.**
- **A server waits for a request from a client.**
- **A client is a process that sends a request to an existing server and (usually) waits for a reply.**

A Client-Server Transaction



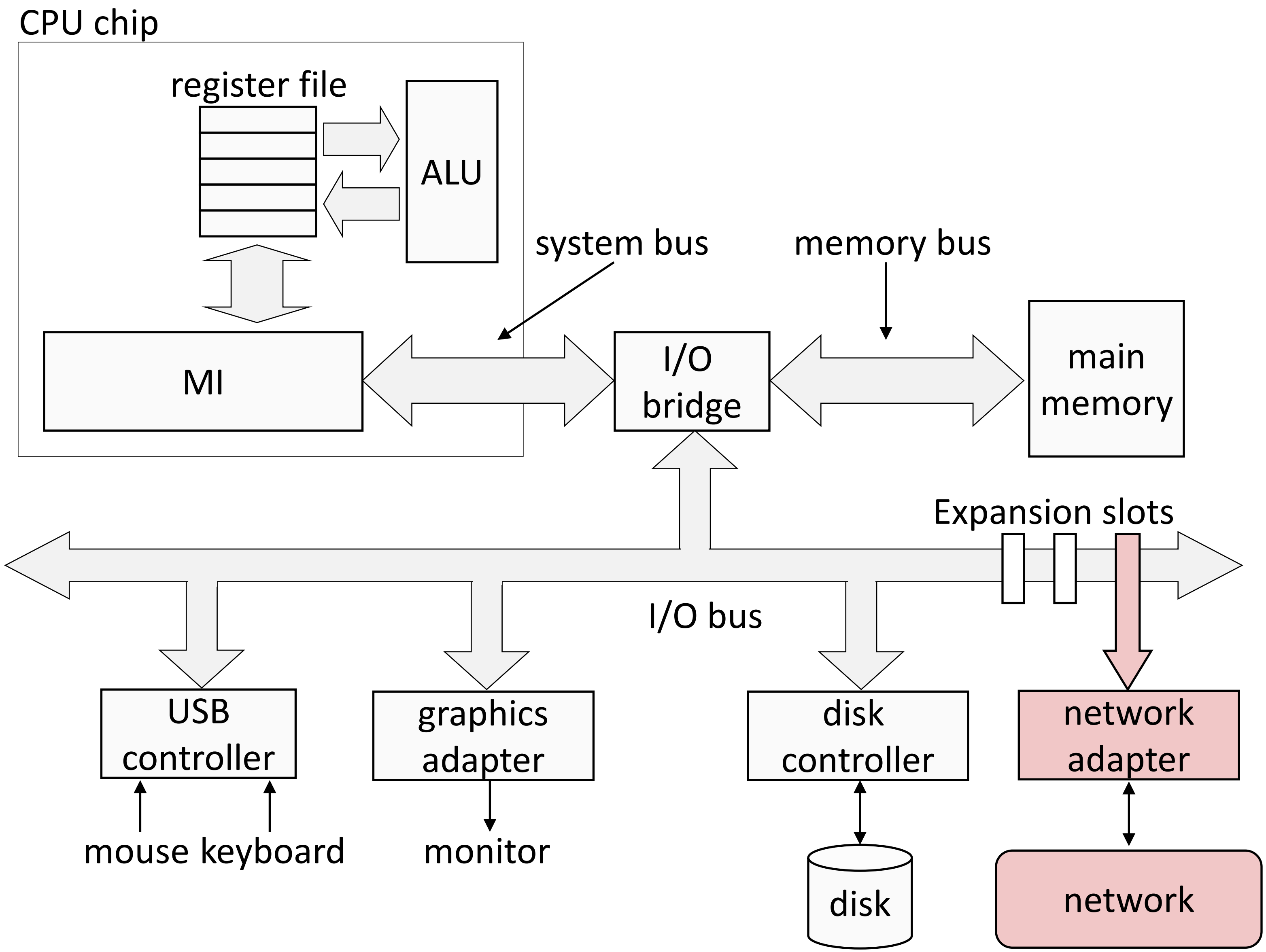
Note: clients and servers are processes running on hosts (can be the same or different hosts)

Example: A Web server manages a set of disk files that it retrieves and executes on behalf of clients.

Client-Server Examples

- **Web Server: Client request for a web page, server delivers**
- **Mail Server: Email serves can be used for sending and receiving emails.**
- **File Servers: They are the centralized locations for the files. Example: One drive, Google drive.**

Hardware Organization of a Network Host

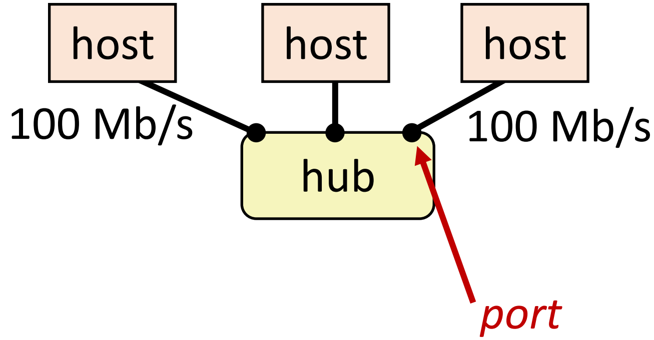


Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Computer Networks

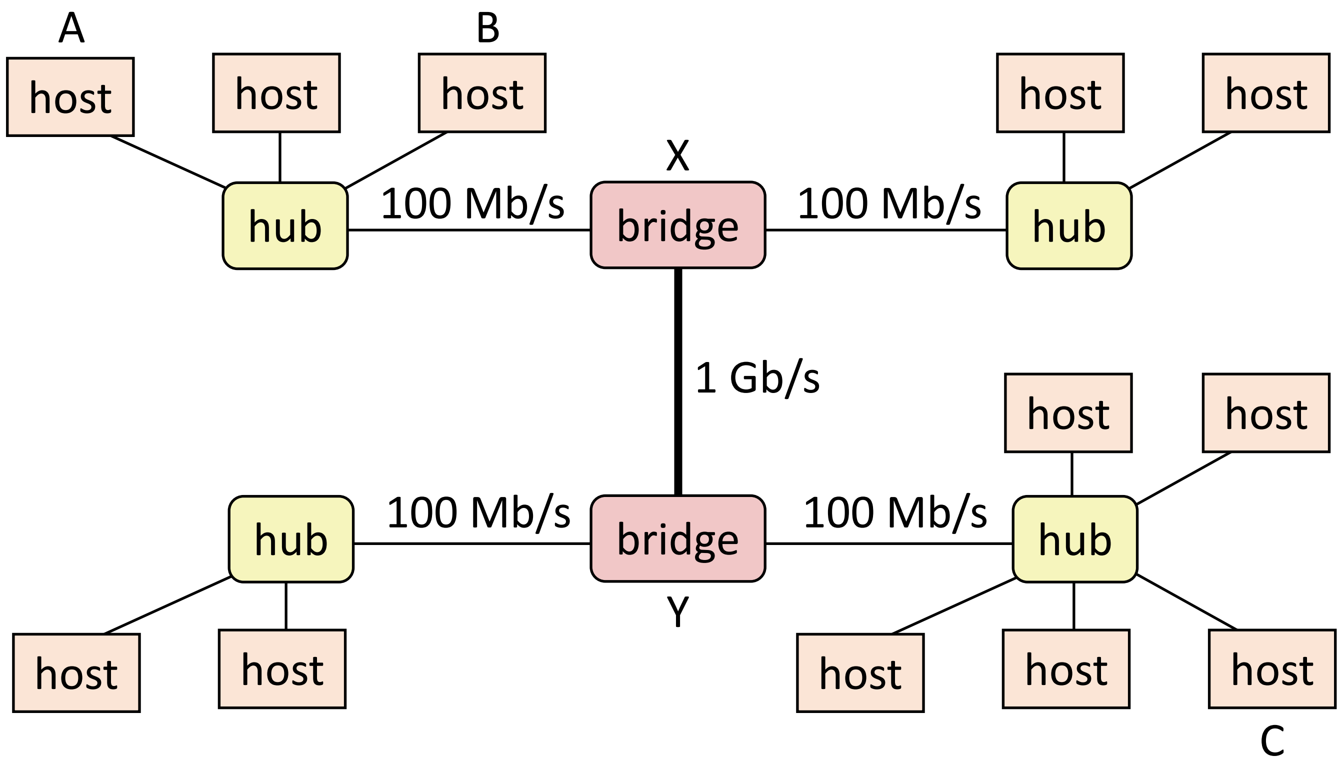
- **A *network* is a hierarchical system of boxes and wires organized by geographical proximity**
  - SAN (System Area Network) spans cluster or machine room
    - Switched Ethernet,
  - LAN (Local Area Network) spans a building or campus
    - Ethernet is most prominent example
  - WAN (Wide Area Network) 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 copies each bit from each port to every other port
    - Every host sees every bit
    - Note: Hubs are on their way out. Bridges (switches, routers) became cheap enough to replace them

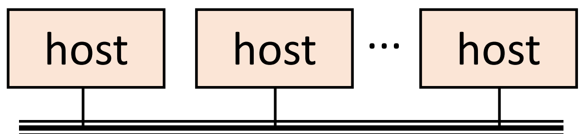
Next Level: Bridged Ethernet Segment



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

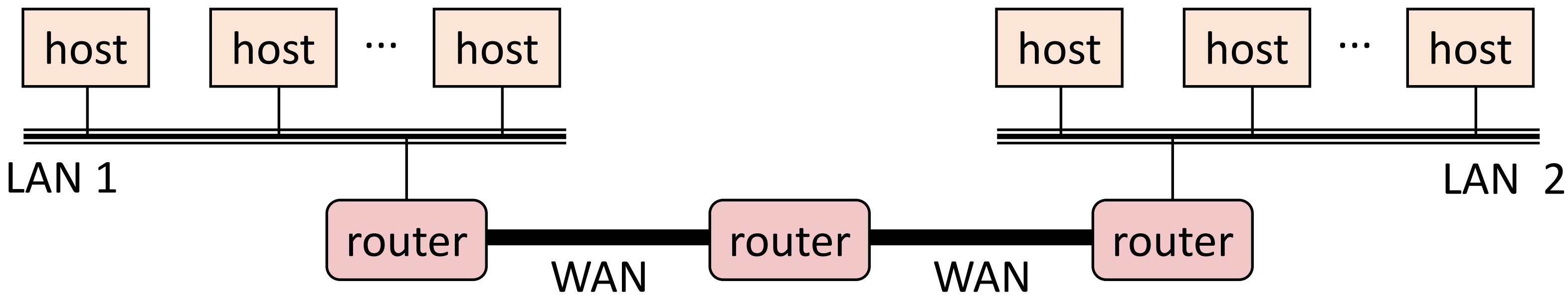
Conceptual View of LANs

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



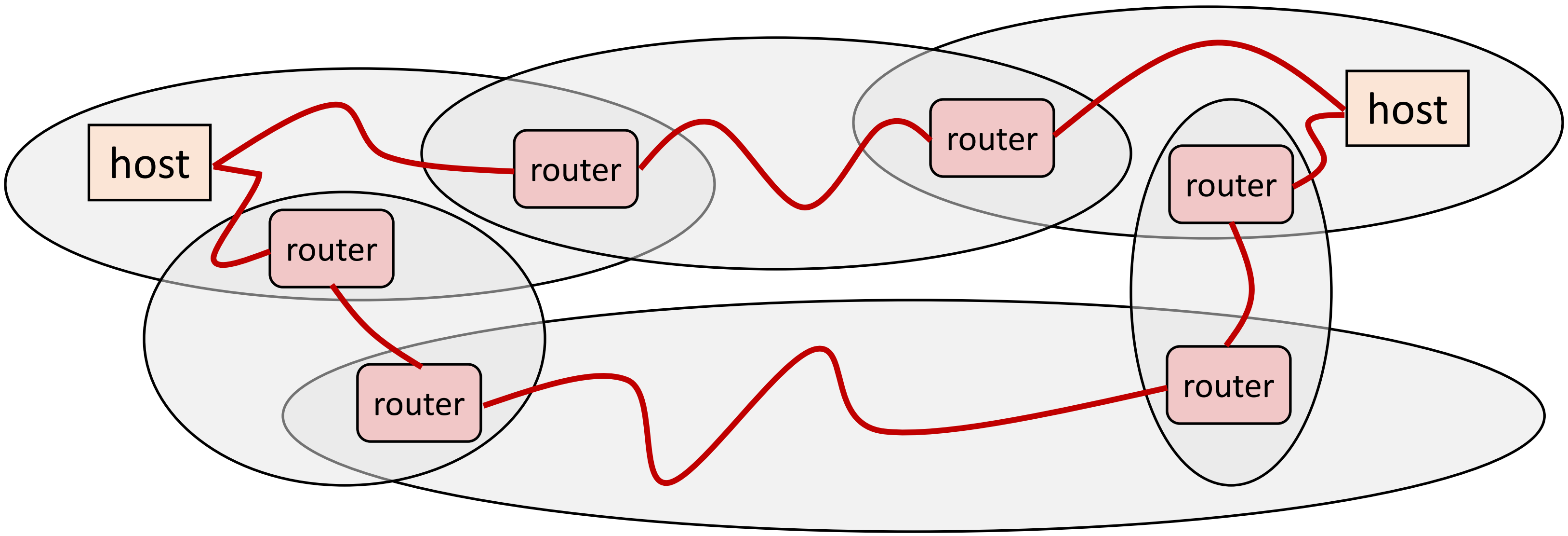
Next Level: internets

- 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, Fiber 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

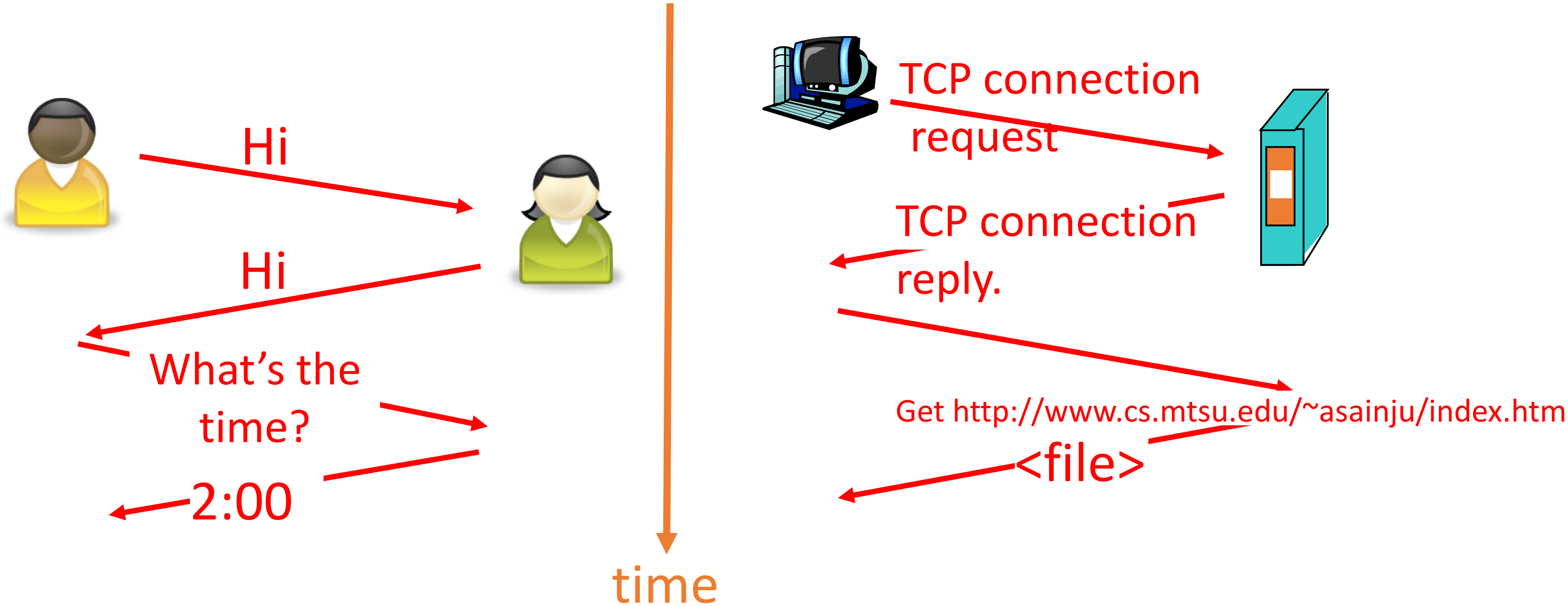
The Notion of an internet Protocol

- **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's a protocol?

- **a human protocol and a computer network protocol:**



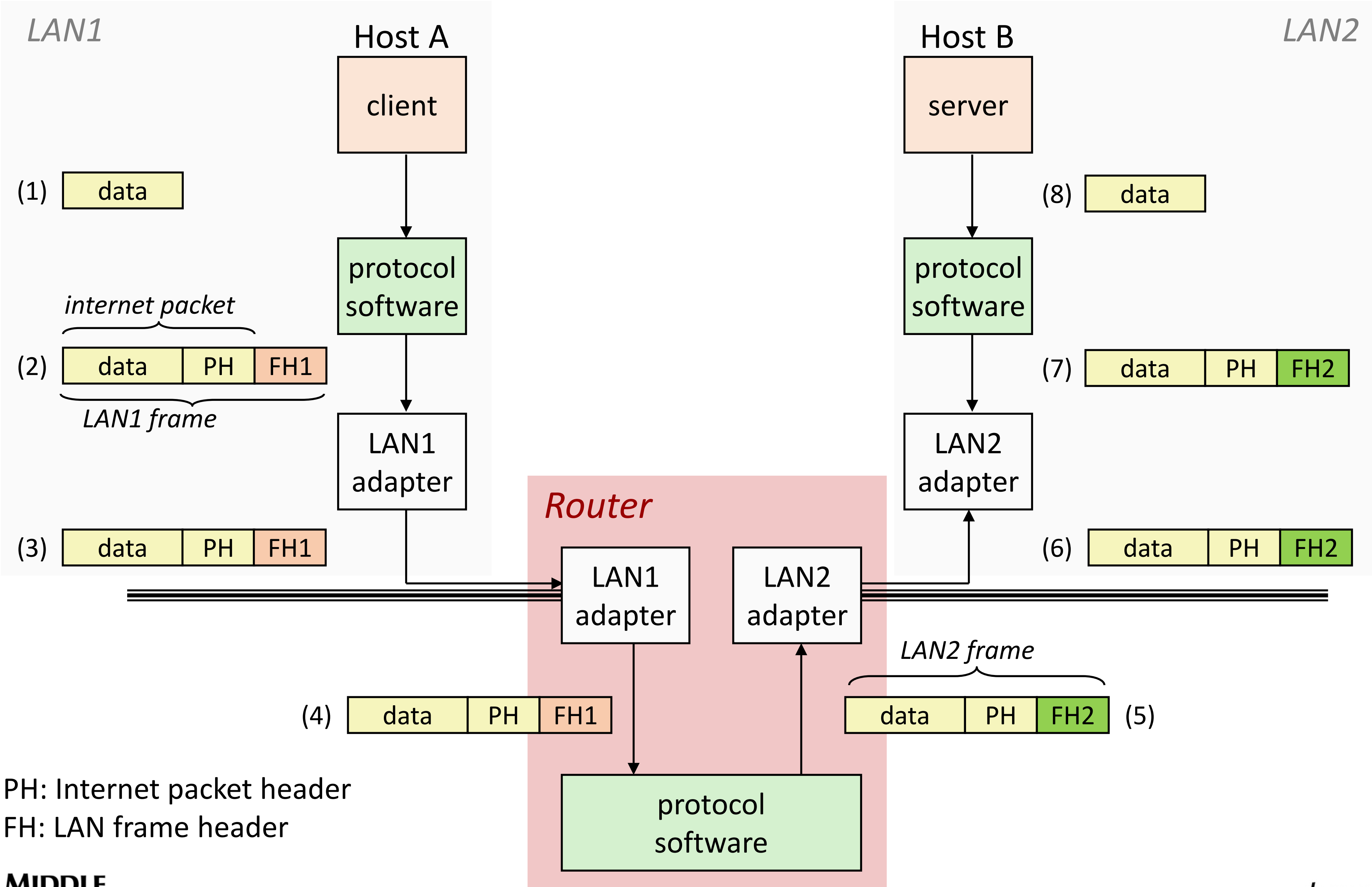
protocols define format, order of msgs sent and received among network entities, and actions taken on msg transmission, receipt

What Does an internet Protocol Do?

- **Provides a *naming scheme***
  - Different LAN technologies have different and incompatible ways of assigning address to host.
  - An internet protocol smooths these differences by defining 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 the 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***
  - *CSCI 4300 – Data Communication and Network*

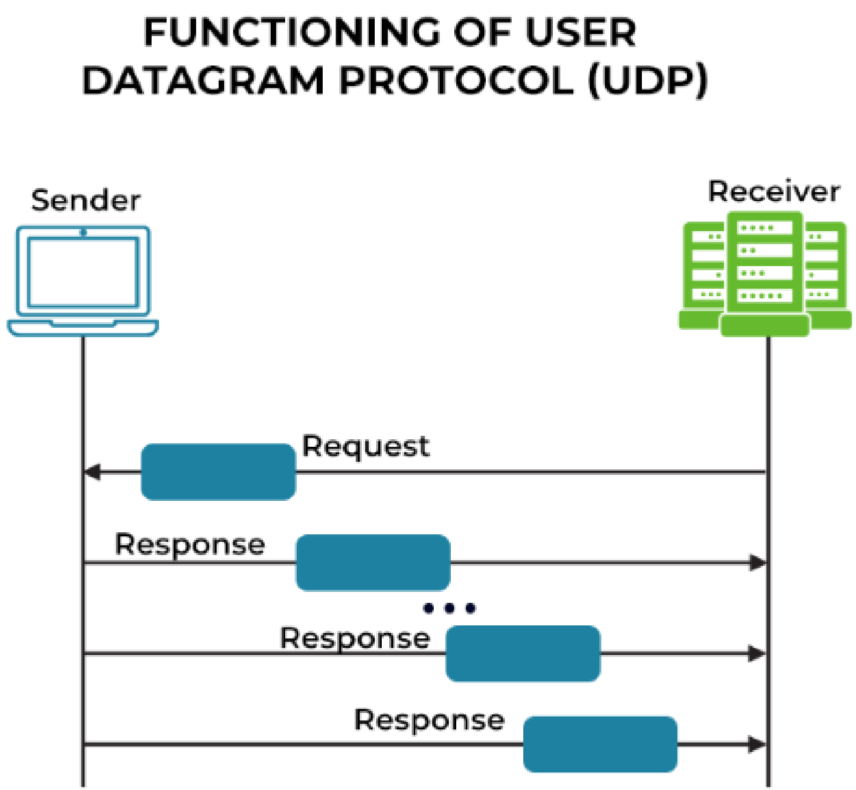
Global IP Internet (upper case)

- **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*
    - IP mechanism is unreliable in the sense that it makes no effort to recover if datagrams are lost or duplicated in the network.
  - UDP (Unreliable Datagram Protocol)
    - extends IP slightly 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***

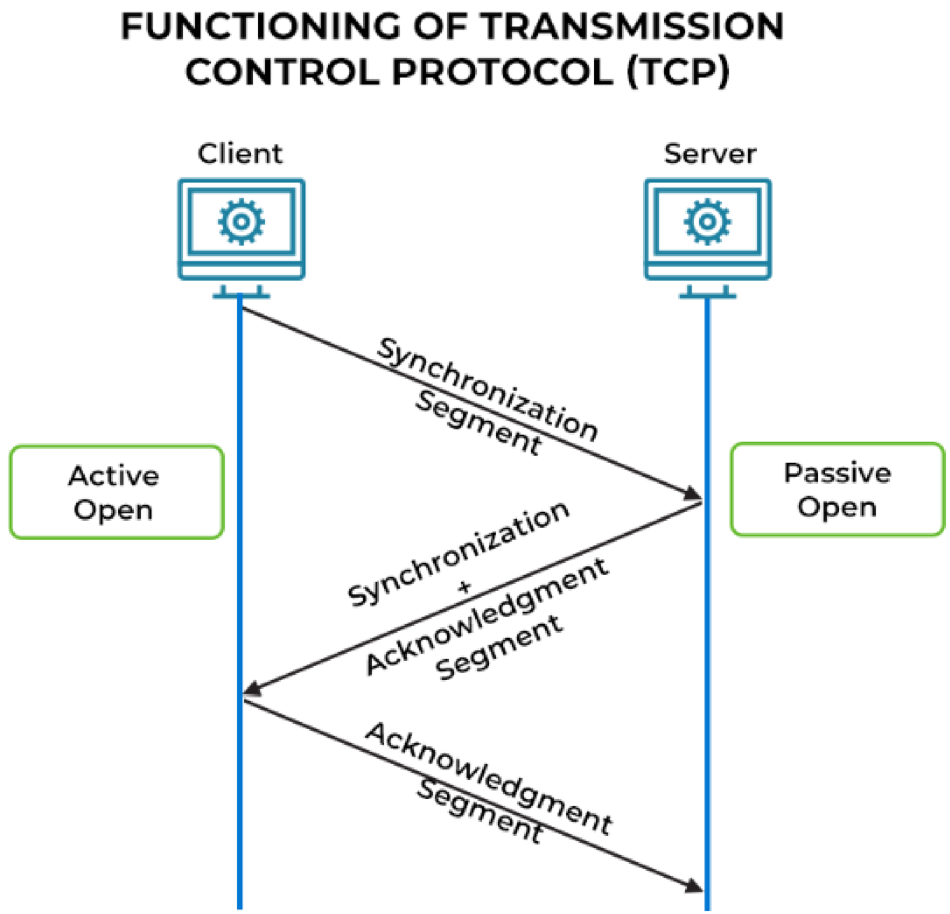
UDP



UDP enables continuous data transmission (i.e., response) without acknowledging or confirming the connection

https://www.spiceworks.com/tech/networking/articles/tcp-vs-udp/

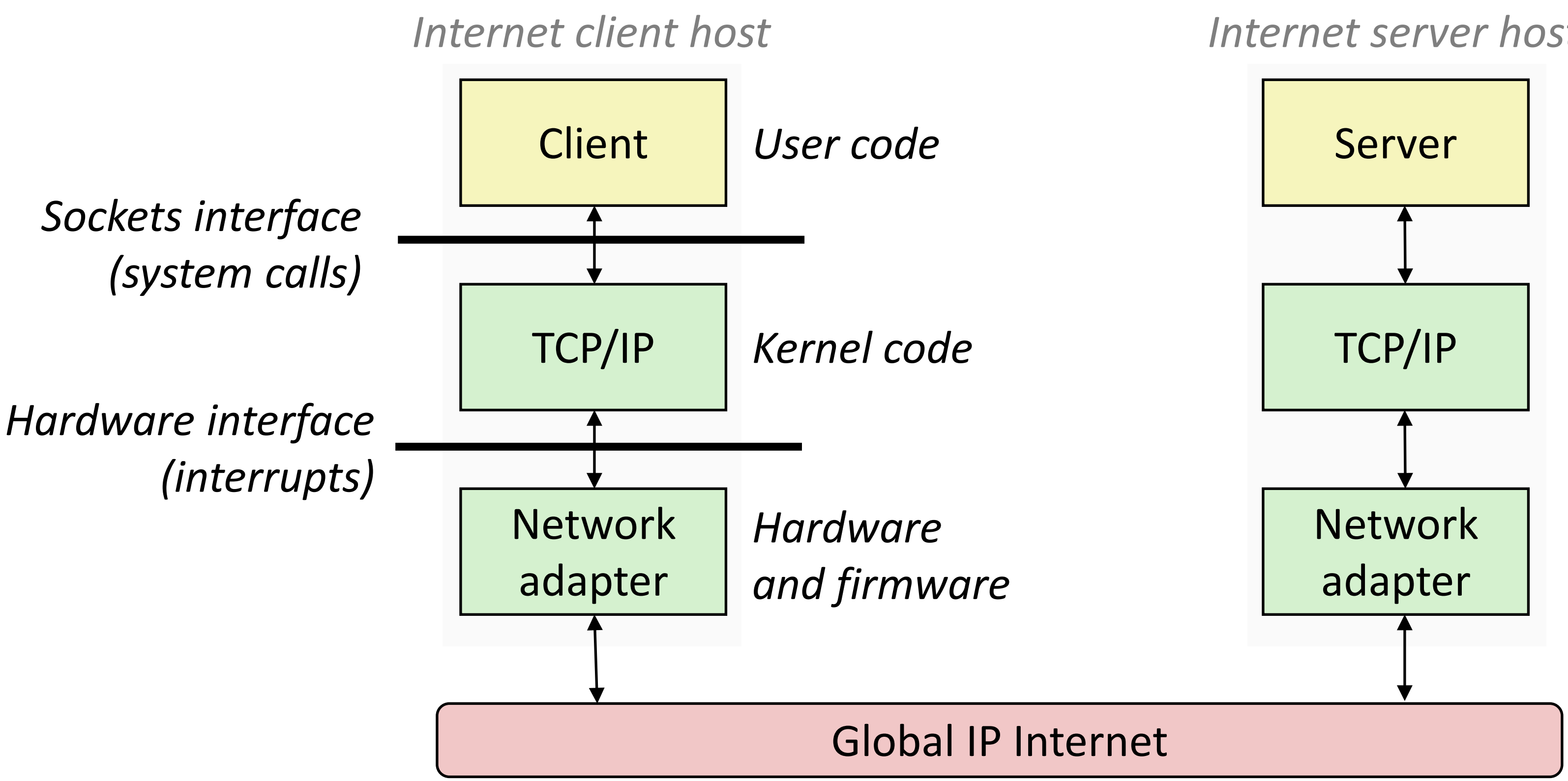
TCP



TCP relies on a three-way handshake (synchronization, synchronization acknowledgment, and final acknowledgment)

https://www.spiceworks.com/tech/networking/articles/tcp-vs-udp/

Hardware and Software Organization of an Internet Application



- Each Internet host runs software that implements the TCP/IP protocol.
- Internet client and servers communicate using a mix of socket interface functions and Unix I/O functions.
- The socket functions are typically implemented as system call.

A Programmer's View of the Internet

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

- 161.45.162.100

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

- 161.45.162.100 is mapped to www.cs.mtsu.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

- **As of 2022, vast majority of Internet traffic still carried by IPv4**
  - Only around 38% of users access Google services using IPv6.

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

(1) IP Addresses

- **An IP address is an unsigned 32-bit integer.**

- **Network program strores IP addresses in the *IP address structure (shown below)***
  - IP addresses are always stored in memory in *network byte order* (big-endian byte order)
  - Because Internet hosts can have different host byte order, TCP/IP defines a uniform network byte order (big-endian byte order) for any integer data item, such as IP address, that is carried across the network in a packet header.
  - Next slides shows a list of Unix functions for converting network and host byte order.

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

Unix Functions for byte order conversion

Useful network byte-order conversion functions
("l" = 32 bits and "s" = 16 bits)
(n => network, h => host)

`htonl`: convert unit32_t from host to network byte order
`htons`: convert unit16_t from host to network byte order
`ntohl`: convert unit32_t from network to host byte order
`ntohs`: convert unit16_t from network to host byte order

```
#include <arpa/inet.h>
//Return value in network byte order
uint32_t htonl(uint32_t host);
uint16_t htons(uint16_t host);

//Return value in host byte order
uint32_t ntohl(uint32_t net);
uint16_t ntohs(uint16_t net);
```

Dotted Decimal Notation

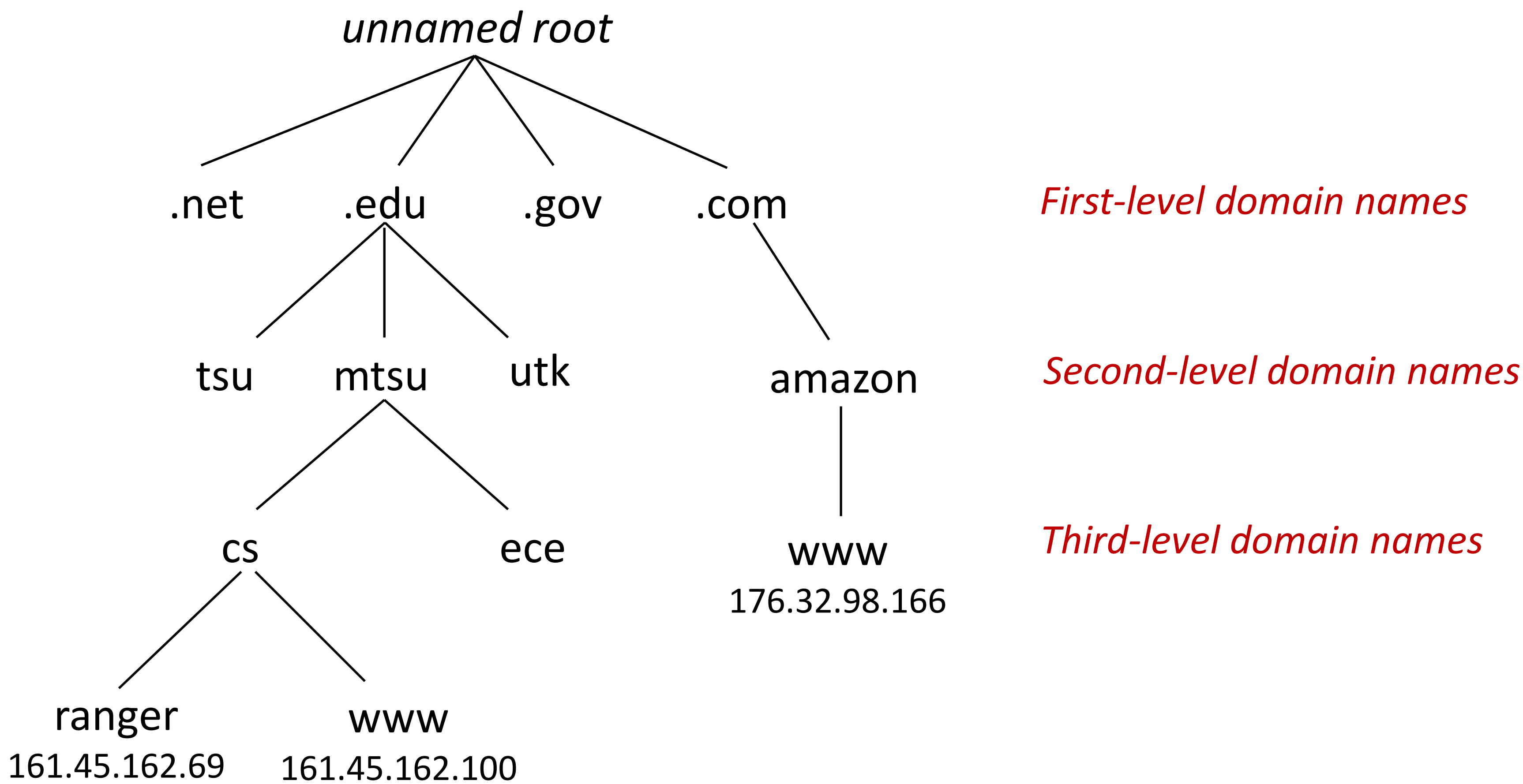
- By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period
    - `IP address:` 0x8002C2F2 = 128.2.194.242

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

Exercise

- **Convert the following 32-bit addresses into dotted decimal notation.**
  - 0xffffffff
  - 0x0aff090c

(2) Internet Domain Names



History

- **Before Domain Name System(DNS), all mappings were in hosts.txt**
  - /etc/hosts on linux
  - C:\Windows\System32\drivers\etc\hosts on windows

- **Centralized, manual system**
  - SRI was the first organization to assign website address such as www.sri.com with extensions such as ".com", ".org".
  - These addresses were assigned to network hosts by the Network Information Center (NIC), managed by SRI from 1970 until 1991.
  - Changes were submitted to SRI via email
  - Machines periodically FTP new copies of *hosts.txt*
  - Administrators could pick names at their discretion
  - Any name was allowed

Towards DNS

- **Eventually, the *hosts.txt* system fell apart**
  - Not scalable, SRI couldn't handle the load
  - Hard to enforce uniqueness of names
    - e.g MIT
      - Massachusetts Institute of Technology?
      - Melbourne Institute of Technology?
  - Many machines had inaccurate copies of *hosts.txt*
- **Thus, DNS was born**

Domain Naming System (DNS)

- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*
  - No Centralization

- 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.

- Hierarchical namespace
  - As opposed to original, flat namespace
  - E.g. .com -> google.com -> mail.google.com

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
csci3240-00.cs.mtsu.edu
```

Properties of DNS Mappings (cont)

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

```
linux> nslookup csci3240-00.cs.mtsu.edu
Address: 161.45.164.116
```

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

```
linux> nslookup www.mtsu.edu
Address: 10.14.0.116
linux> nslookup w1.mtsu.edu
Address: 10.14.0.116
```

Properties of DNS Mappings (cont)

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

```
linux> nslookup www.twitter.com
Address: 199.16.156.6
Address: 199.16.156.70
Address: 199.16.156.102
Address: 199.16.156.230

linux> nslookup twitter.com
Address: 199.16.156.102
Address: 199.16.156.230
Address: 199.16.156.6
Address: 199.16.156.70
```

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

```
linux> nslookup csc.mtsu.edu
*** Can't find csc.mtsu.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)

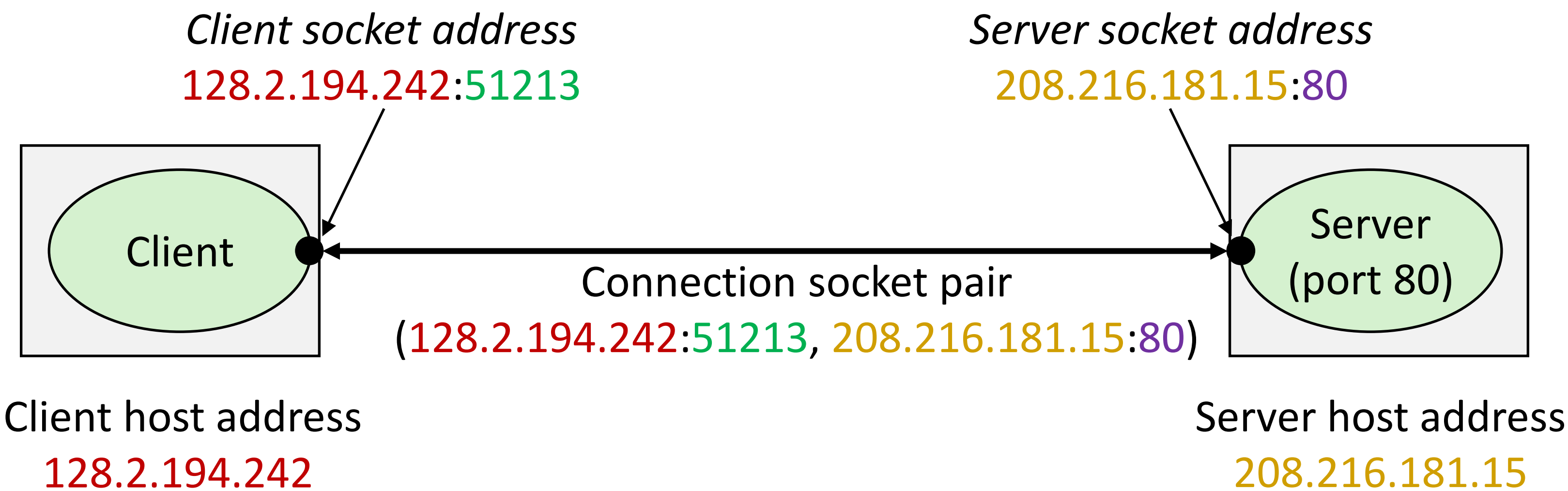
Well-known Ports and Service Names

- **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

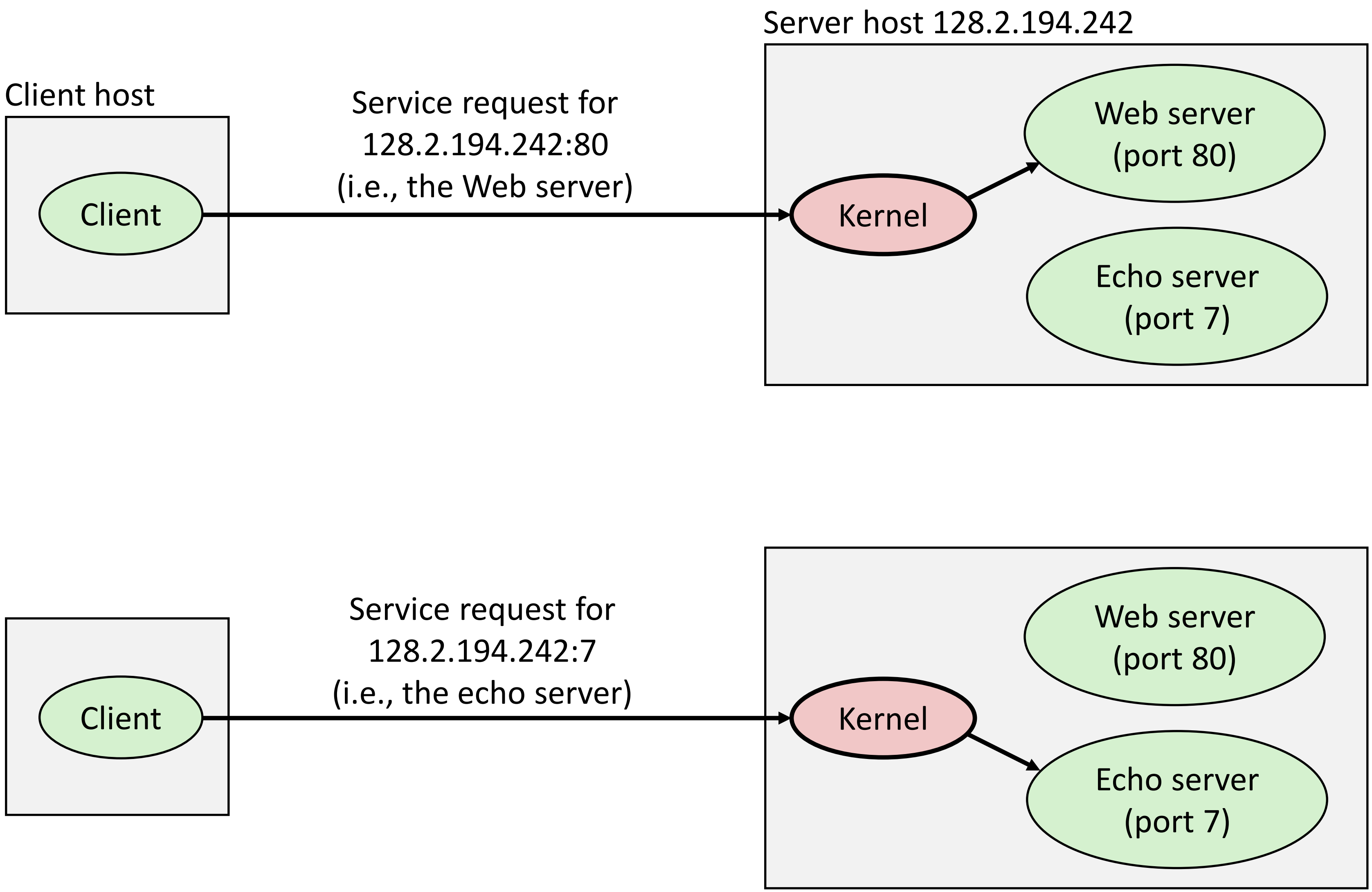
- **A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)**
  - **`(cliaddr:cliport, servaddr:servport)`**



51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

Using Ports to Identify Services



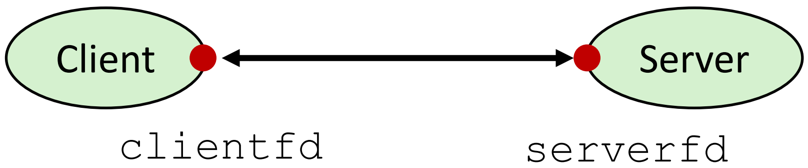
Sockets Interface

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

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

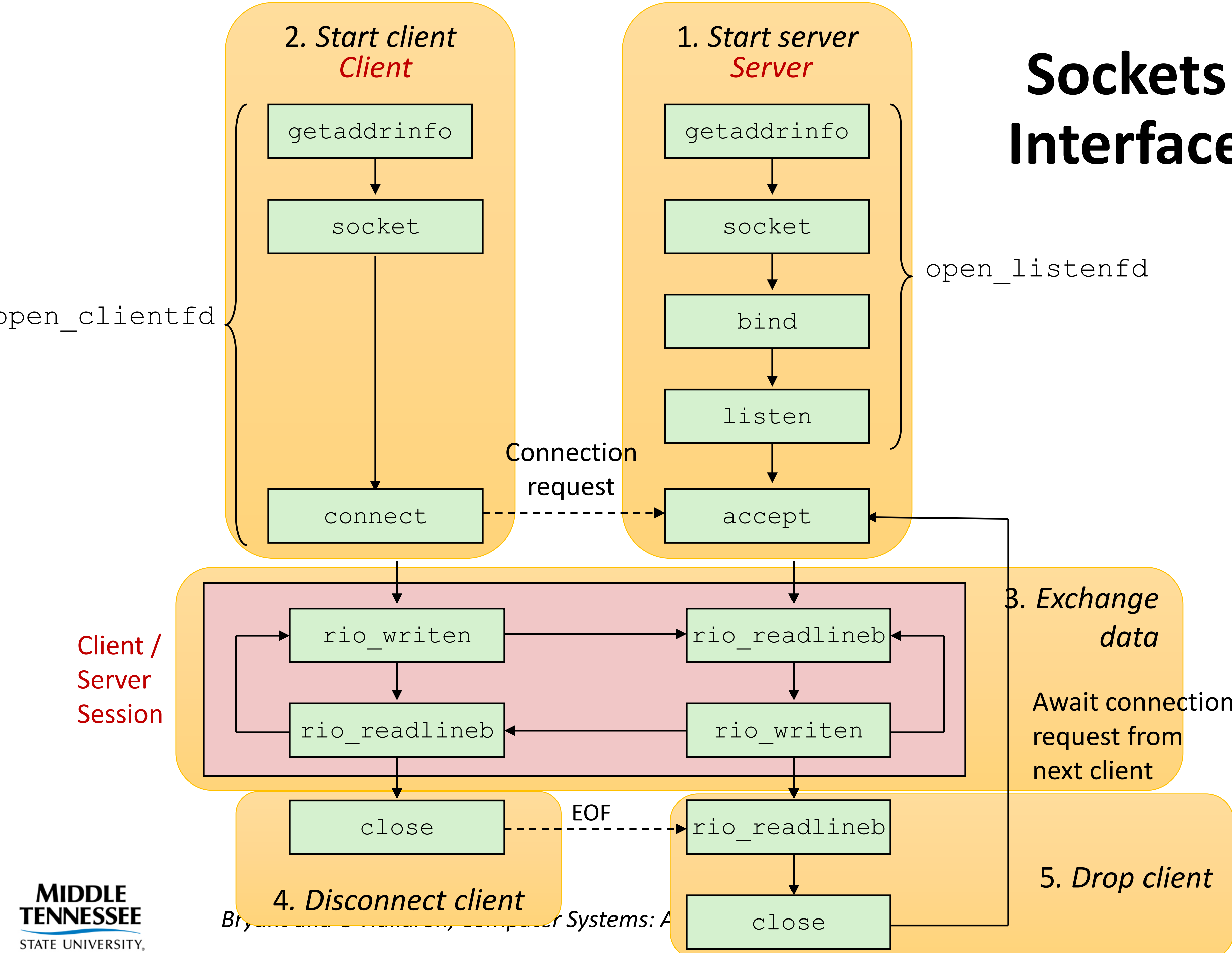
Sockets

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application (program), 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**



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

Sockets Interface



Sockets Address Structure

- Internet socket addresses are stored in 16-byte structures having the type **`sockaddr_in`**
- For Internet applications, **`sin_family`** field is AF_INET
- The **`sin_port`** filed is a 16-bit port number
- And, the **`sin_addr`** field contains a 32-bit IP address.
- The IP address and port number are always stored in network byte order

```
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) */
};
```

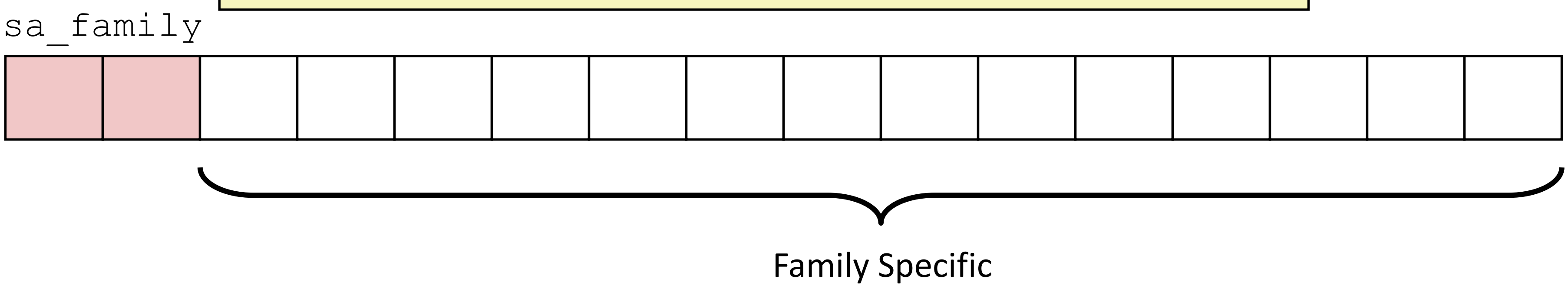
Here, the _in suffix in sockaddr_in refers to internet, not input.

Socket Address Structures

- **Generic socket address:**
  - The **`connect`**, **`bind`**, and **`accept`** functions requires a pointer to a protocol specific socket address structure.
  - The problem faced by the designer of the sockets interface was how to define these functions to accept any kind of socket address structure.
  - Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed.
  - The solution was to define sockets function to expect a pointer to a generic `sockaddr` structure and then require applications to cast any pointers to protocol-specific structures to this generic structure.
  - For casting convenience, we can use typedef:

```
typedef struct sockaddr SA;
```

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

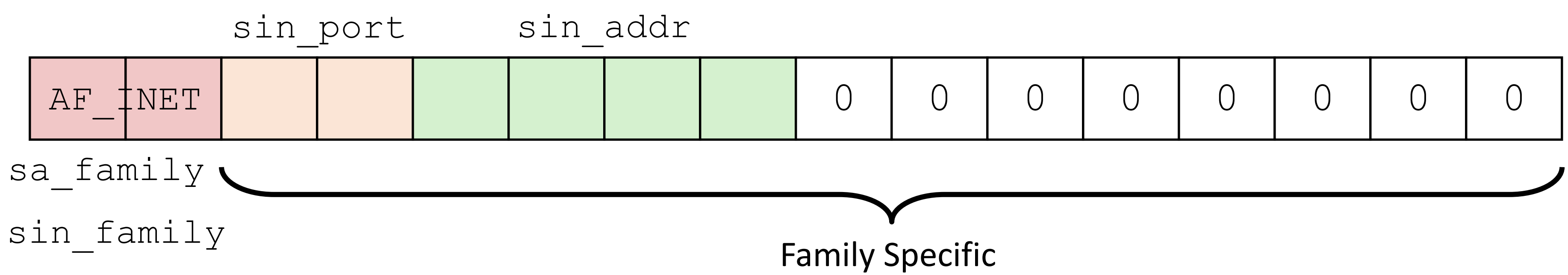


Socket Address Structures

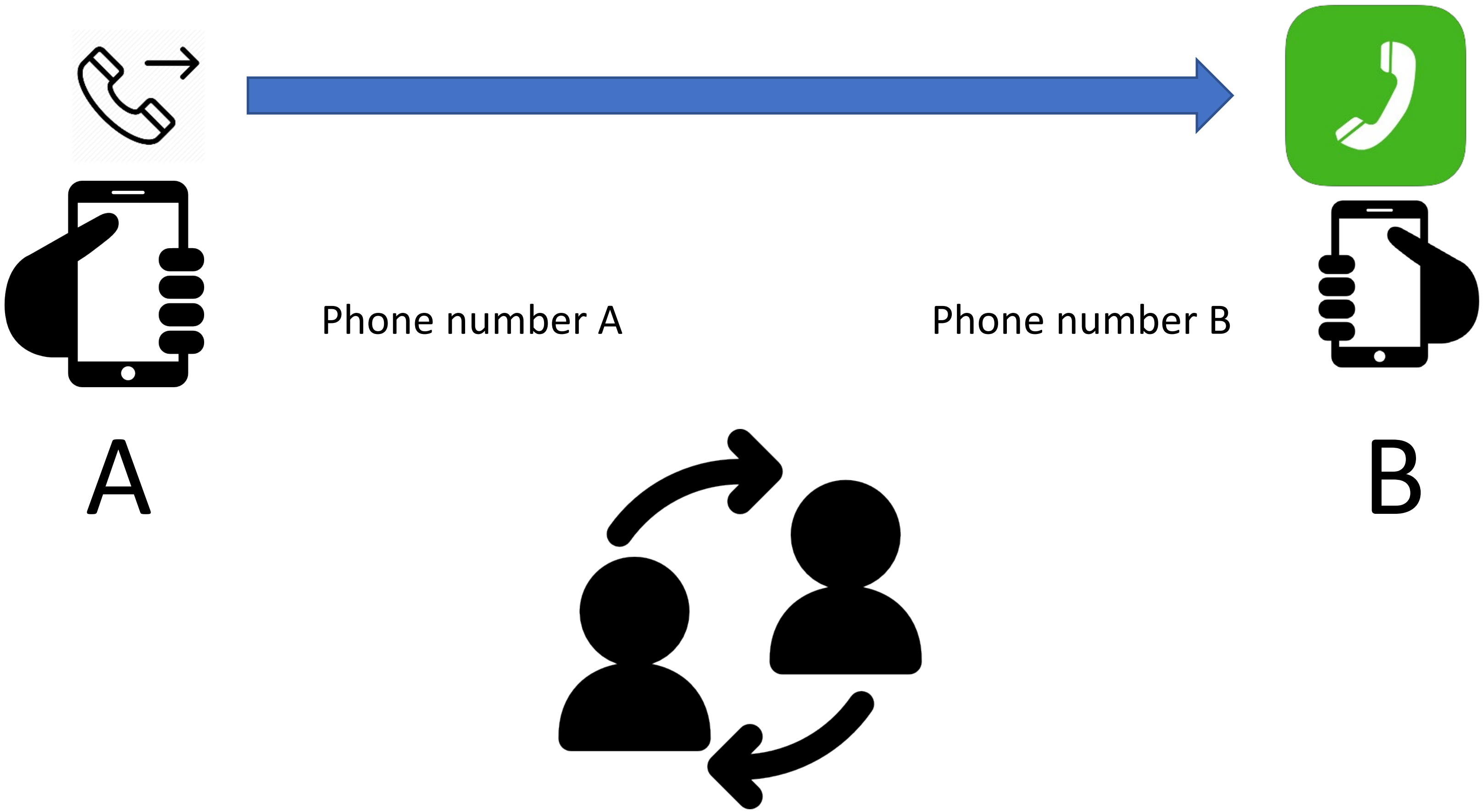
- **Internet-specific socket address:**
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

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

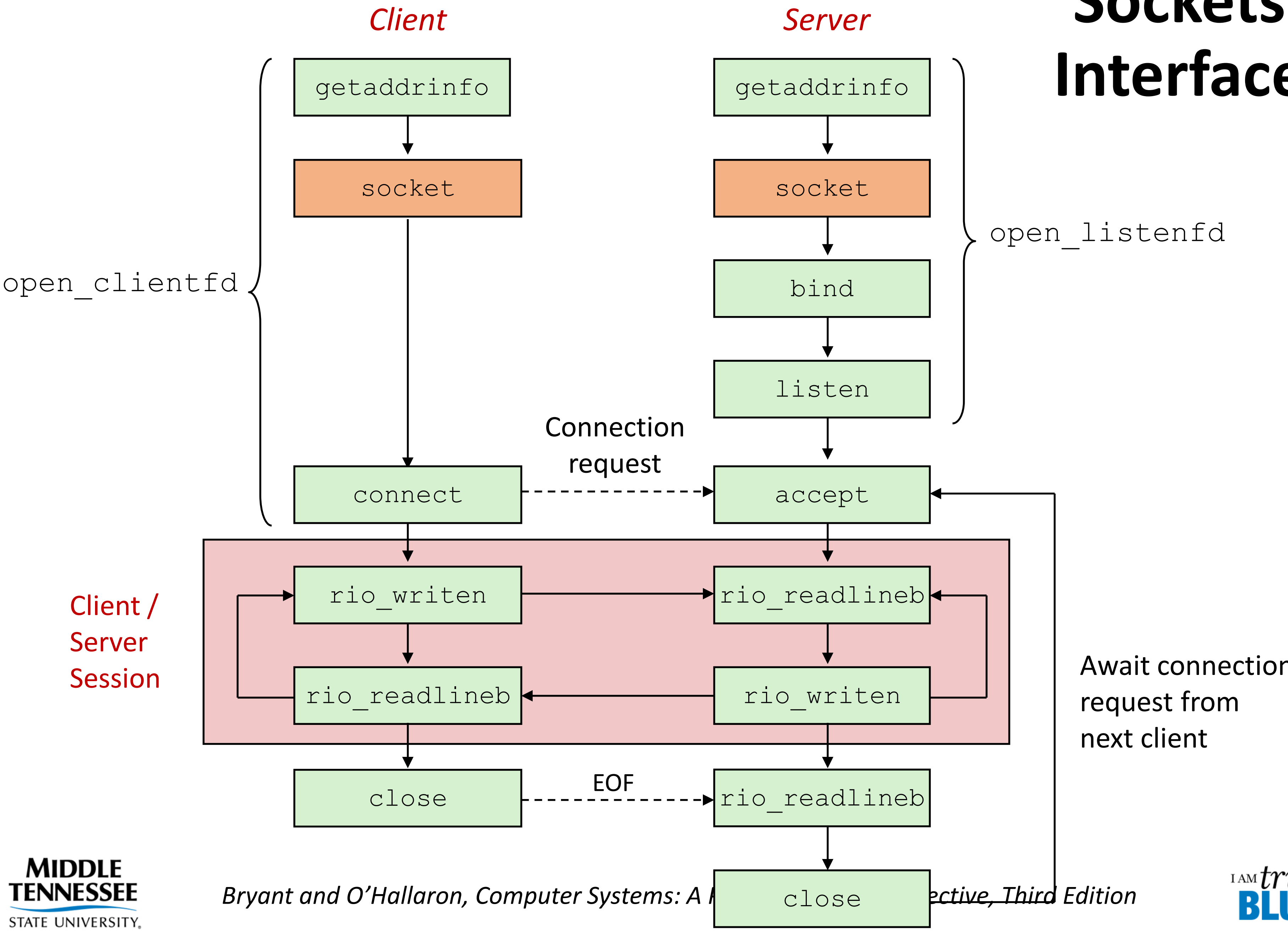
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```



Socket Programming: An Analogy with a Phone call



Sockets Interface



Sockets Interface: `socket`

- TCP Socket can be compared to a phone
  - We need a device to call someone or accept their phone calls, right?

- To perform network I/O, the first thing a process must do is, call the socket function to create a specific type of socket by specifying the type of communication protocol desired, protocol family, etc.

Sockets Interface: `socket`

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

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

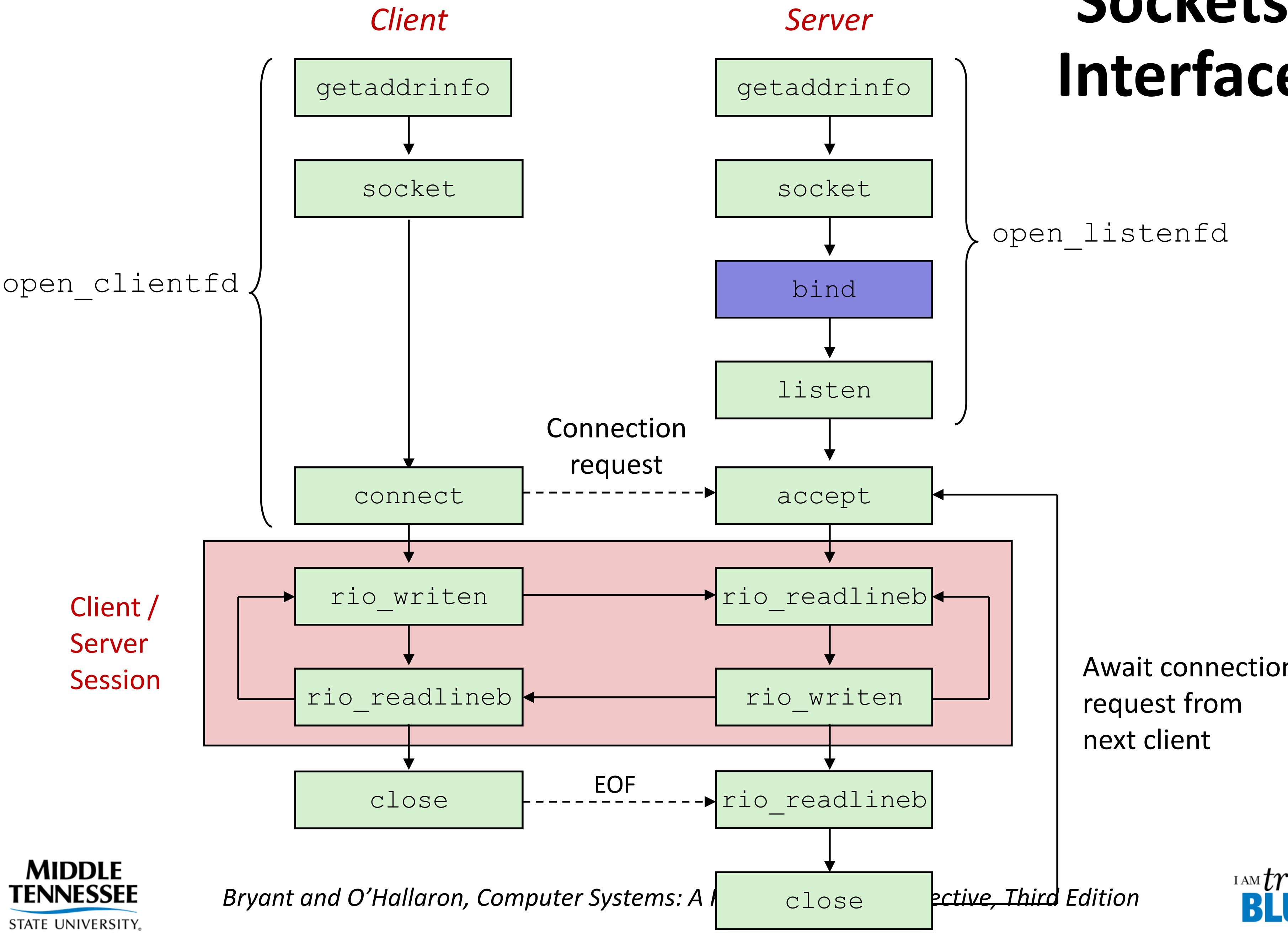
- ```
  int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
  ```

Indicates that we are using 32-bit IPV4 addresses

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

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

Sockets Interface



Sockets Interface: `bind`

- It is like registering a phone number to the phone (to connect to other phones).

- Sockets do not have a complete address in the beginning to start transferring the data, so we bind a Socket to a port.
  - Analogy: Phones do not have a phone number at the beginning to make phone calls, so we register a phone number.

- The process of allocating a port number to a socket is called ‘binding’.

Sockets Interface: `bind`

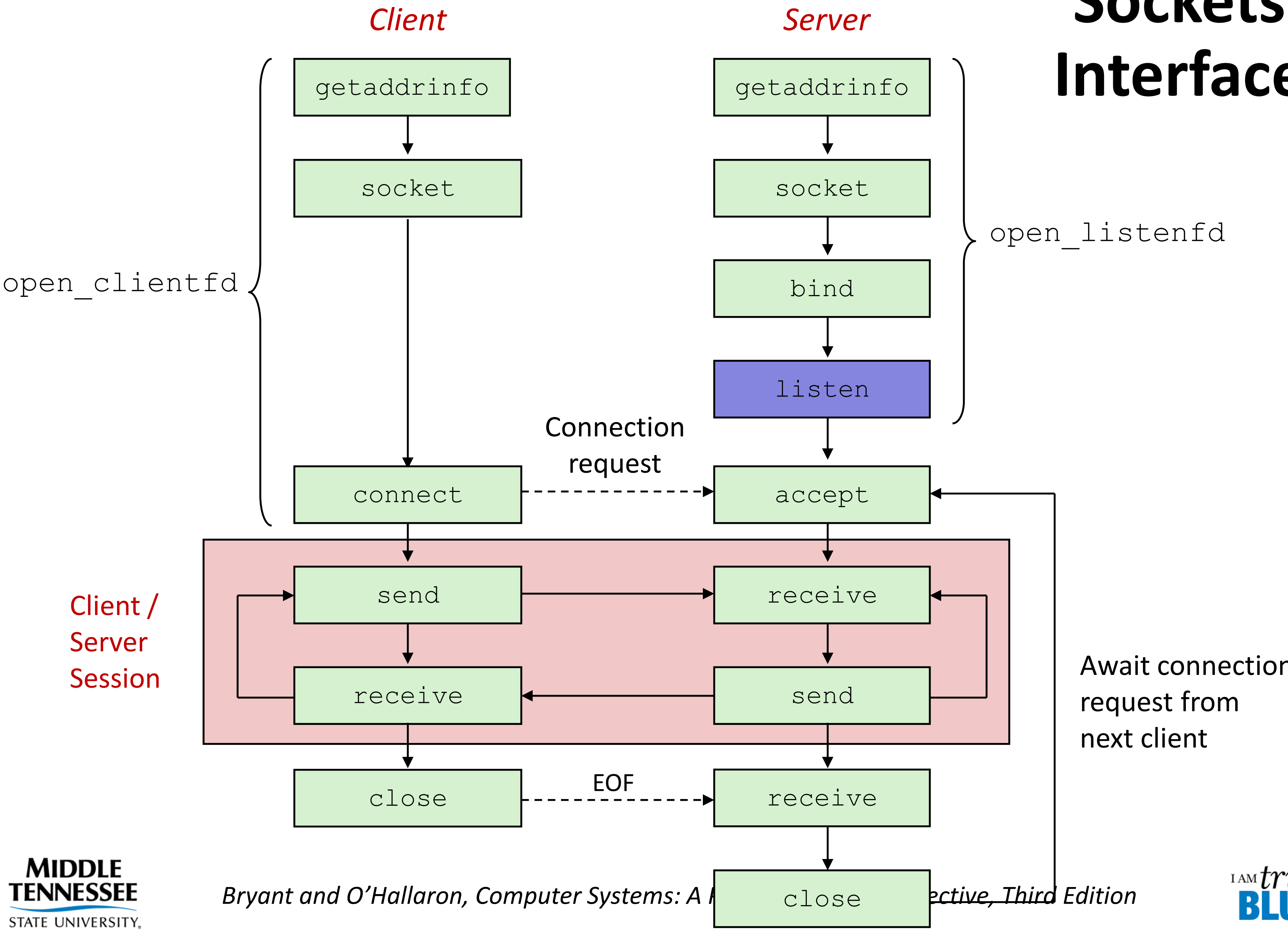
- A server uses bind to ask the kernel to associate the server's socket address with a **socket descriptor**:

```
#include <sys/socket.h>
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

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

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

Sockets Interface



Sockets Interface: `listen`

- Clients are active entities that initiate connection requests.

- Servers are passive entities that wait for connection requests from clients.

- By default, the kernel assumes that a descriptor created by the socket function corresponds to an *active* socket that will be live on the client end of a connection.

- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

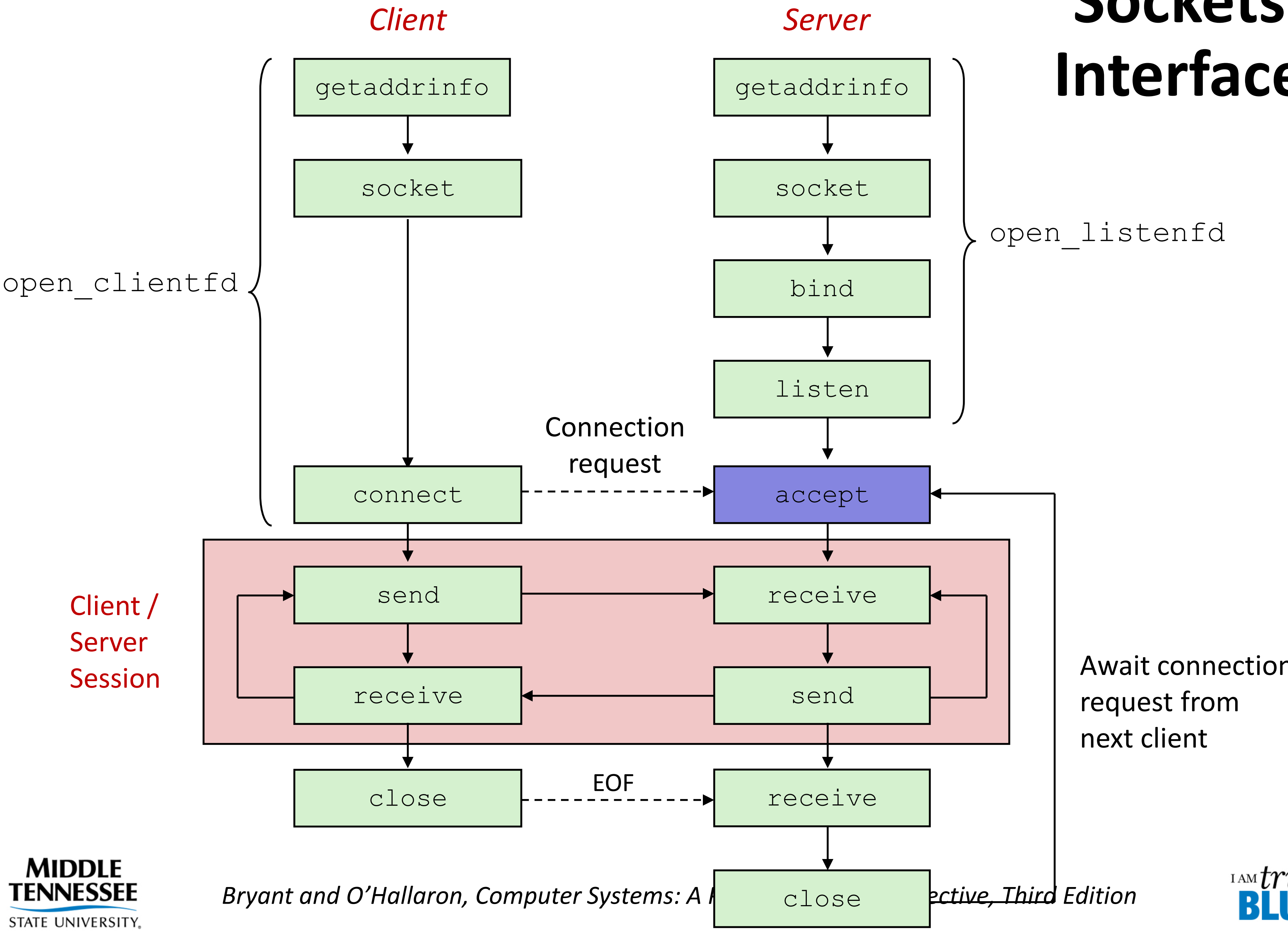
```
int listen(int sockfd, int backlog);
```

Sockets Interface: `listen`

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

- The listen function converts sockfd from an active socket to a *listening socket* that can accept connection requests from clients.
- The backlog is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.
- We will typically set it to a large value, such as 1024.

Sockets Interface



Sockets Interface: `accept`

- Accept a connection request
  - Transition of the connection request from listen() method to an actual Socket
- Servers wait for connection requests from clients by calling `accept` function:

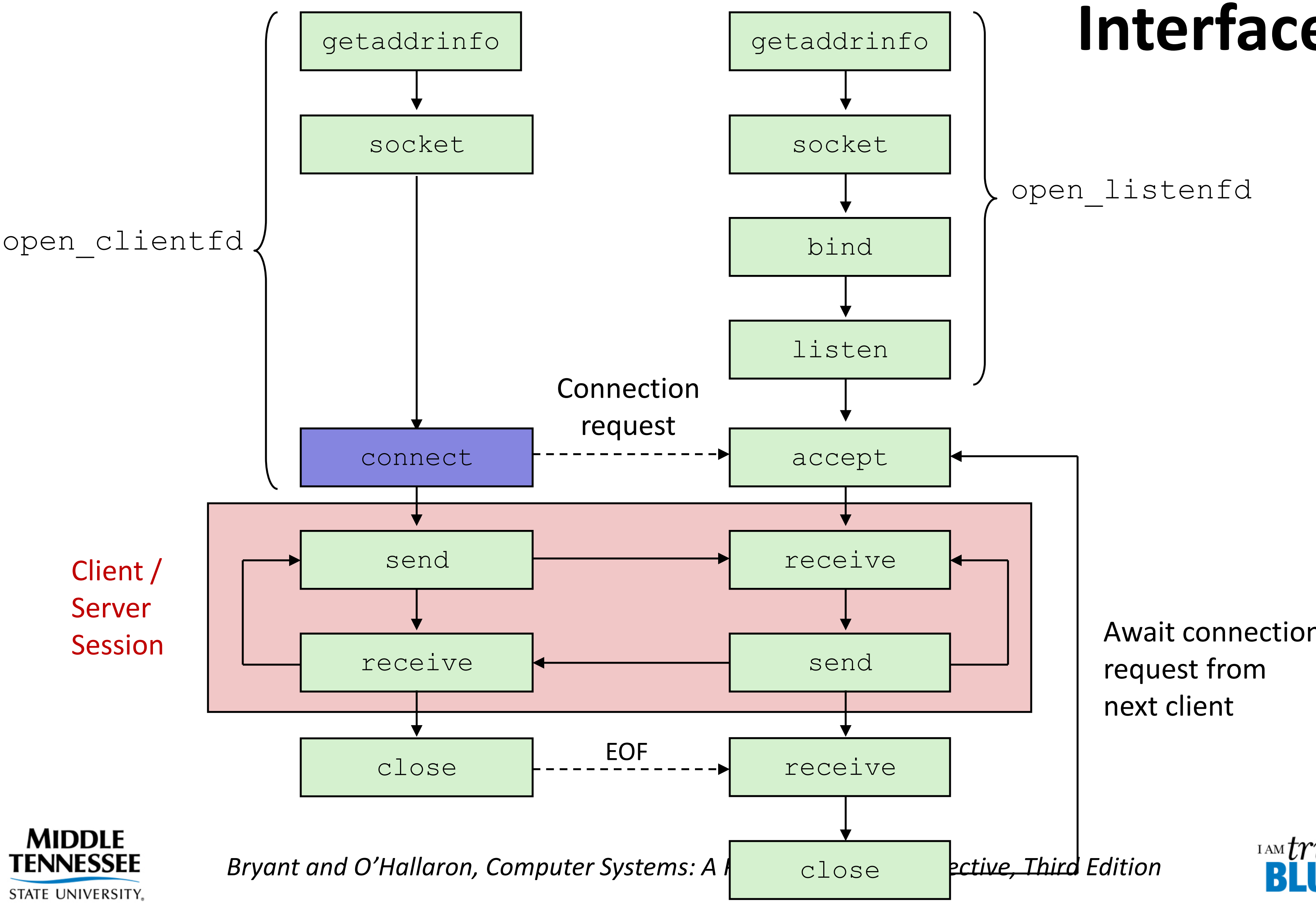
```
int accept(int listenfd, SA *addr, int *addrlen);
```

- The accept function waits for connection request to arrive on the connection bound to listening descriptor `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.
- Finally, it returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.

Make the logical steps from the 4 previous parts

- **Sequence**
  - Step 1. Creating the socket (socket function call)
  - Step 2. Bind the socket to a port (bind function call)
  - Step 3. Listen for incoming connection requests & identify ones (listen function call)
  - Step 4. Accept the identified connection request and open the socket (accept function call)

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

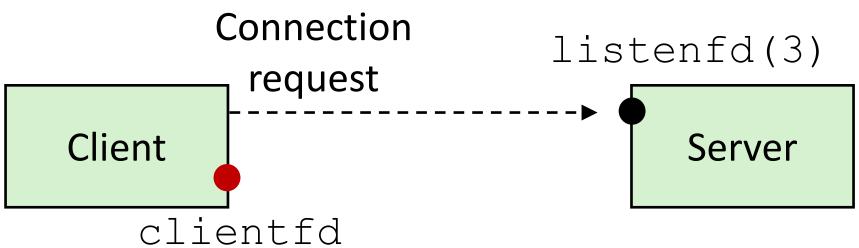
```
(x:y, addr.sin_addr:addr.sin_port)
```

    - 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.*

`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`

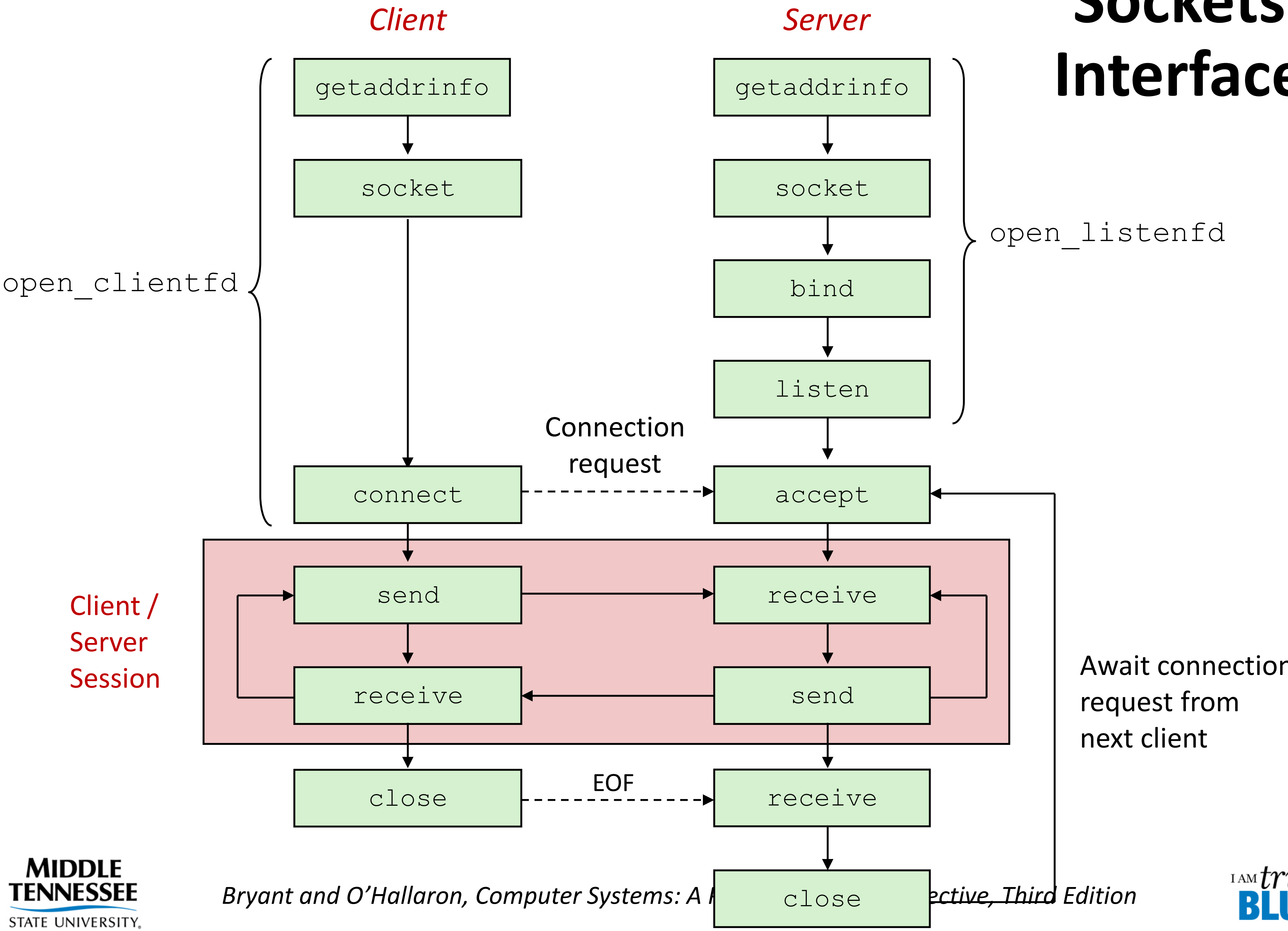
Connected vs. Listening Descriptors

- **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

Sockets Interface



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

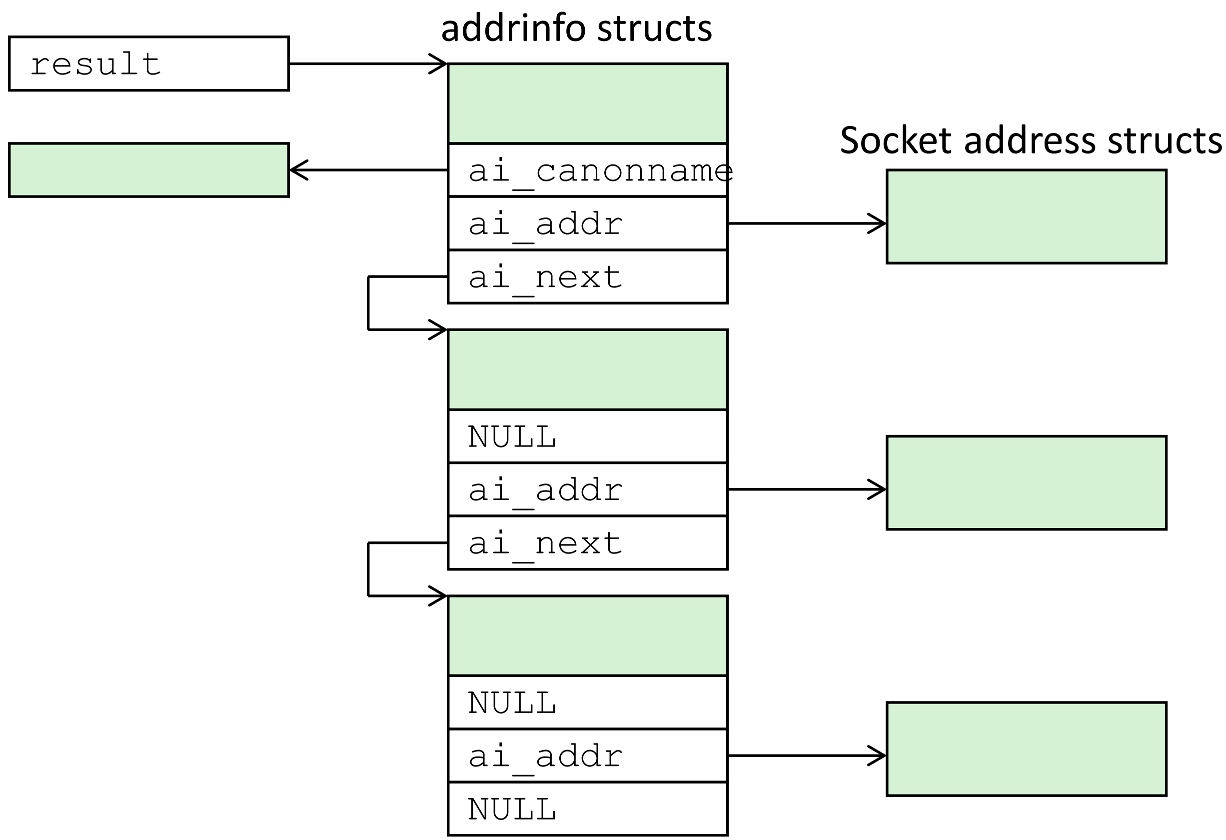
```
int getaddrinfo(const char *host,            /* Hostname or address */
                const char *service,         /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);   /* Output linked list */

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`



- **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

hostinfo.c

```
#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

```
csci3240-00> ./hostinfo localhost
127.0.0.1

csci3240-00 > ./hostinfo www.cs.mtsu.edu
161.145.162.100

csci3240-00 > ./hostinfo google.com
142.250.10.147
142.250.10.104
142.250.10.106
142.250.10.99
142.250.10.103
142.250.10.105
```