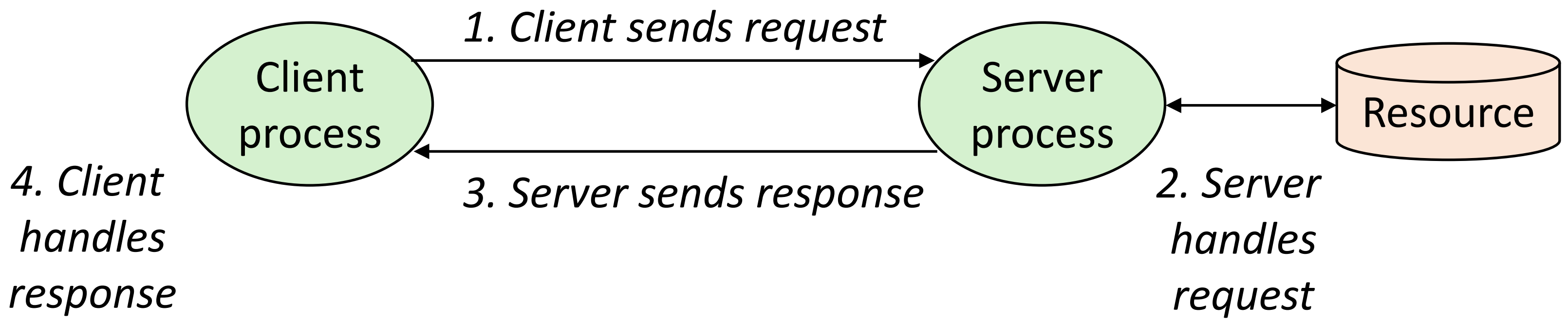
CS 449 - Intro to Systems Software

Network Programming

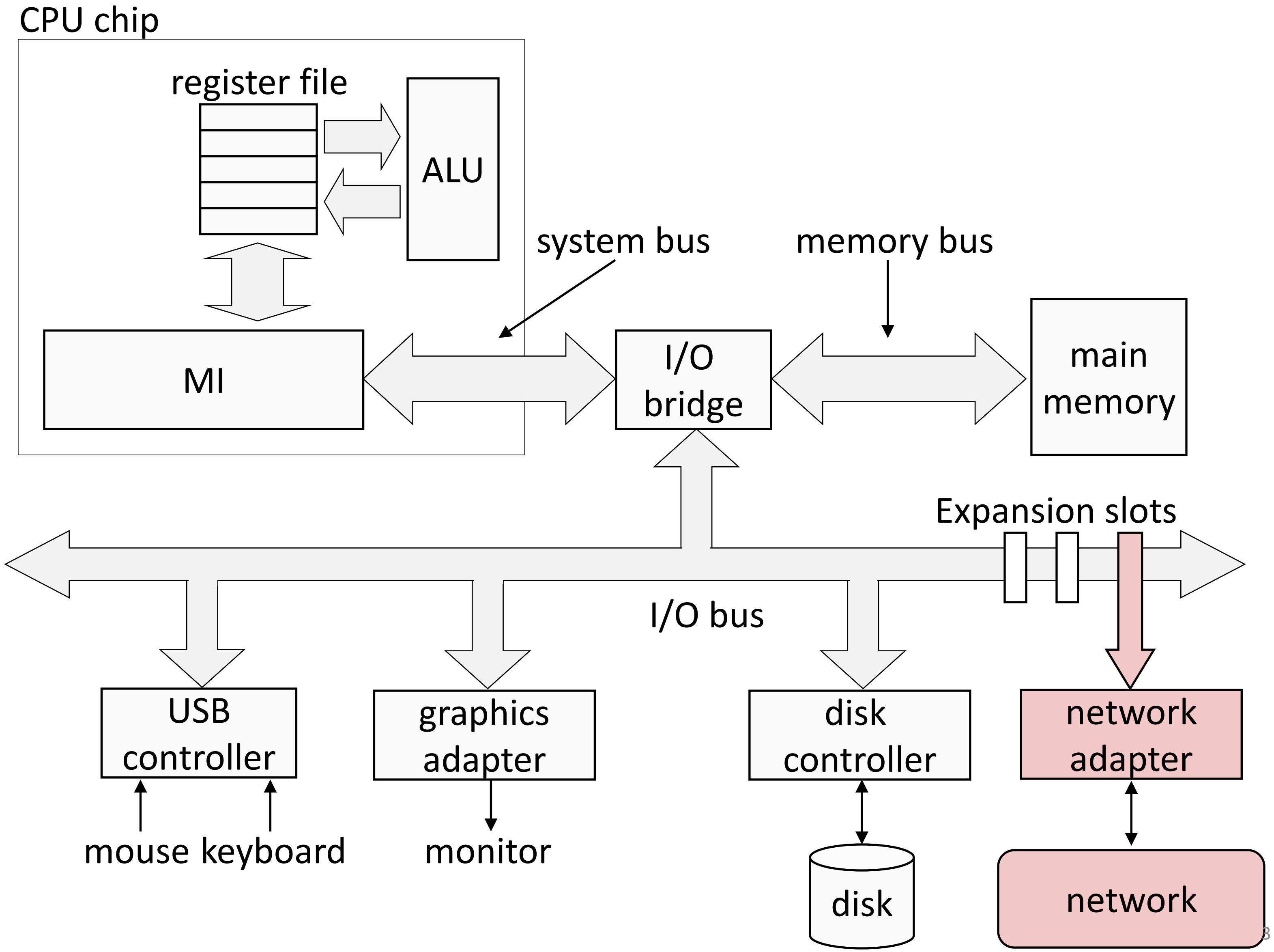
A Client-Server Transaction

- 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)

Hardware Organization of a Network Host

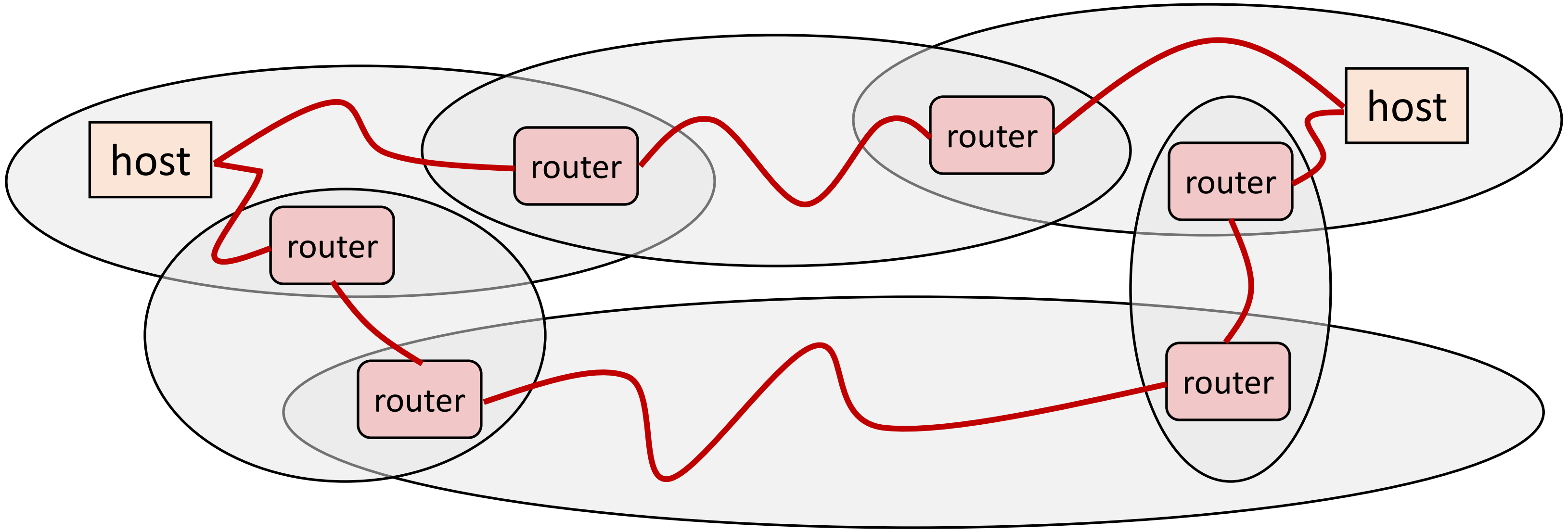


Computer Networks

- 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")

* Not to be confused with a Storage Area Network

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

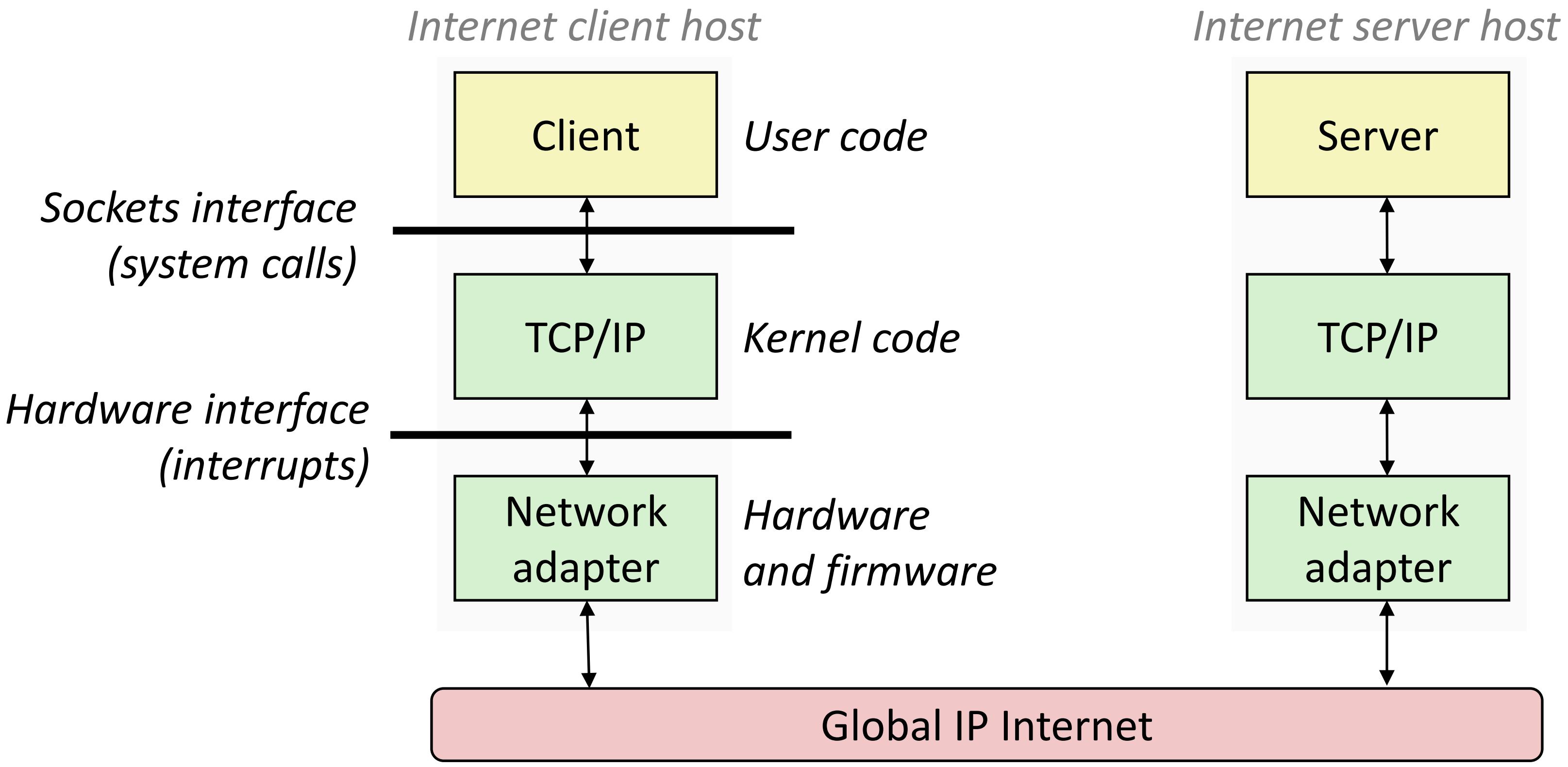
Global IP Internet (upper case)

- 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



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*
    - 136.142.156.73 is mapped to www.cs.pitt.edu

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

(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.

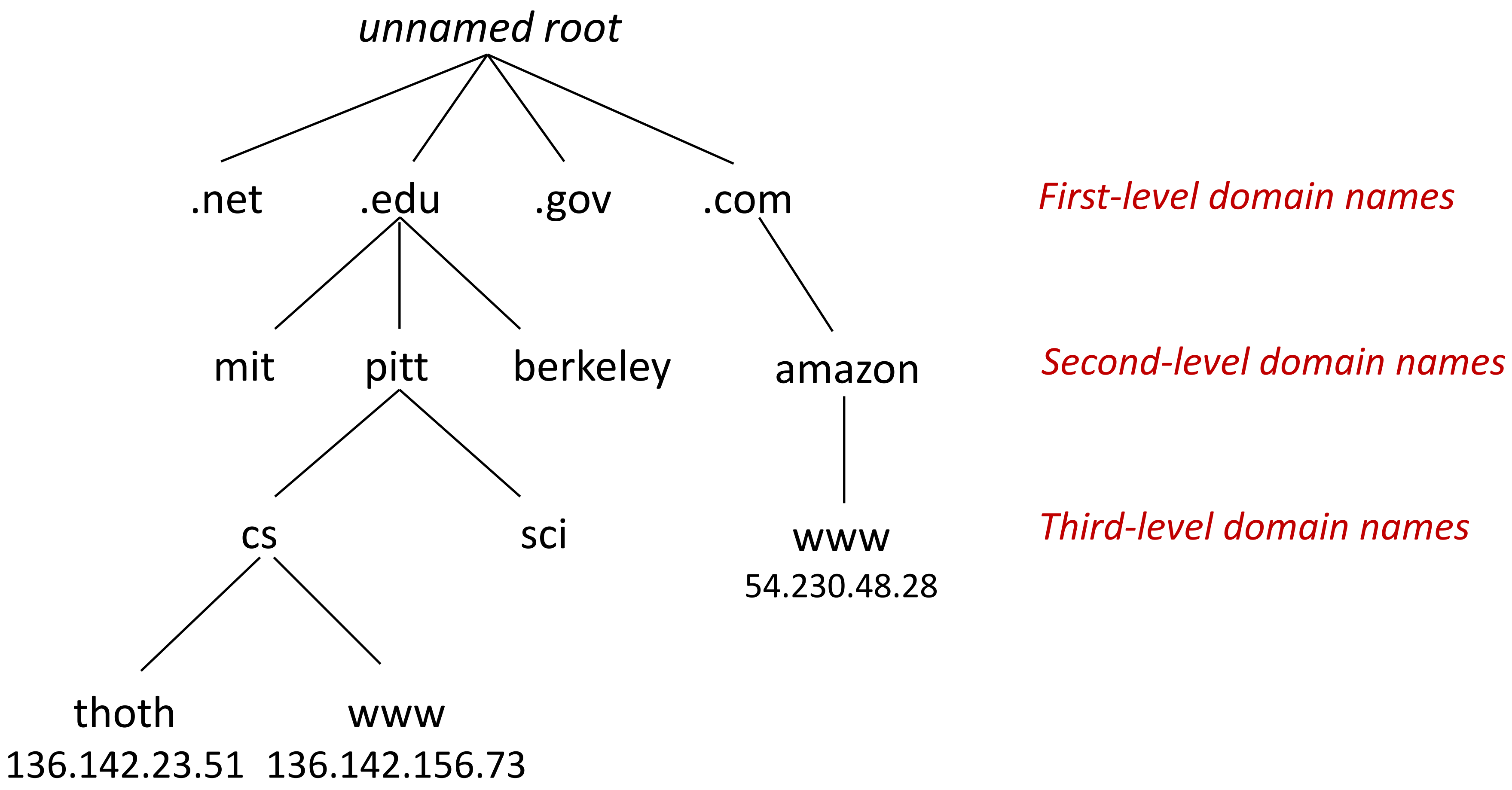
```
/* Internet address structure */
struct in_addr {
    uint32_t  s_addr; /* network byte order (big-endian) */
};
```

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 to convert between IP addresses and dotted decimal format.

(2) Internet Domain Names



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.

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
thoth.cs.pitt.edu
```

Properties of DNS Mappings (cont)

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

```
linux> nslookup thoth.cs.cmu.edu
Address: 136.142.23.51
```

- Multiple domain names mapped to the same IP address:

```
linux> nslookup cs.pitt.edu
Address: 136.142.156.73
linux> nslookup sci.pitt.edu
Address: 136.142.156.73
```

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 bla.cs.pitt.edu
(No Address given)
```

(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)

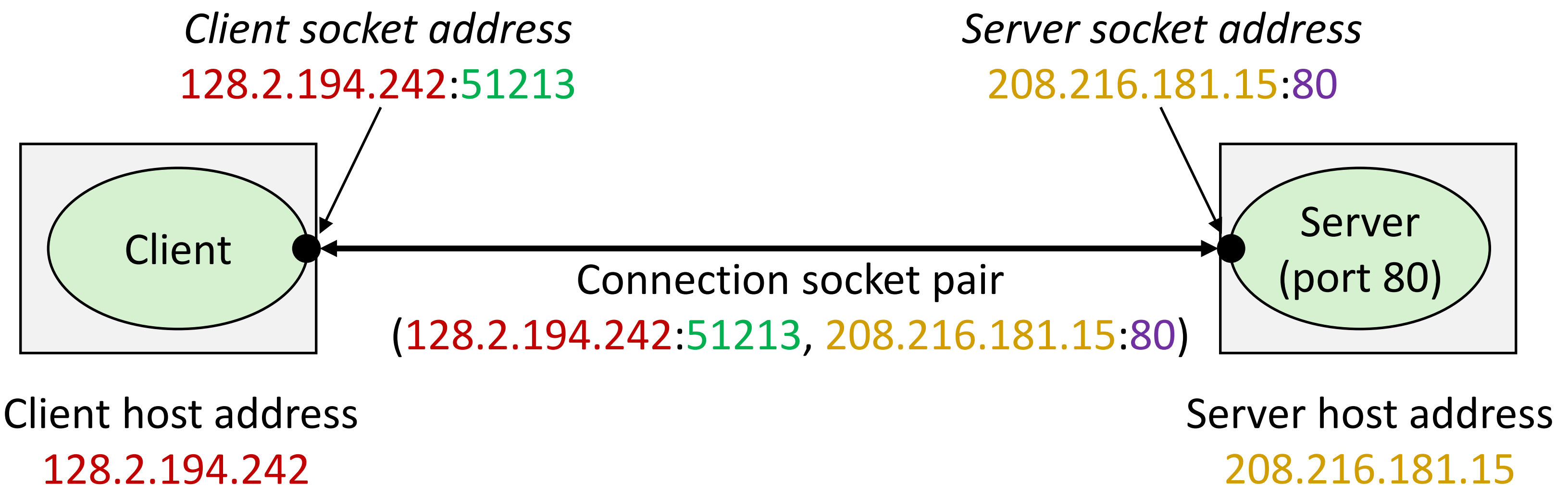
Well-known Service Names and Ports

- 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
  - Web servers: http 80

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

Anatomy of a Connection

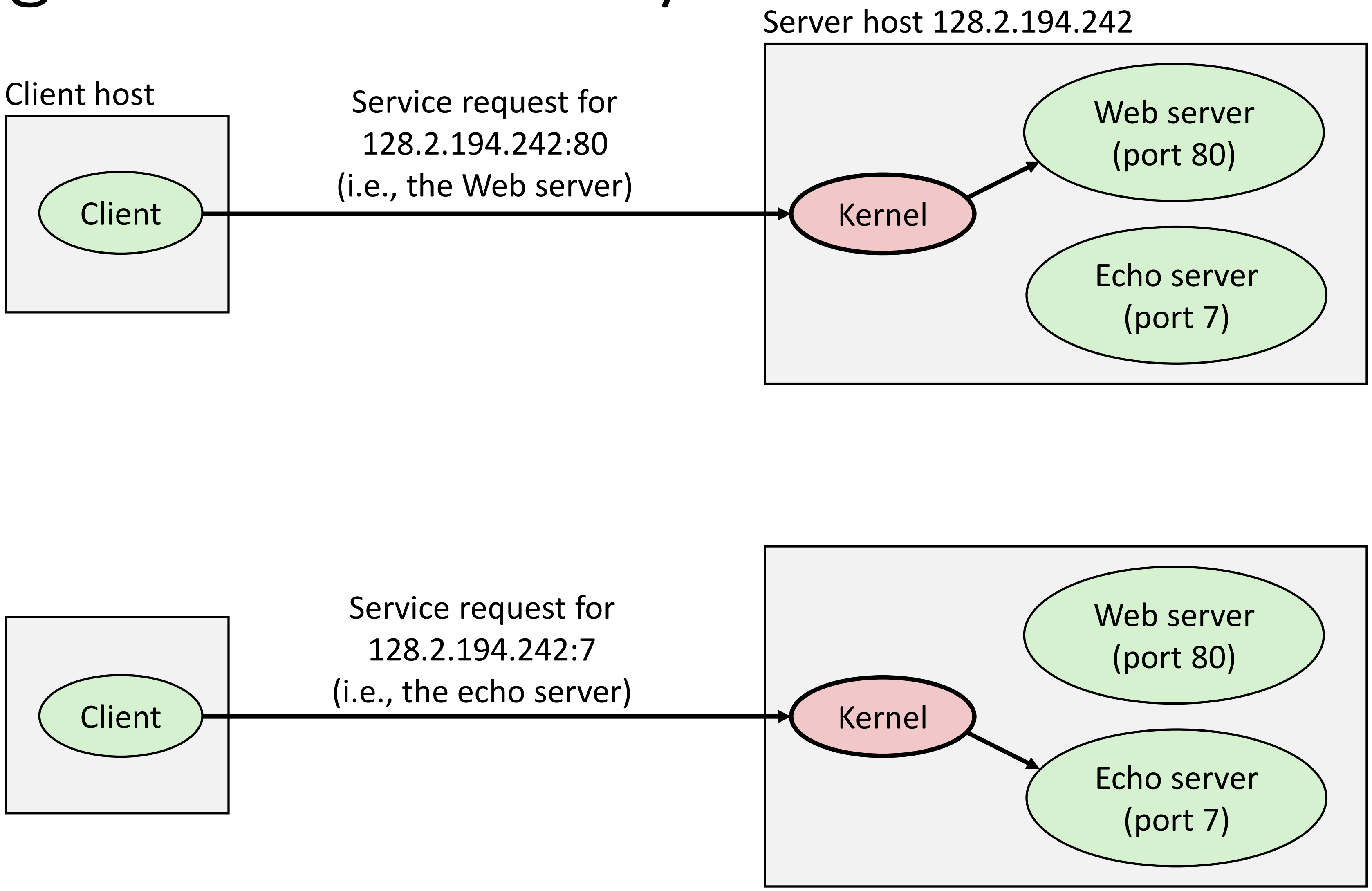
- A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)
  - `(cliaddr:cliport, servaddr:servport)`



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

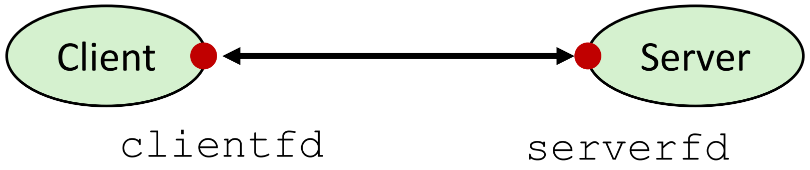
- Set of system-level functions used in conjunction with Unix I/O to build network applications.

- 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



- 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

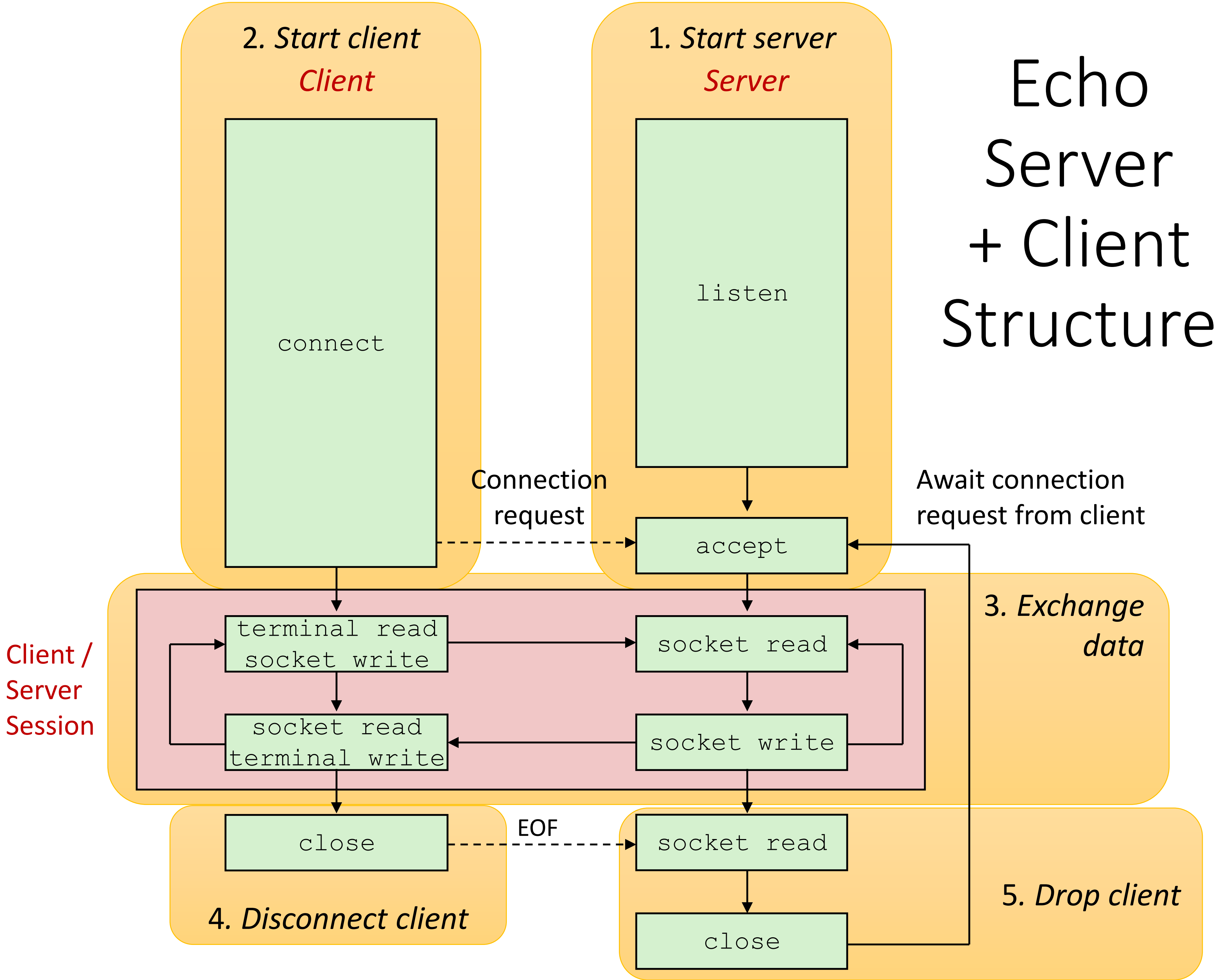
Client

```
thoth $ ./echoclient                                             (A)
This line is being echoed                                        (B)
This line is being echoed
This one is, too                                                 (C)
This one is, too
^D
```

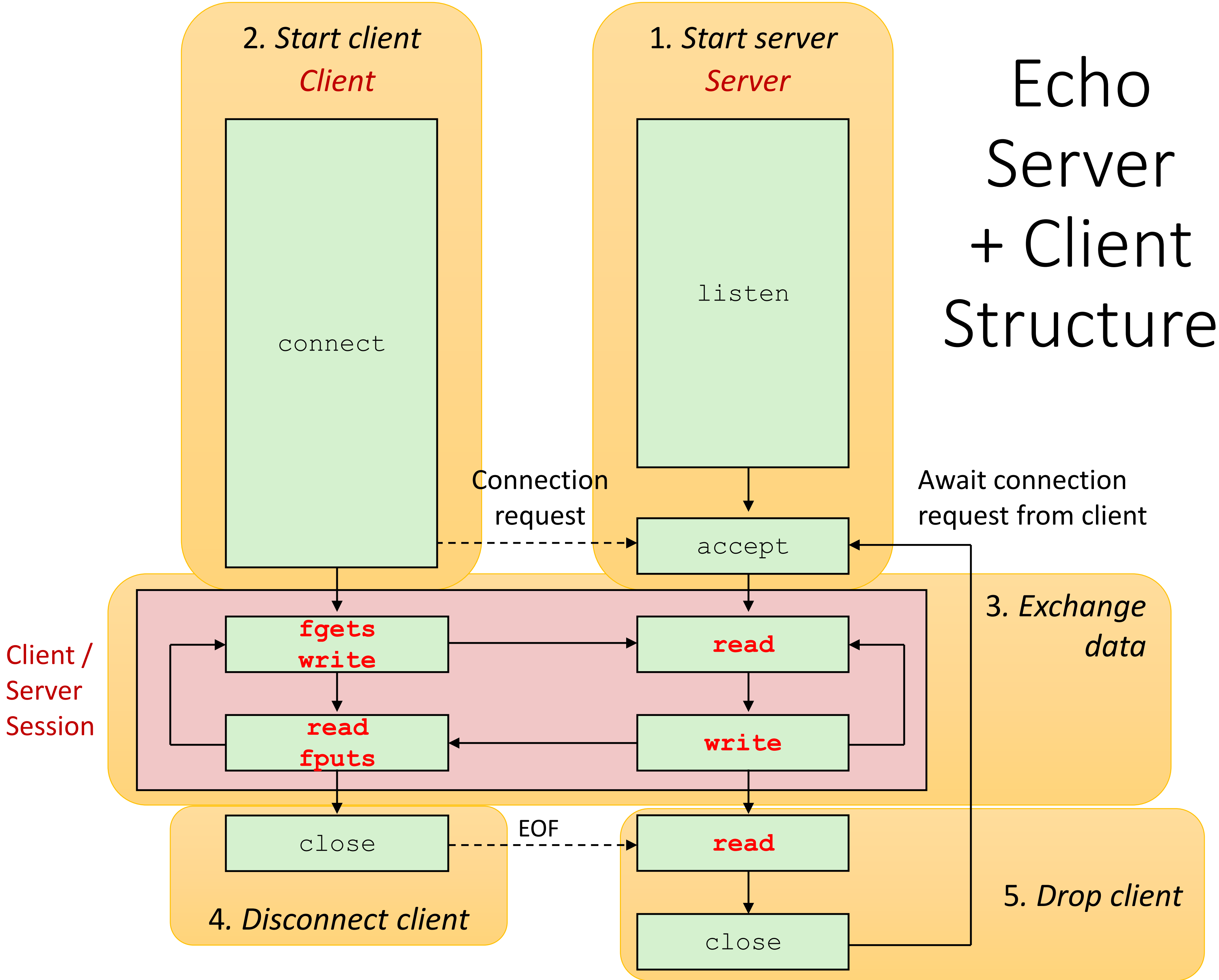
Server

```
thoth $ ./echoserver
Server connected to client.                                 (A)
server received 26 bytes                                    (B)
server received 17 bytes                                    (C)
```

Echo Server + Client Structure



Echo Server + Client Structure



Echo Server: Main Routine

C (gcc -o echoserver echoserver.c)

```c
#include <stdio.h>      // fgets, etc
#include <sys/socket.h> // socket API
#include <arpa/inet.h>  // inet functions, htons
#include <unistd.h>     // read/close system calls
#include <stdlib.h>     // exit
#include <string.h>     // strlen
#define PORT 9997

int main(void) {
  // Creating socket file descriptor (using internet protocol)
  int server_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (server_fd == 0) {
    perror("socket failed");
    exit(EXIT_FAILURE);
  }
```

We create a socket.

```c
  // We want to use the internet protocol
  struct sockaddr_in address;
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = INADDR_ANY;
  address.sin_port = htons(PORT);
```

We define what port and protocol we want

```c
  // Bind socket to the port (so it listens to that port)
  int result = bind(server_fd, (struct sockaddr *)&address, sizeof(address));
  if (result < 0) {
    perror("bind failed");
    exit(EXIT_FAILURE);
  }
```

We bind ourselves to that port.

```c
  // Listen for connections
  result = listen(server_fd, 3);
  if (result < 0) {
    perror("listen");
    exit(EXIT_FAILURE);
  }
```

We wait until somebody requests a connection.

```c
  // Listen will return when a connection is requested...
  // Accept that connection
  int addrlen = sizeof(address);
  int new_socket = accept(server_fd, (struct sockaddr *)&address,
                                     (socklen_t*)&addrlen);
  if (new_socket < 0) {
    perror("accept");
    exit(EXIT_FAILURE);
  }
```

We accept that connection.

```c
  printf("Server connected to client.\n");

  char buffer[1024] = {0};
  int count = 0;
  do {
    // Read data (it waits until data is available)
    count = read(new_socket, buffer, 1024);
    printf("Server received %d bytes.\n", count);
    buffer[count] = '\0';
    write(new_socket, buffer, strlen(buffer));
  } while(count);
```

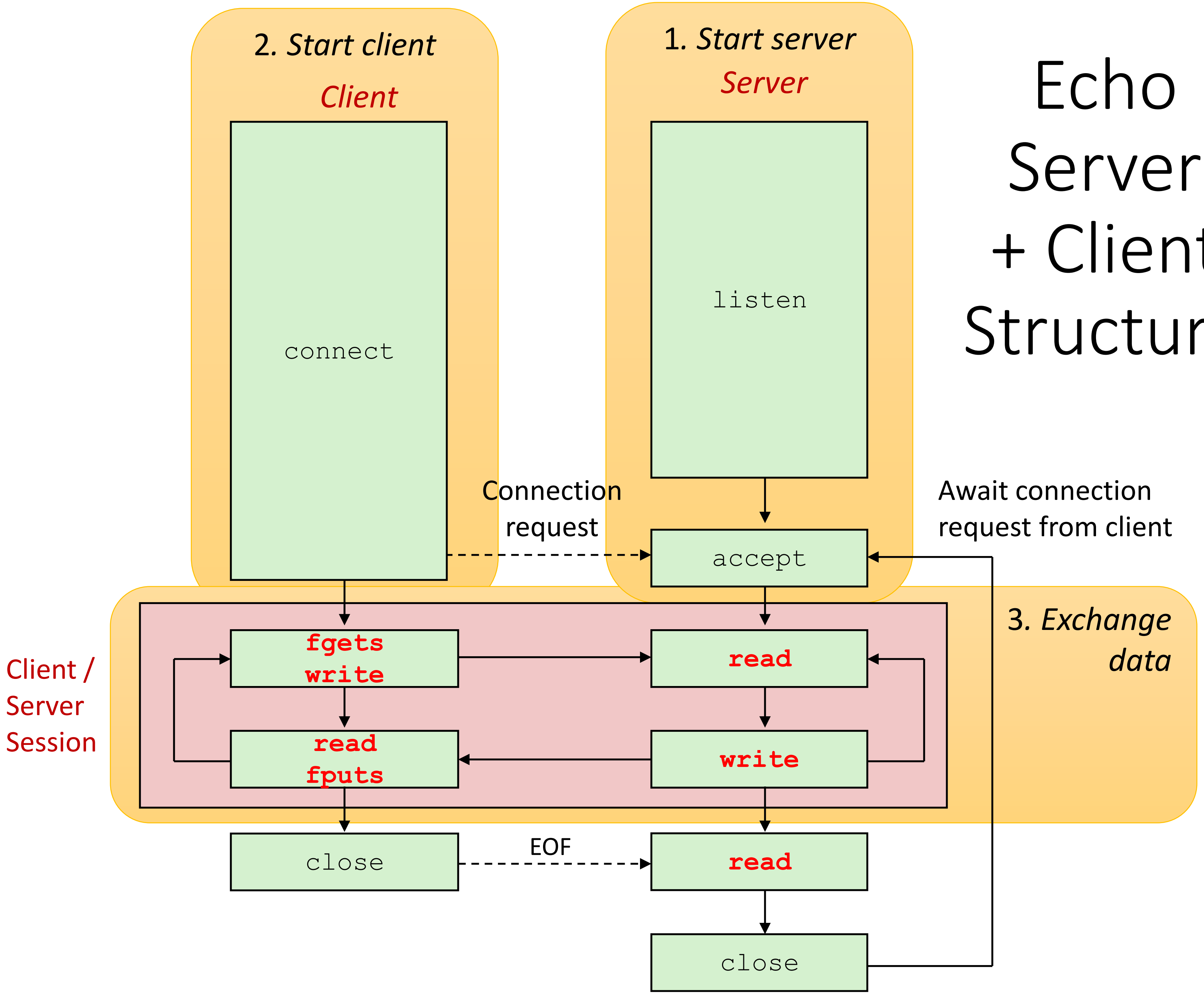
We wait until data arrives and read it.

We write it back out. Stopping our loop when nothing was read.

```c
  close(new_socket);
  close(server_fd);
  return 0;
}
```

We close all of our connections.

Echo Server + Client Structure



Echo Client: Main Routine

C (gcc -o echoclient echoclient.c)

```c
#include <stdio.h>      // fgets, etc
#include <sys/socket.h> // socket API
#include <arpa/inet.h>  // inet functions, htons
#include <unistd.h>     // read/close system calls
#include <string.h>     // strlen
#define PORT 9997

int main(void) {
  // Creating socket file descriptor (using internet protocol)
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0) {
    printf("\n Socket creation error \n");
    return -1;
  }
```

We create a socket.

Using the Internet protocol.

```c
  struct sockaddr_in serv_addr;
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(PORT);
```

Connecting to localhost

```c
  // Convert IPv4 and IPv6 addresses from text to binary form
  if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0) {
    printf("\nInvalid address / Address not supported \n");
    return -1;
  }
```

Actually request a connection.

```c
  int result = connect(sock,
                       (struct sockaddr *)&serv_addr,
                       sizeof(serv_addr));

  if (result < 0) {
    printf("\nConnection Failed \n");
    return -1;
  }
```

If we got here, the server accepted our connection!

```c
  char buffer[1024] = {0};
  int count = 0;
  do {
    if (fgets(buffer, 1024, stdin) == NULL) {
      break; // Exit when line is empty (CTRL+D is pressed)
    }
    write(sock, buffer, strlen(buffer));
    count = read(sock, buffer, 1023);
    buffer[count] = '\0';
    fputs(buffer, stdout);
  } while(count);

  close(sock);
  return 0;
}
```

This loop reads from stdin (user input)

We write everything to the server!

And print out everything the server sends us.

We clean up when the loop ends (when no user input via CTRL+D)

Read and write system calls

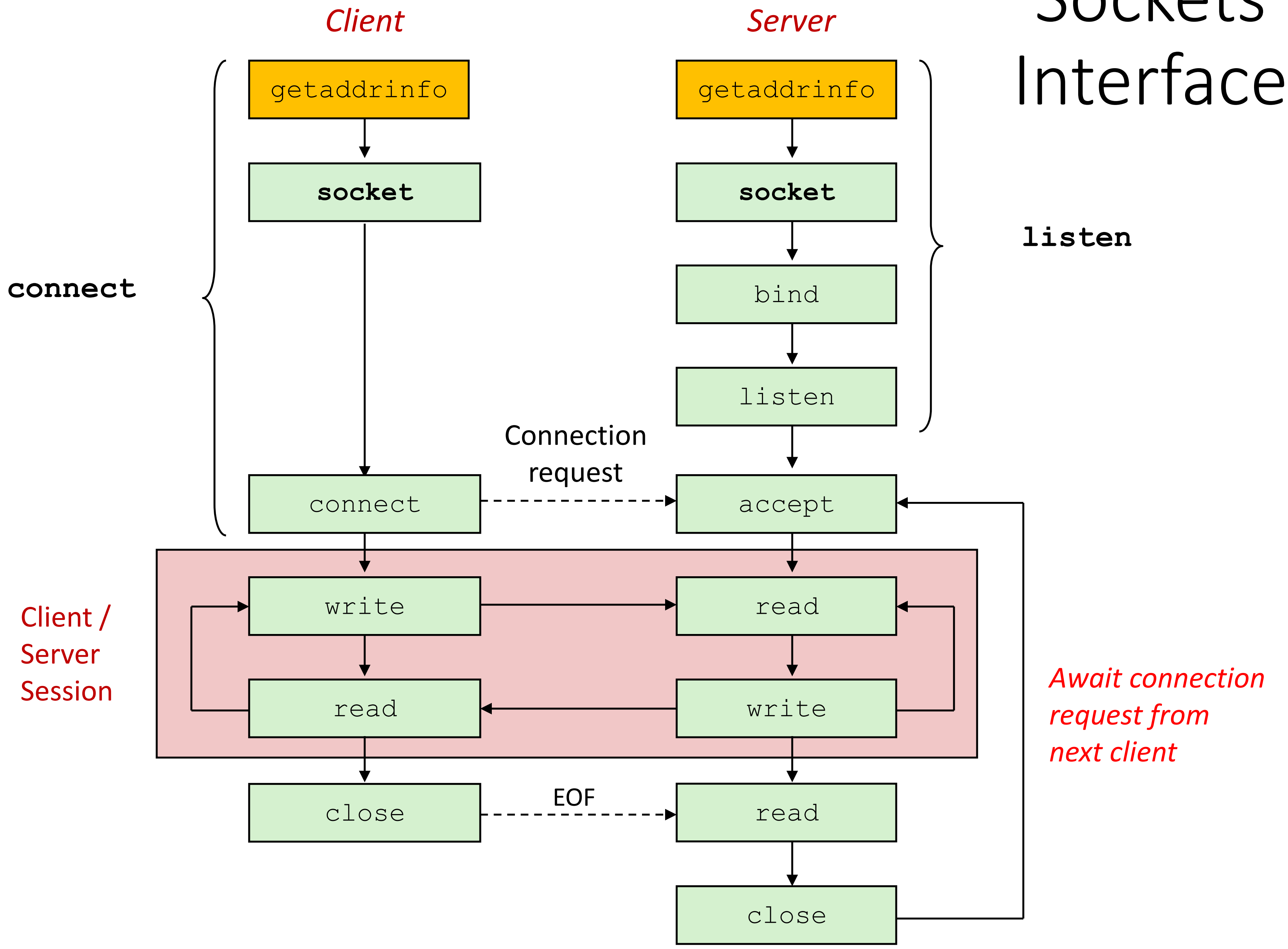
- Same interface used to read/write files.
  - Because sockets are also files! Neat.

```
#include <unistd.h>

ssize_t read(int fd, void *usrbuf, size_t n);
ssize_t write(int fd, void *usrbuf, size_t n);
```

Return: number of bytes transferred if OK, 0 on EOF (`read` only), -1 on error

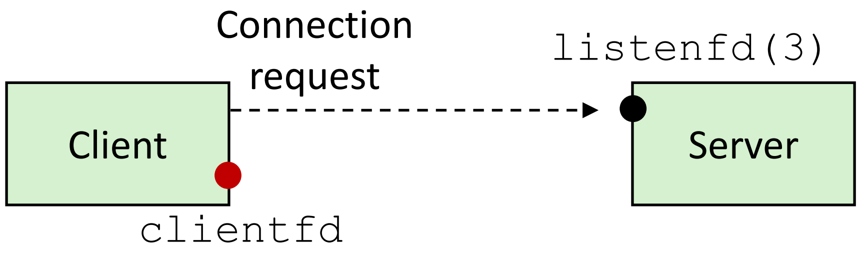
- **`read`** returns a count of 0 only if it encounters EOF
  - So, it is useful to notice if the other machine disconnected.
- Calls to **`read`** and **`write`** can be interleaved arbitrarily on the same file descriptor (socket, file on disk, etc)

Sockets Interface



`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

Testing Servers Using `telnet`

- The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections
  - Our simple echo server
  - Web servers
  - Mail servers

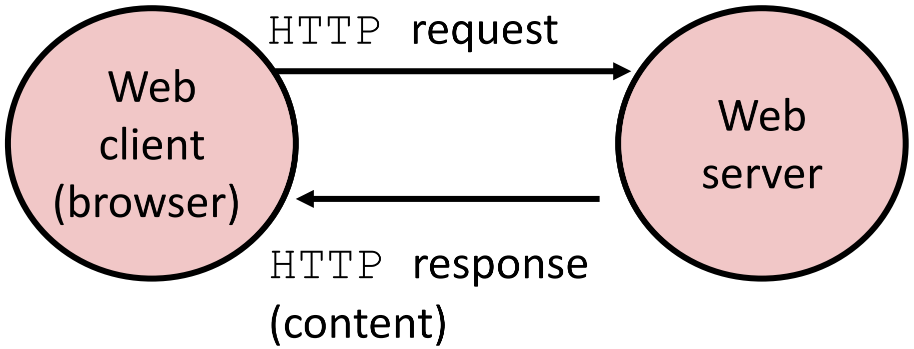
- Usage:
  - **`linux>`** ***`telnet <host> <portnumber>`***
  - Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

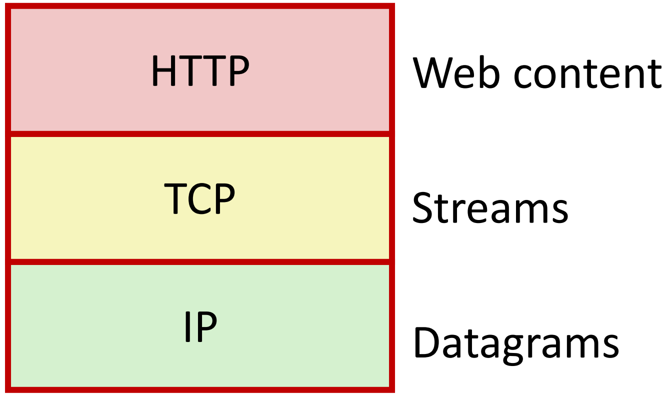
Testing the Echo Server With `telnet`

```
wilkiepedia.org $ ./echoserver
Server connected to client.
Server received 11 bytes
Server received 8 bytes




thoth $ telnet wilkiepedia.org 9997
Trying 128.2.210.175...
Connected to wilkiepedia.org (128.2.210.175).
Escape character is '^]'.
Hi there!
Hi there!
Howdy!
Howdy!
^]
telnet> quit
Connection closed.
thoth $
```

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/1.1
  - 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`	*Binar 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

- Bottom line: *Web content is 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.google.edu:80/index.html`
- Clients use *prefix* (`http://www.google.edu:80`) to infer:
  - What kind (protocol) of server to contact (**HTTP**)
  - Where the server is (**`www.google.com`**)
  - 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 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: empty 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

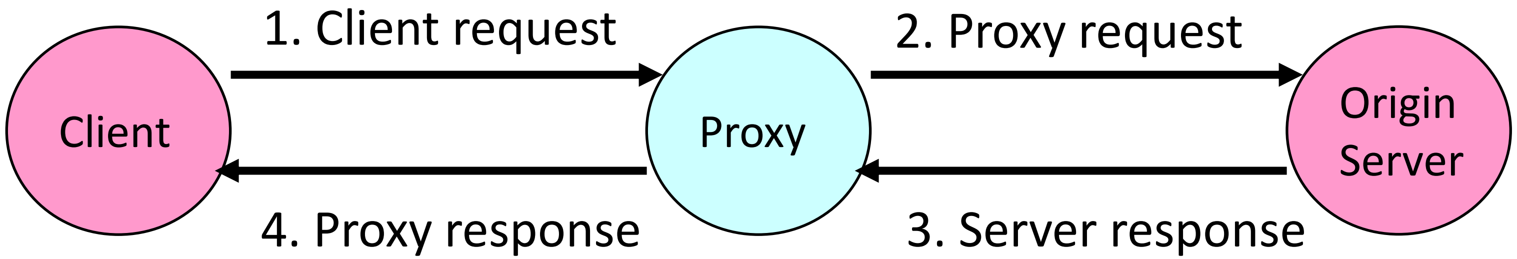
```
whaleshark> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.42.52...                       Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1                   Client: request line
Host: www.cmu.edu                           Client: required HTTP/1.1 header
                                            Client: empty 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
```

EXTRA SLIDES

(Useful material for Proxy Lab)
WHICH WE ARE NOT DOING
DO NOT WORRY

Proxies

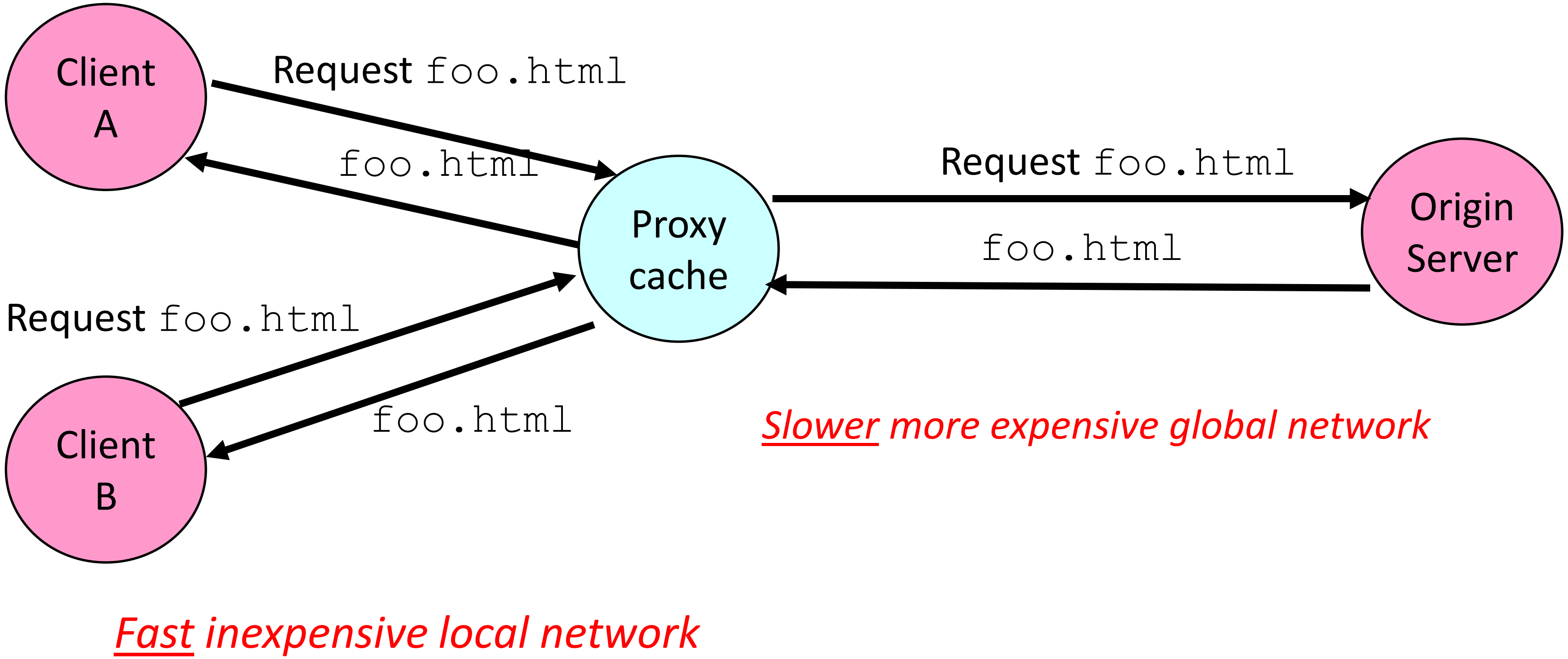
- A *proxy* is an intermediary between a client and an *origin server*
  - To the client, the proxy acts like a server
  - To the server, the proxy acts like a client



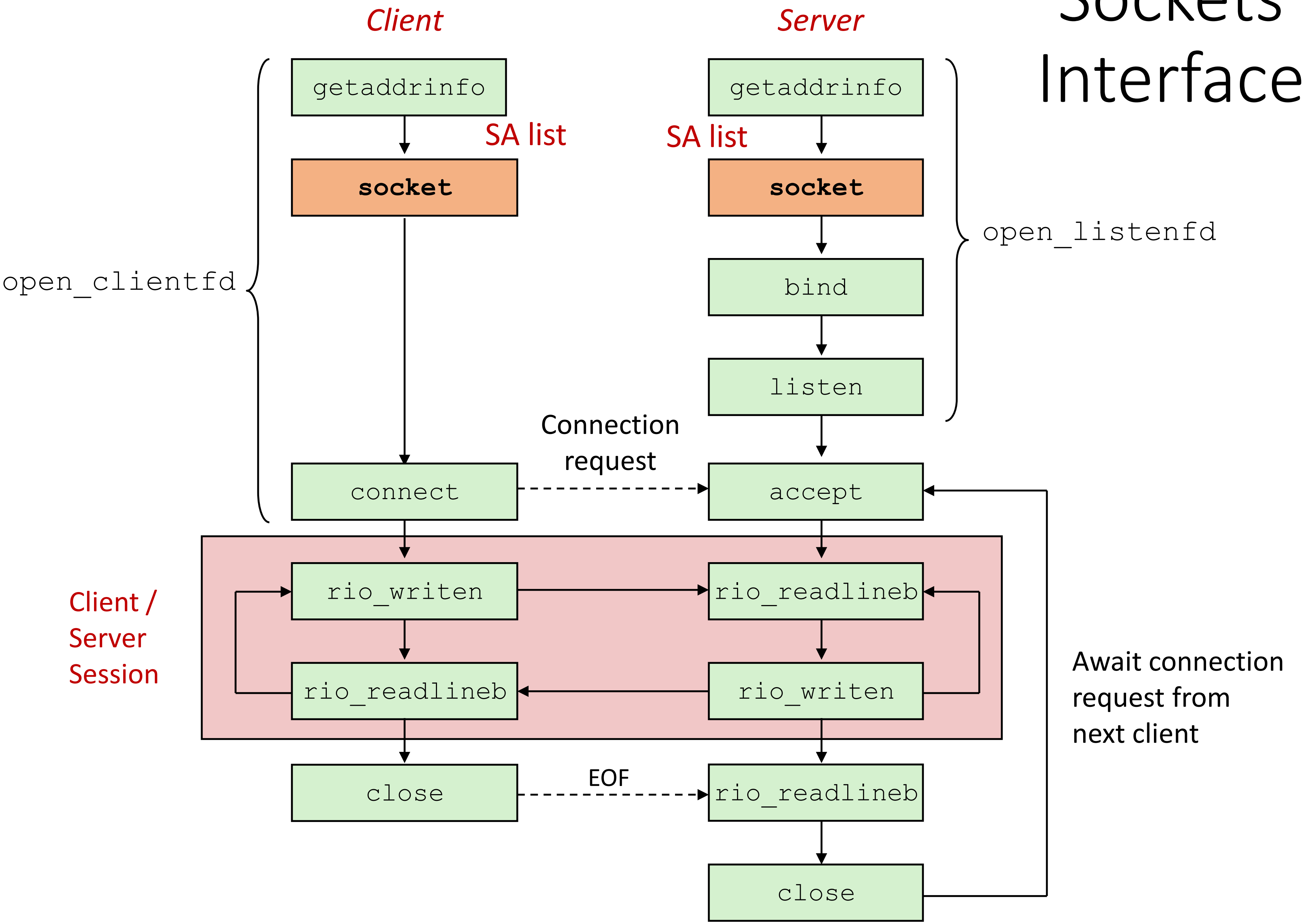
- **This is what you will be implementing in <u>Proxy Lab</u>**

Why Proxies?

- Can perform useful functions as requests and responses pass by
  - Examples: Caching, logging, anonymization, filtering, transcoding



Sockets Interface



Sockets Interface: `socket`

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

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

- Example:

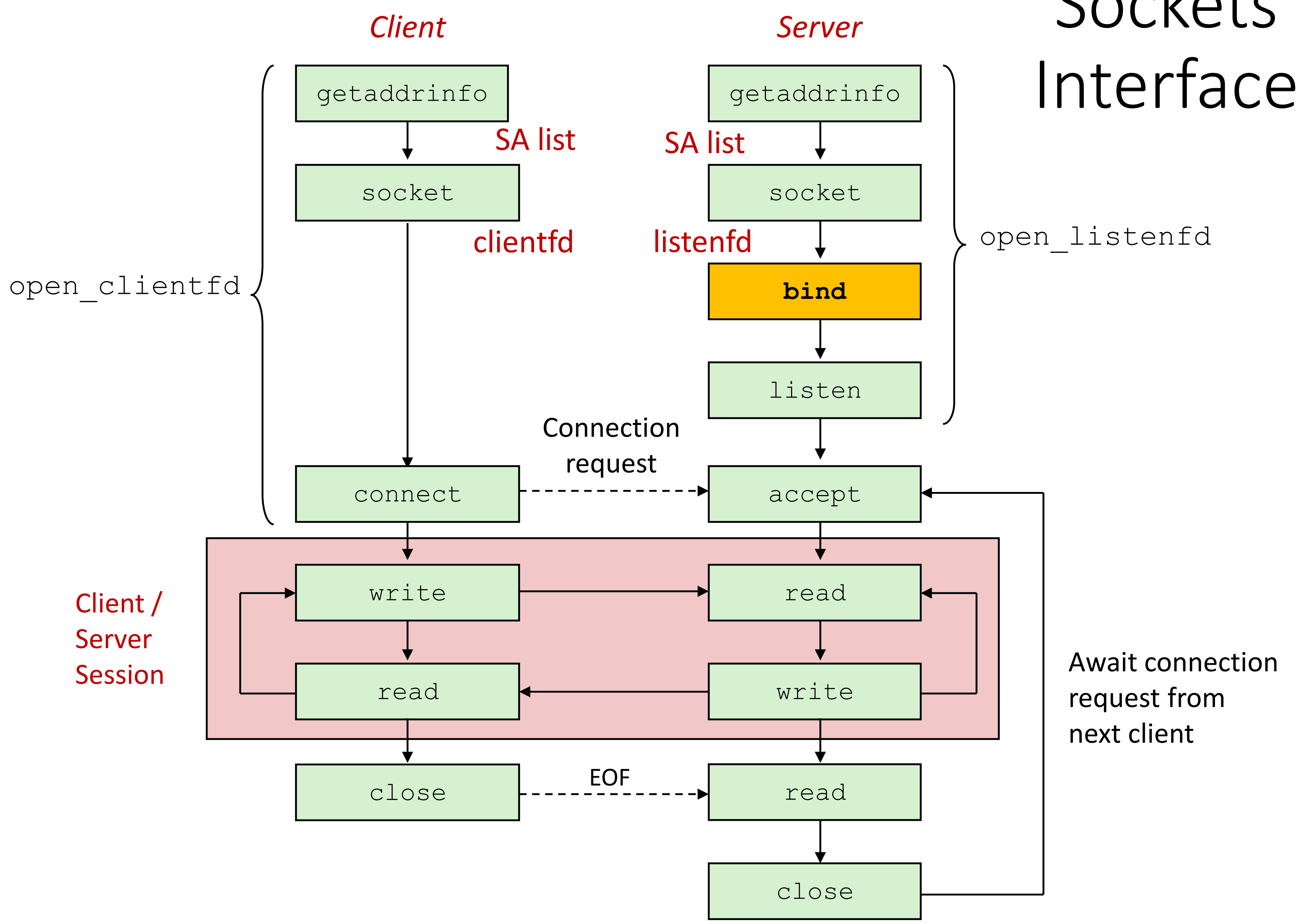
```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

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

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

Sockets Interface



Sockets Interface: `bind`

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

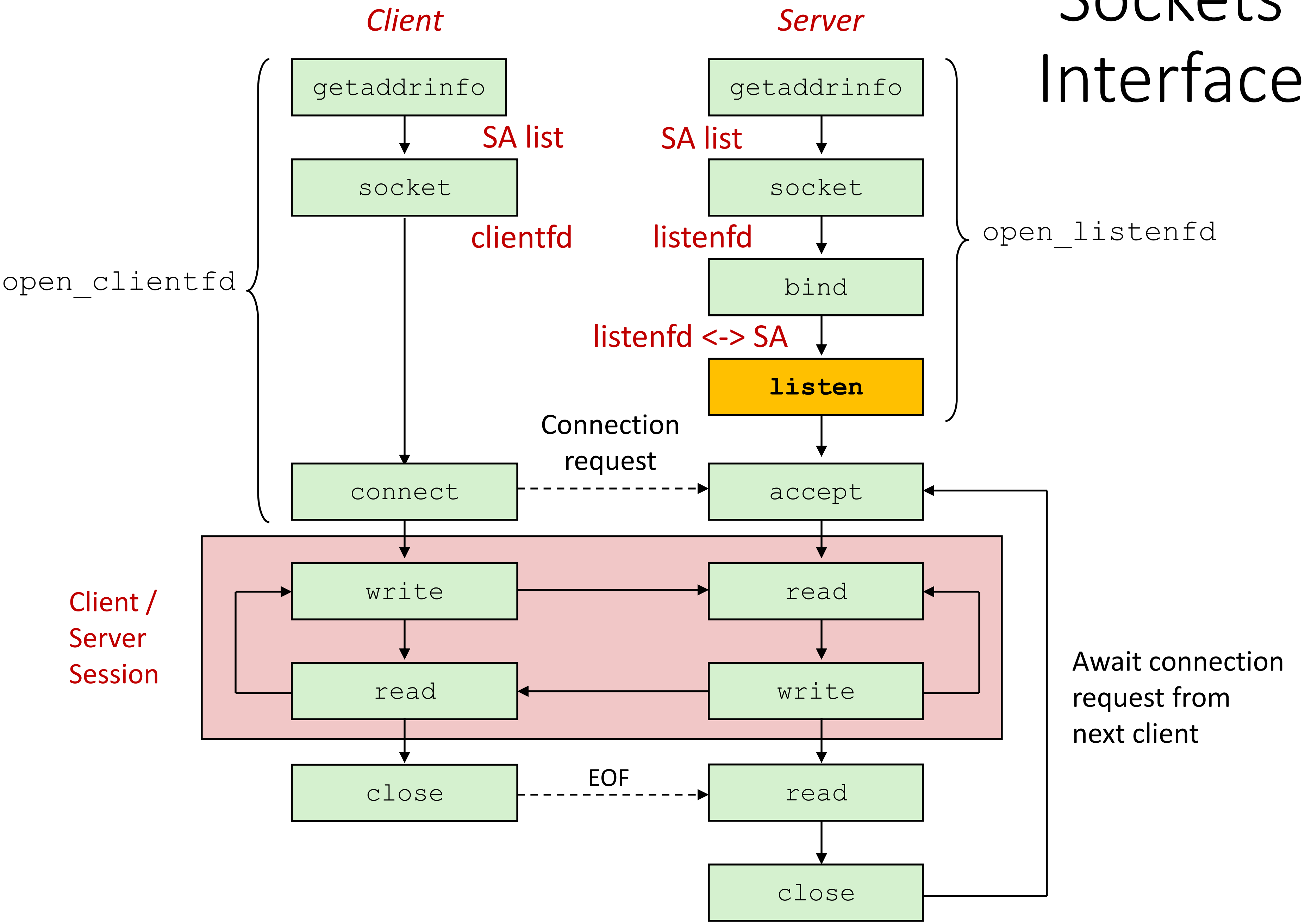
```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

Recall: `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`

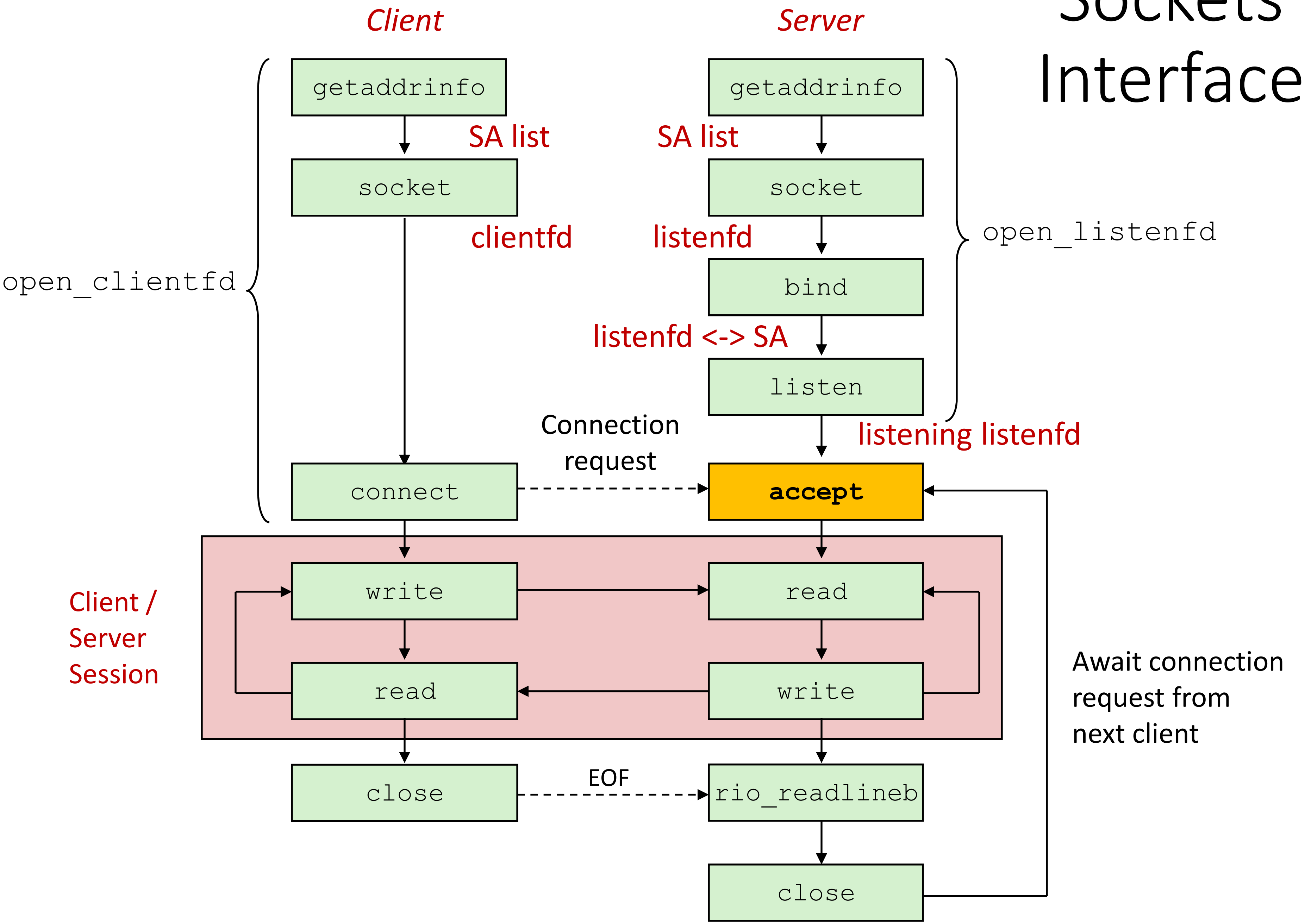
- By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection.
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

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

- 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.

Sockets Interface



Sockets Interface: `accept`

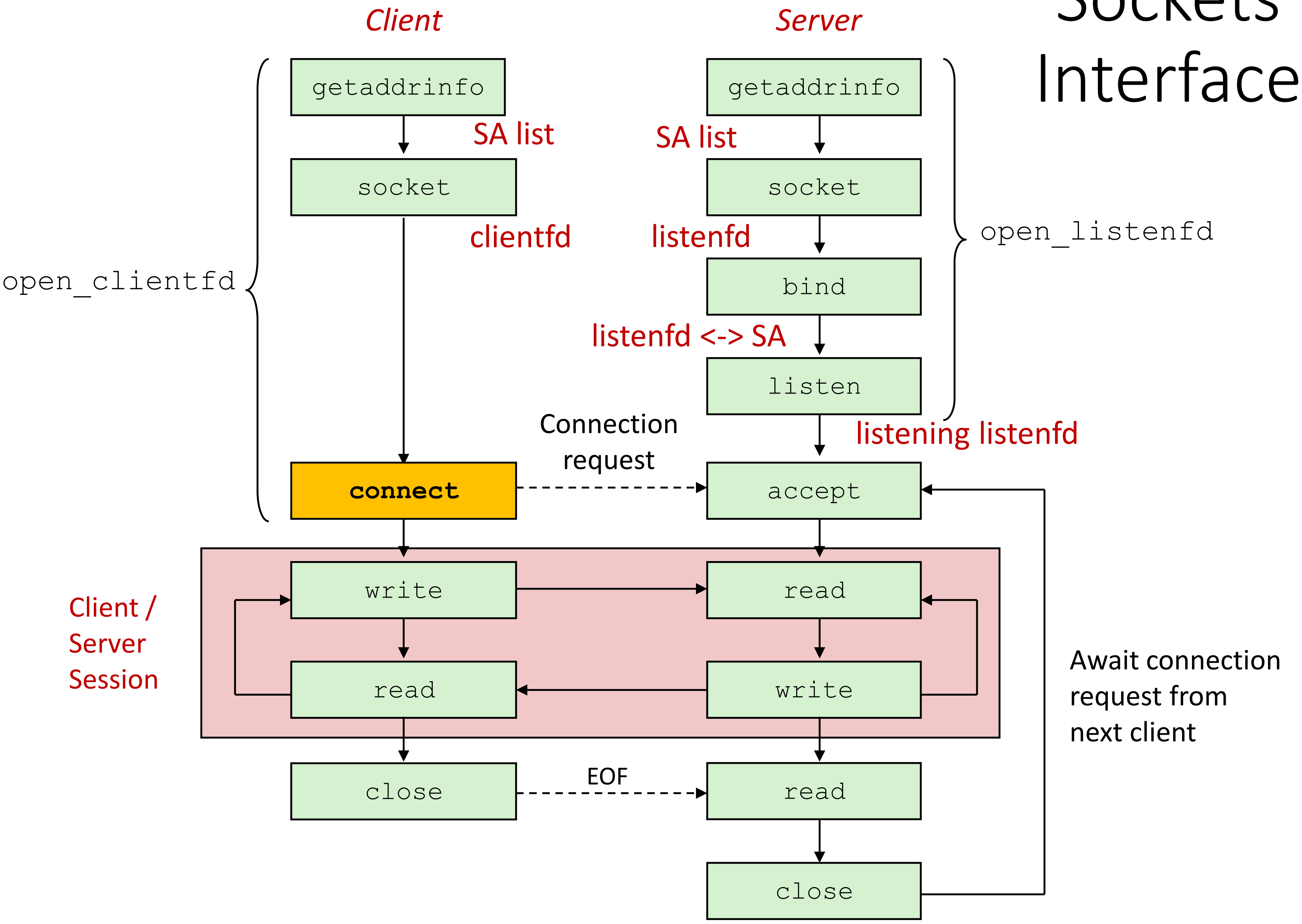
- Servers wait for connection requests from clients by calling `accept`:

```
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* 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);
```

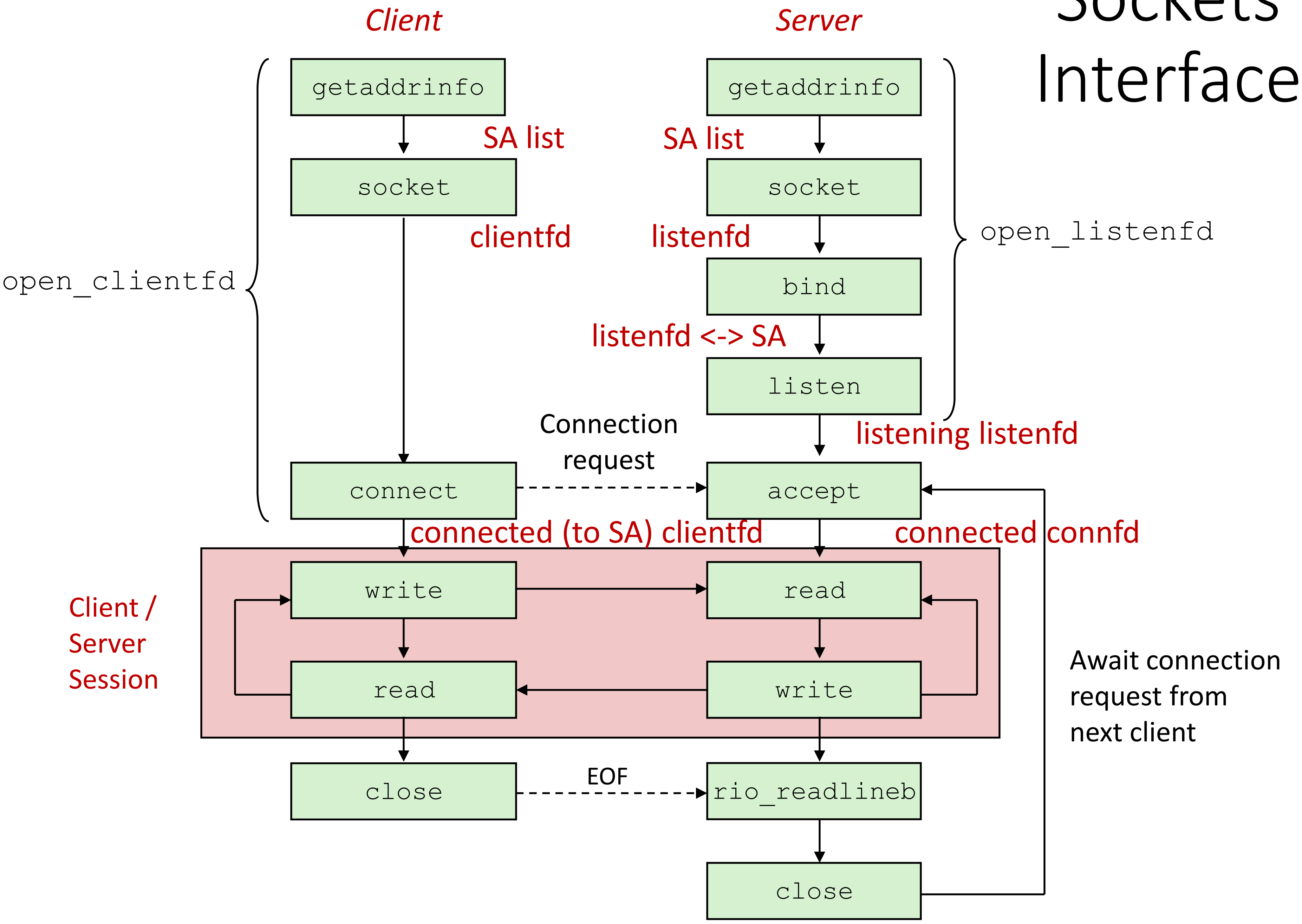
- Attempts to establish a connection with server at socket address `addr`
  - If successful, then **`clientfd`** is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

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

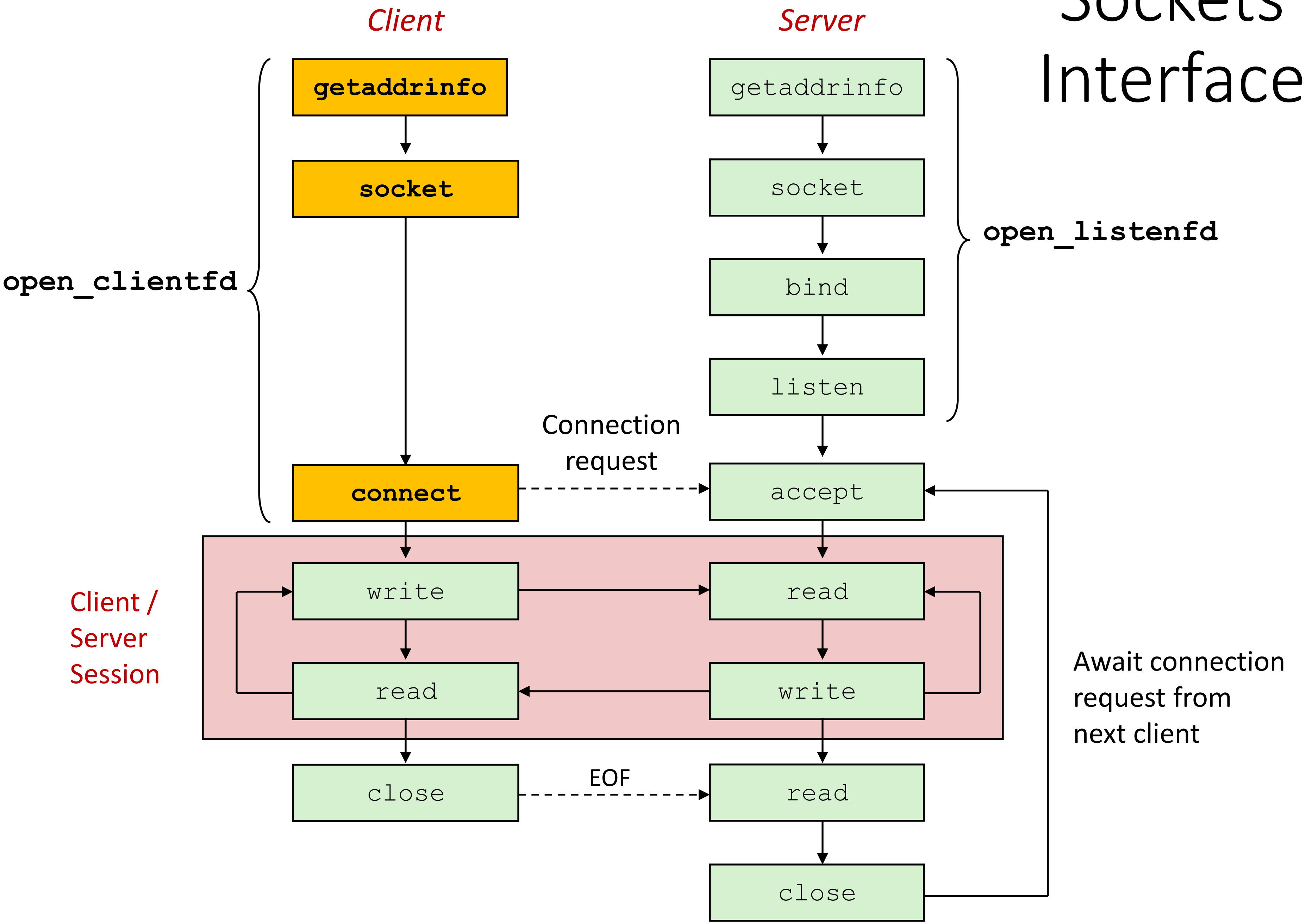
    - **`x`** is client address
    - **`y`** is ephemeral port that uniquely identifies client process on client host

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

Sockets Interface



Sockets Interface



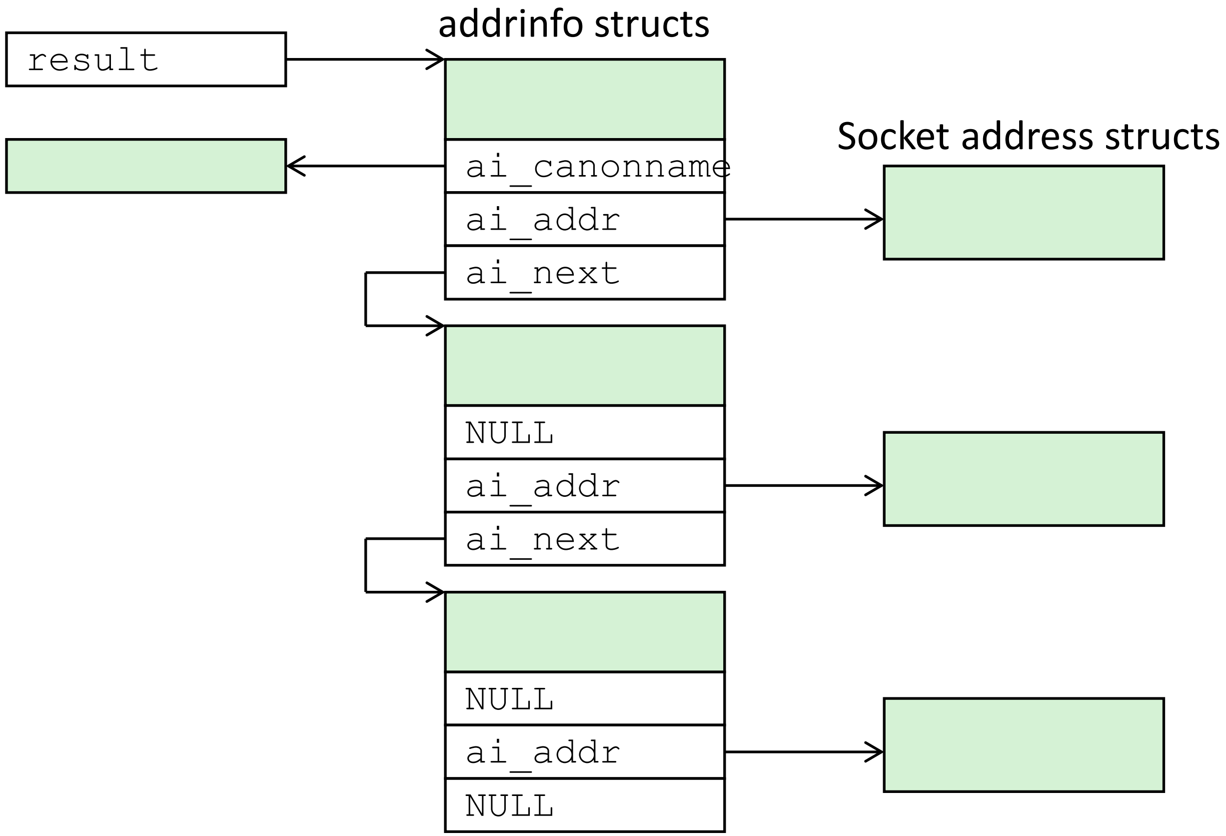
Sockets Helper: `open_clientfd`

- Establish a connection with a server

```
int open_clientfd(char *hostname, char *port) {
  int clientfd;
  struct addrinfo hints, *listp, *p;

  /* Get a list of potential server addresses */
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
  hints.ai_flags = AI_NUMERICSERV;  /* …using numeric port arg. */
  hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
  getaddrinfo(hostname, port, &hints, &listp);
```

csapp.c

`getaddrinfo` Linked List



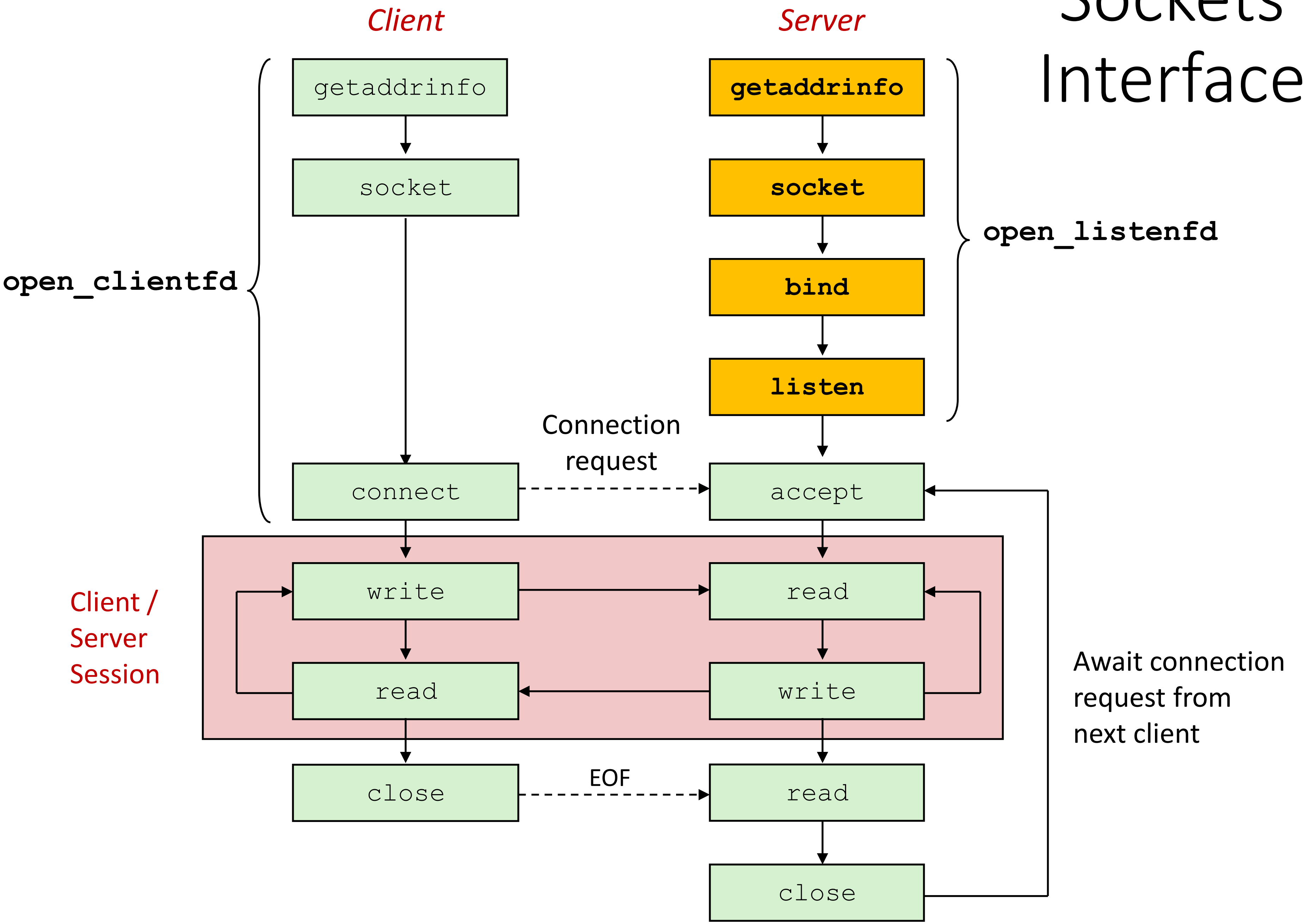
- Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- Servers: walk the list until calls to `socket` and `bind` succeed.

Sockets Helper: `open_clientfd` (cont)

```
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */

        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; /* Success */
        close(clientfd); /* Connect failed, try another */
    }

    /* Clean up */
    freeaddrinfo(listp);
    if (!p) /* All connects failed */
        return -1;
    else    /* The last connect succeeded */
        return clientfd;
}
```

csapp.c

Sockets Interface



Sockets **Helper:** `open_listenfd`

- Create a listening descriptor that can be used to accept connection requests from clients.

```
 int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;             /* Accept connect. */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* …on any IP addr */
    hints.ai_flags |= AI_NUMERICSERV;            /* …using port no. */
    getaddrinfo(NULL, port, &hints, &listp);
```

csapp.c

Sockets Helper: `open_listenfd` (cont)

```
    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue;  /* Socket failed, try the next */

        /* Eliminates "Address already in use" error from bind */
        setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));

        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        close(listenfd); /* Bind failed, try the next */
    }
```

csapp.c

Sockets Helper: `open_listenfd` (cont)

```
    /* Clean up */
    freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;

    /* Make it a listening socket ready to accept conn. requests */
    if (listen(listenfd, LISTENQ) < 0) {
        close(listenfd);
        return -1;
    }
    return listenfd;
}
```

csapp.c

- **Key point: `open_clientfd` and `open_listenfd` are both independent of any particular version of IP.**

Case Study

Tiny Web Server

Tiny Web Server

- Tiny Web server described in textbook (CS:APP)
  - Tiny is a sequential Web server
  - Serves static and dynamic content to real browsers
    - text files, HTML files, GIF, PNG, and JPEG images
  - 239 lines of commented C code
  - Not as complete or robust as a real Web server
    - You can break it with poorly-formed HTTP requests (e.g., terminate lines with "**`\n`**" instead of "**`\r\n`**")

Tiny Operation

- Accept connection from client
- Read request from client (via connected socket)
- Split into <method> <uri> <version>
  - If method not GET, then return error
- If URI contains "`cgi-bin`" then serve dynamic content
  - (Would do wrong thing if had file "**`abcgi-bingo.html`**")
  - Fork process to execute program
- Otherwise serve static content
  - Copy file to output

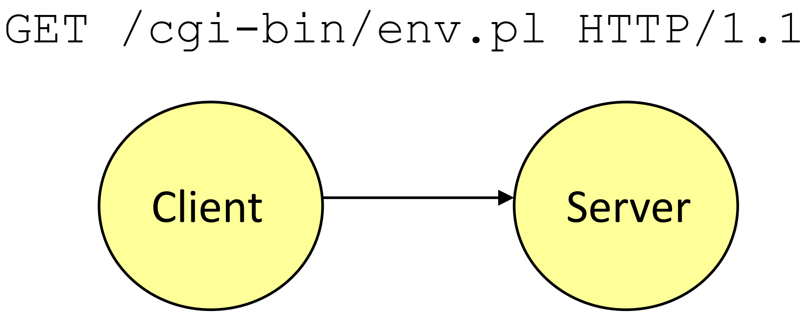
Tiny Serving Static Content

```
void serve_static(int fd, char *filename, int filesize)
{
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    /* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    write(fd, buf, strlen(buf));

    /* Send response body to client */
    srcfd = open(filename, O_RDONLY, 0);
    srcp = mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    close(srcfd);
    write(fd, srcp, filesize);
    munmap(srcp, filesize);
}
```

tiny.c

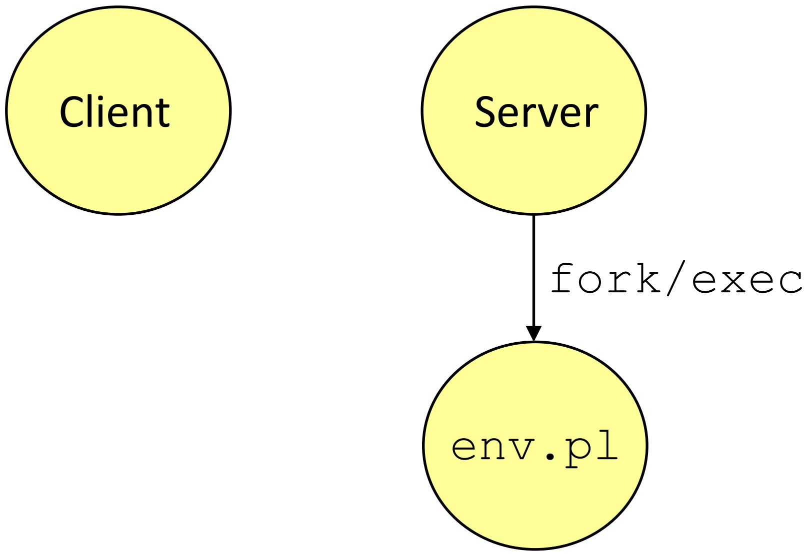
Serving Dynamic Content

- Client sends request to server
- If request URI contains the string “`/cgi-bin`”, the Tiny server assumes that the request is for dynamic content



Serving Dynamic Content (cont)

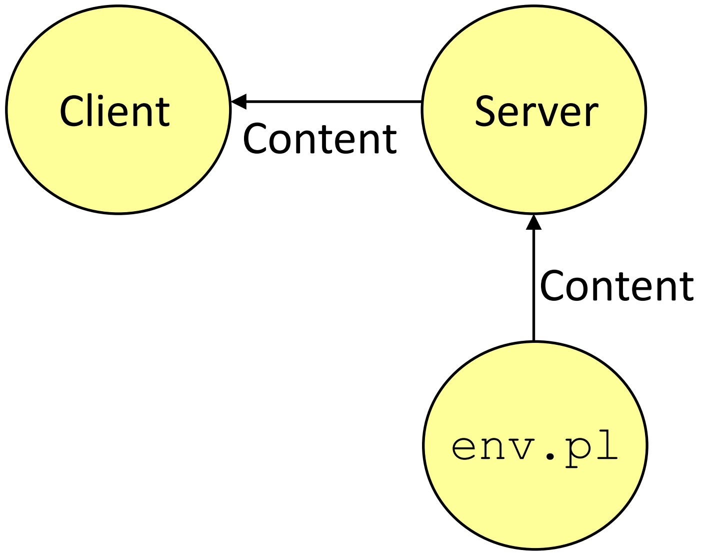
- The server creates a child process and runs the program identified by the URI in that process



Serving Dynamic Content (cont)

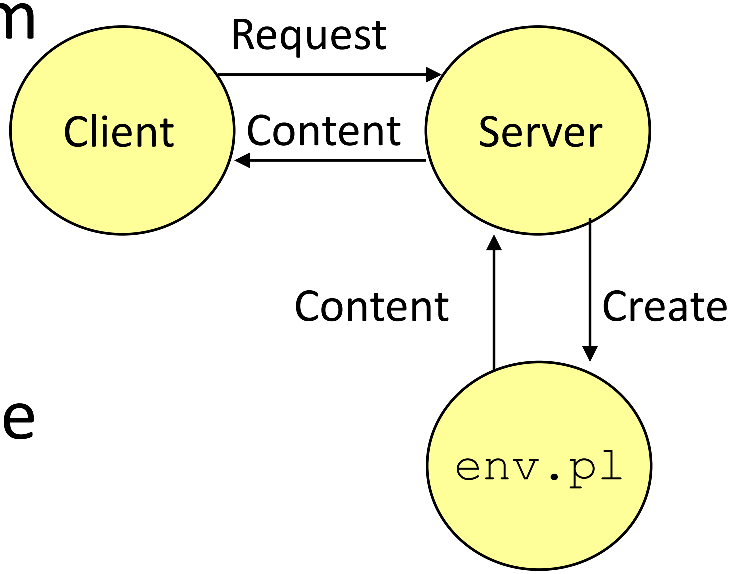
- The child runs and generates the dynamic content

- The server captures the content of the child and forwards it without modification to the client



Issues in Serving Dynamic Content

- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the Common Gateway Interface (CGI) specification.



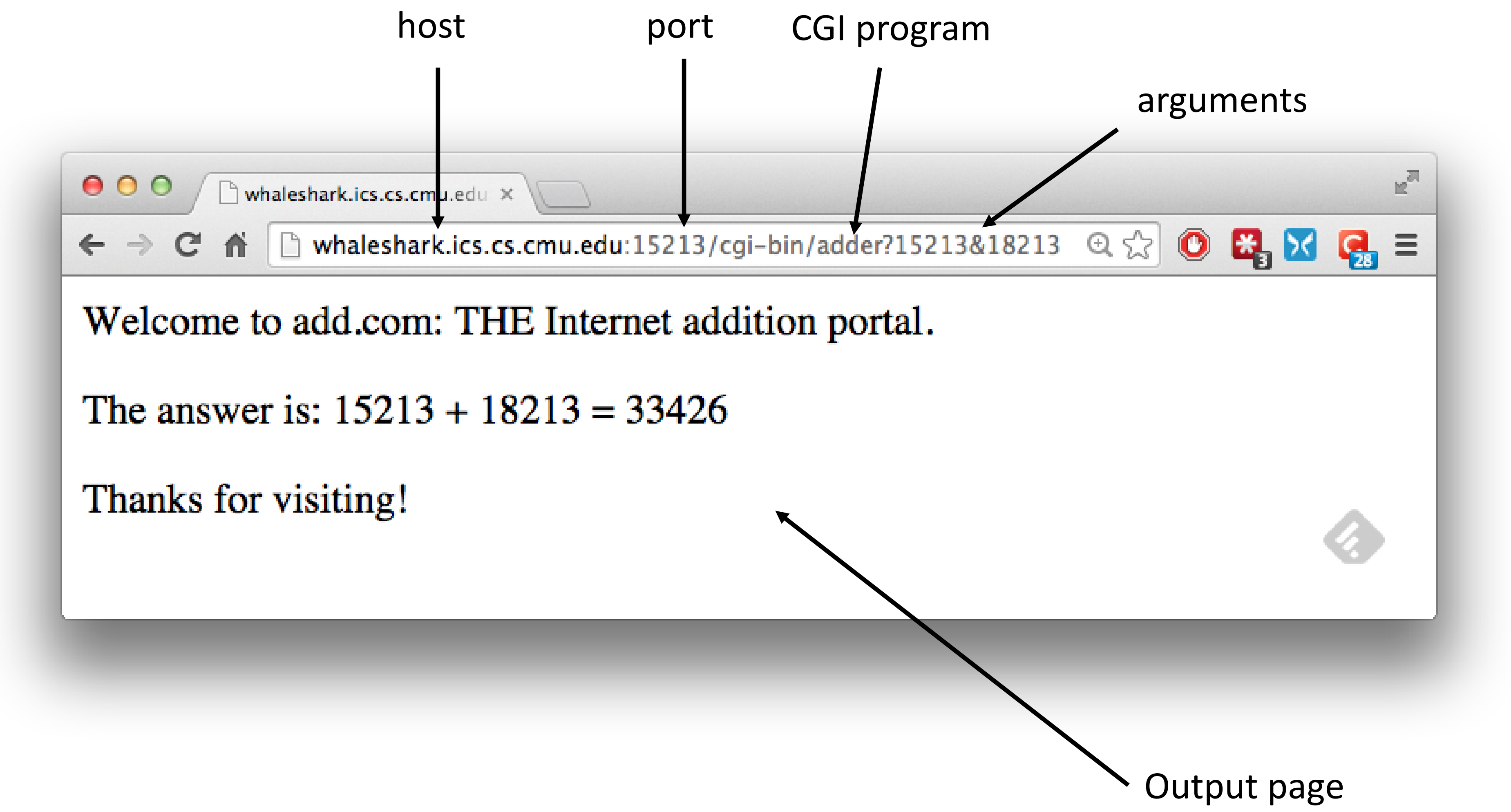
CGI

- Because the children are written according to the CGI spec, they are often called *CGI programs.*

- However, CGI really defines a simple standard for transferring information between the client (browser), the server, and the child process.

- CGI is the original standard for generating dynamic content. Has been largely replaced by other, faster techniques:
  - E.g., fastCGI, Apache modules, Java servlets, Rails controllers
  - Avoid having to create process on the fly (expensive and slow).

The add.com Experience



Serving Dynamic Content With GET

- <u>Question:</u> How does the client pass arguments to the server?
- <u>Answer:</u> The arguments are appended to the URI

- Can be encoded directly in a URL typed to a browser or a URL in an HTML link
  - **`http://add.com/cgi-bin/adder?15213&18213`**
  - **`adder`** is the CGI program on the server that will do the addition.
  - argument list starts with "**`?`**"
  - arguments separated by "**`&`**"
  - spaces represented by "**`+`**" `or` "**`%20`**"

Serving Dynamic Content With GET

- URL suffix:
  - **`cgi-bin/adder?15213&18213`**

- Result displayed on browser:

```
Welcome to add.com: THE Internet addition portal.

The answer is: 15213 + 18213 = 33426

Thanks for visiting!
```

Serving Dynamic Content With GET

- Question: How does the server pass these arguments to the child?
- Answer: In environment variable QUERY_STRING
  - A single string containing everything after the “**?**”
  - For add: **QUERY_STRING** = “**15213&18213**”

```
/* Extract the two arguments */
if ((buf = getenv("QUERY_STRING")) != NULL) {
    p = strchr(buf, '&');
*p = '\0';
    strcpy(arg1, buf);
    strcpy(arg2, p+1);
    n1 = atoi(arg1);
    n2 = atoi(arg2);
}
```

adder.c

Serving Dynamic Content with GET

- <u>Question:</u> How does the server capture the content produced by the child?
- <u>Answer:</u> The child generates its output on `stdout`. Server uses `dup2` to redirect `stdout` to its connected socket.

```
void serve_dynamic(int fd, char *filename, char *cgiargs)
{
    char buf[MAXLINE], *emptylist[] = { NULL };

    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    if (Fork() == 0) { /* Child */
        /* Real server would set all CGI vars here */
        setenv("QUERY_STRING", cgiargs, 1);
        Dup2(fd, STDOUT_FILENO);          /* Redirect stdout to client */
        Execve(filename, emptylist, environ); /* Run CGI program */
    }
    Wait(NULL); /* Parent waits for and reaps child */
}
```

tiny.c

Serving Dynamic Content with GET

- **Notice that only the CGI child process knows the content type and length, so it must generate those headers.**

```
    /* Make the response body */
    sprintf(content, "Welcome to add.com: ");
    sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
    sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
            content, n1, n2, n1 + n2);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* Generate the HTTP response */
    printf("Content-length: %d\r\n", (int)strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
    fflush(stdout);

    exit(0);
```

adder.c

Serving Dynamic Content With GET

```
bash:makoshark> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
GET /cgi-bin/adder?15213&18213 HTTP/1.0

HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
Content-length: 117
Content-type: text/html

Welcome to add.com: THE Internet addition portal.
<p>The answer is: 15213 + 18213 = 33426
<p>Thanks for visiting!
Connection closed by foreign host.
bash:makoshark>
```

HTTP request sent by client

HTTP response generated by the server

HTTP response generated by the CGI program

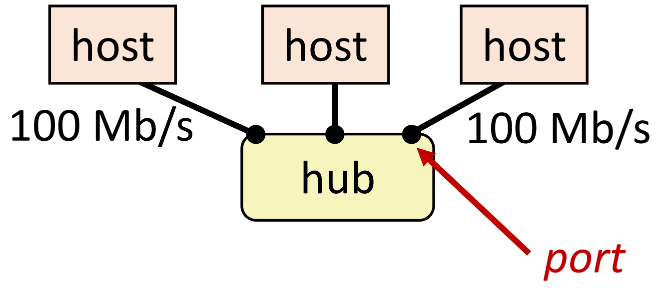
For More Information

- W. Richard Stevens et. al. “Unix Network Programming: The Sockets Networking API”, Volume 1, Third Edition, Prentice Hall, 2003
  - THE network programming bible.
- Michael Kerrisk, “The Linux Programming Interface”, No Starch Press, 2010
  - THE Linux programming bible.
- Code examples
  - csapp.{.c,h}, hostinfo.c, echoclient.c, echoserveri.c, tiny.c, adder.c
  - You can use any of this code in your assignments.

BONUS SLIDES

The following slides are for those curious.
You will NOT be expected to know this material.

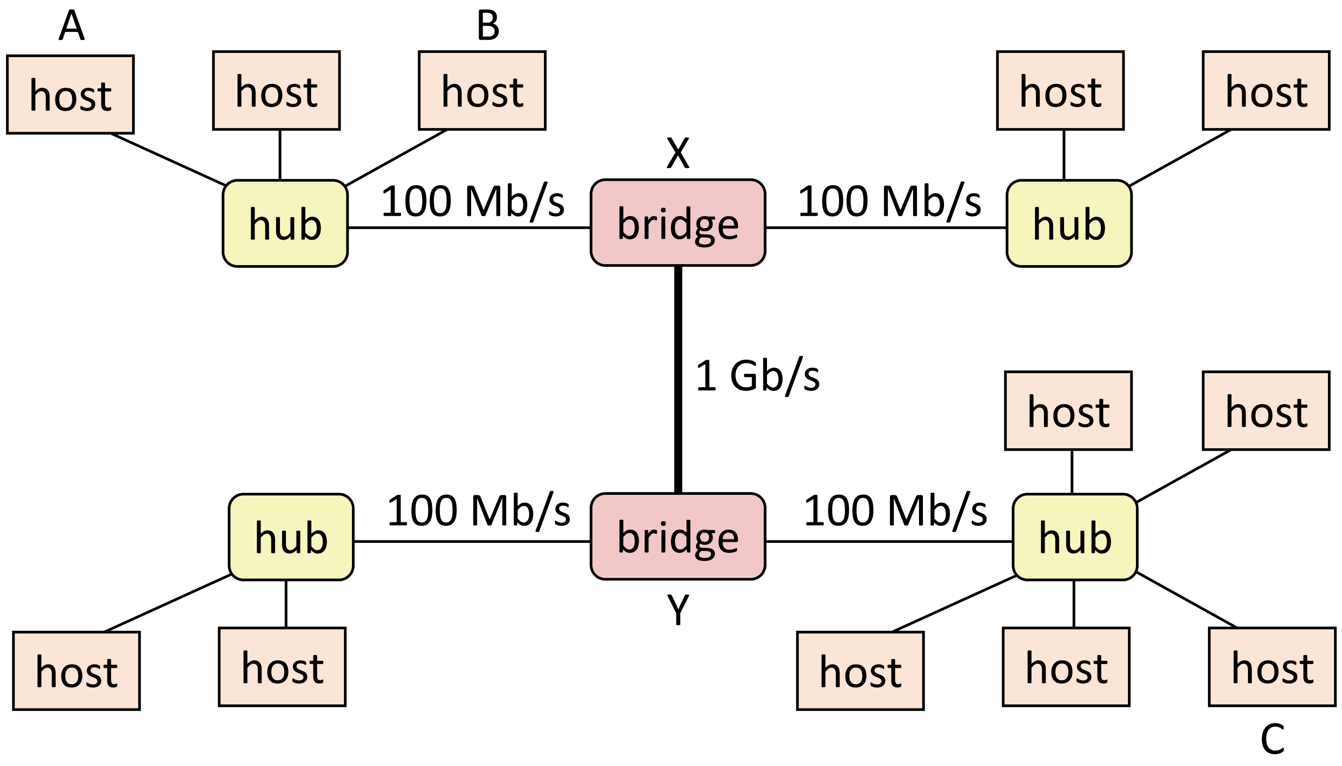
Lowest Level: Ethernet Segment



- 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]

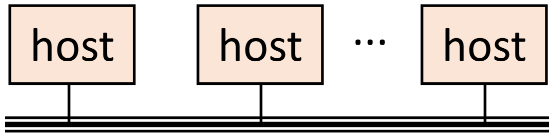
Next Level: Bridged Ethernet Segment



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

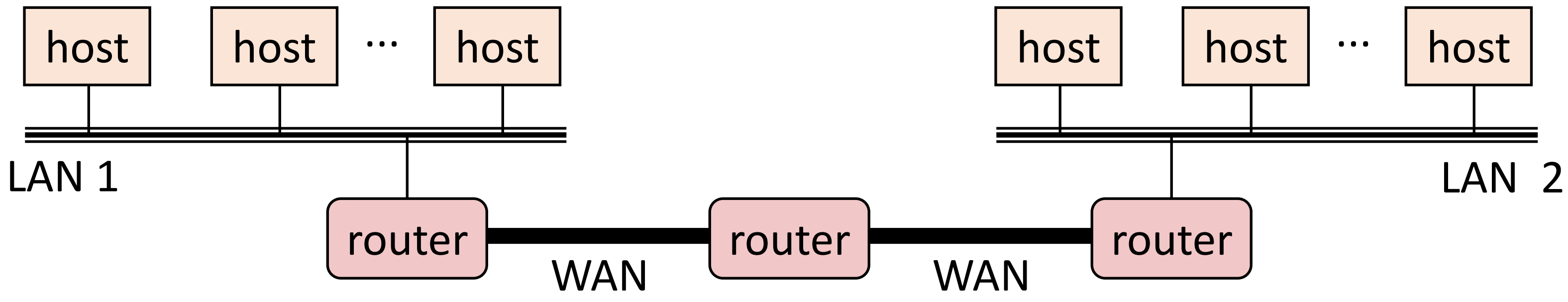
Conceptual View of LANs

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



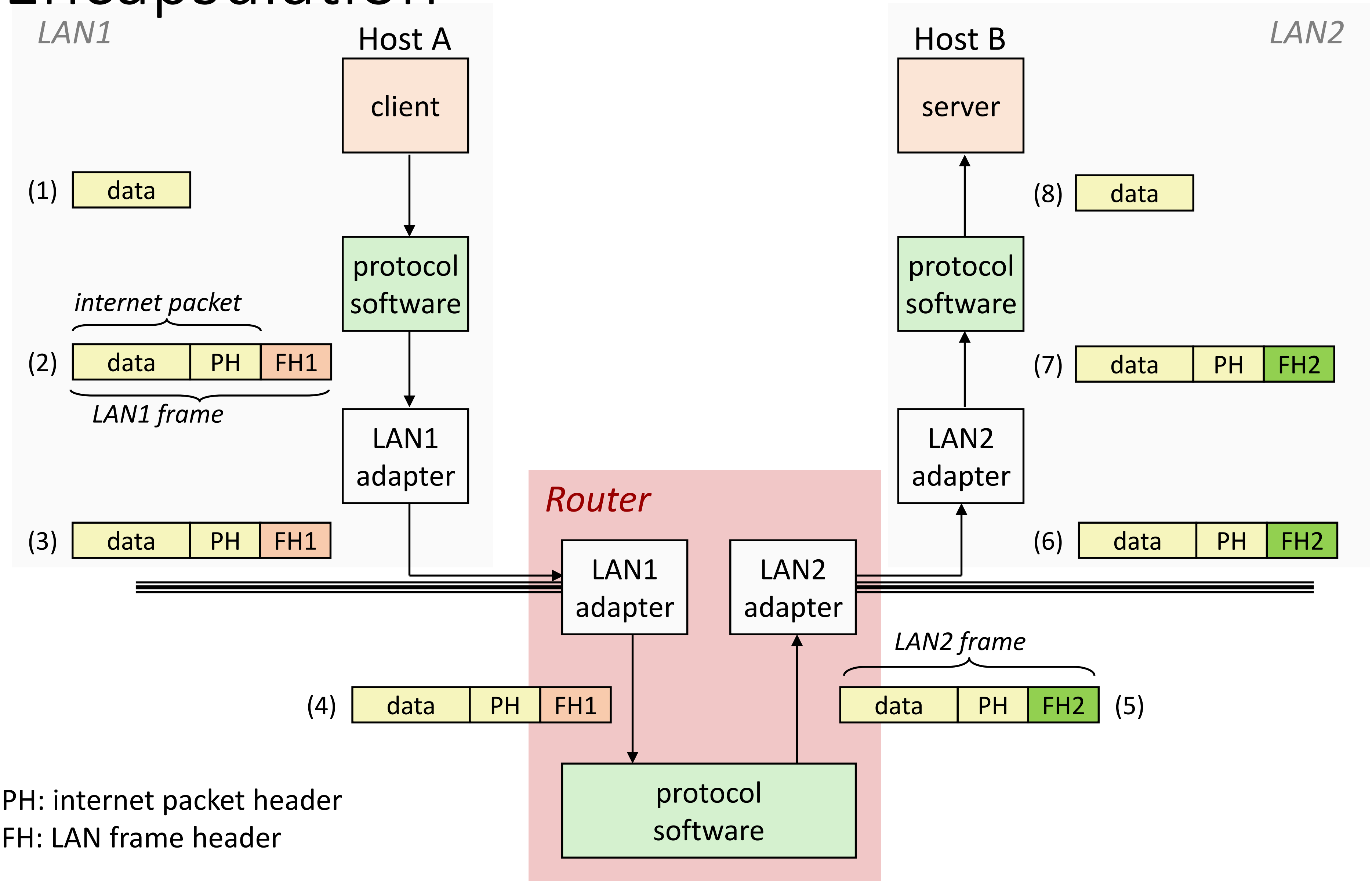
Next Level: internets

- 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, …)*

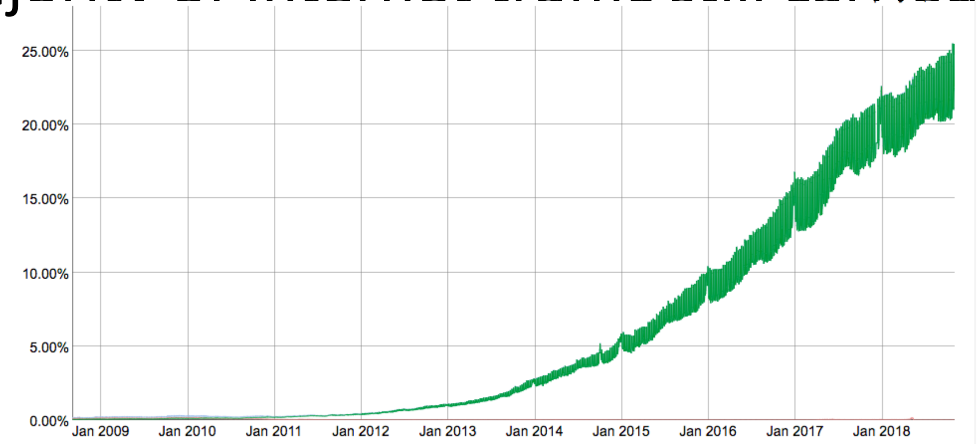
Transferring internet Data Via Encapsulation



PH: internet packet header
FH: LAN frame header

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



IPv6 traffic at Google

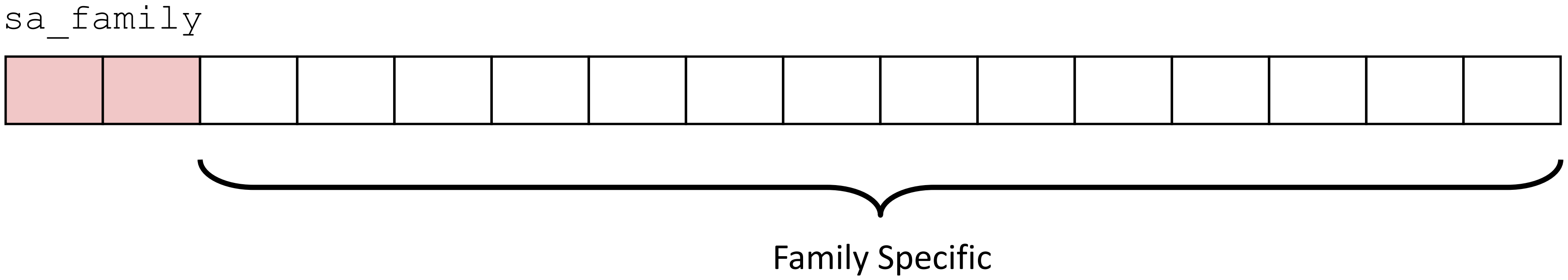
- We will focus on IPv4, but will show you how to write networking code that is protocol-independent.

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;`

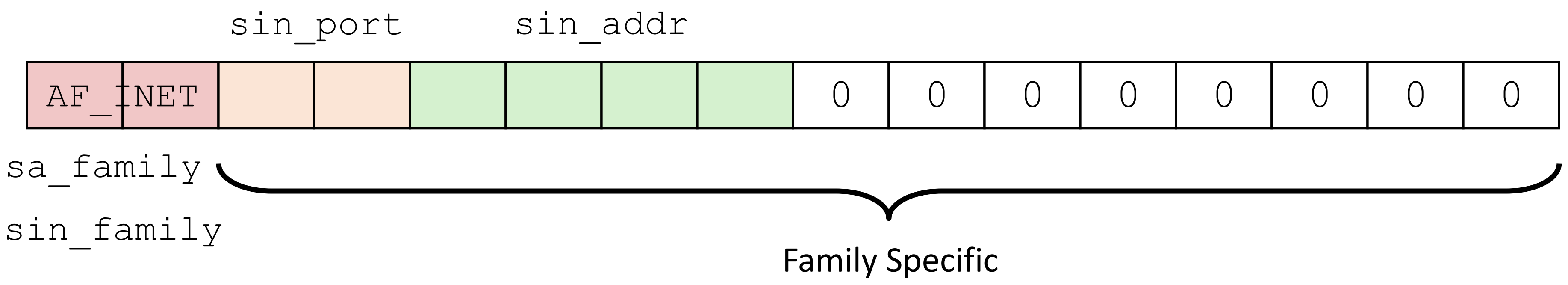
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data  */
};
```



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

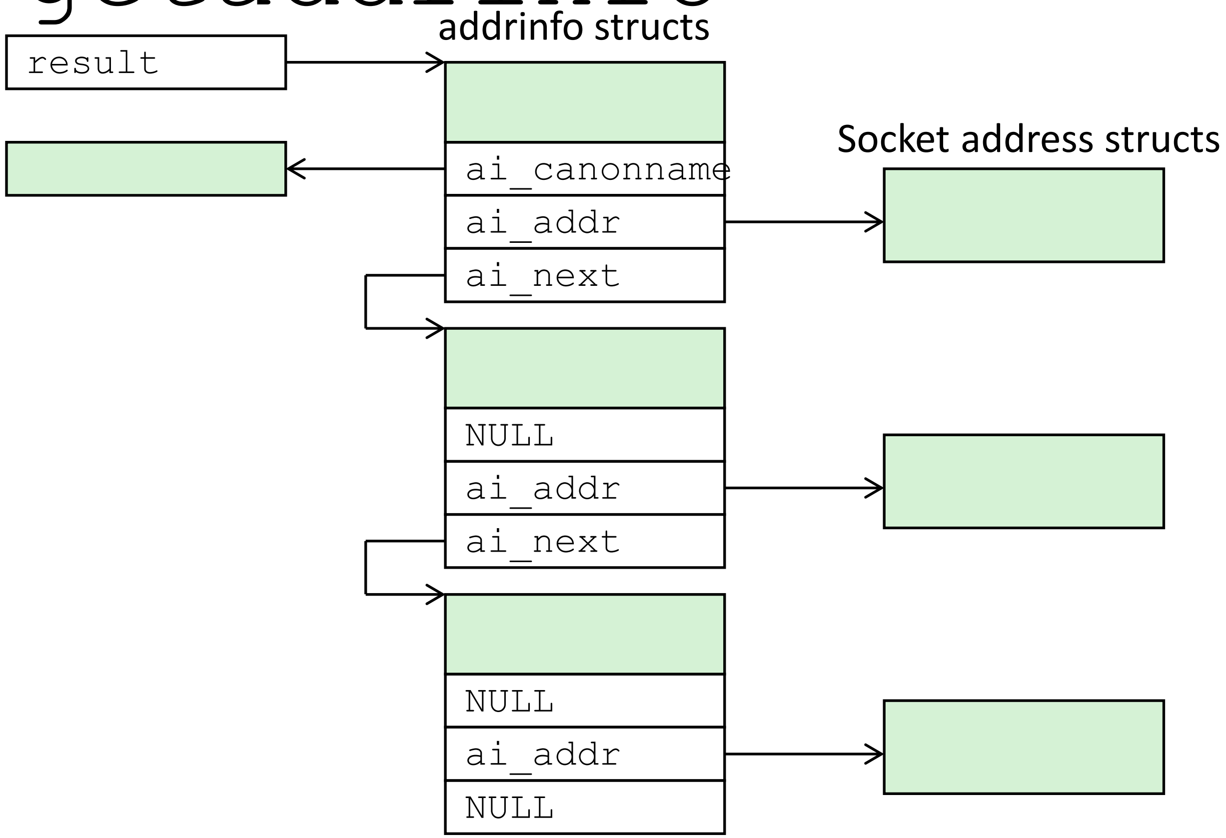
- Disadvantages
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

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`



- 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 (writing our own nslookup)

```
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

whaleshark> ./hostinfo google.com
172.217.15.110
2607:f8b0:4004:802::200e
```