Network Programming: Part I

CS230 System Programming
15th Lecture

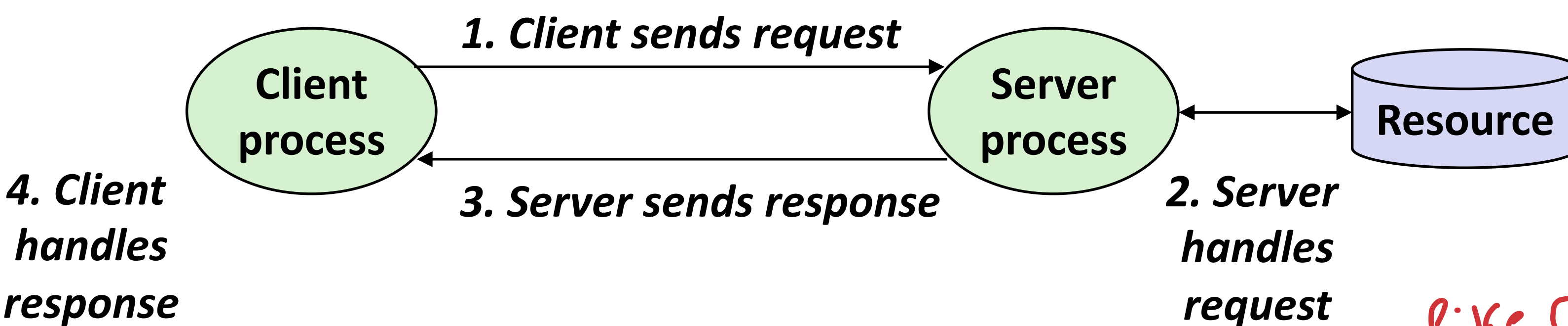
Instructors:

Jongse Park

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)

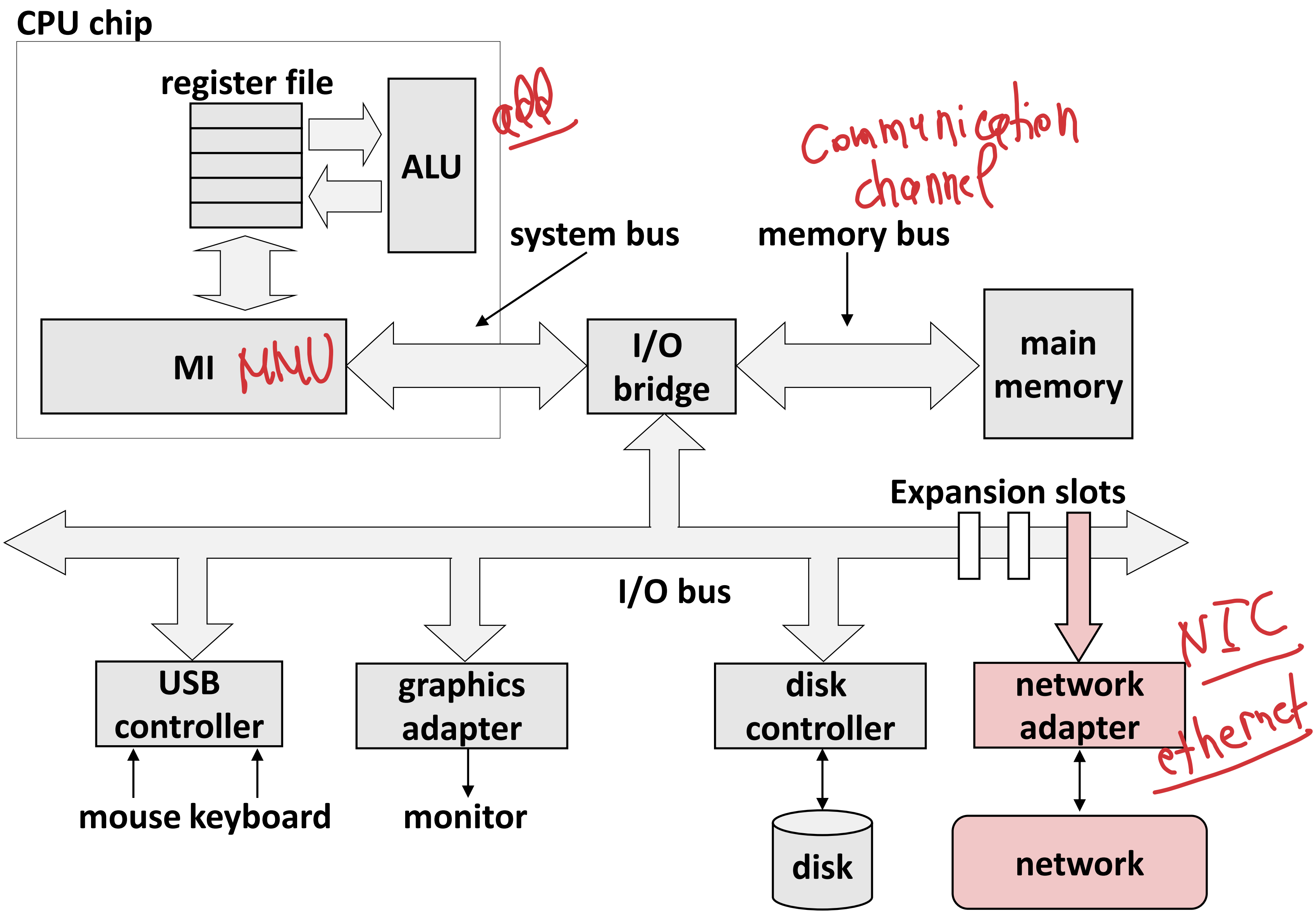
network transactions



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

→ like computers with names

Hardware Organization of a Network Host



Computer Networks

comp's

- **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, … (wi-fi router maybe, office)
  - LAN (Local Area Network) spans a building or campus $\boxed{148.248.}$
    - 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**

ultimate internet

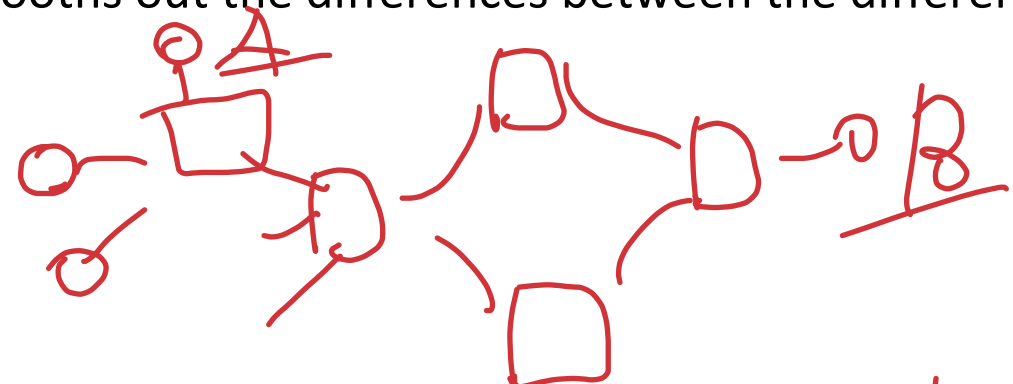
  - The Global IP Internet (uppercase "I") is the most famous example of an internet (lowercase "i")

The Notion of an internet Protocol

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

→ set of contracts between computers

- **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



A ⇔ B ?

how communicate

computer2computer networking

What Does an internet Protocol Do?

- **Provides a *naming scheme***
  - An internet protocol defines a uniform format for ***host addresses***
  - Each host → computer (and router) is assigned at least one of these internet addresses that uniquely identifies it → should have ≥1 name
- **Provides a *delivery mechanism***
  - An internet protocol defines a standard transfer unit (***packet***)
  - Packet consists of ***header*** and ***payload*** → actual data
    - Header: contains info such as packet size, source and destination addresses
    - Payload: contains data bits sent from source host

→ where and to?

Global IP Internet (upper case)

- **Most famous example of an internet**

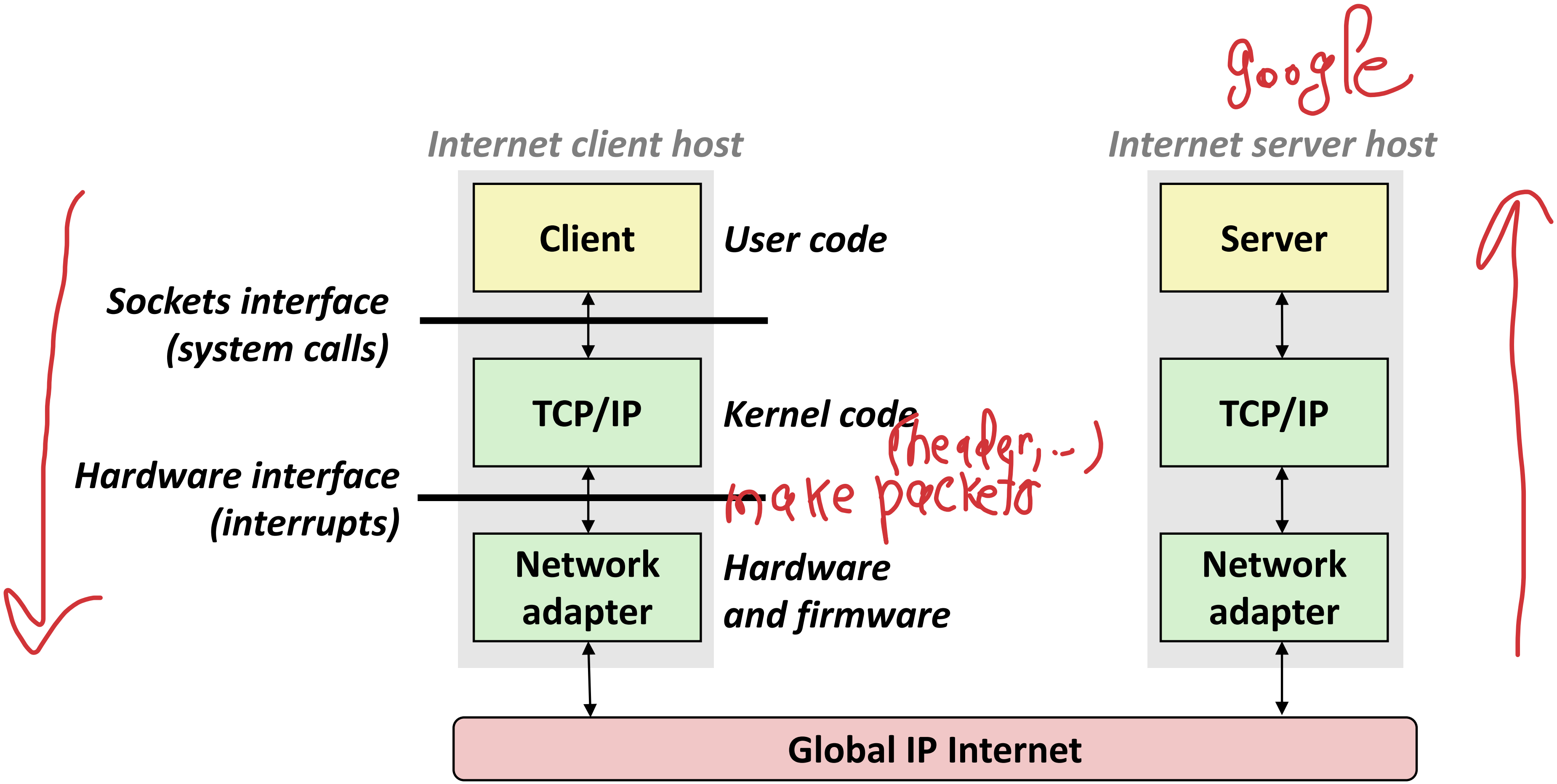
networking stack

- **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* youtube video, gaming
  - TCP (Transmission Control Protocol)
    - Uses IP to provide *reliable* byte streams from *process-to-process* over *connections* → packet received (used in internet)

- **Accessed via a mix of Unix file I/O and functions from the *sockets interface*** ← network stack (kernel)

abstract as file

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

CMU 148.248 names

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

url is also address

- 128.2.203.179 is mapped to www.cs.cmu.edu → domain (handled by DNS distributor)

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) in packet → header
  - True in general for any integer transferred in a packet header from one machine to another. → header should remember which port
    - 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) */
};
```

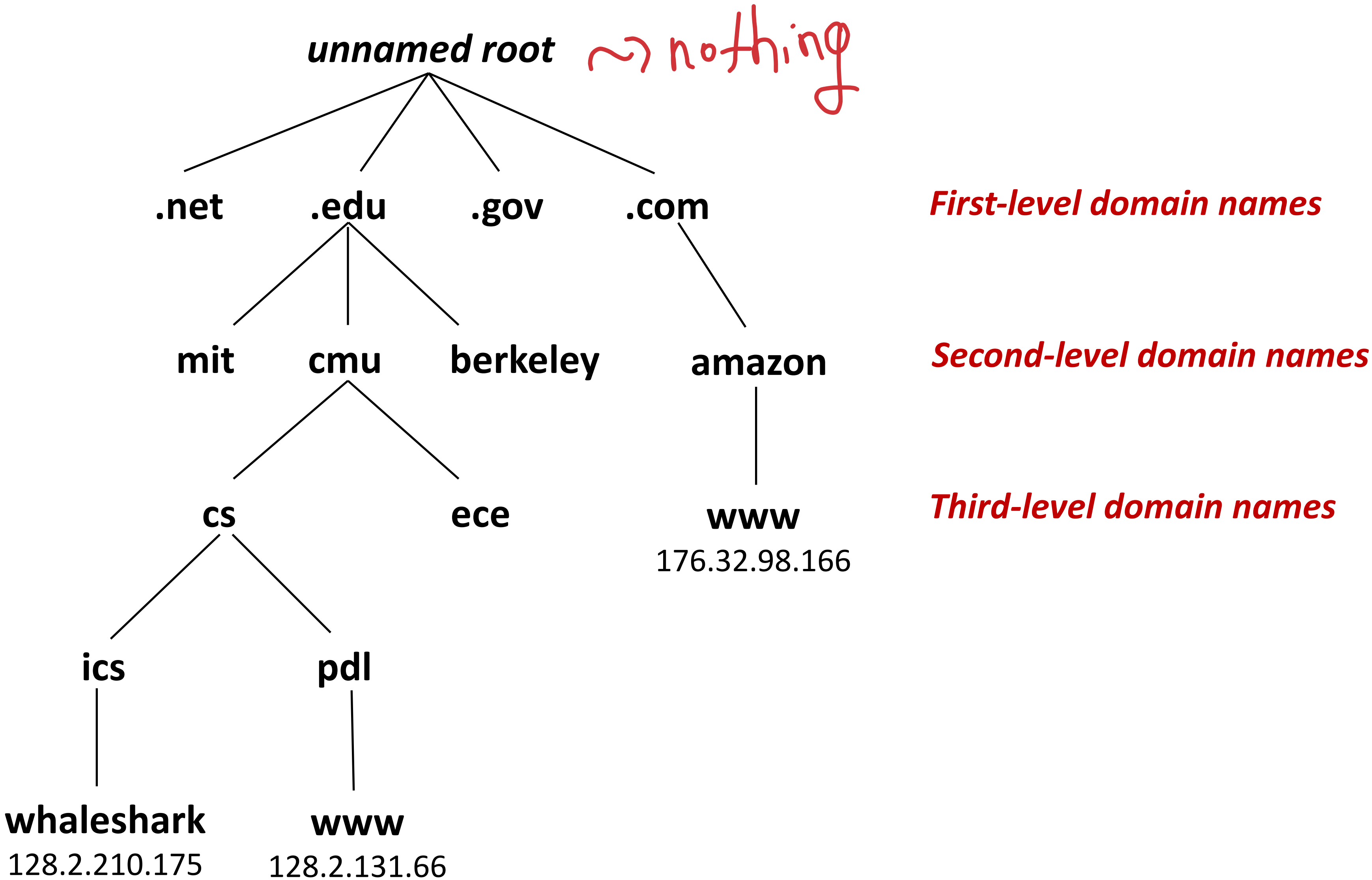
network programming + big/little-endian → big issue

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



Domain Naming System (DNS)

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

not single server

- **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.

mcel4801. kaist.ac.kr

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
```

access yourself

when design ur website
(how looks from other server)

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

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

Properties of DNS Mappings (cont)

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

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

- Multiple domain names mapped to the same IP address:

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

haver → multiple IP's
distribute IP traffic

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

communicate
build connections to

(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. (process: server and client) think of comp → but process
  - *Full-duplex*: data can flow in both directions at the same time, write, read
  - *Reliable*: stream of bytes sent by the source is eventually received by the destination in the same order it was sent. TCP

website ← TCP/UDP → zoom: ; okay with losing info

IP

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

↳ Identify computer system

- **A *port* is a 16-bit integer that identifies a process:** which 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) http://kc...:80

HTTP → reliable
TCP/UDP
IP

packets
routers
NIC

port ~ processes sending/receiving

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: $\boxed{22}$/ssh
  - email server: 25/smtp → email
  - Web servers: 80/http

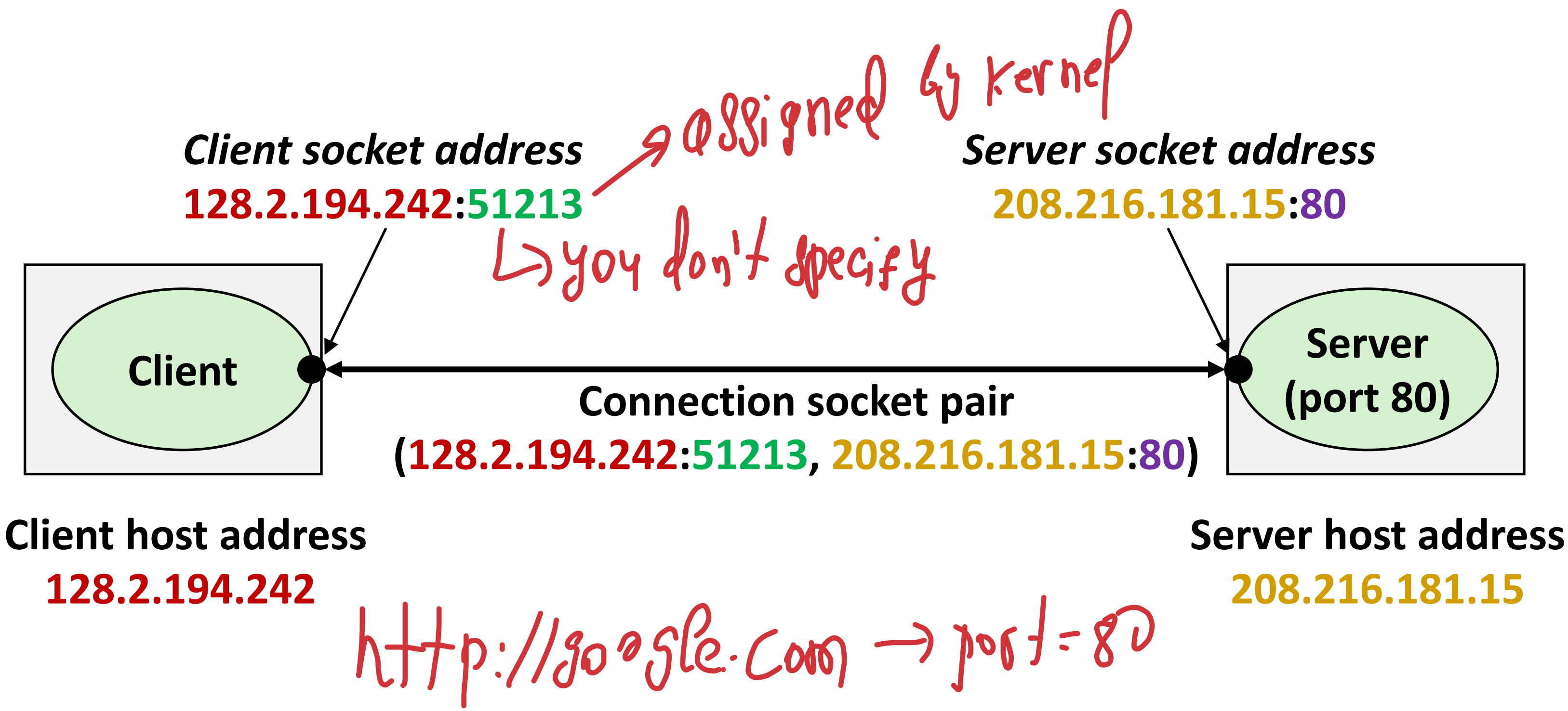
dangerous due to hackers
Sol: two-way authentic. number
ssh -p ☐ jsport@ —
↑port

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

/etc/ssh/ssh.cat

Anatomy of a Connection

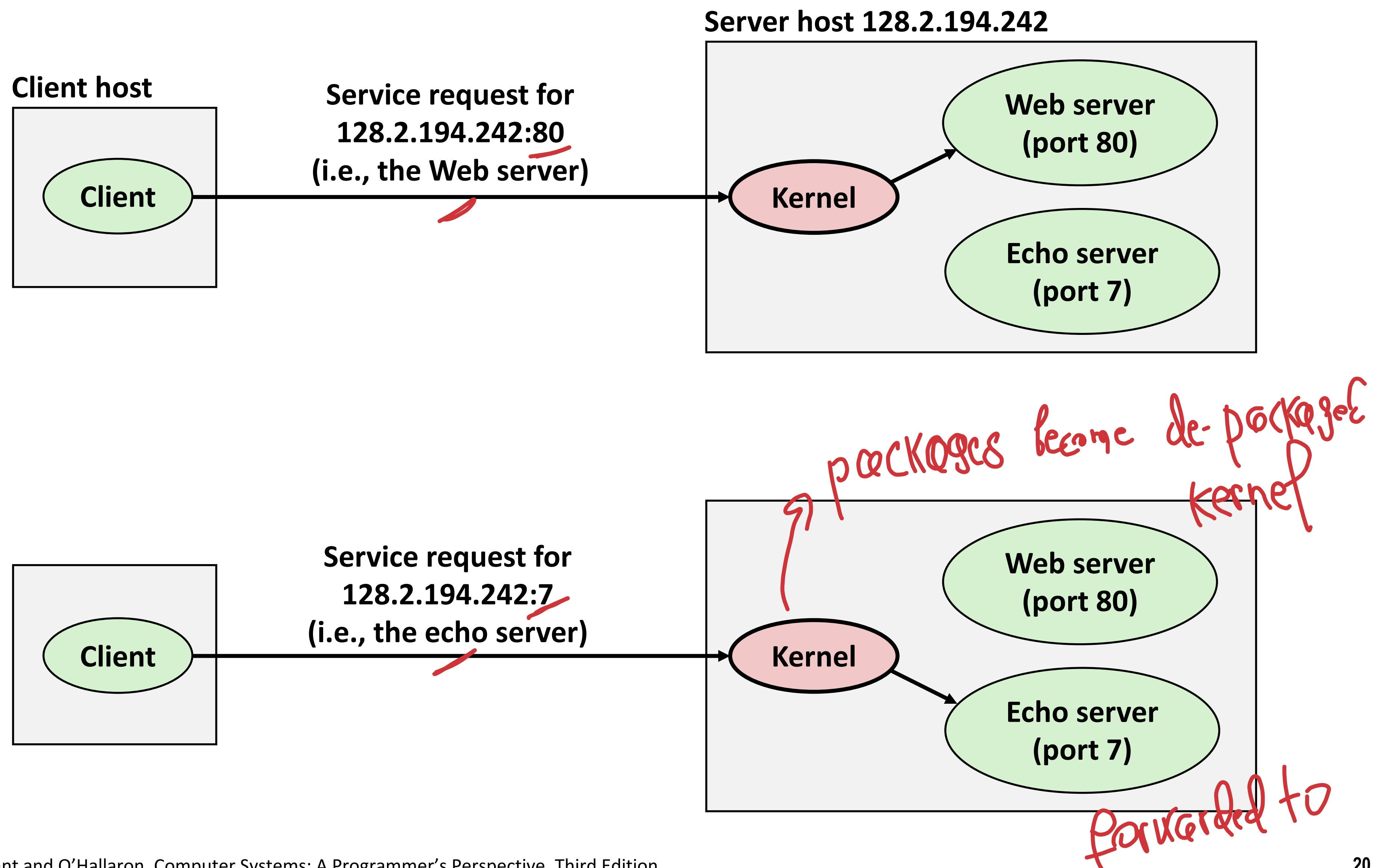
- A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)
  - `(cliaddr:cliport, servaddr:servport)`



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 $\boxed{\text{Unix I/O}}$ to build network applications.**

  ↳ used after connection's built

- **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

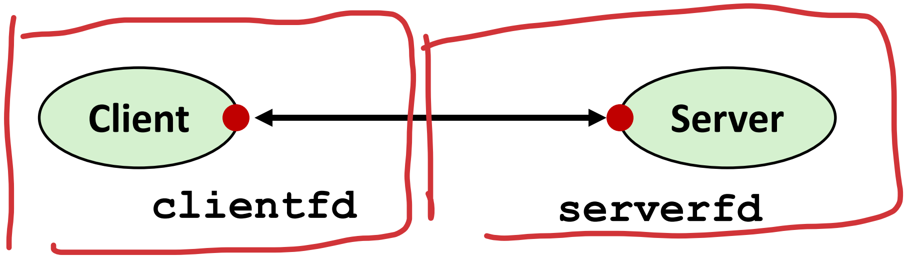
Sockets

App
file descriptor
socket
Net
kernel
HTTP
TCP
IP

- **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**

own descriptors

2 different processes



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

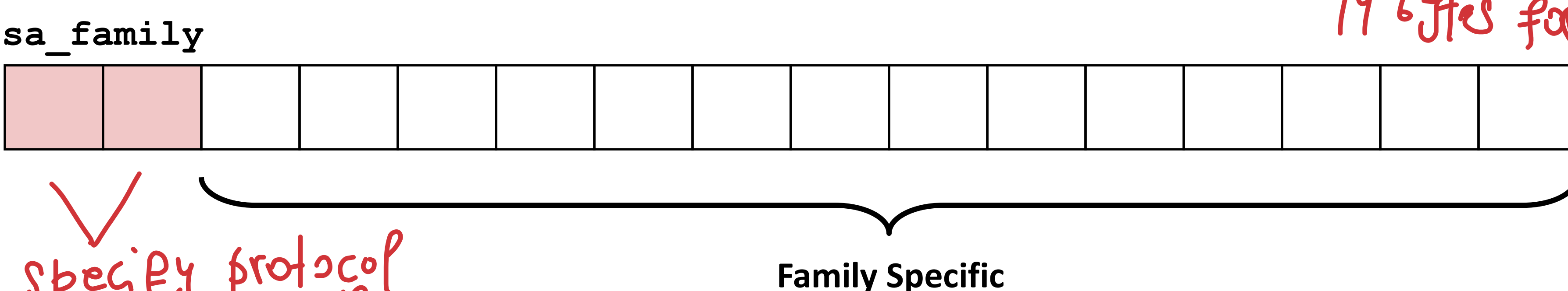
Socket Address Structures

functions to build connections

- **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
    ↳ pointer to any DS
  - For casting convenience, we adopt the Stevens convention:
    cast to other type
    `typedef struct sockaddr SA;`

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

use 14 bytes for



specify protocol family

Socket Address Structures

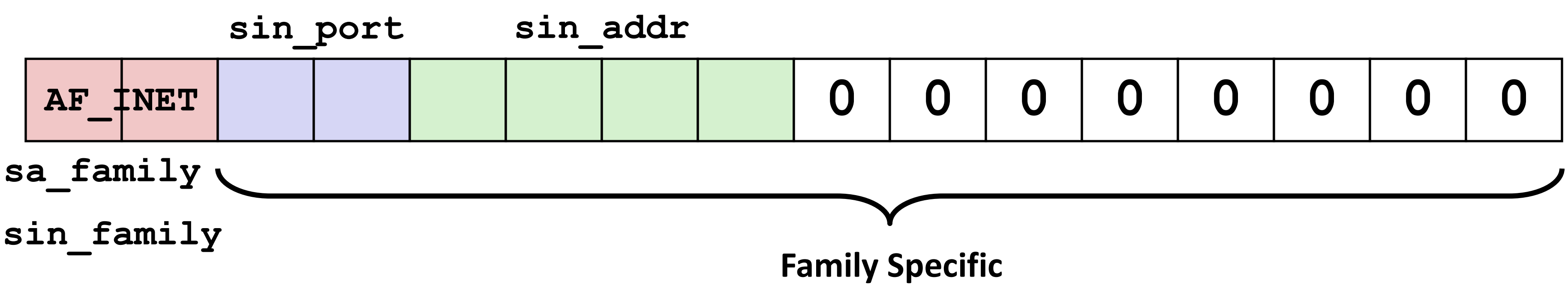
- **Internet-specific socket address:**
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

→ depending on protocol can have multiple variants

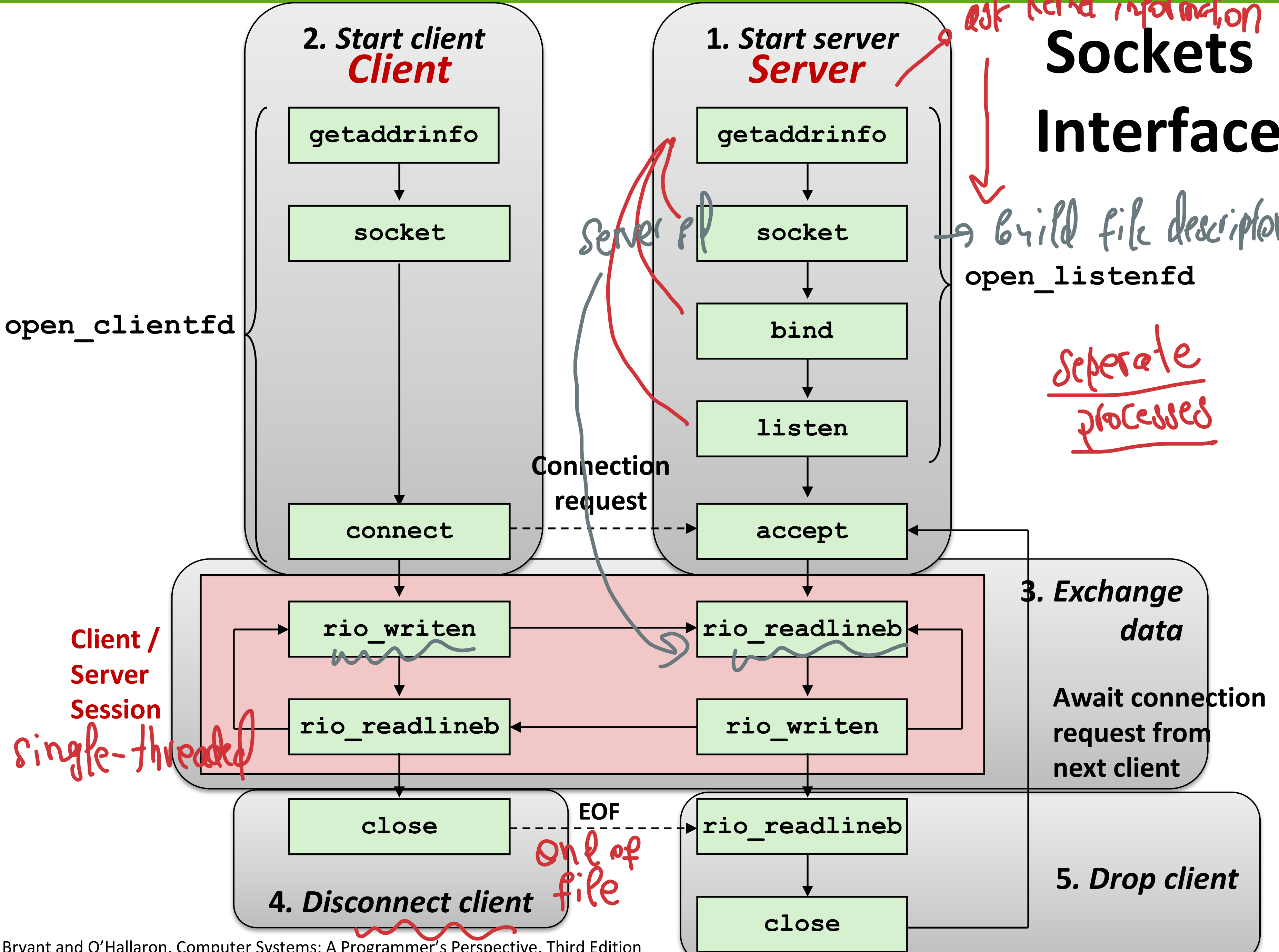
If OOP then we could inherit from sockaddr

```
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) */
};
```

↳ specify protocol family



Sockets Interface



function provided by kernel

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.

google.com → IP address
port
service
} what you know

- **Advantages:** concurrent processes ✓
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

↳ not specific to protocol
protocol-specific provided by function is given by kernel

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

TCP, Internet-protocol (same pattern)

Host and Service Conversion: getaddrinfo

"a.b." "google.com"

IP

→predefined service name

```
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 */
```

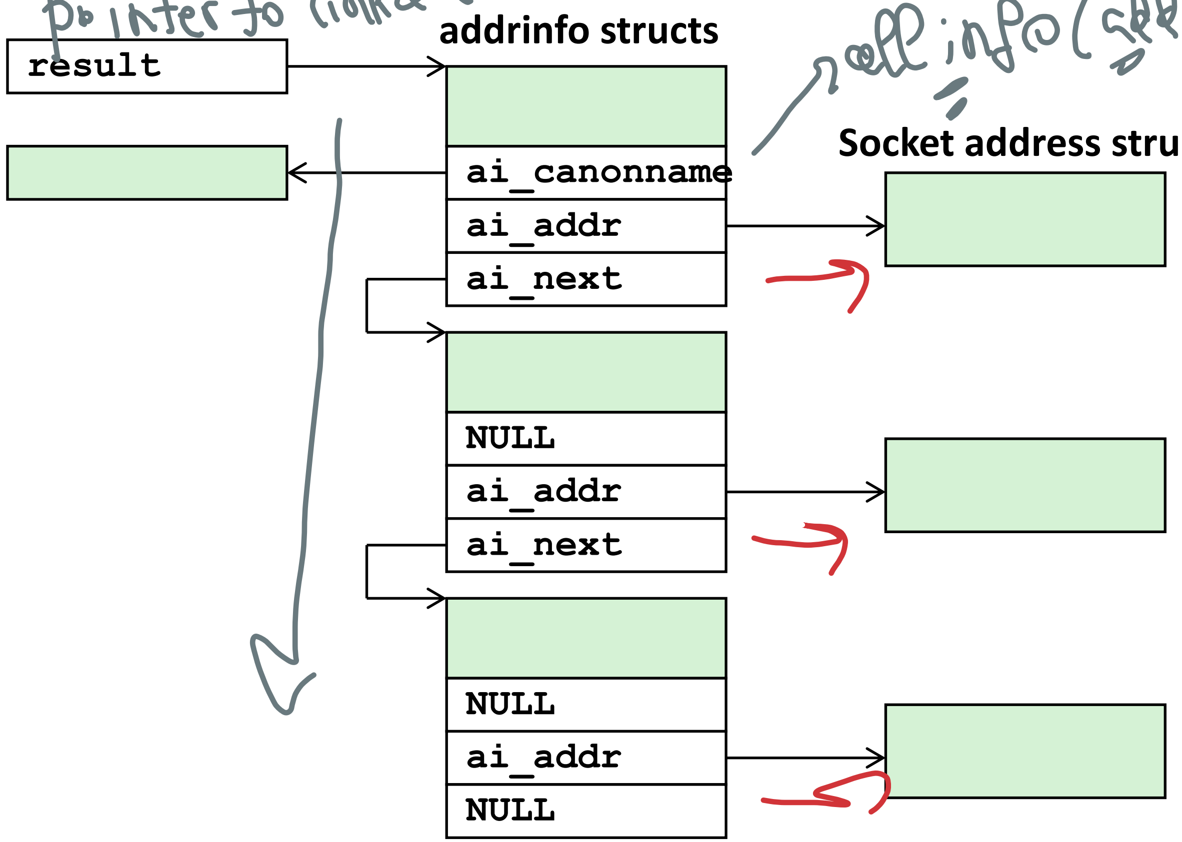
optionally constrain

- **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`

pointer to linked list

addrinfo (address)



bind socket address into socket descriptor in App

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

and connect

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

addrinfo Struct

singly linked-list elem

```
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 */
};
```

args for func

only next

- 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.

all these will be required for connect/bind inverse of getaddrinfo

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

```c
#include "csapp.h"

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

memory set

reserve space for initial linked-list

```c
    /* 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 */
```

optional

protocol-family

TCP

```c
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
```

hostinfo.c

result

Conversion Example (cont)

initial address of linked-list

```
    /* 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);
}
```

IP address of host name

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
```

address info

Sockets Interface: `socket`

fields of address info

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

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

get

socket

- Example:

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

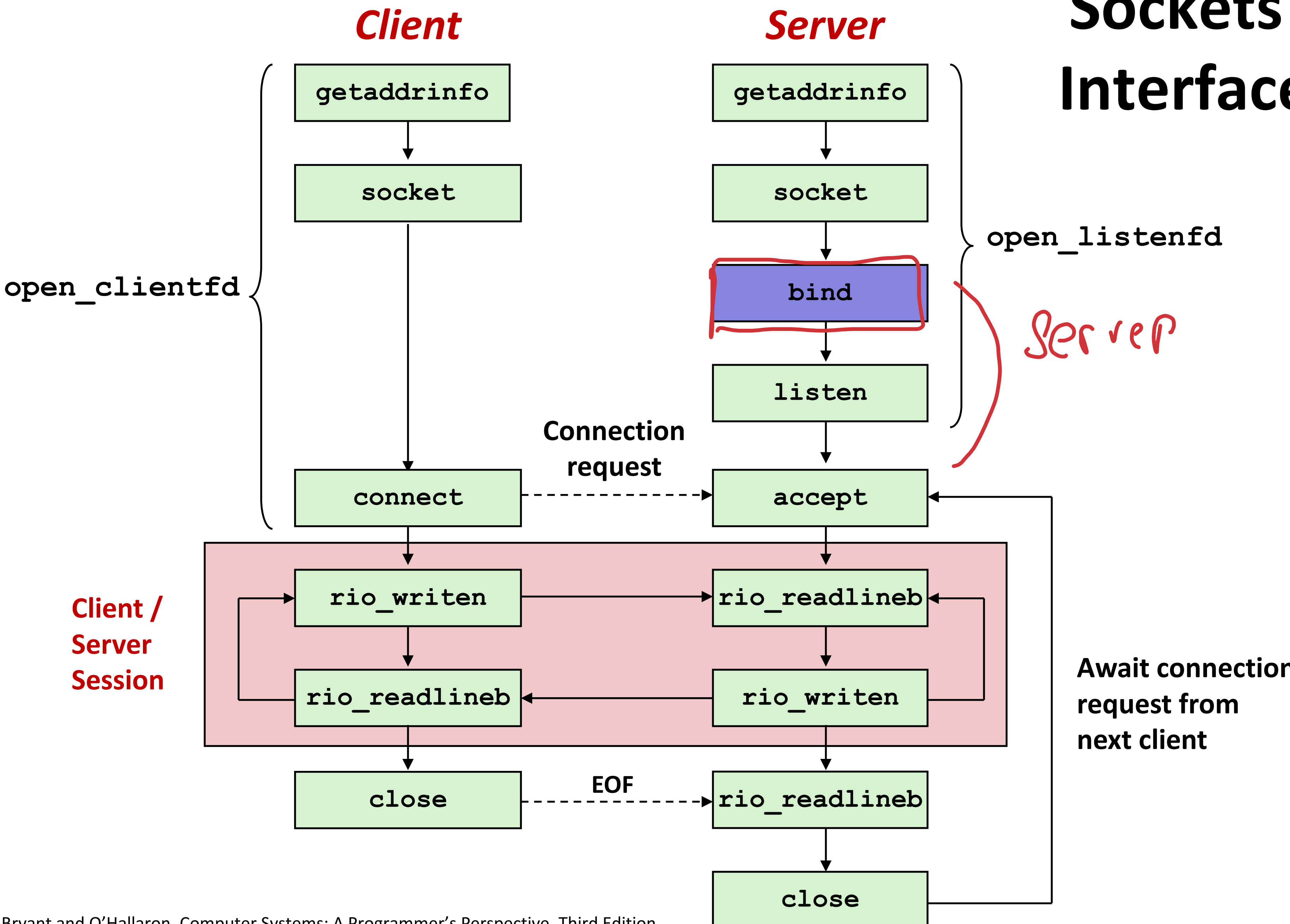
Indicates that we are using 32-bit IPV4 addresses

p.ai_addr
(pass from getinfo)

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: after socket function

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

return of socket

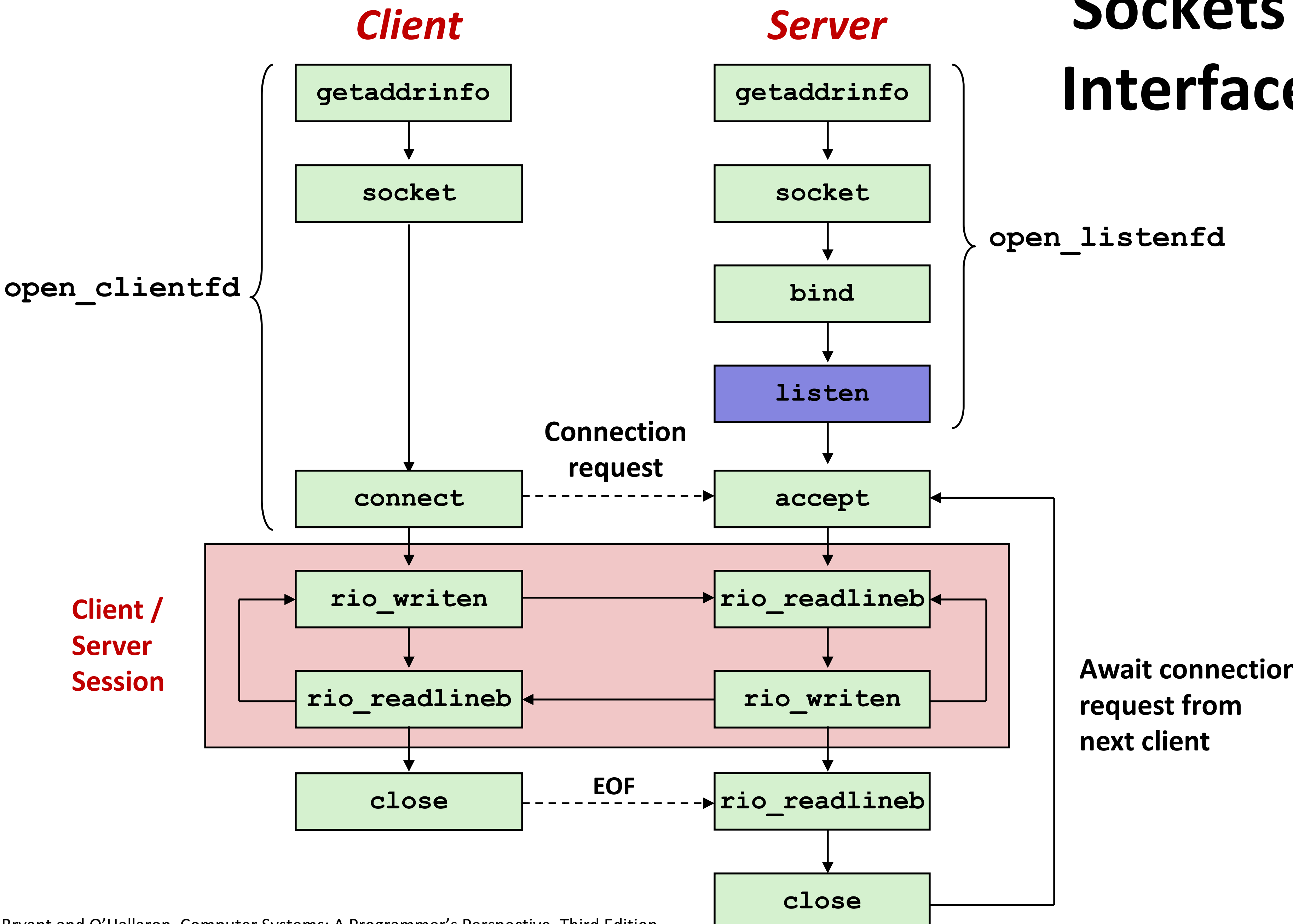
managed by kernel

provided by getaddrinfo

- 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`

a server

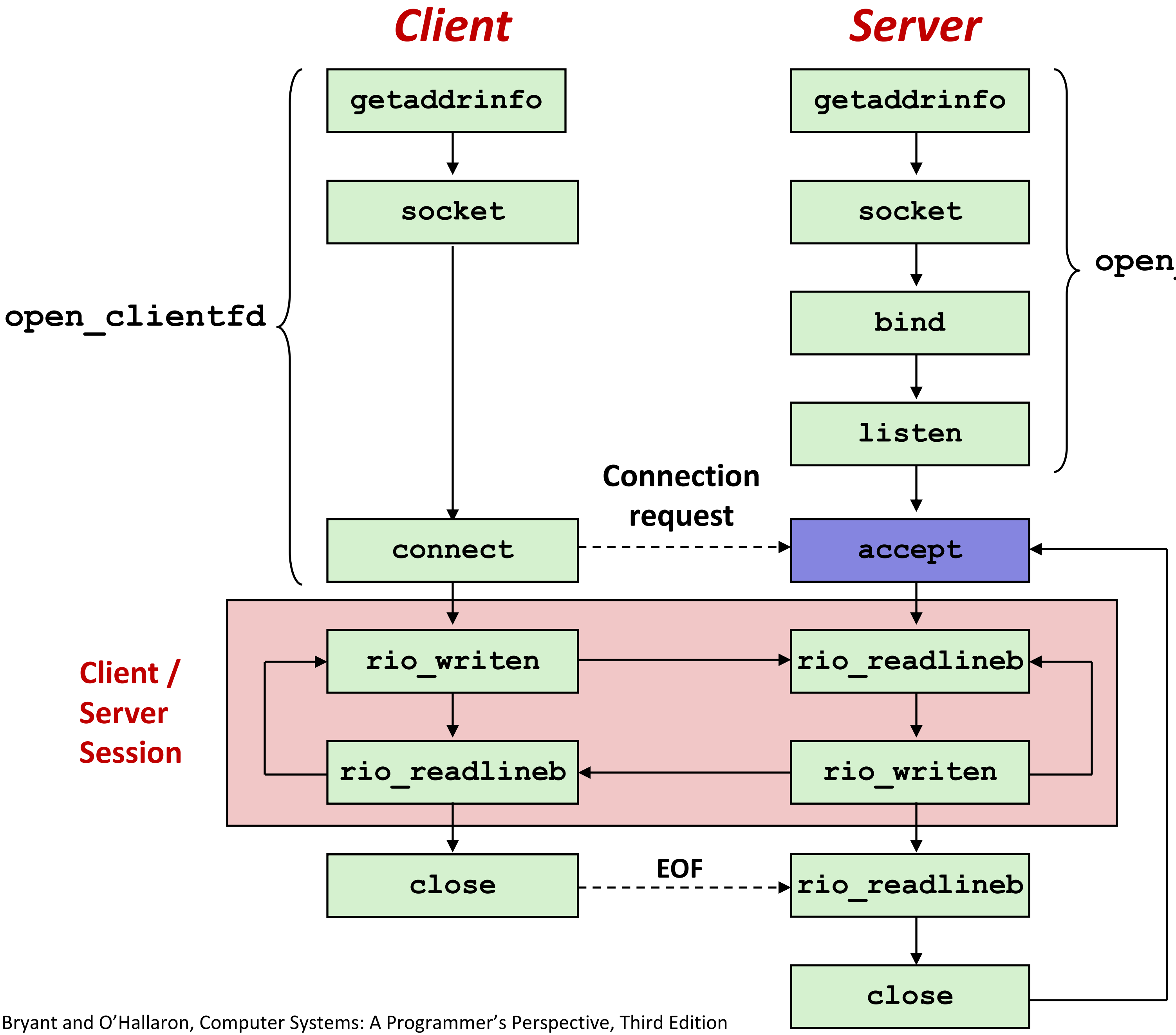
Hey kernel, this sd will be used as

- By default, kernel assumes that descriptor from socket function is an ***active socket*** that will be on the client end of a connection. Dont forget → you're server
- 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



listen → hey kernel, you'll be listen socket (addr)

Sockets Interface: accept

accept → now, listen! (do)

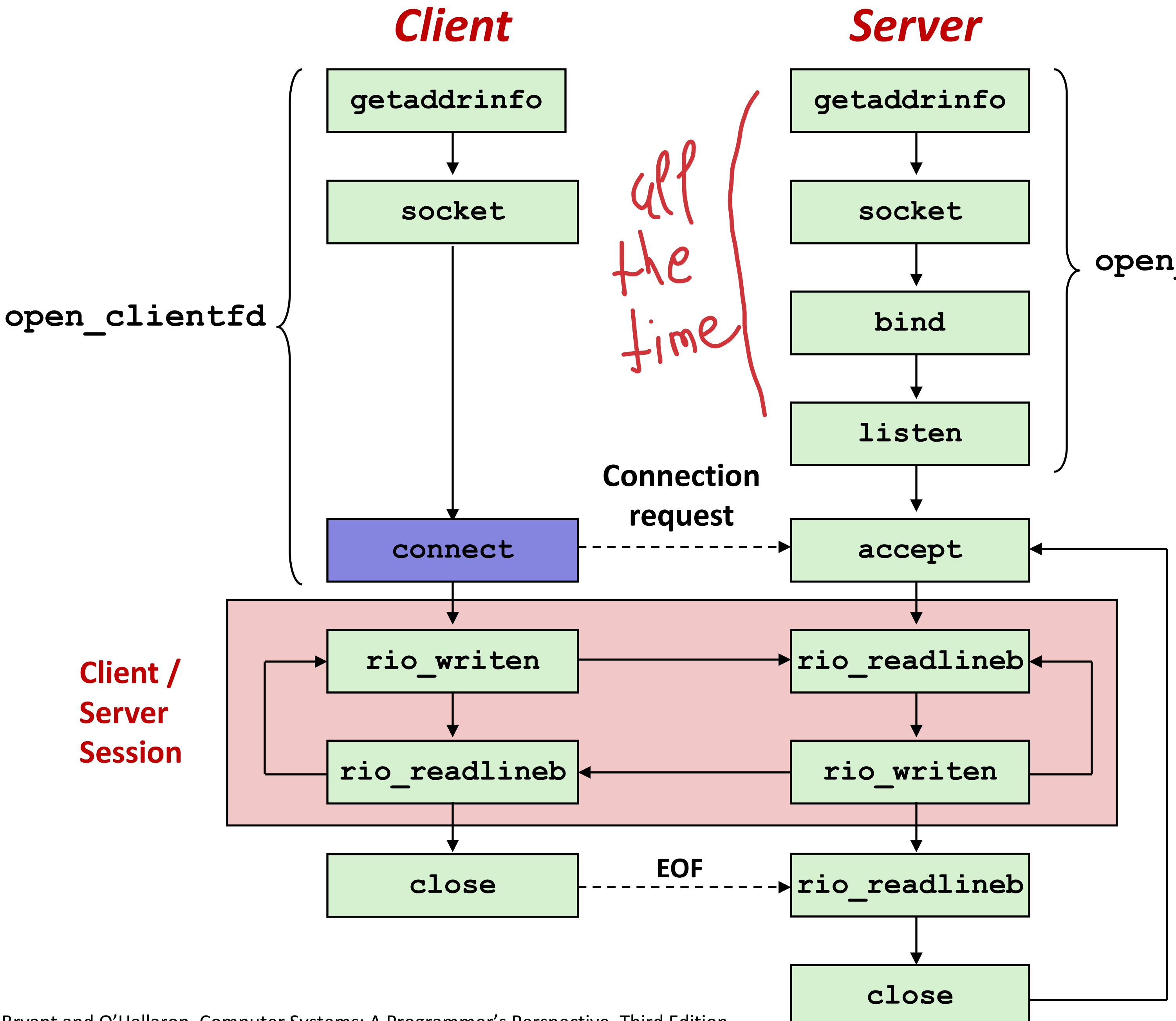
- **Servers wait for connection requests from clients by calling `accept`:**

```
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.**

(read/write)

Sockets Interface



Sockets Interface: connect

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

from getaddrinfo

```
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)`

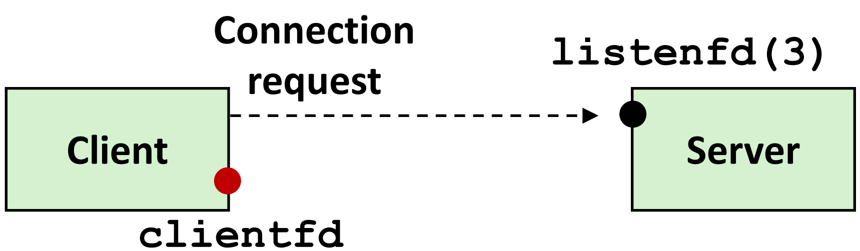
    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host

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

accept Illustrated

creates socket descriptor

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



built upon networking stack

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

(host, IP address, port)

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

↳ communicate actual data

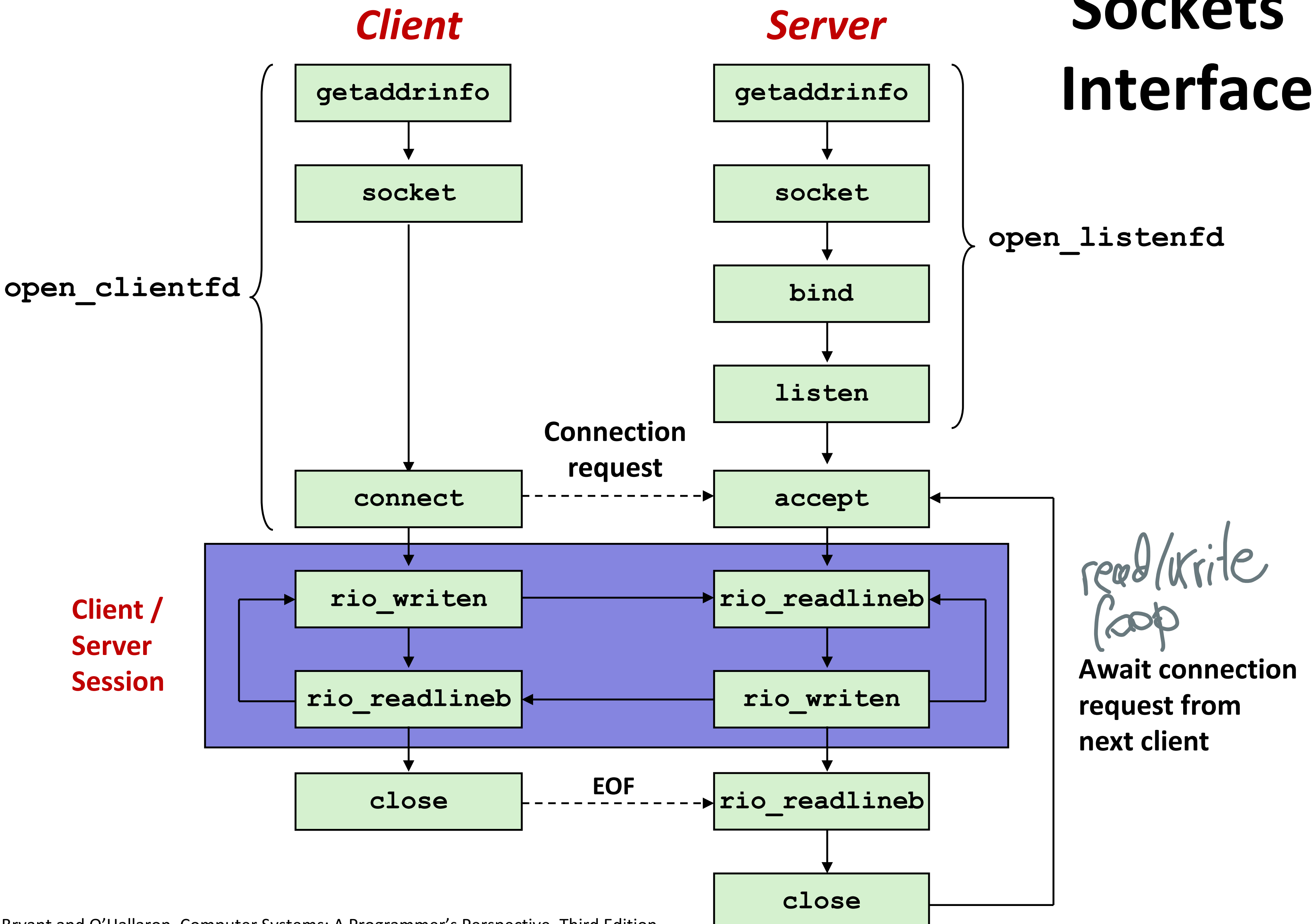
Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server
- **Connected descriptor** Can create multiple connected descriptors
  - 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

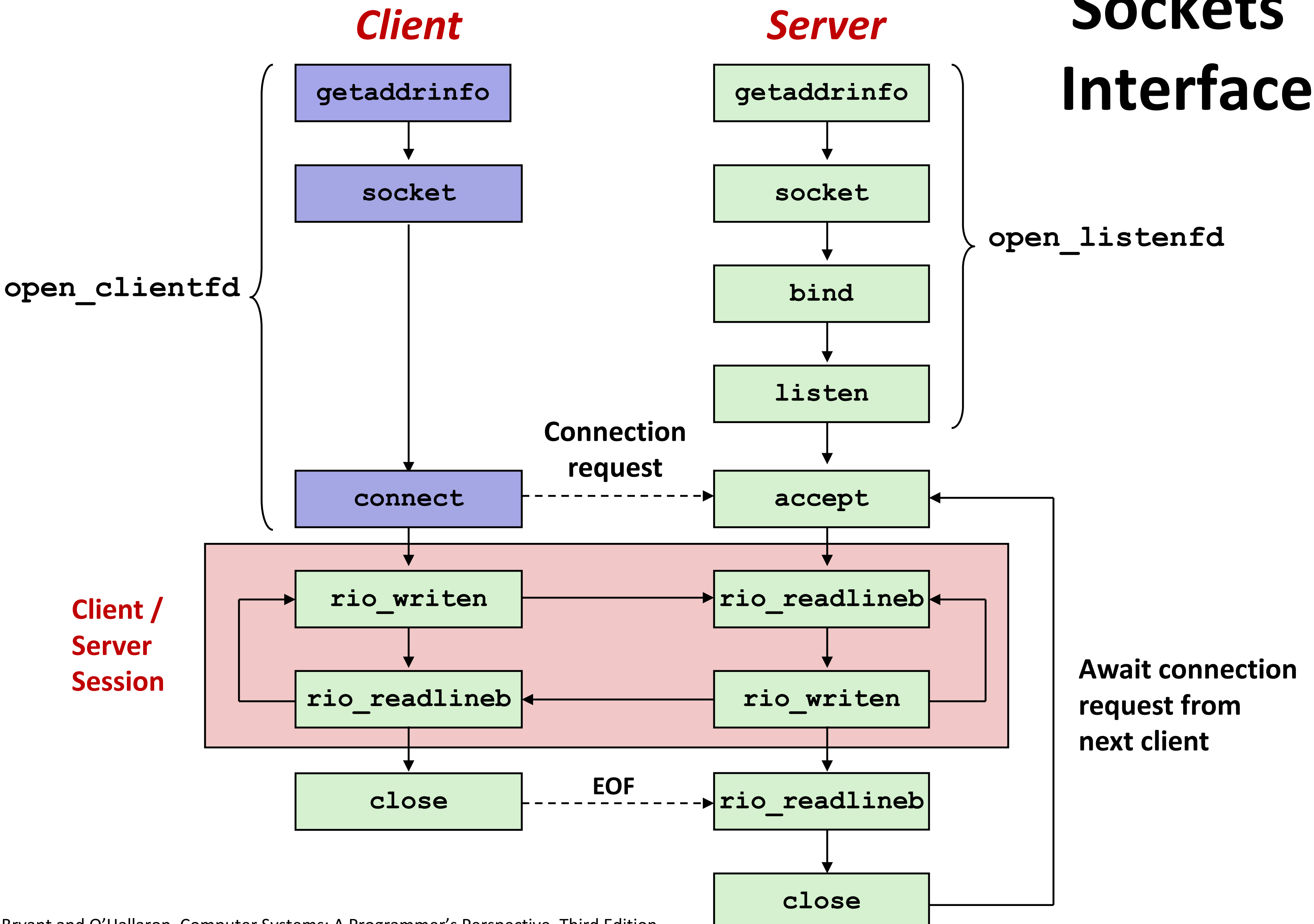
closes whenever certain client finishes

- **Why the distinction?** multiple threads
  - 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



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;
```

specify hints with certain goals

```
  /* 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);
```

→ will have address info

csapp.c

Sockets Helper: open_clientfd (cont)

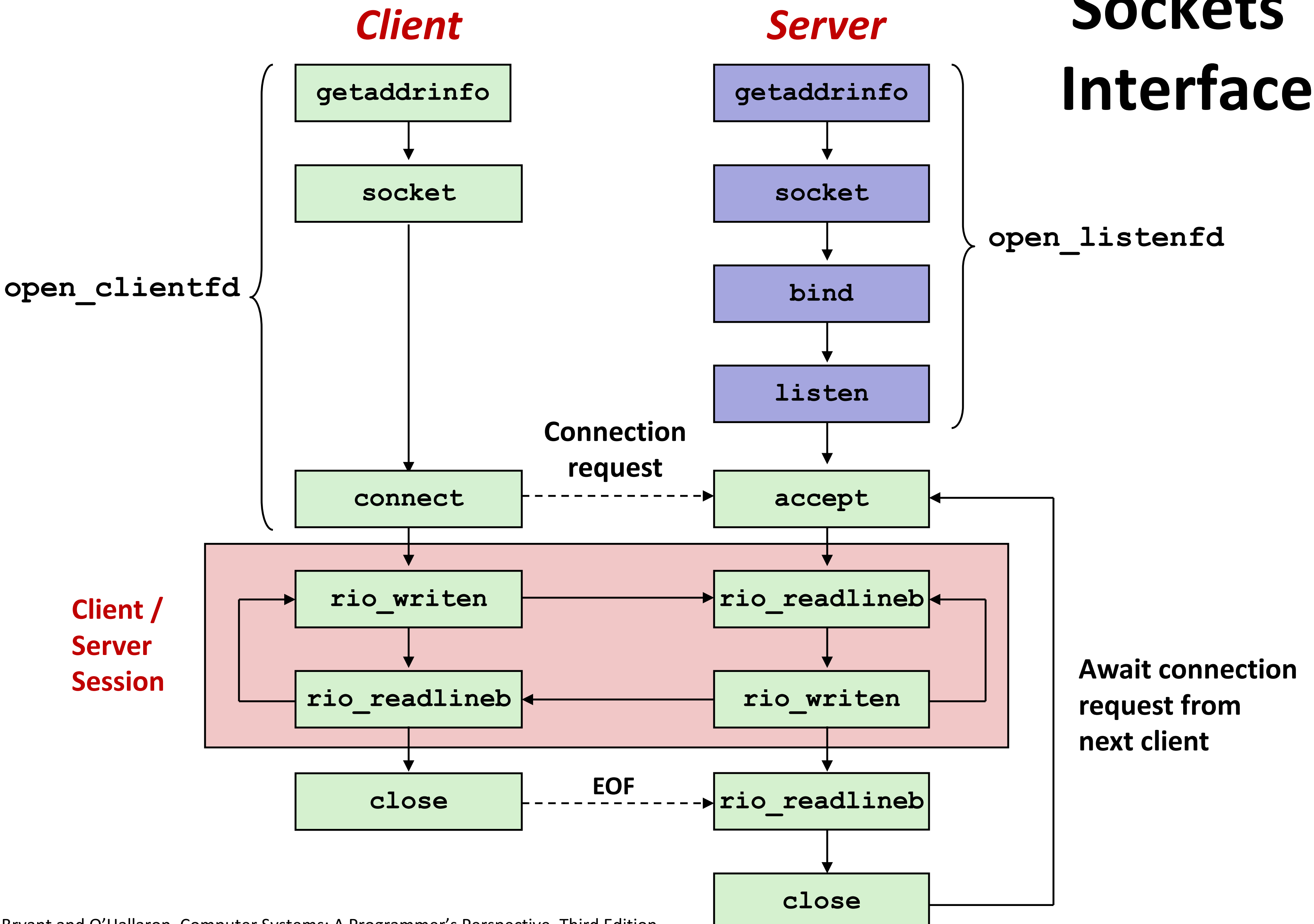
```
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket failed, 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;   Null
    else    /* The last connect succeeded */
        return clientfd;  ~> socket descriptor
}
```

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 */
}
```

SA type

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

↳ listening file descriptor

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

Echo Client: Main Routine

```
#include "csapp.h"

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

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

    clientfd = Open_clientfd(host, port);   ~> opened socket descriptor
    Rio_readinitb(&rio, clientfd);

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

echoclient.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);
}
```

2 socket descriptors

host name of server

obtain those

Connected socket descriptor

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

read/write
loop

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:**

client function

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

Testing the Echo Server With `telnet`

```
whaleshark> ./echoserveri 15213
Connected to (MAKOSHARK.ICS.CS.CMU.EDU, 50280)
server received 11 bytes
server received 8 bytes



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 '^]'.
Hi there!
Hi there!
Howdy!
Howdy!
^]
telnet> quit
Connection closed.
makoshark>
```

client server

telnet

testing

echo server