

# Network Programming: Part II

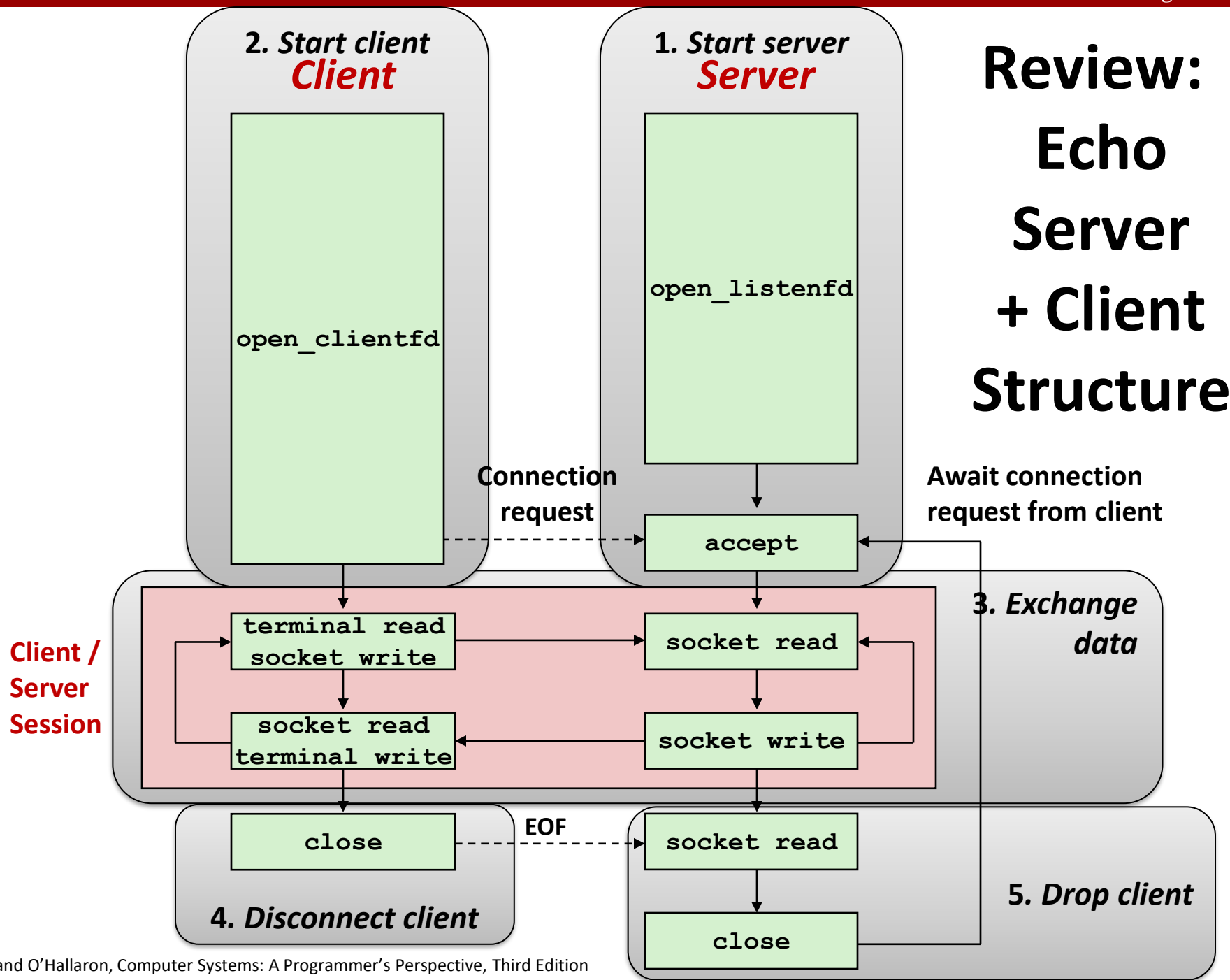
15-213/15-513: Introduction to Computer Systems  
22<sup>nd</sup> Lecture, July 19, 2024

## Instructors:

Brian Railing

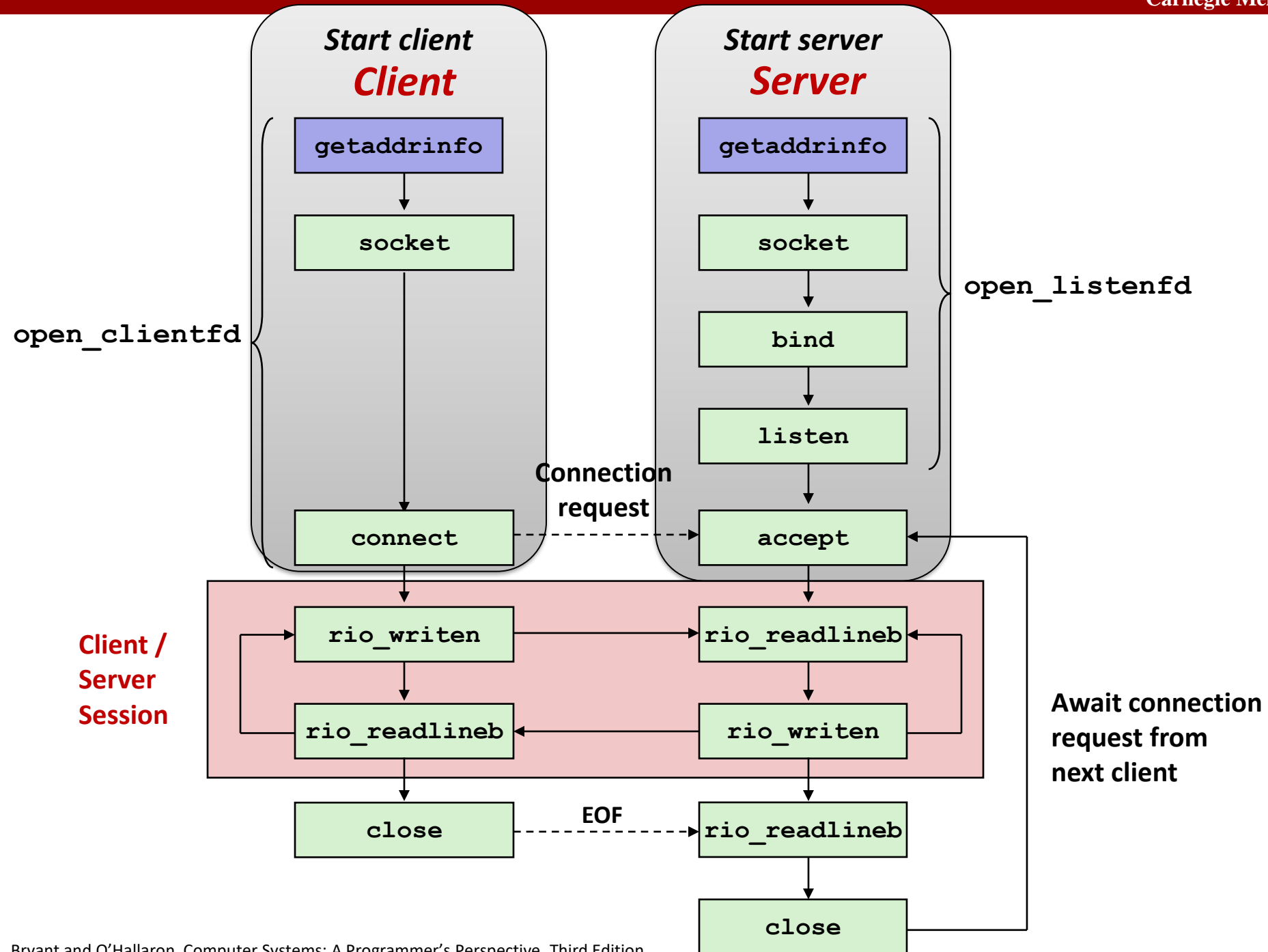
# Reminders

- **Shell lab due on Friday, July 19 at 11:59pm EDT**
- **Proxy lab out tonight**
  - Final due Friday, August 2 (no extensions!)
- **SFS lab out tonight (probably)**
  - Also due Friday, August 2 (no extensions!)
- **Final exam: Thursday August 1 (unless google form)**



# Today

- **Setting up connections**
- Application protocol example: HTTP

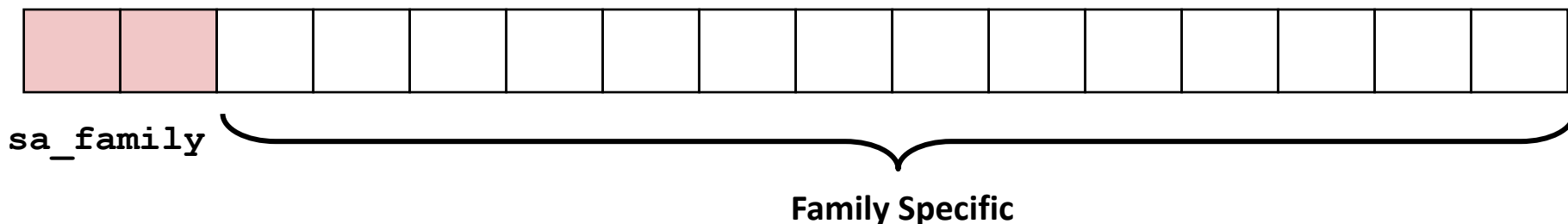


# Review: Generic Socket Address

## ■ Generic socket address:

- For address arguments to **connect**, **bind**, and **accept**

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



# Review: Socket Address Structures

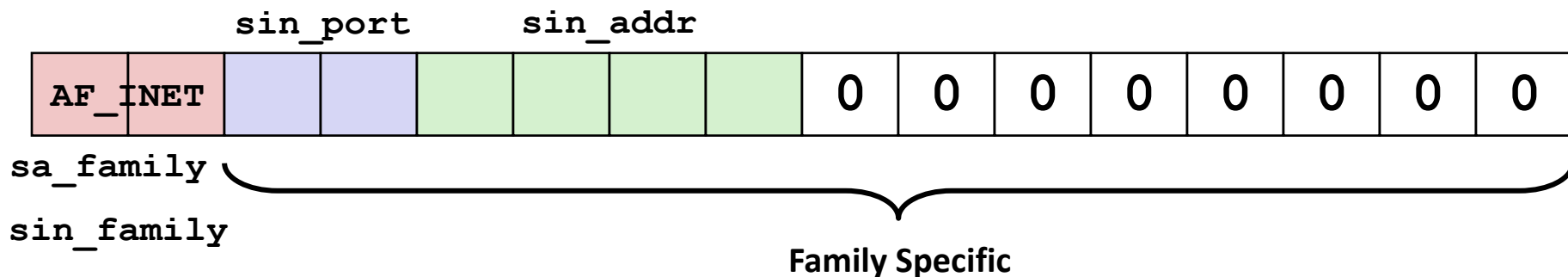
## ■ Internet (IPv4) specific socket address:

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

```

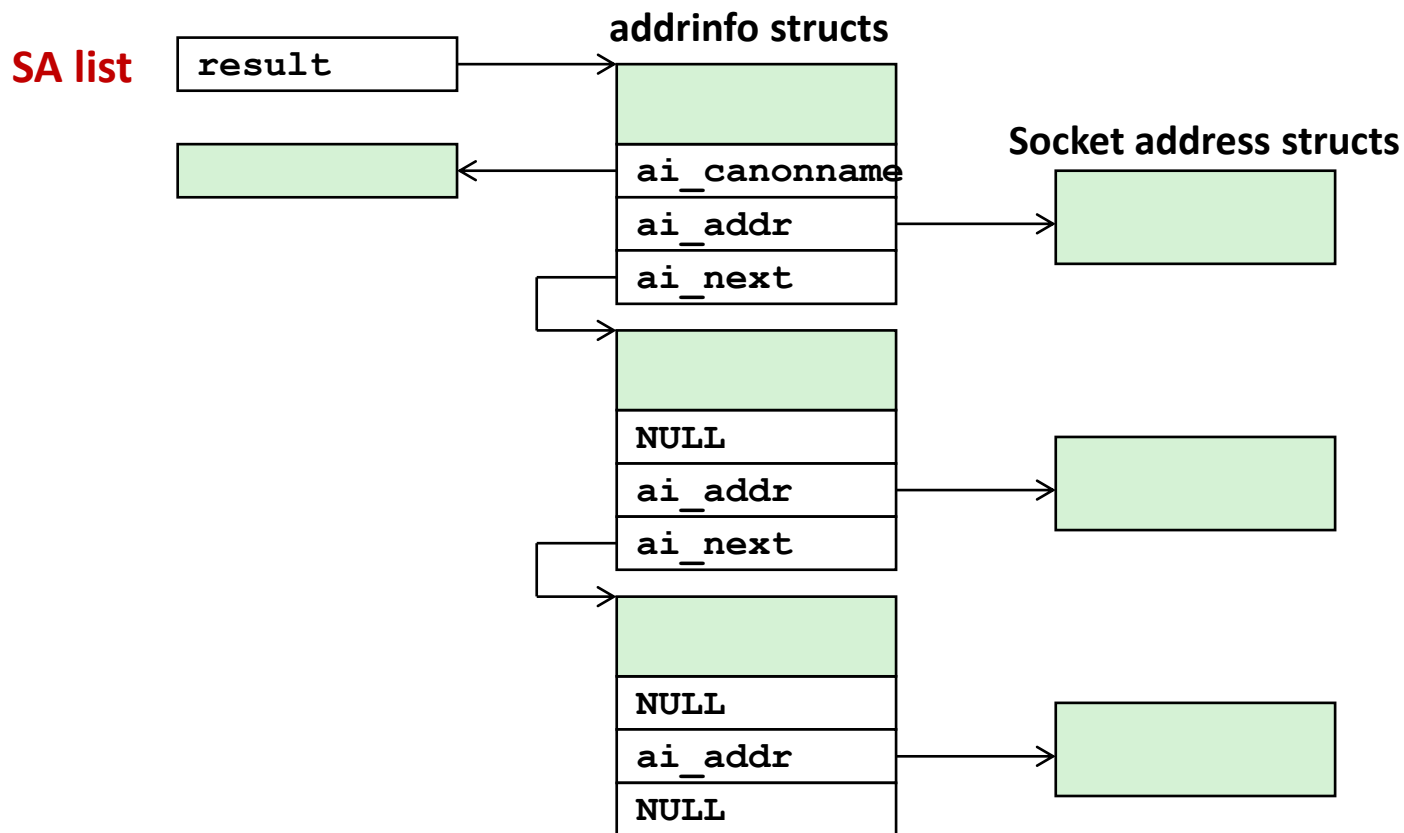
struct sockaddr_in {
    uint16_t      sin_family; /* Protocol family (always AF_INET) */
    uint16_t      sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};

```

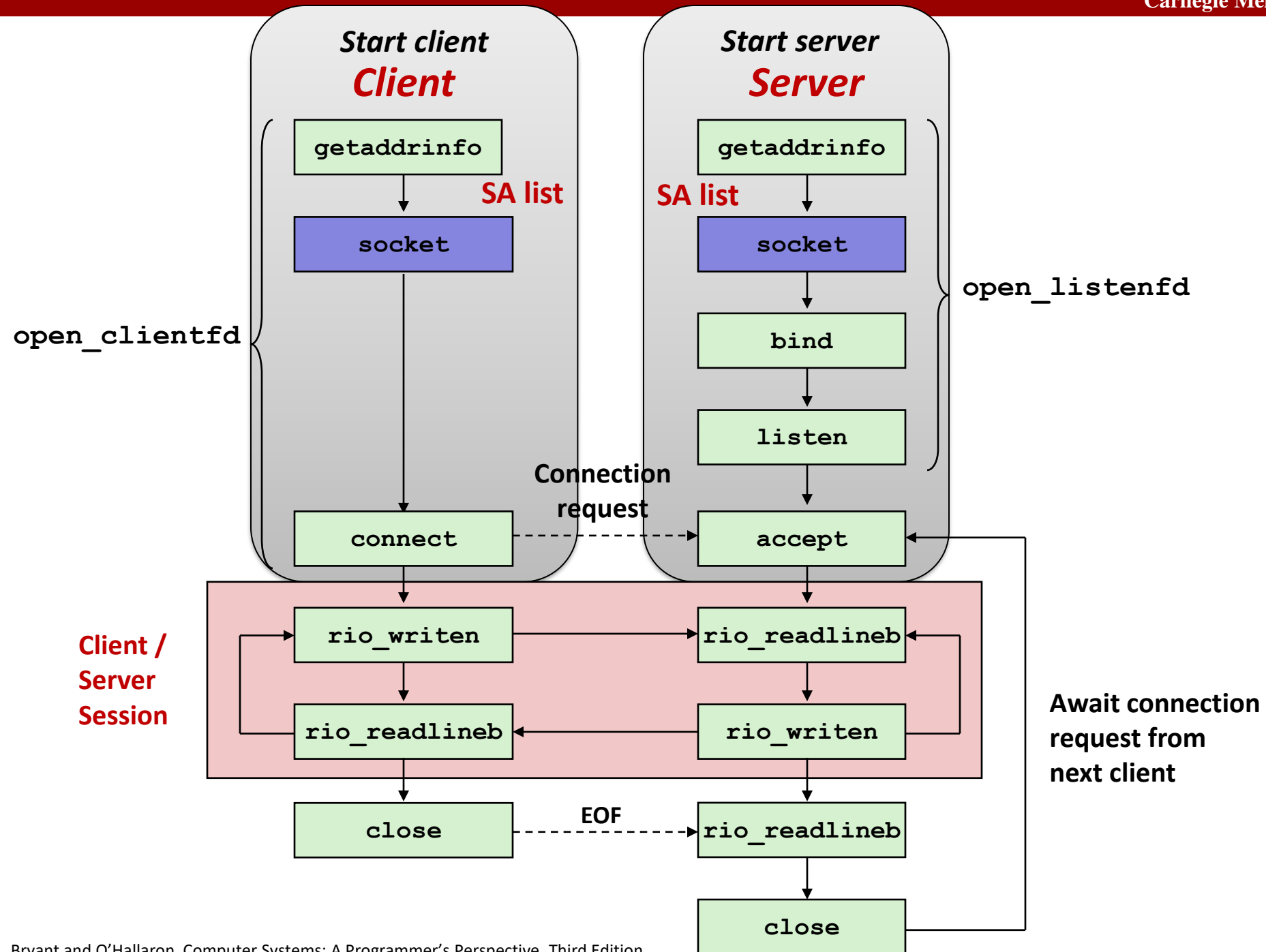


# Review: getaddrinfo

- `getaddrinfo` converts string representations of hostnames, host addresses, ports, service names to socket address structures







# Sockets Interface: `socket`

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

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

- **Example:**

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

*Protocol specific!*

Indicates that we are using  
32-bit IPV4 addresses

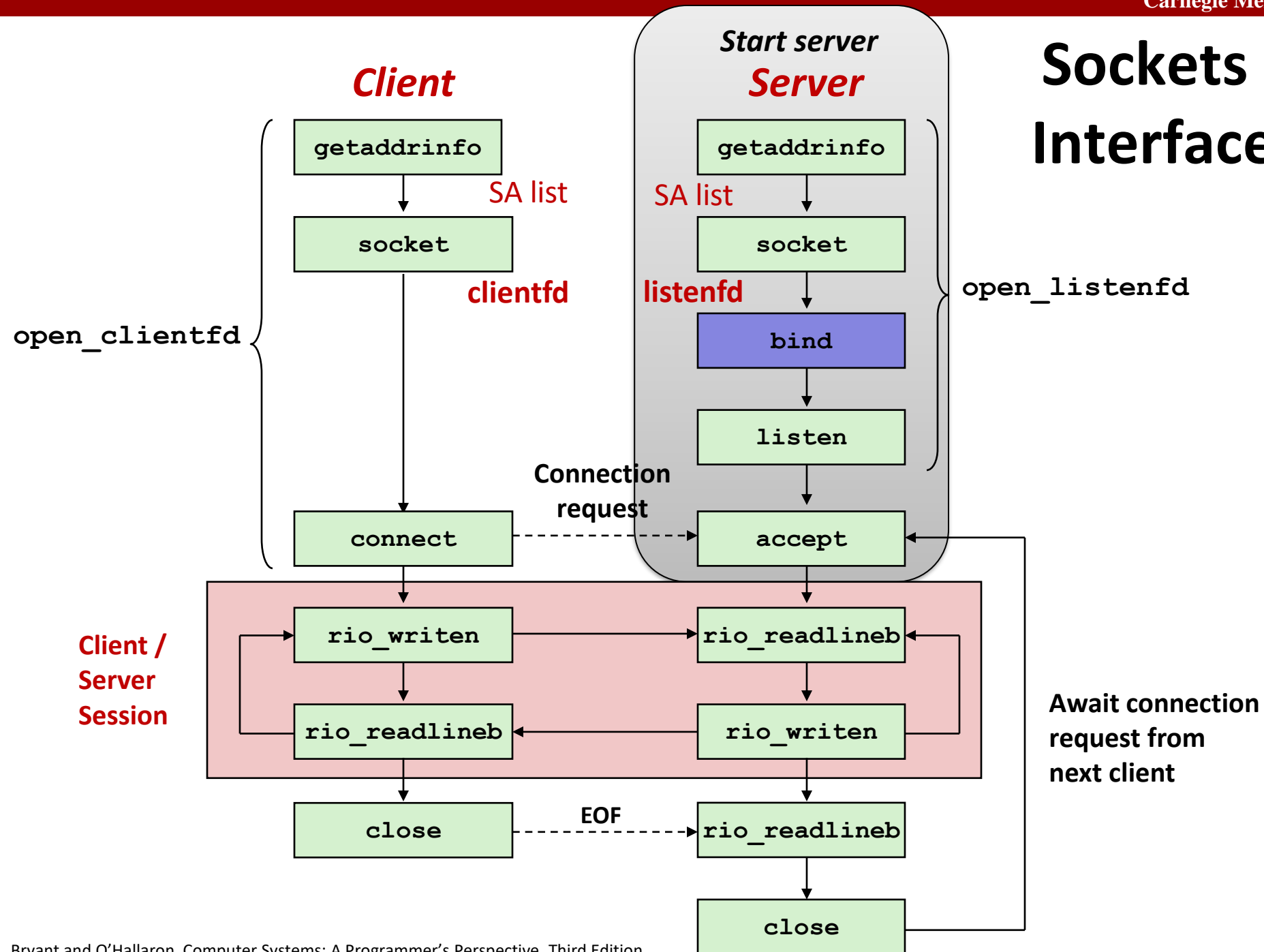
Indicates that the socket  
will be the end point of a  
reliable (TCP) connection

- **Example:**

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

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

# Sockets Interface



# Sockets Interface: `bind`

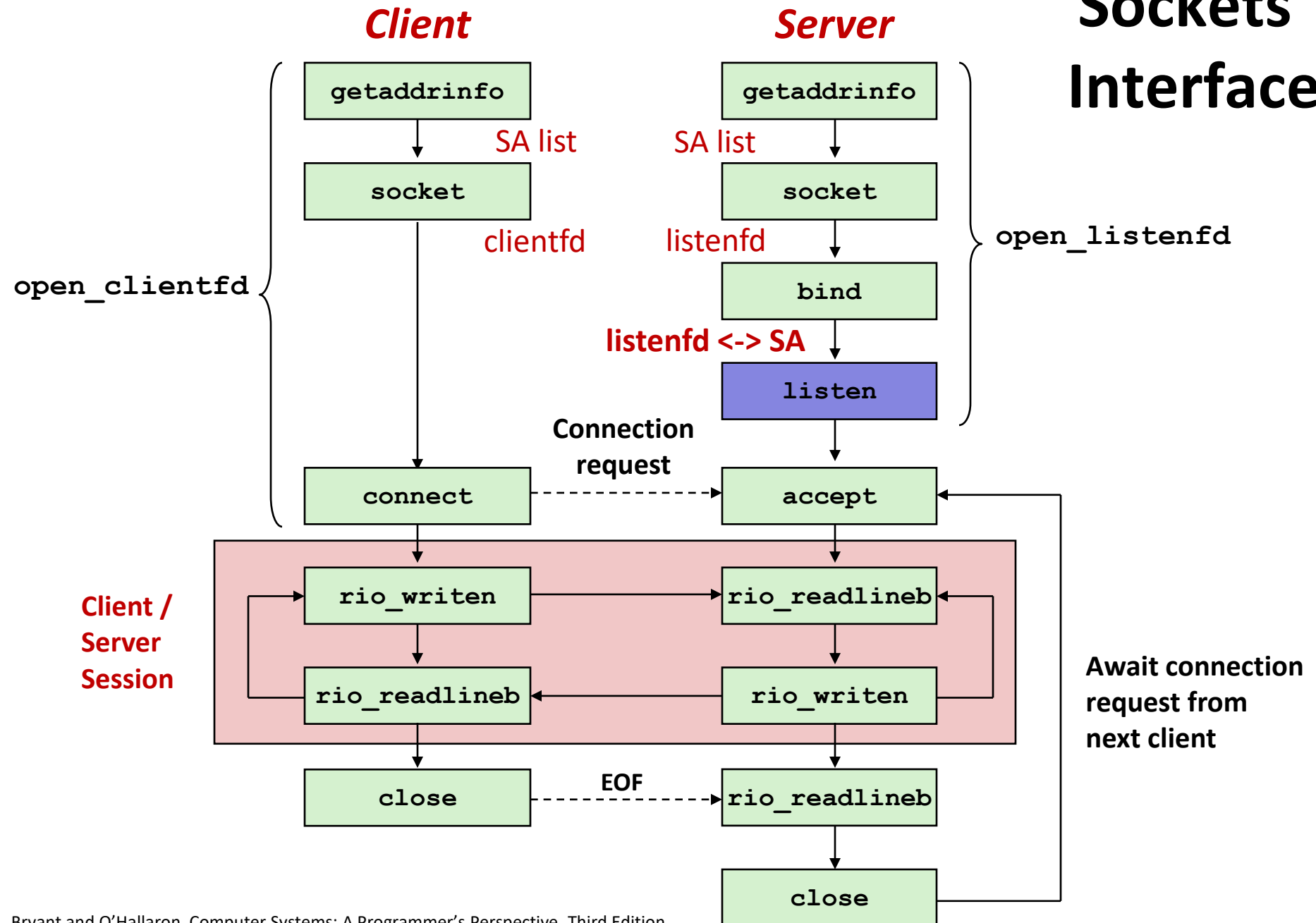
- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

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

Our convention: `typedef struct sockaddr SA;`

- Process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`
- Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

# Sockets Interface



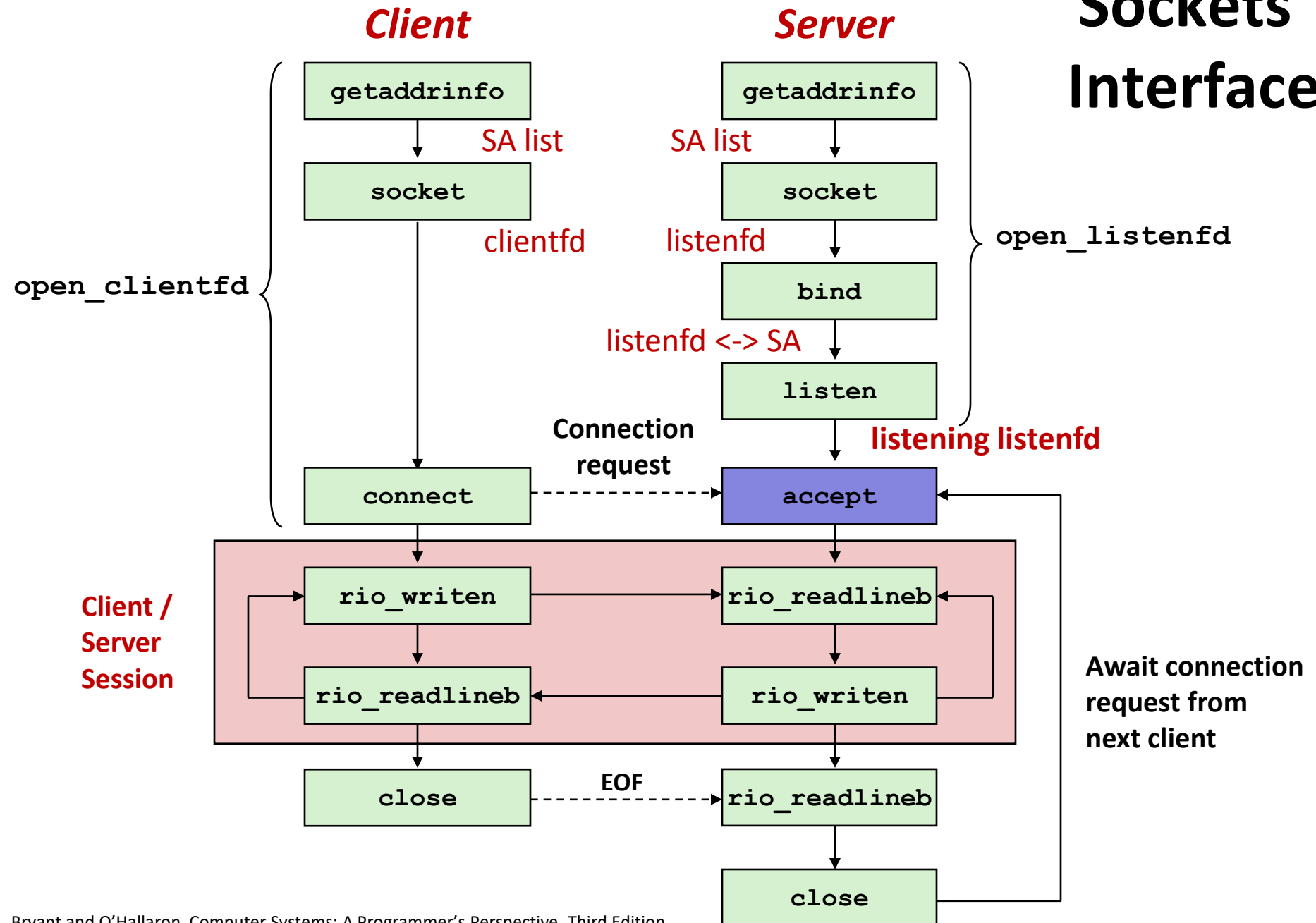
# Sockets Interface: `listen`

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

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

- Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.
- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests (128-ish by default)

# Sockets Interface



# Sockets Interface: `accept`

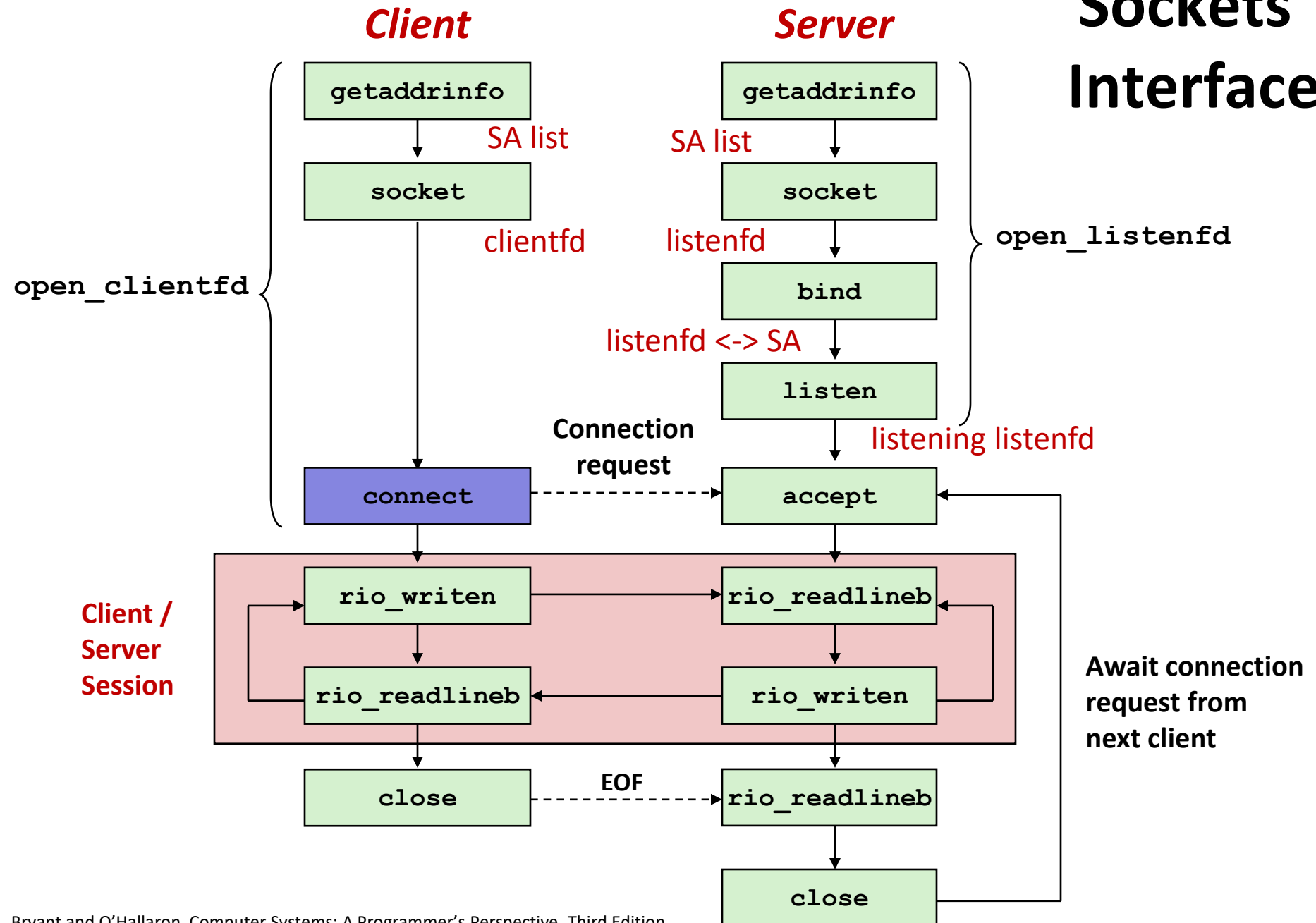
- Servers wait for connection requests from clients by calling `accept`:

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

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.
- Returns a ***connected descriptor*** `connfd` that can be used to communicate with the client via Unix I/O routines.



# Sockets Interface



# Sockets Interface: connect

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

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

- Attempts to establish a connection with server at socket address **addr**

- If successful, then **sockfd** is now ready for reading and writing.

- Resulting connection is characterized by socket pair

(**x:y**, **addr.sin\_addr:addr.sin\_port**)

- **x** is client address

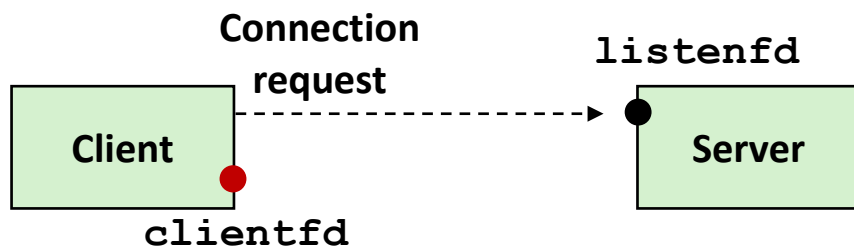
- **y** is ephemeral port that uniquely identifies client process on client host

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

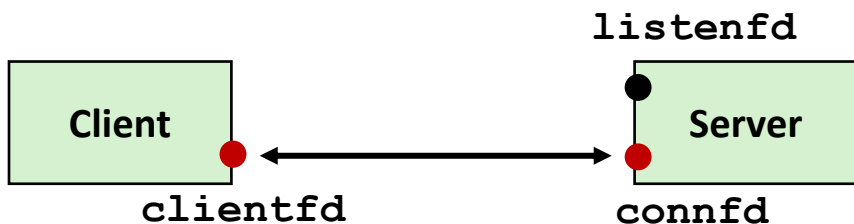
# connect/accept Illustrated



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



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



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

# Connected vs. Listening Descriptors

## ■ Listening descriptor

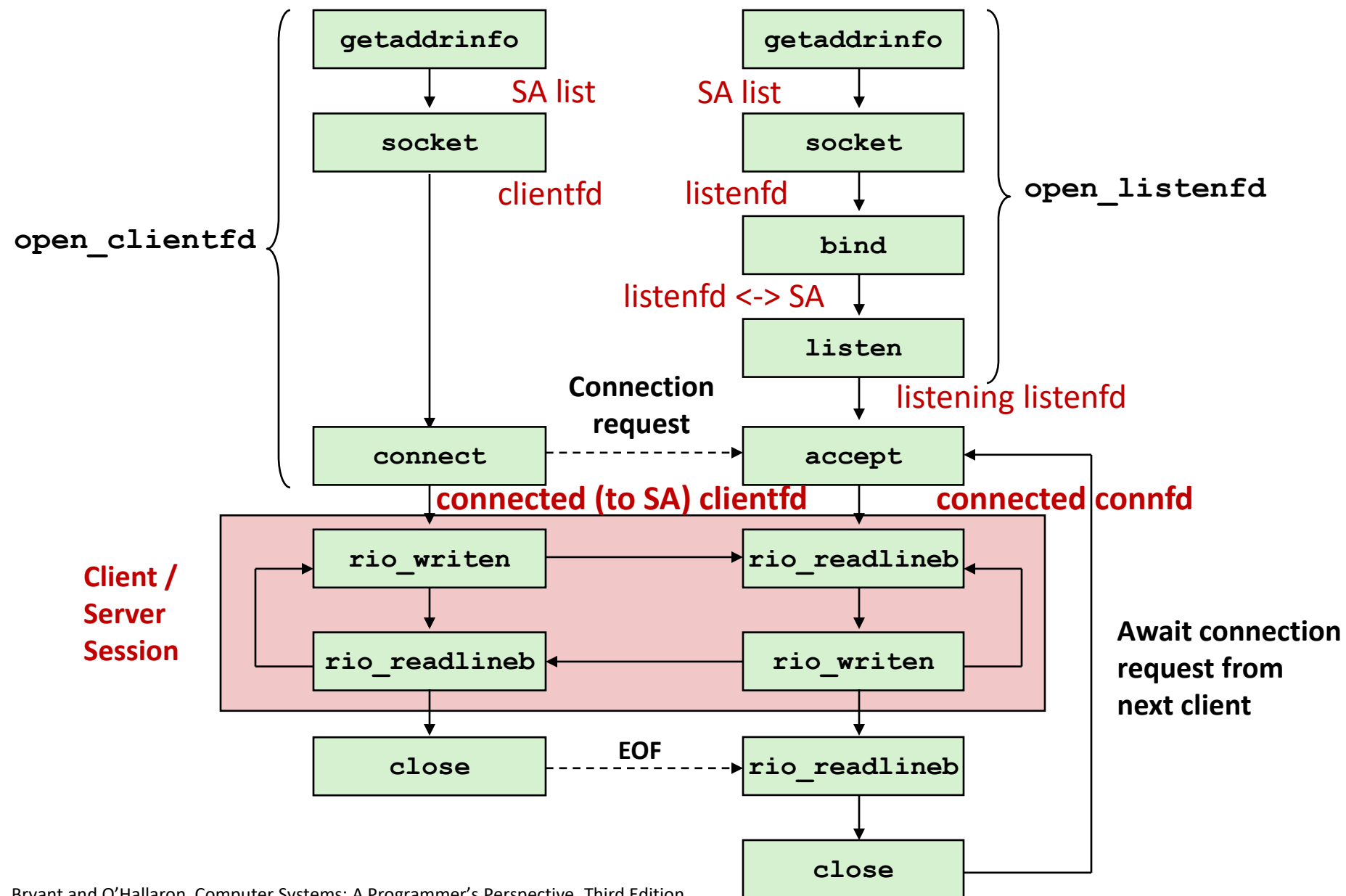
- End point for client connection requests
- Created once and exists for lifetime of the server

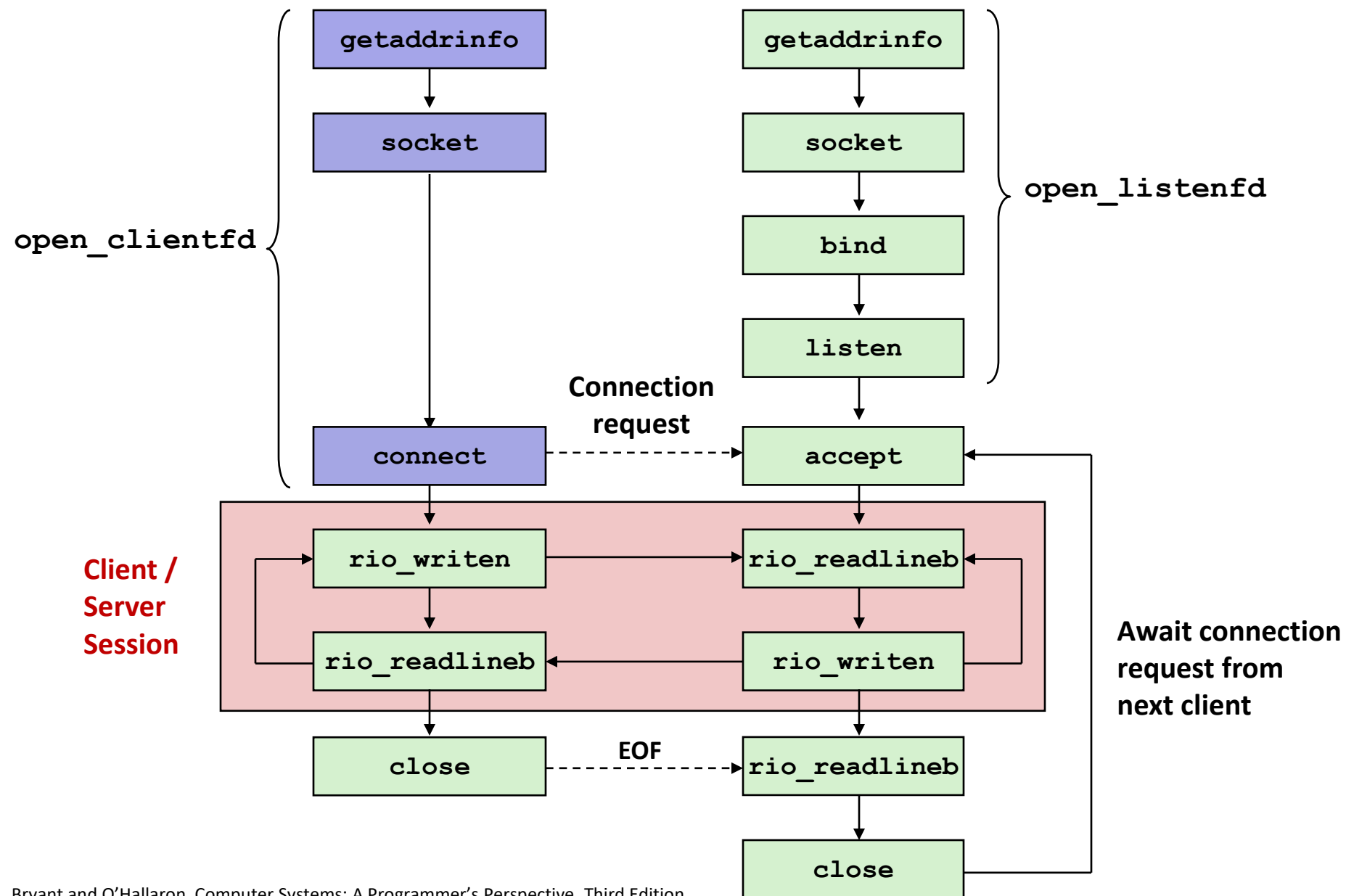
## ■ Connected descriptor

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

## ■ Why the distinction?

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

**Client****Server**

**Client****Server**

# Sockets Helper: `open_clientfd`

- Establish a connection with a server

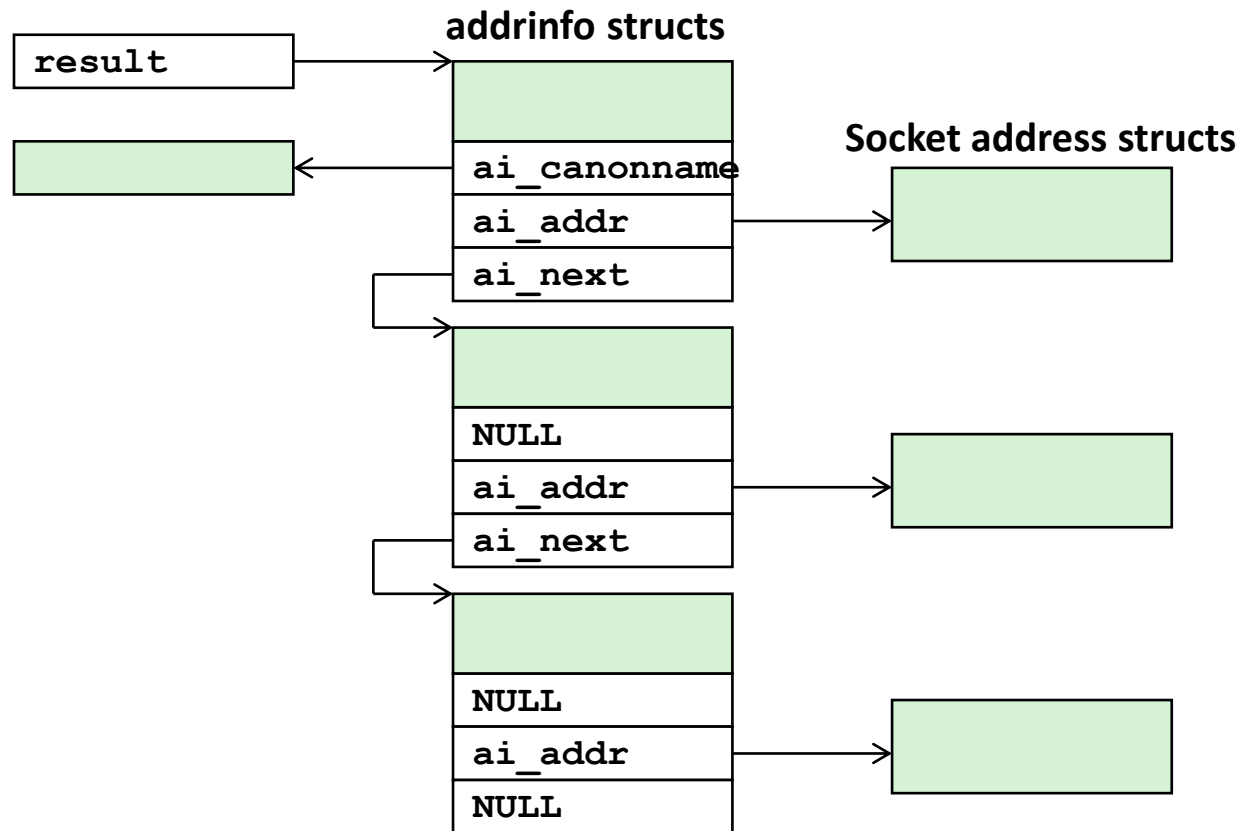
```
int open_clientfd(char *hostname, char *port) {
    int clientfd;
    struct addrinfo hints, *listp, *p;

    /* 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

**AI\_ADDRCONFIG** means “use whichever of IPv4 and IPv6 works on this computer”. Good practice for clients, not for servers.

# getaddrinfo



- **Clients:** walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- **Servers:** walk the list calling `socket`, `listen`, `bind` for *all* addresses, then use `select` to accept connections on any of them (beyond our scope)



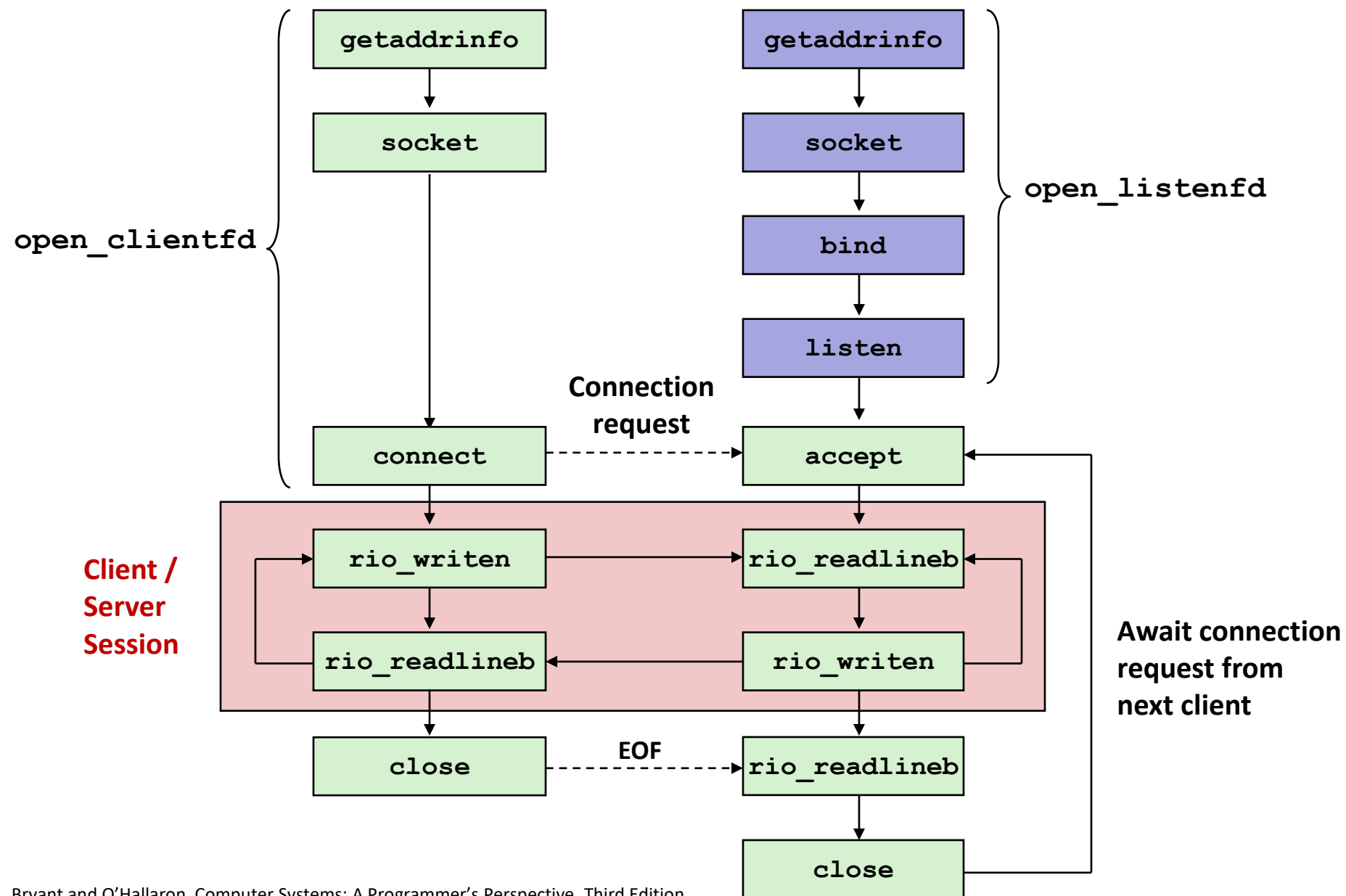
# Sockets Helper: `open_clientfd` (cont)

```
/* 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

**Client****Server**

# 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

`AI_PASSIVE` means “I plan to listen on this socket.”

`AI_ADDRCONFIG` normally not used for servers, but we use it for convenience

# 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

A production server would not break out of the loop on the first success.  
We do that for simplicity only.

# 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.

# 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

```
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>
```

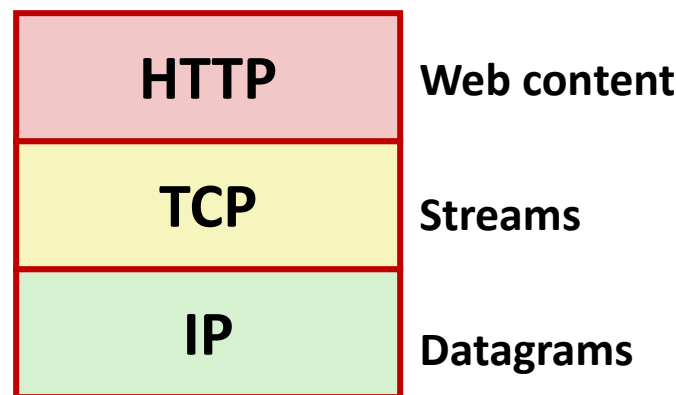
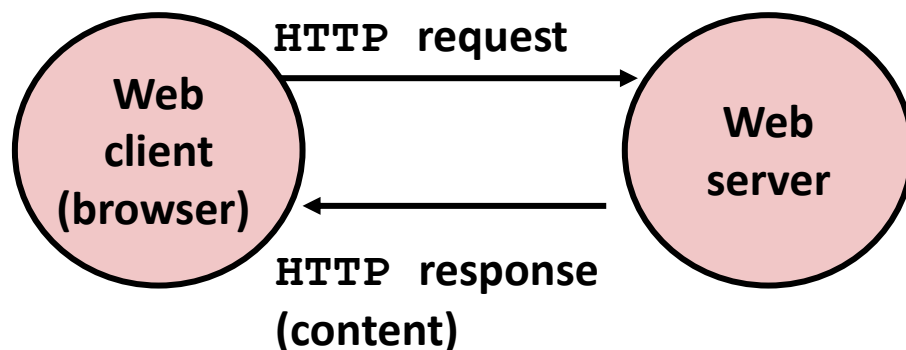
# Today

- Setting up connections
- **Application protocol example: HTTP**



# Web Server Basics

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



<http://www.w3.org/Protocols/rfc2616/rfc2616.html>

# Web Content

## ■ Web servers return *content* to clients

- *content*: a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

## ■ Example MIME types

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

You can find the complete list of MIME types at:

<http://www.iana.org/assignments/media-types/media-types.xhtml>

# Static and Dynamic Content

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

# URLs and how clients and servers use them

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

# HTTP Request Example

```
GET / HTTP/1.1  
Host: www.cmu.edu
```

Client: request line

Client: required HTTP/1.1 header

Client: blank line terminates headers

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

# HTTP Requests

- HTTP request is a *request line*, followed by zero or more *request headers*
- Request line: `<method> <uri> <version>`
  - `<method>` is one of GET, POST, OPTIONS, HEAD, PUT, DELETE, or TRACE
  - `<uri>` is typically URL for proxies, URL suffix for servers
    - A URL is a type of URI (Uniform Resource Identifier)
    - See <http://www.ietf.org/rfc/rfc2396.txt>
  - `<version>` is HTTP version of request (HTTP/1.0 or HTTP/1.1)
- Request headers: `<header name>: <header data>`
  - Provide additional information to the server

# HTTP Responses

- HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line (“\r\n”) separating headers from content.
- Response line:
  - `<version> <status code> <status msg>`
    - `<version>` is HTTP version of the response
    - `<status code>` is numeric status
    - `<status msg>` is corresponding English text
      - **200 OK** Request was handled without error
      - **301 Moved** Provide alternate URL
      - **404 Not found** Server couldn't find the file
- Response headers: `<header name>: <header data>`
  - Provide additional information about response
  - **Content-Type**: MIME type of content in response body
  - **Content-Length**: Length of content in response body

# Example HTTP Transaction

whaleshark> telnet www.cmu.edu 80	Client: open connection to server
Trying 128.2.42.52...	Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.	
Escape character is '^]'. <b>GET / HTTP/1.1</b>	Client: request line
Host: www.cmu.edu	Client: required HTTP/1.1 header
	Client: blank line terminates headers
HTTP/1.1 <b>301 Moved Permanently</b>	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: <b>http://www.cmu.edu/index.shtml</b>	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
Trying 128.2.42.52...
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1
Host: www.cmu.edu

HTTP/1.1 200 OK
Date: Wed, 05 Nov 2014 17:37:26 GMT
Server: Apache/1.3.42 (Unix)
Transfer-Encoding: chunked
Content-Type: text/html; charset=...

1000
<html ..>
...
</html>
0
Connection closed by foreign host.

```

Client: open connection to server  
 Telnet prints 3 lines to terminal

Client: request line  
 Client: required HTTP/1.1 header  
 Client: blank line terminates headers  
 Server: response line  
 Server: followed by 4 response headers

Server: empty line terminates headers  
 Server: begin response body  
 Server: first line of HTML content

Server: end response body  
 Server: close connection

# Example HTTP(S) Transaction, Take 3

```
whaleshark> openssl s_client www.cs.cmu.edu:443
CONNECTED(00000005)
...
Certificate chain
...
-
Server certificate
-----BEGIN CERTIFICATE-----
MIIGDjCCBPagAwIBAgIRAMiF7LBPDoySilnNoU+mp+gwDQYJKoZIhvcNAQELBQAw
djELMAkGA1UEBhMCVVMxCzAJBgNVBAGTAk1JMRIwEAYDVQQHEw1Bbm4gQXJib3Iu
EjAQBgNVBAoTCUluZGVybmV0MjERMA8GA1UECzMISW5Db21tb24xHzAdBgNVBAMT
wkWkvDVBBCwKXrShVxQNsJ6J
...
-----END CERTIFICATE-----
subject=/C=US/postalCode=15213/ST=PA/L=Pittsburgh/street=5000 Forbes
Ave/O=Carnegie Mellon University/OU=School of Computer
Science/CN=www.cs.cmu.edu issuer=/C=US/ST=MI/L=Ann
Arbor/O=Internet2/OU=InCommon/CN=InCommon RSA Server CA
SSL handshake has read 6274 bytes and written 483 bytes
...
>GET / HTTP/1.0

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