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

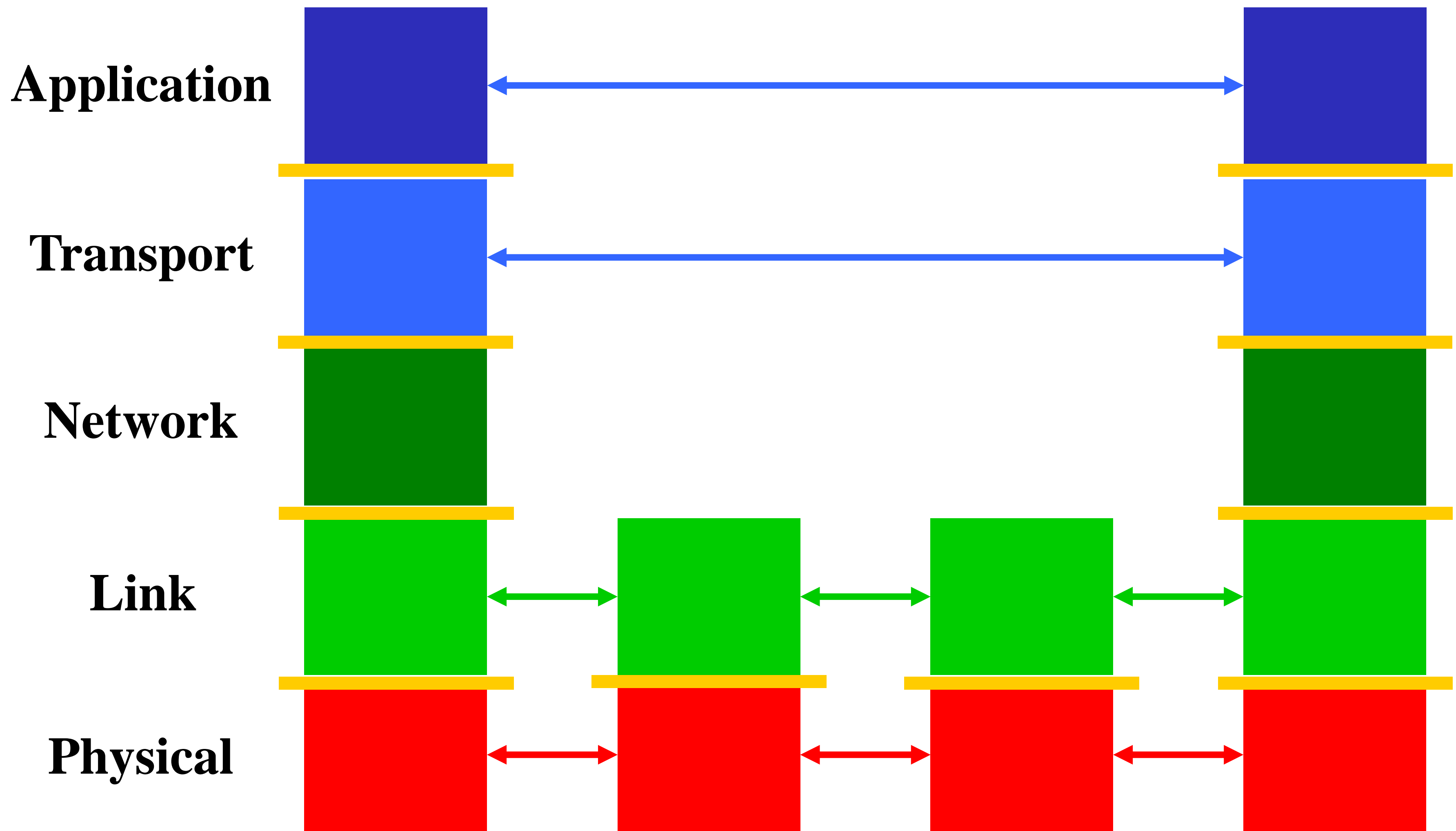
Network Programming: Part II

18-213/18-613:
Introduction to Computer Systems
21st Lecture, March 31, 2022

Today

- **Network Layers: Birds Eye View**
- The Sockets Interface — CSAPP 11.4
- Web Servers — CSAPP 11.5.1-11.5.3
- The Tiny Web Server — CSAPP 11.6
- Serving Dynamic Content — CSAPP 11.5.4
- Proxy Servers

Protocol and Service Levels



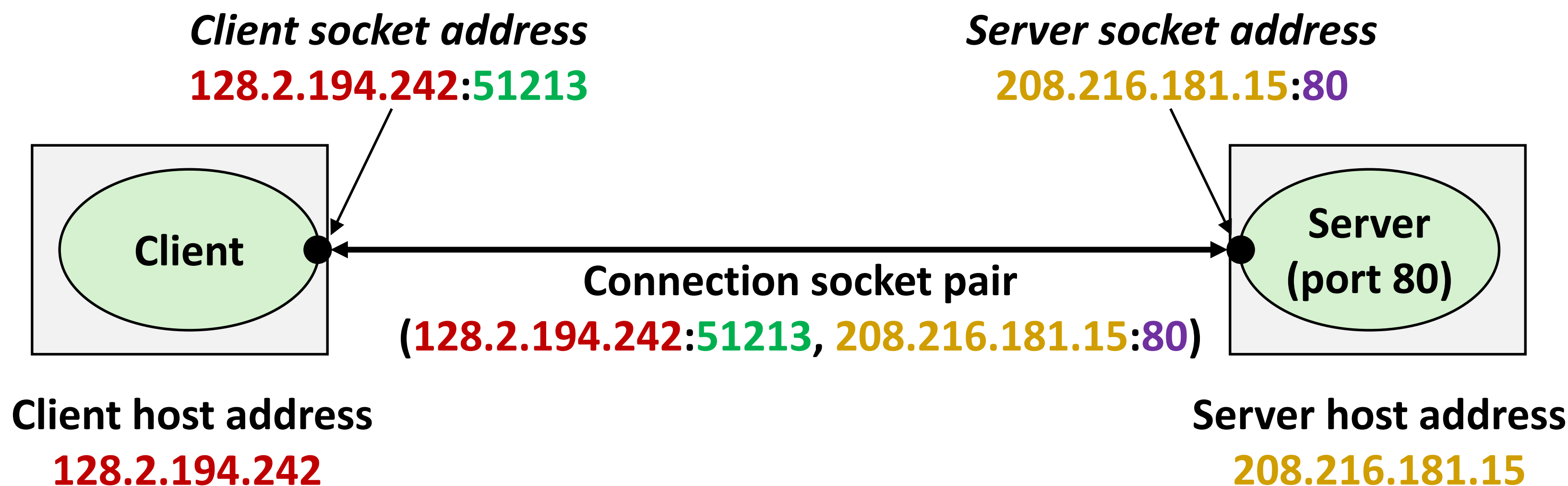
Layering: modular approach to network functionality

Today

- Network Layers: Birds Eye View
- **The Sockets Interface** — **CSAPP 11.4**
- Web Servers — CSAPP 11.5.1-11.5.3
- The Tiny Web Server — CSAPP 11.6
- Serving Dynamic Content — CSAPP 11.5.4
- Proxy Servers

Recall: Anatomy of a Connection

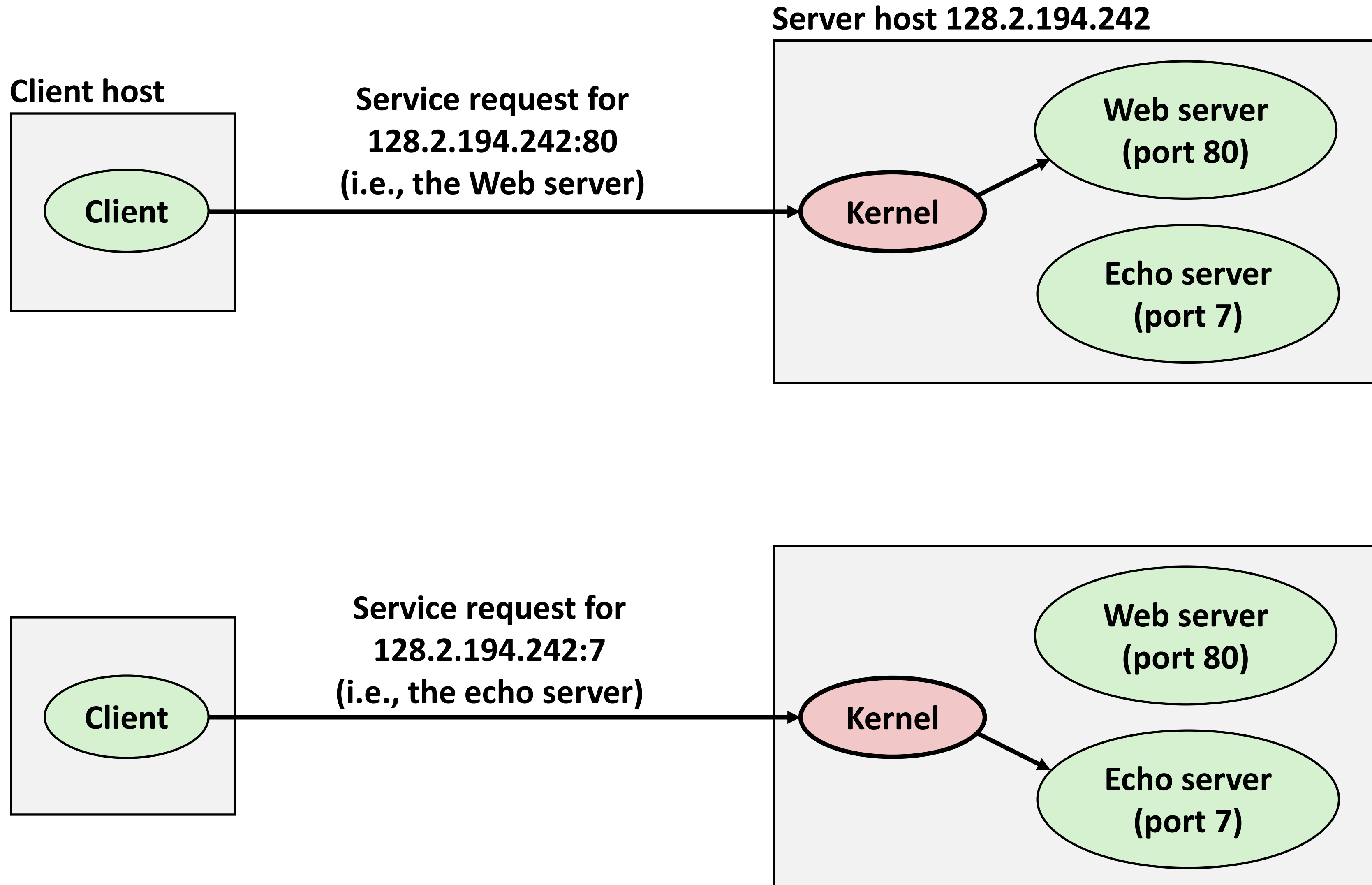
- A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)
  - `(cliaddr:cliport, servaddr:servport)`



51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

Recall: Using Ports to Identify Services



Sockets Interface

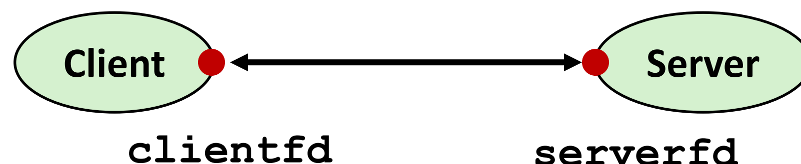
- **Set of system-level functions used in conjunction with Unix I/O to build network applications.**

- **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**

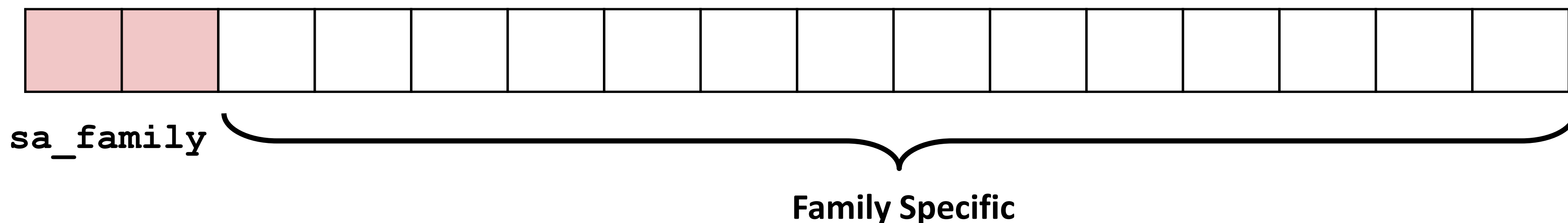


- **The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors**

Representing a socket: Generic Socket Address

- **Generic socket address:**
  - For address arguments to **`connect`**, **`bind`**, and **`accept`**

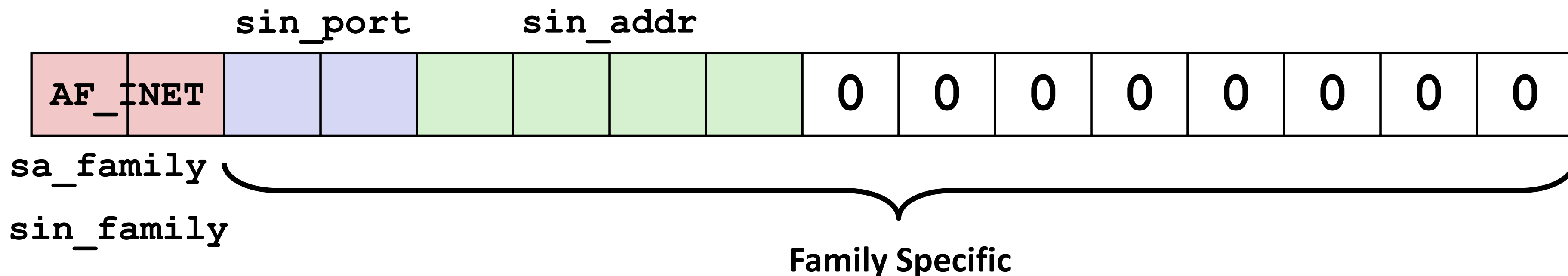
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```



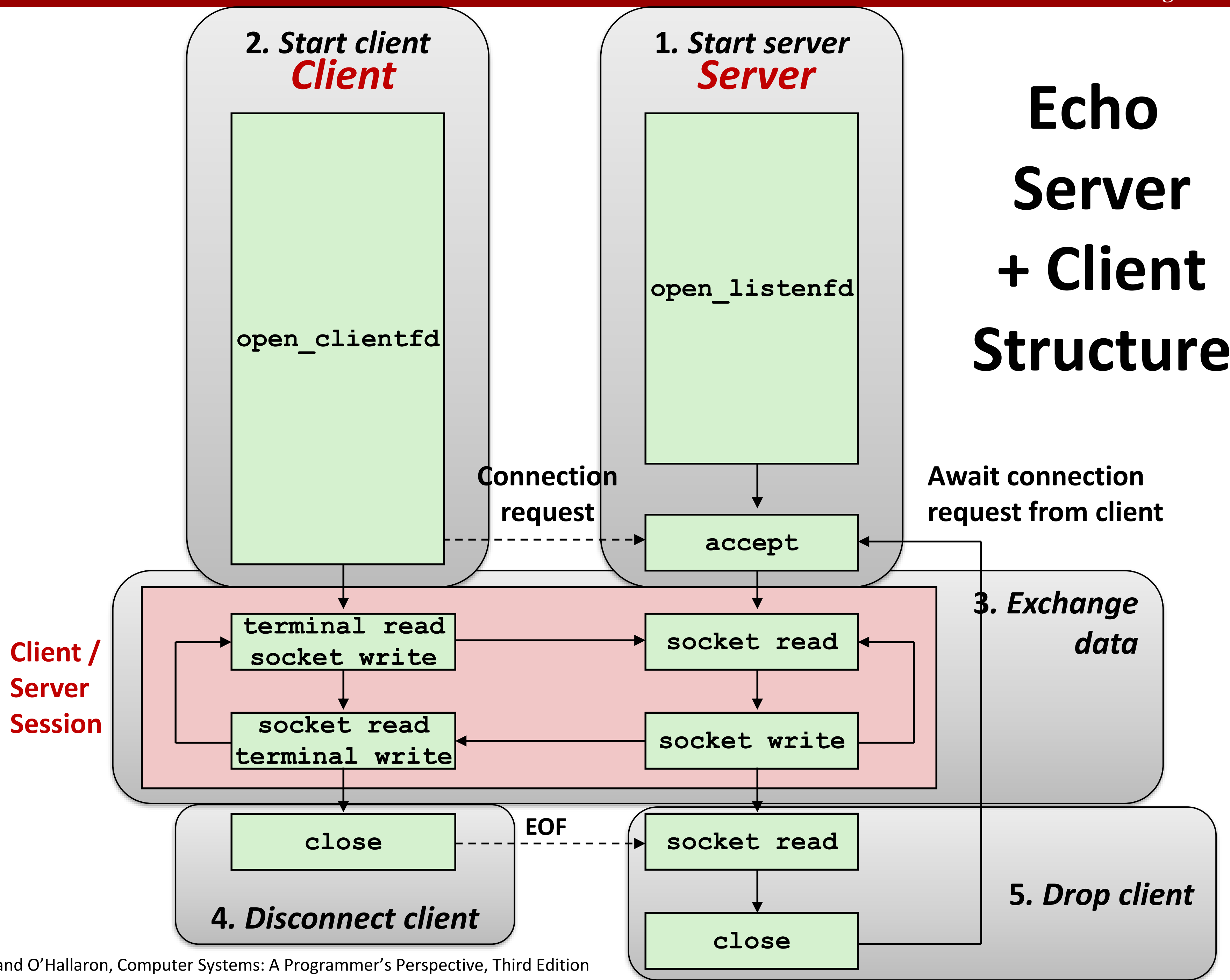
Representing a Socket: Socket Address Structures

- **Internet (IPv4) specific socket address:**
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

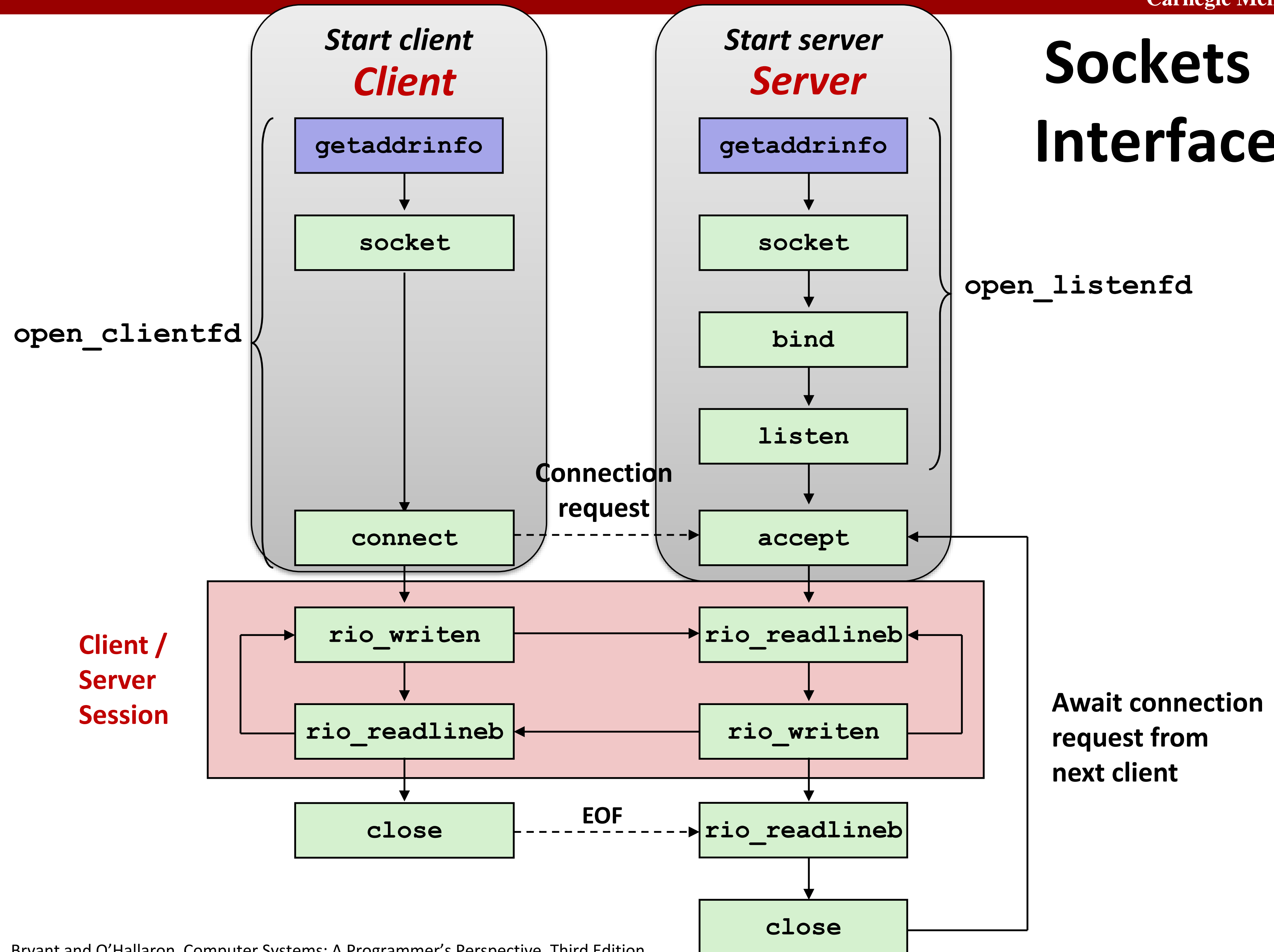
```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};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



Echo Server + Client Structure



Sockets Interface



Host and Service Conversion: `getaddrinfo`

- **`getaddrinfo` is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.**
  - Replaces obsolete `gethostbyname` and `getservbyname` funcs.

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

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

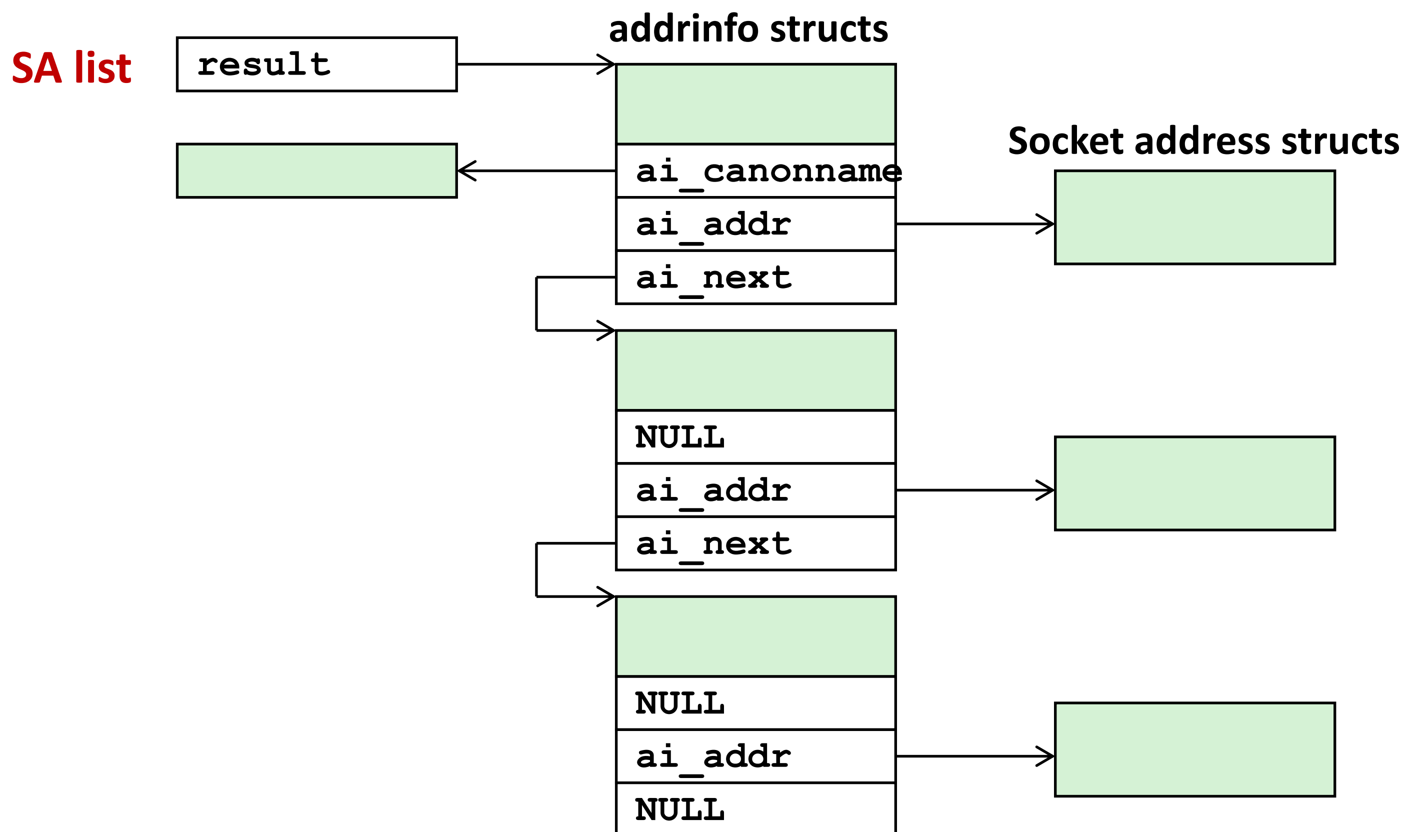
Host and Service Conversion: getaddrinfo

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

void freeaddrinfo(struct addrinfo *result);  /* Free linked list */

const char *gai_strerror(int errcode);        /* Return error msg */
```

- **Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.**
- **Helper functions:**
  - `freeadderinfo` frees the entire linked list.
  - `gai_strerror` converts error code to an error message.

getaddrinfo

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

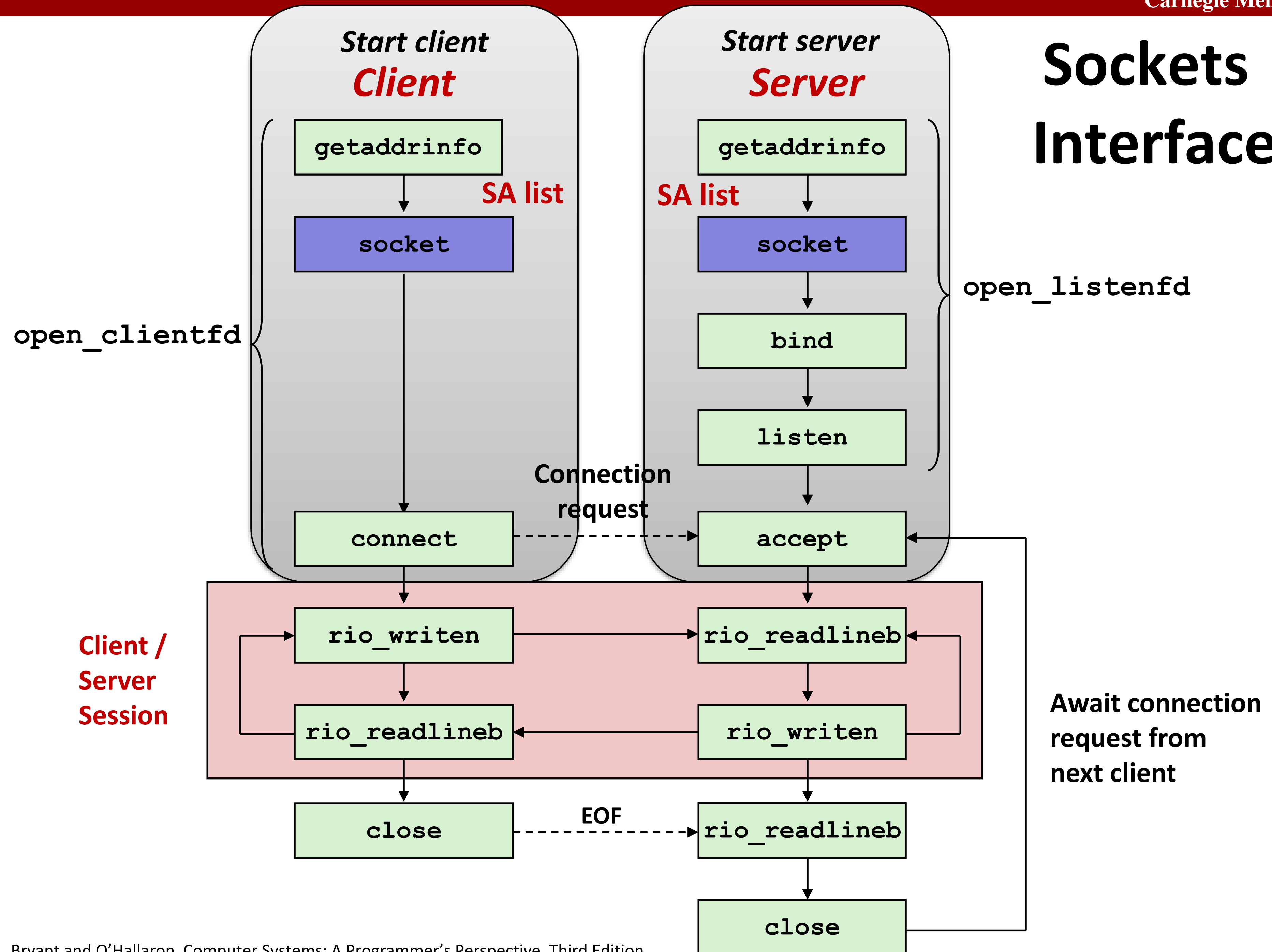


Host and Service Conversion: `getnameinfo`

- **`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 */
```

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:

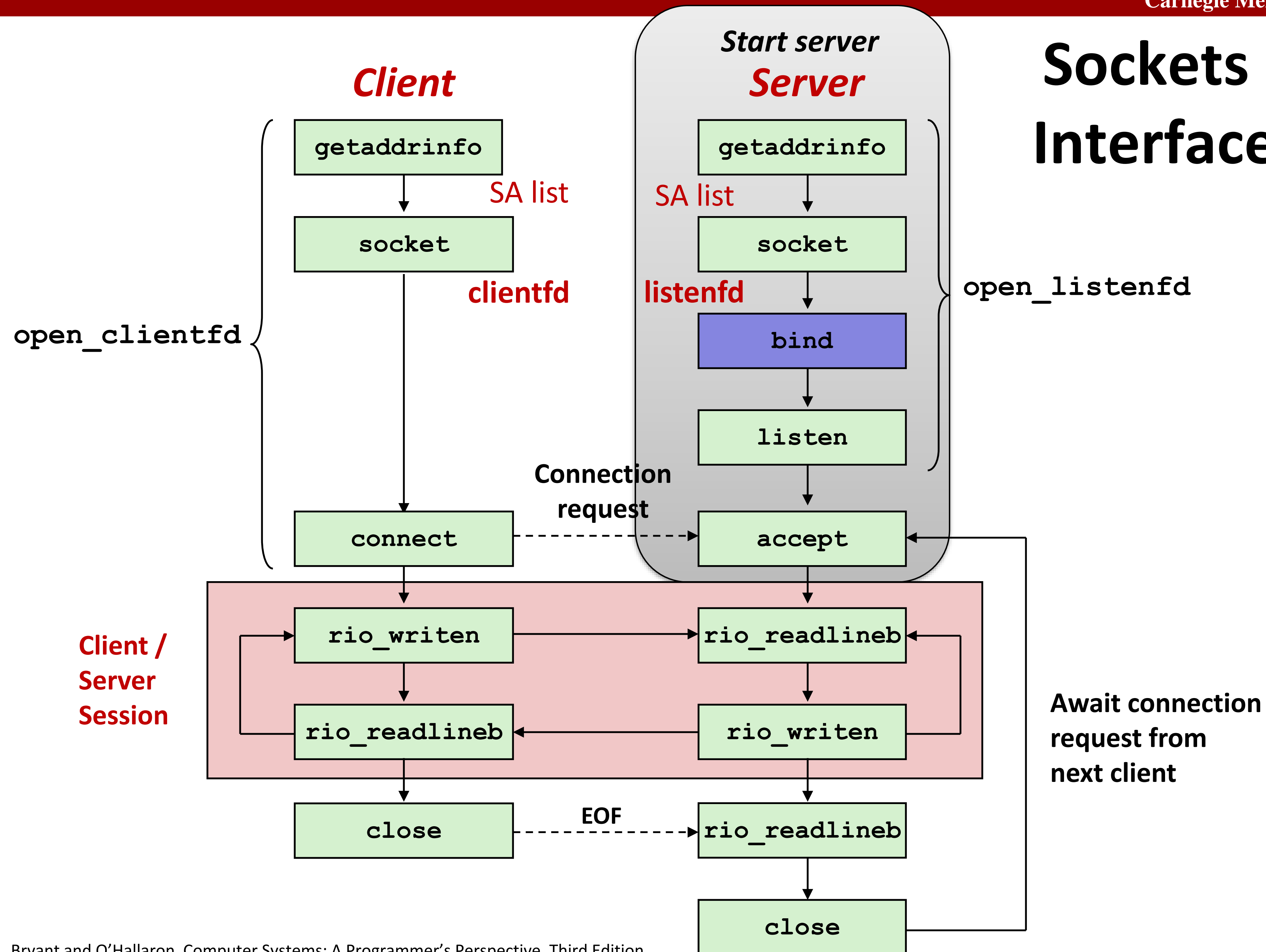
```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

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

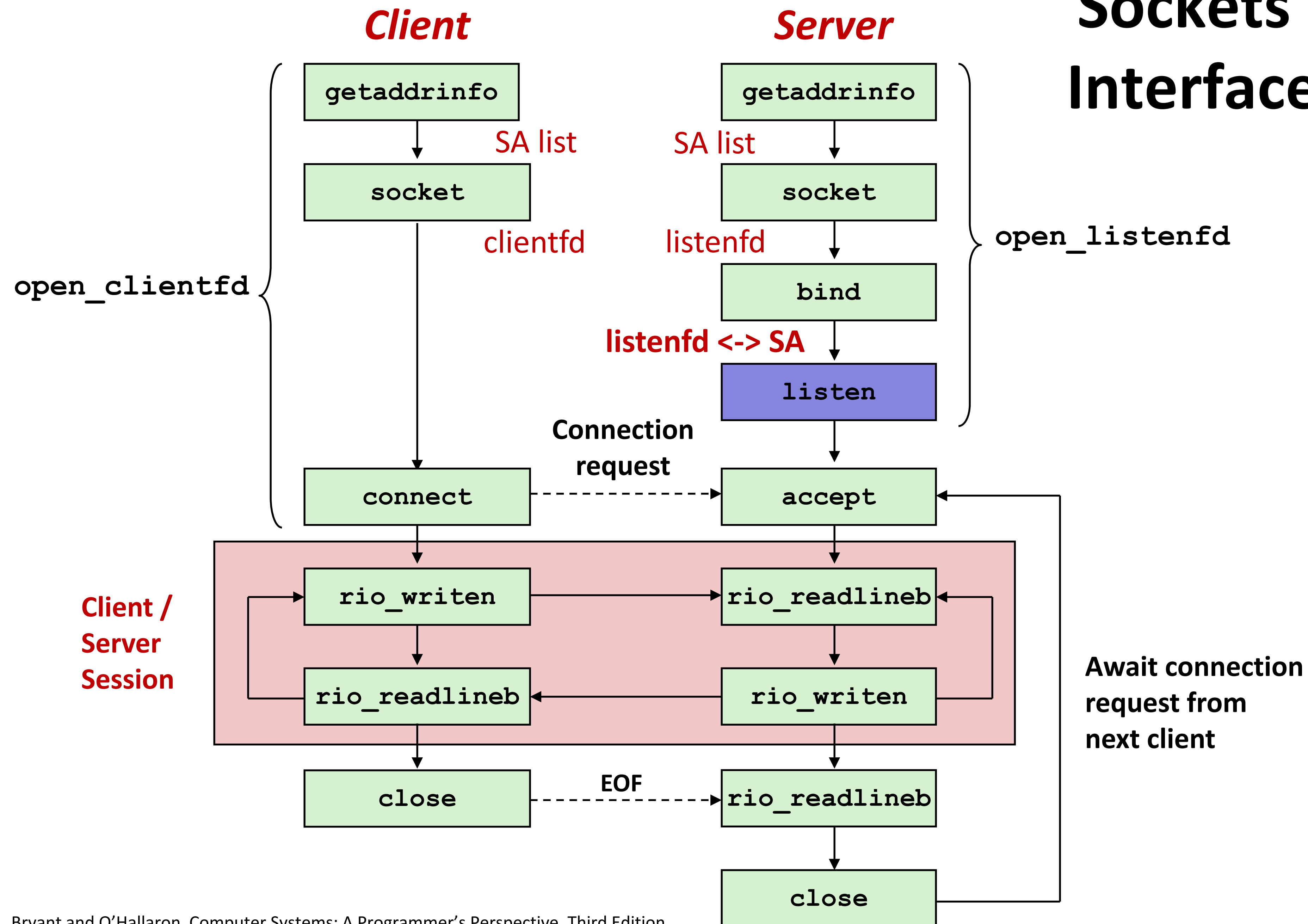
- **A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:**

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

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`**

Sockets Interface



Sockets Interface: `listen`

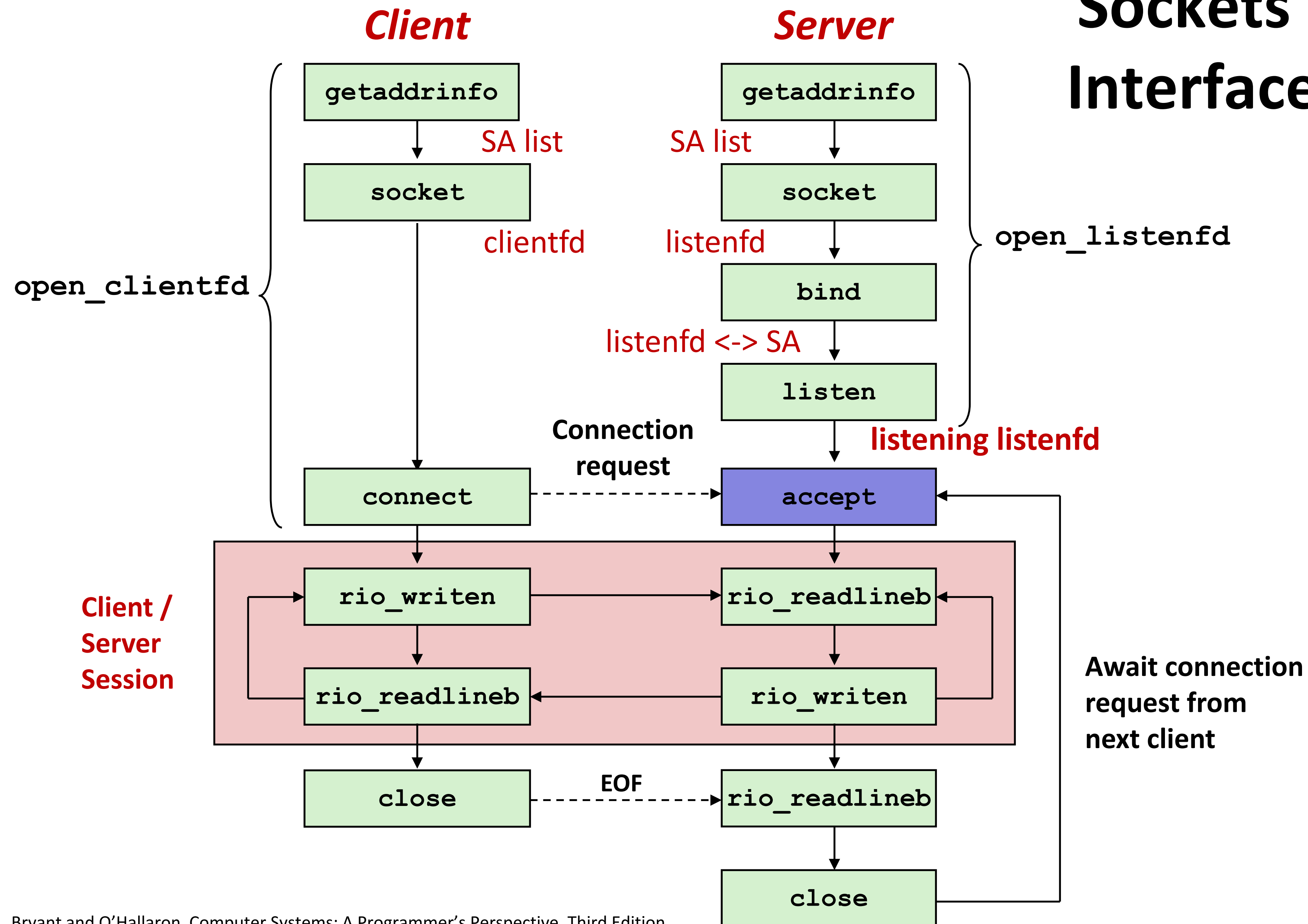
- Kernel assumes that descriptor from socket function is an *active socket* that will be on the client end
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

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

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

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

Sockets Interface



Sockets Interface: `accept`

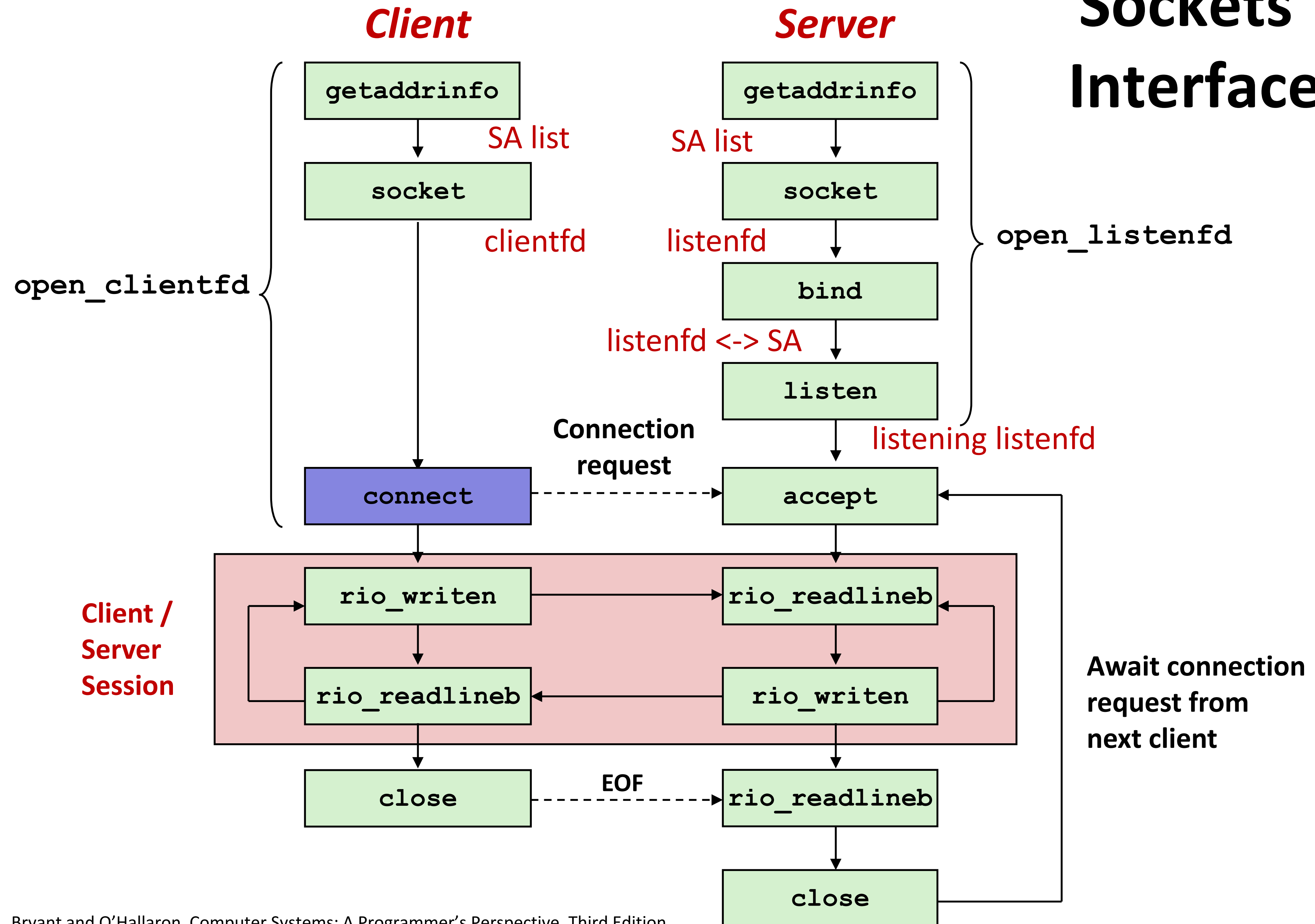
- Servers wait for connection requests from clients by calling `accept`:

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

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

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

Sockets Interface



Sockets Interface: `connect`

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

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

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

    (**`x:y, addr.sin_addr:addr.sin_port`**)
    - **`x`** is client address
    - **`y`** is ephemeral port that uniquely identifies client process on client host

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

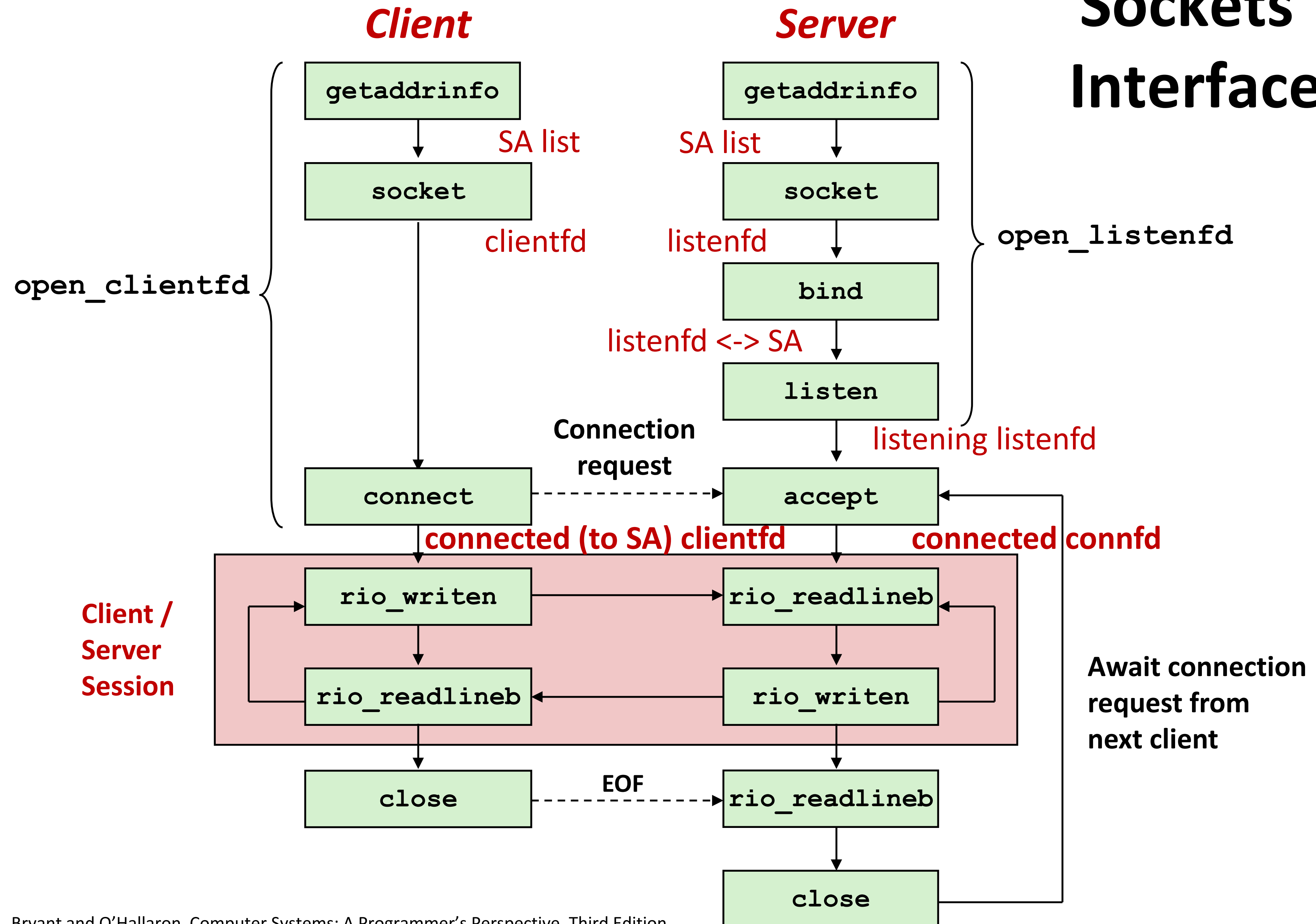
Connected vs. Listening Descriptors

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

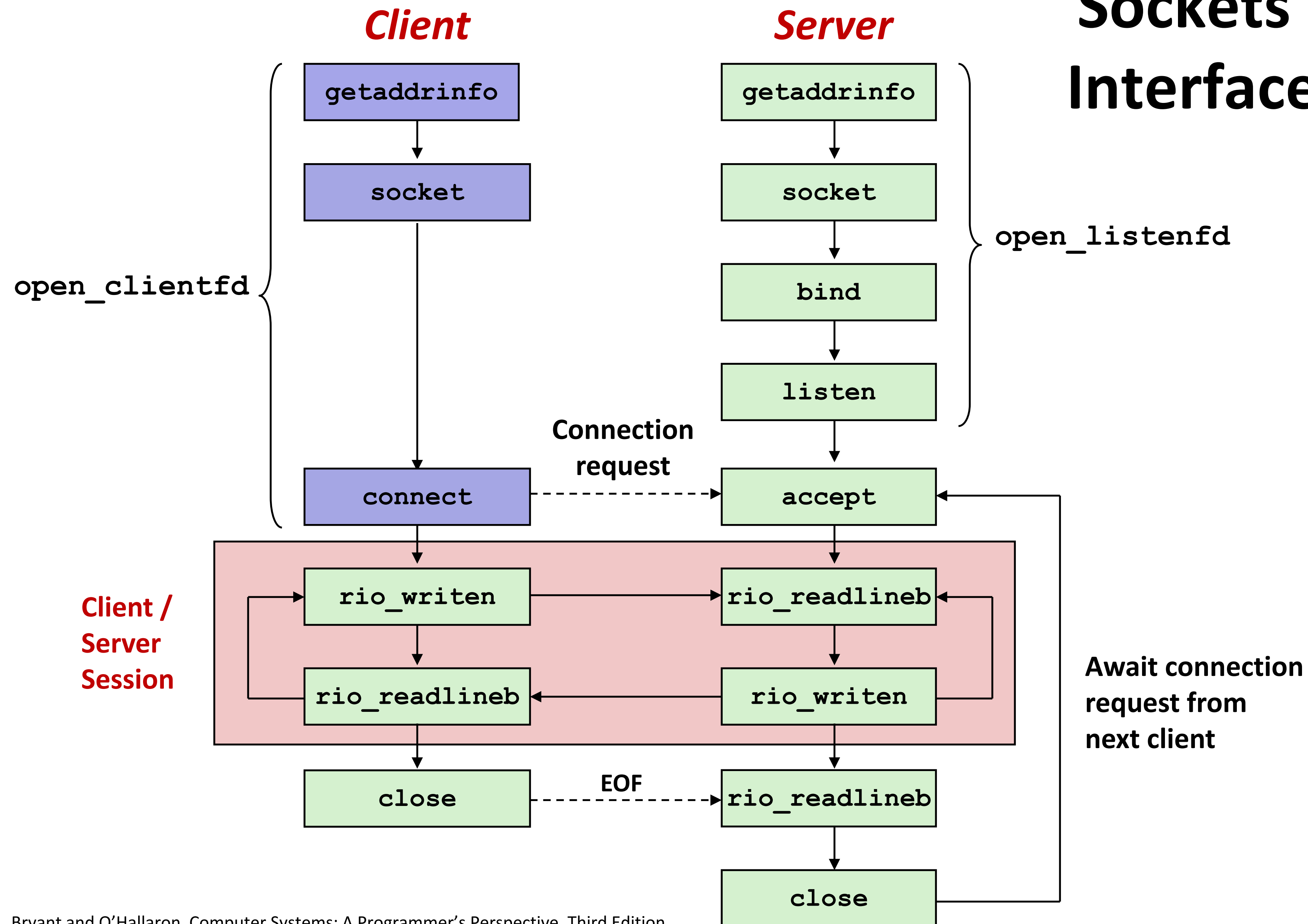
- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

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

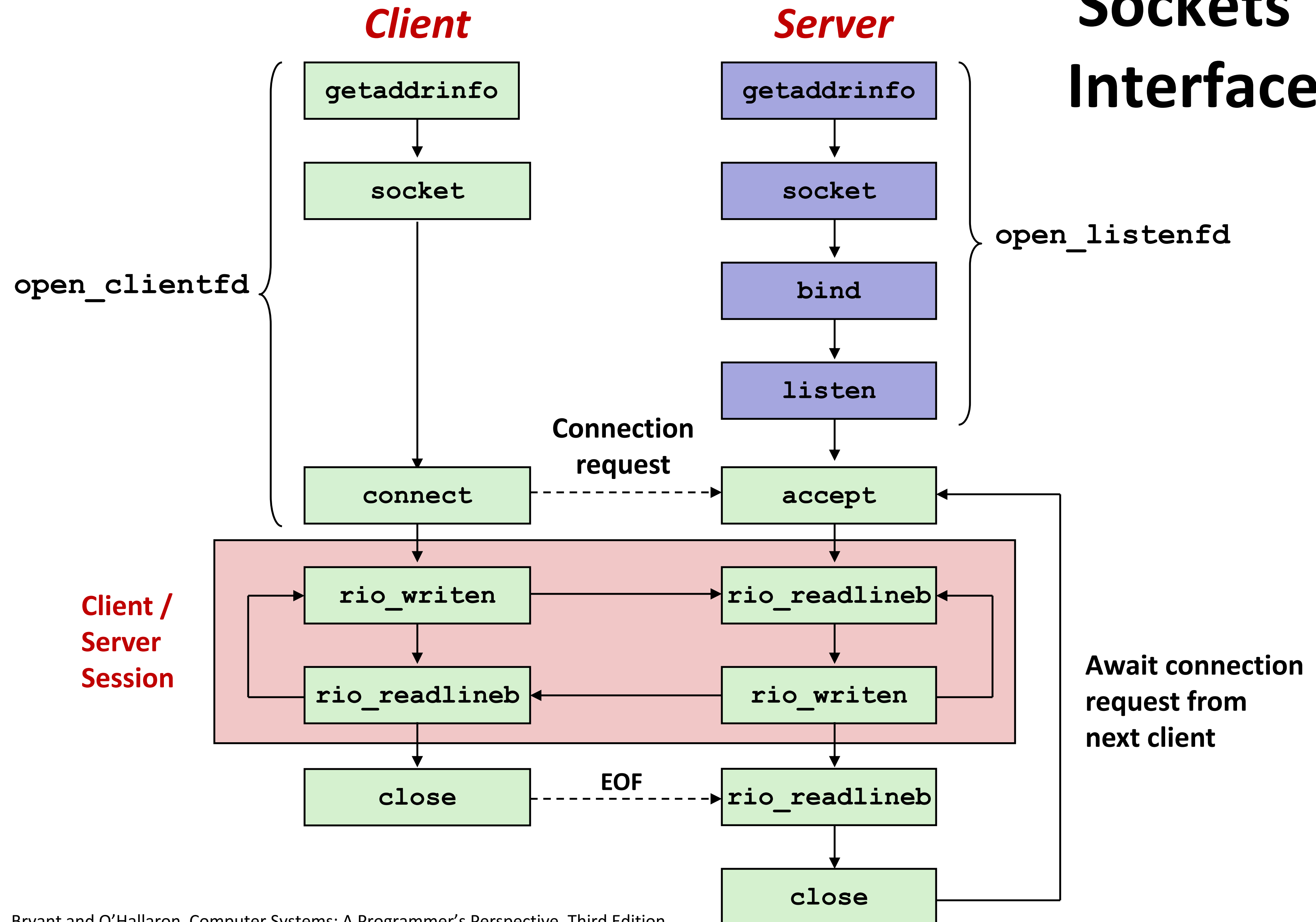
Sockets Interface



Sockets Interface



Sockets Interface



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`

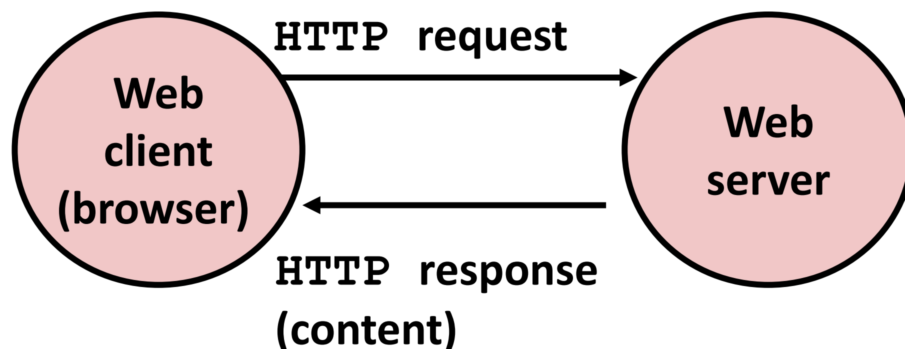
```
whaleshark> ./echoserveri 18213
Connected to (MAKOSHARK.ICS.CS.CMU.EDU, 50280)
server received 11 bytes
server received 8 bytes




makoshark> telnet whaleshark.ics.cs.cmu.edu 18213
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>
```

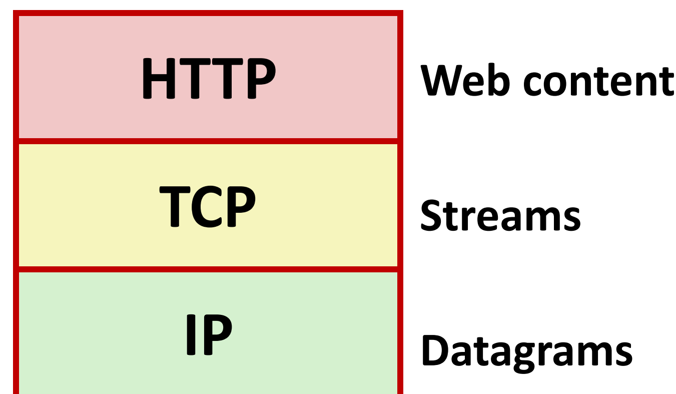
Today

- Network Layers: Birds Eye View
- The Sockets Interface — CSAPP 11.4
- **Web Servers — CSAPP 11.5.1-11.5.3**
- The Tiny Web Server — CSAPP 11.6
- Serving Dynamic Content — CSAPP 11.5.4
- Proxy Servers

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/2 is so different that it might as well be a new protocol.





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
  - Content is identified by its URL (Uniform Resource Locator)

- **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

- ***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
- **Any URL can refer to either static or dynamic content**

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 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

Many more HTTP response codes

201
Created

406
Not Acceptable

414
Request-URI Too Long

Example HTTP Transaction

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

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

Example HTTP Transaction, Take 2

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

Example HTTP(S) Transaction, Take 3

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

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

Quiz Time!

Check Canvas > Networking (part II)

Today

- Network Layers: Birds Eye View
- The Sockets Interface — CSAPP 11.4
- Web Servers — CSAPP 11.5.1-11.5.3
- **The Tiny Web Server — CSAPP 11.6**
- Serving Dynamic Content — CSAPP 11.5.4
- Proxy Servers

Tiny Web Server

- **Tiny Web server described in text**
  - 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);
    Rio_writen(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);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```

tiny.c

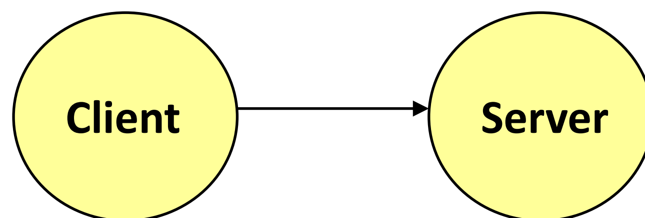
Today

- Network Layers: Birds Eye View
- The Sockets Interface — CSAPP 11.4
- Web Servers — CSAPP 11.5.1-11.5.3
- The Tiny Web Server — CSAPP 11.6
- **Serving Dynamic Content — CSAPP 11.5.4**
- Proxy Servers

Serving Dynamic Content

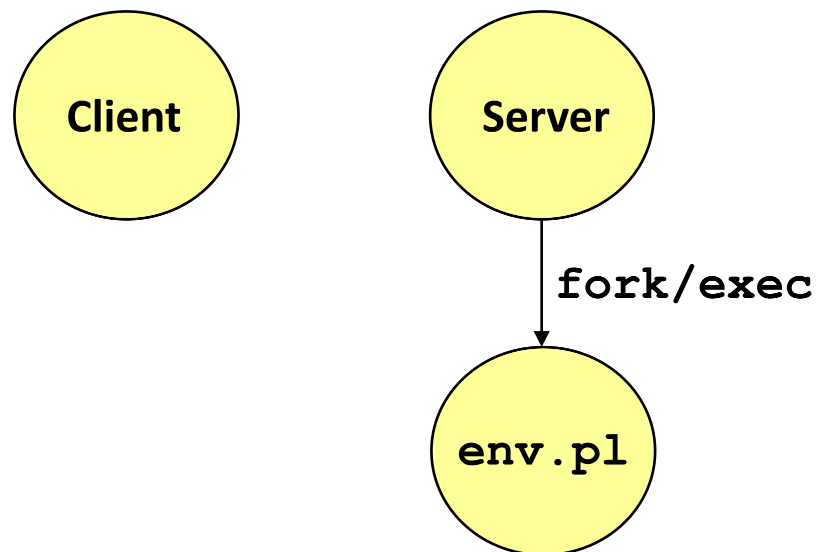
- **Client sends request to server**
- **If request URI contains the string "`/cgi-bin`", the Tiny server assumes that the request is for dynamic content**

`GET /cgi-bin/env.pl HTTP/1.1`



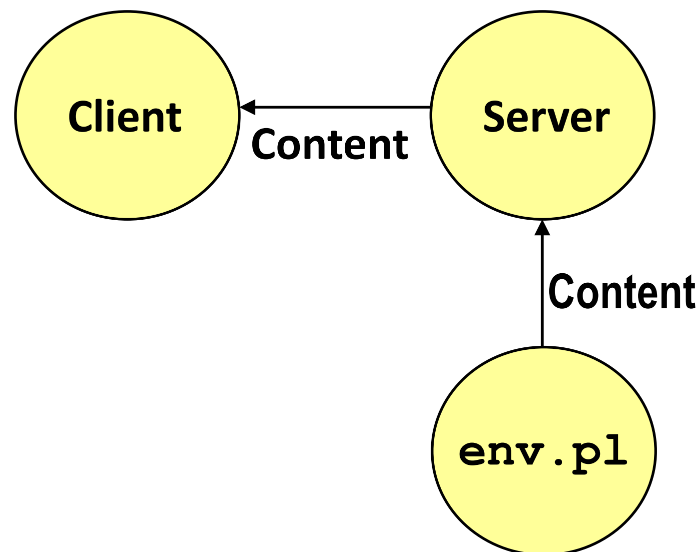
Serving Dynamic Content (cont)

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



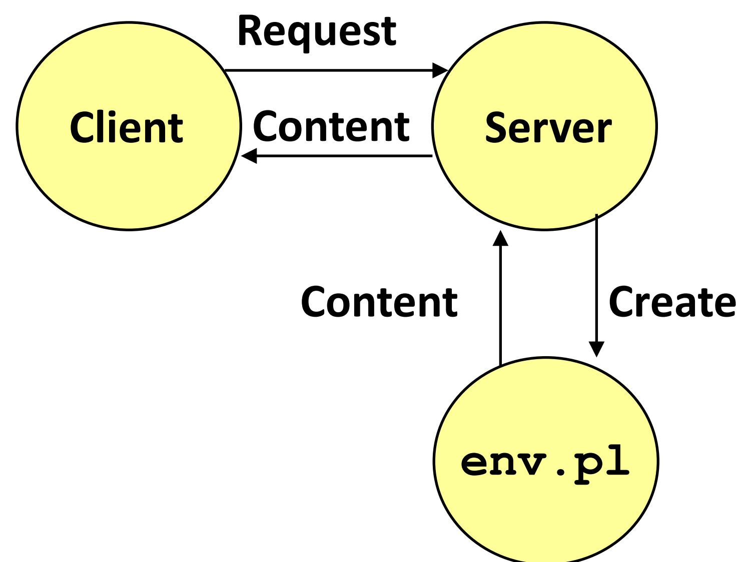
Serving Dynamic Content (cont)

- 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

host

port

CGI program

arguments

whaleshark.ics.cs.cmu.edu

whaleshark.ics.cs.cmu.edu:15213/cgi-bin/adder?15213&18213

Welcome to add.com: THE Internet addition portal.

The answer is: 15213 + 18213 = 33426

Thanks for visiting!

Output page

Serving Dynamic Content With GET

- **Question: How does the client pass arguments to the server?**
- **Answer: 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

- **Question:** How does the server capture the content produced by the child?
- **Answer:** 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

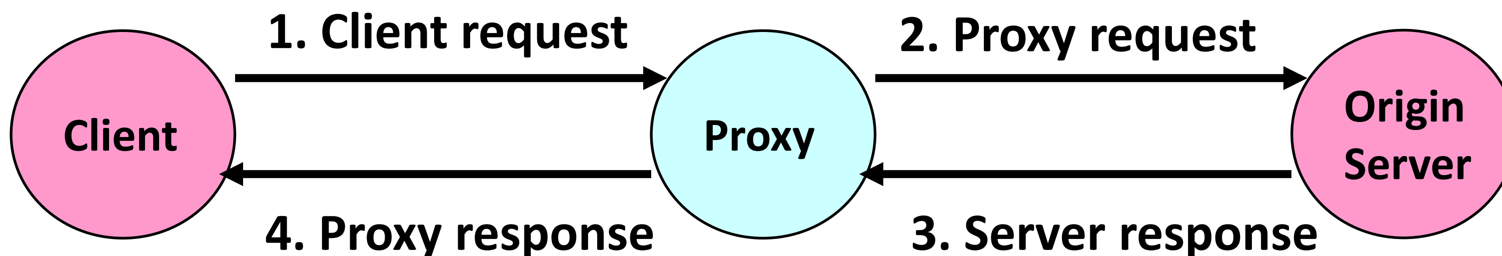
HTTP response generated by the CGI program

Today

- Network Layers: Birds Eye View
- The Sockets Interface — CSAPP 11.4
- Web Servers — CSAPP 11.5.1-11.5.3
- The Tiny Web Server — CSAPP 11.6
- Serving Dynamic Content — CSAPP 11.5.4
- **Proxy Servers**

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



Why Proxies?

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