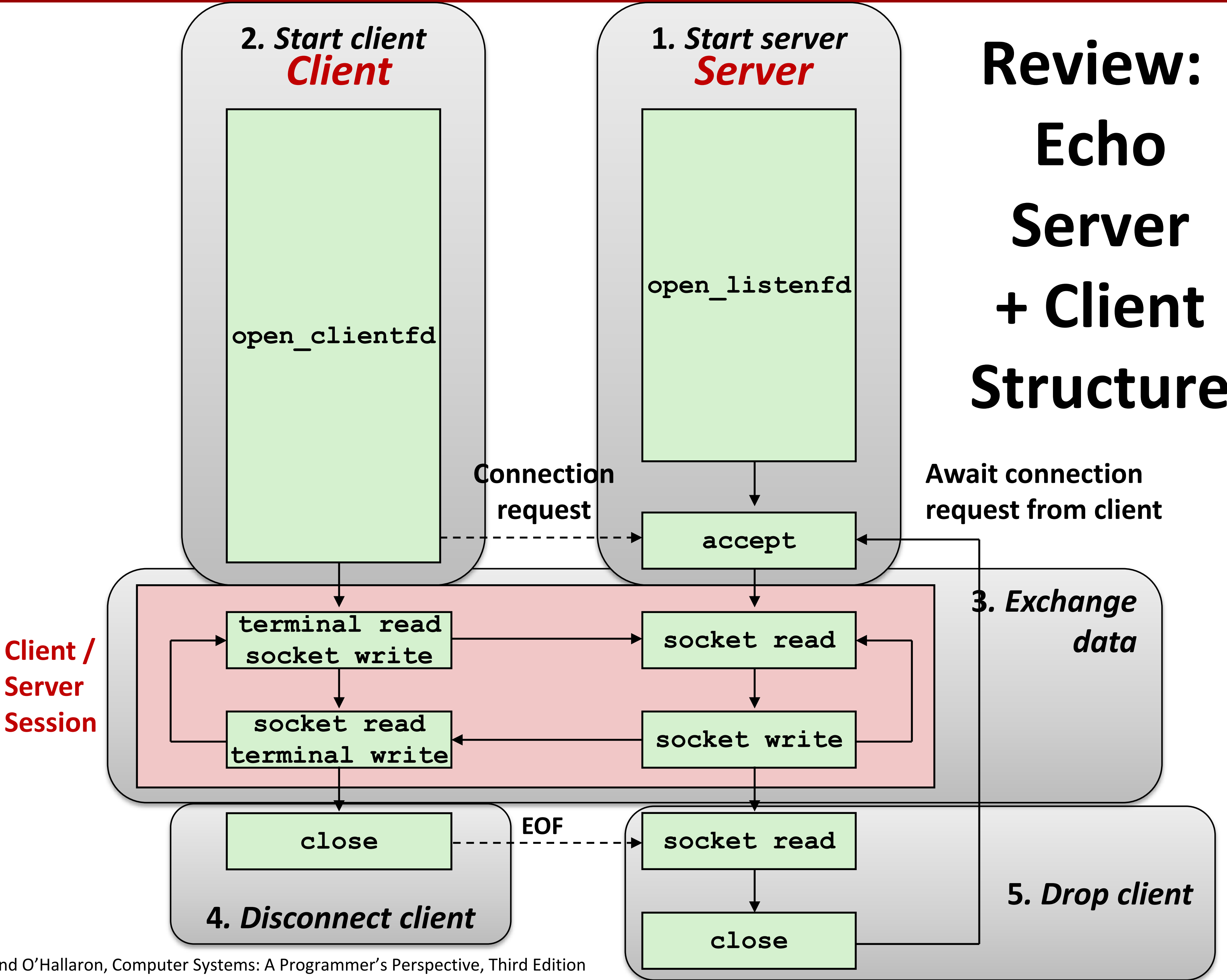
Network Programming: Part II

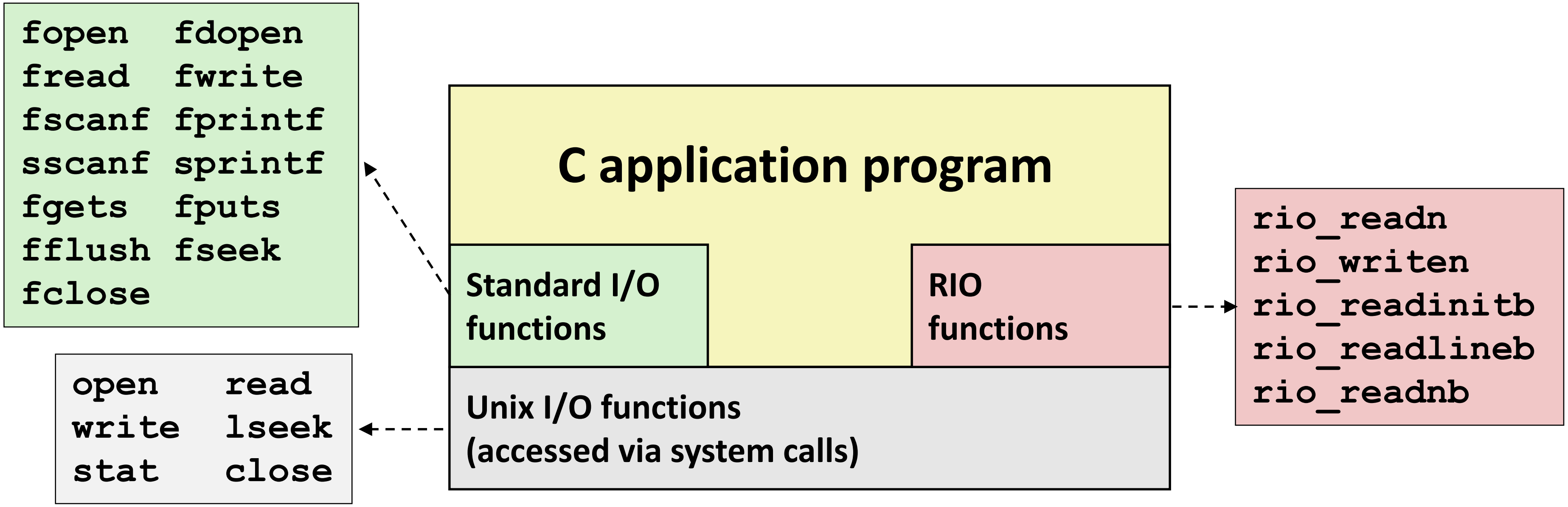
15-213/18-213/14-513/15-513/18-613:
Introduction to Computer Systems
April 9, 2020

Review: Echo Server + Client Structure



Review: C Standard I/O, Unix I/O and RIO

- **Robust I/O (RIO): 15-213 special wrappers**
  good coding practice: handles error checking, signals, and "short counts"



```
fopen   fdopen
fread   fwrite
fscanf  fprintf
sscanf  sprintf
fgets   fputs
fflush  fseek
fclose
```

```
open    read
write   lseek
stat    close
```

C application program

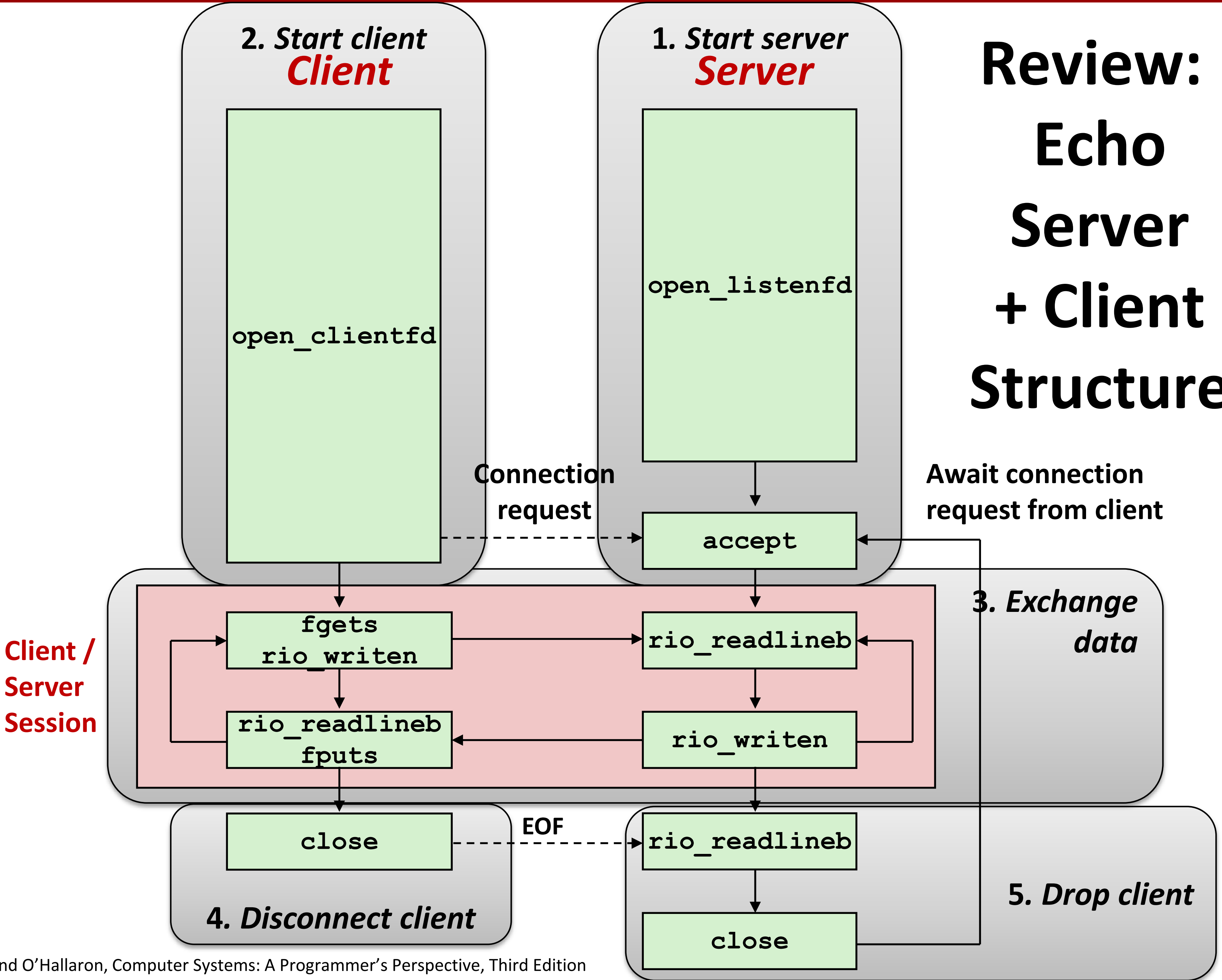
Standard I/O functions

RIO functions

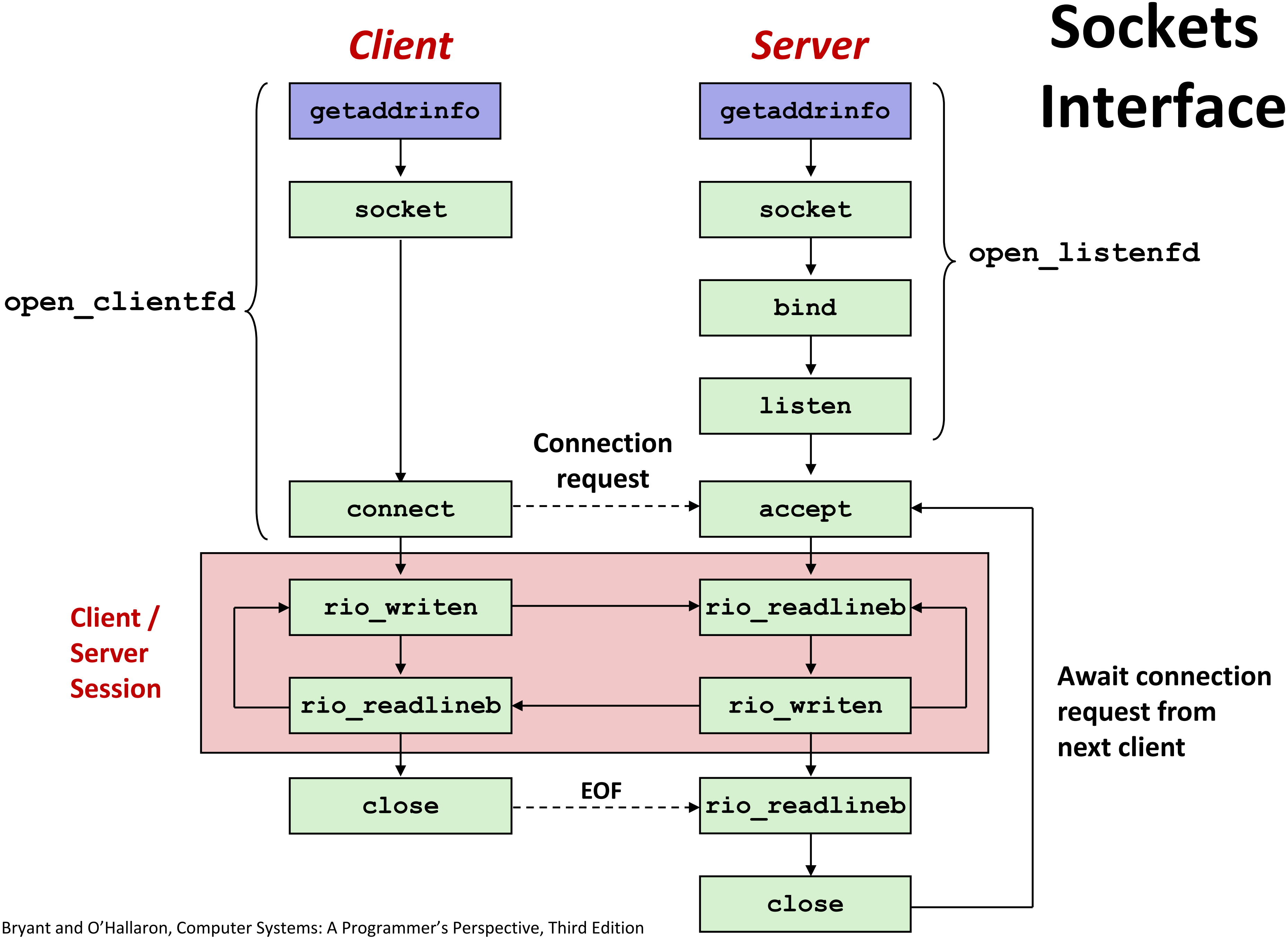
Unix I/O functions (accessed via system calls)

```
rio_readn
rio_writen
rio_readinitb
rio_readlineb
rio_readnb
```

Review: Echo Server + Client Structure



Sockets Interface

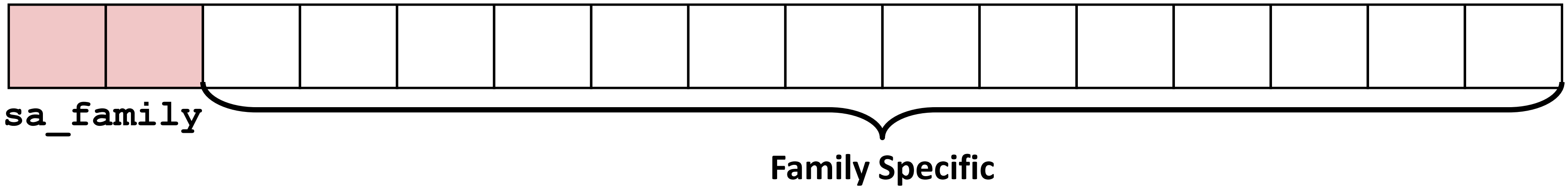


Socket Address Structures & getaddrinfo

- **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.  */
};
```

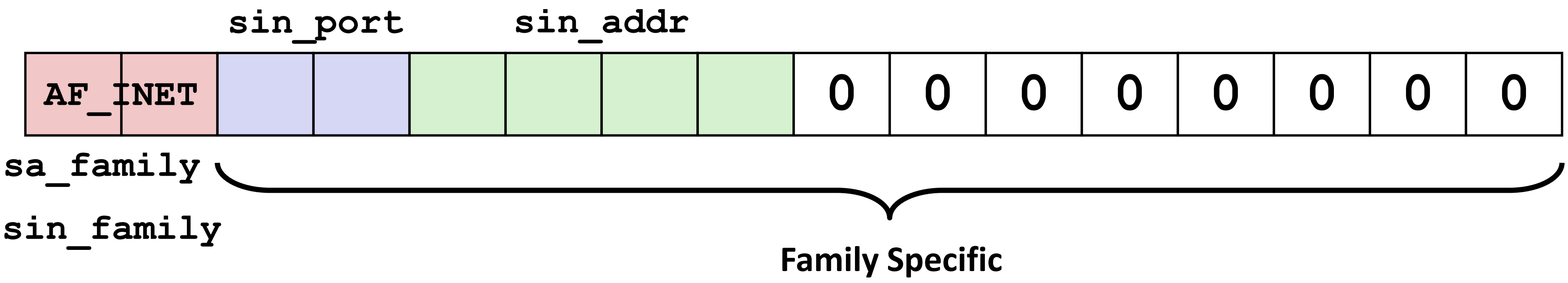


- **getaddrinfo** converts string representations of hostnames, host addresses, ports, service names to socket address structures

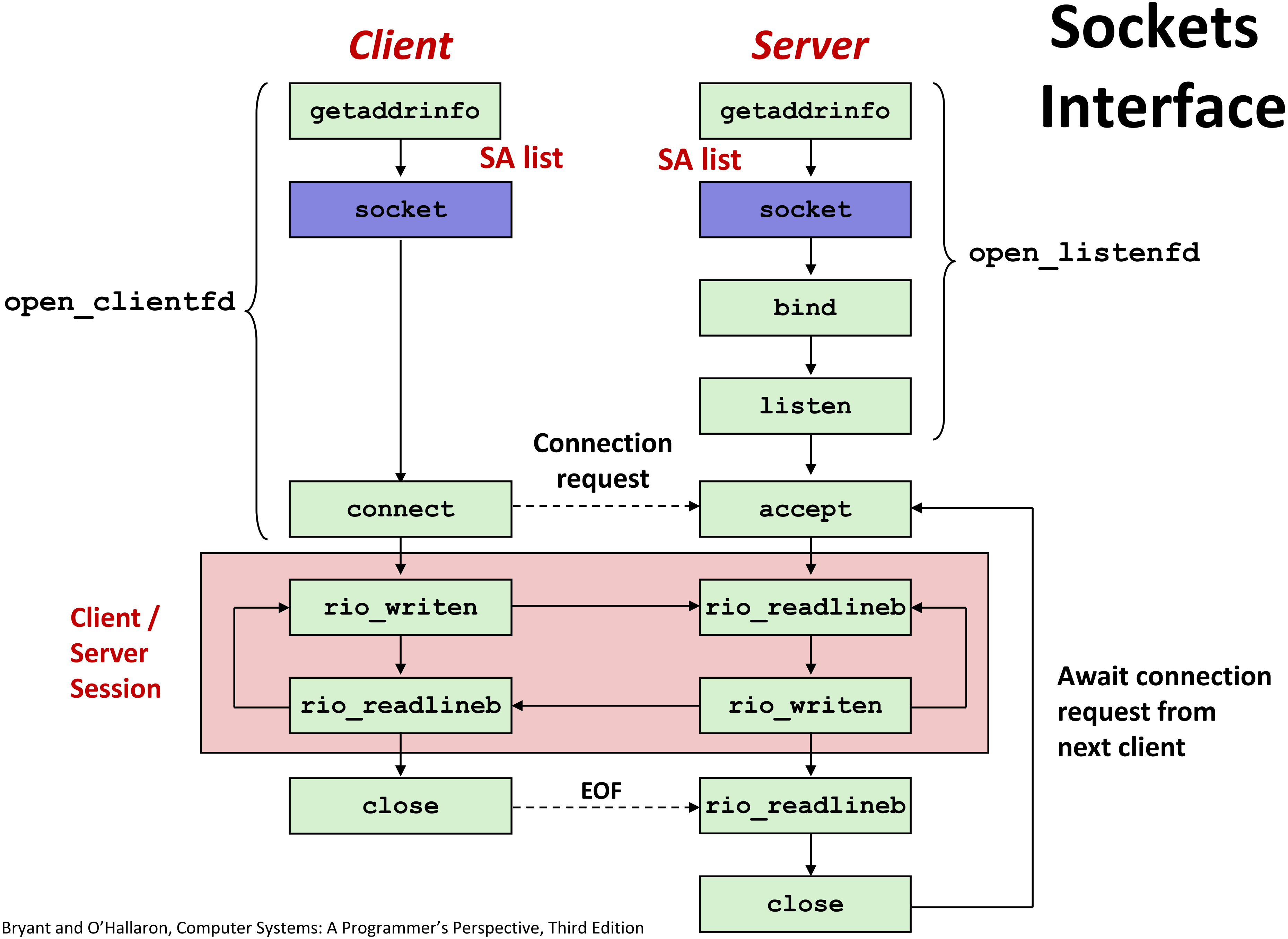
Socket Address Structures

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

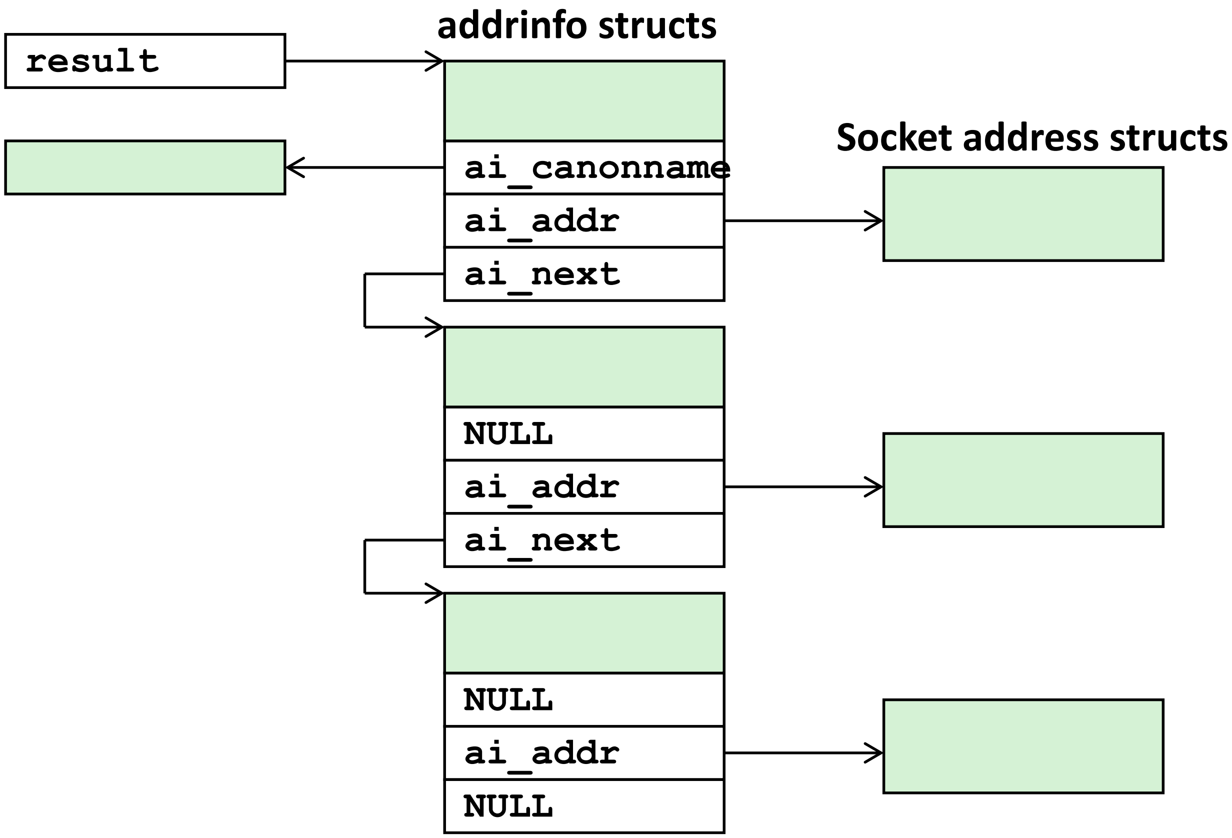
```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};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



Sockets Interface



Linked List Returned by getaddrinfo



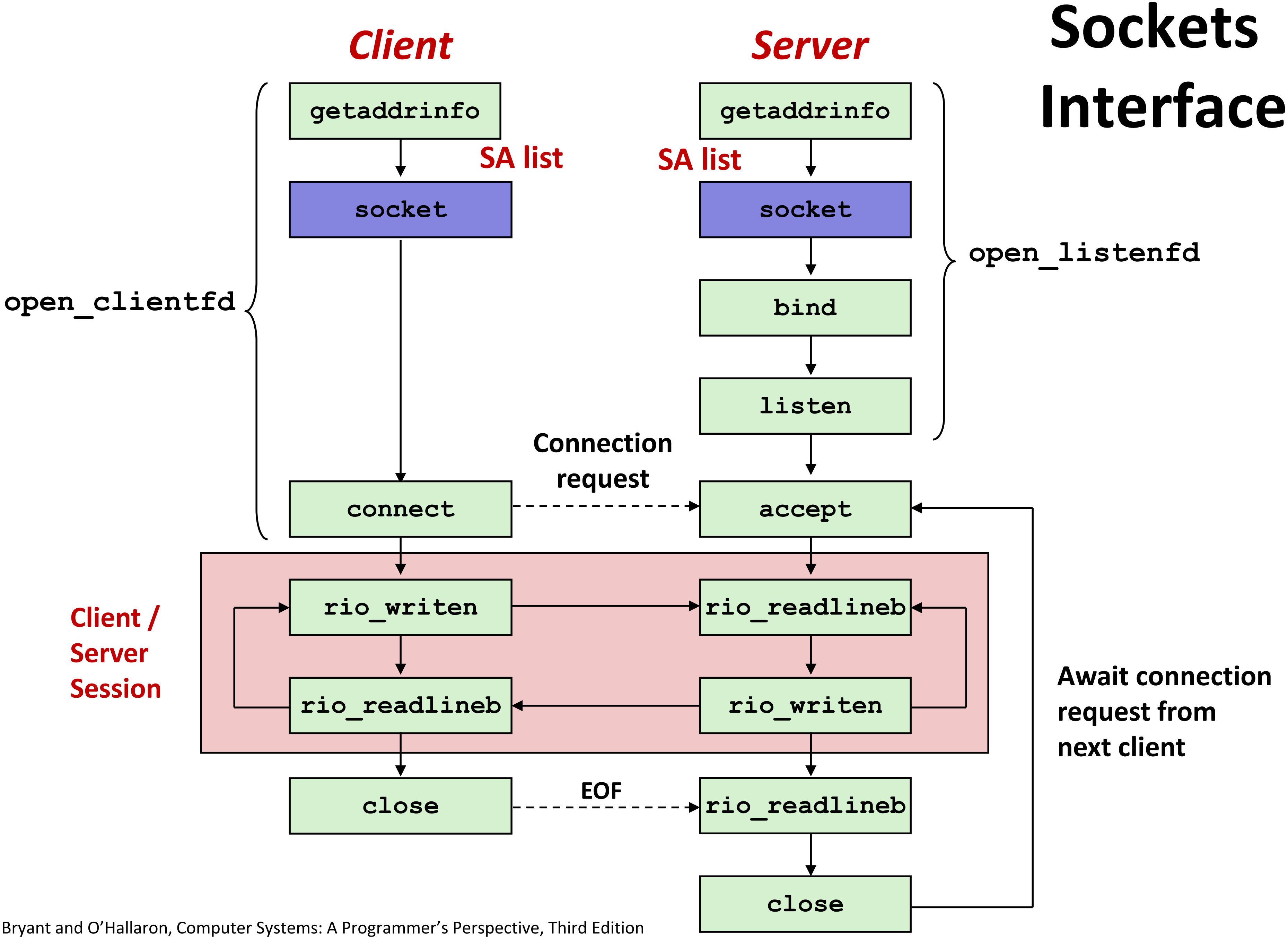
- **Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.**
- **Servers: walk the list until calls to `socket` and `bind` succeed.**

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 .**

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: 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

```
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 */
```

Pro
ge
pro

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);
```

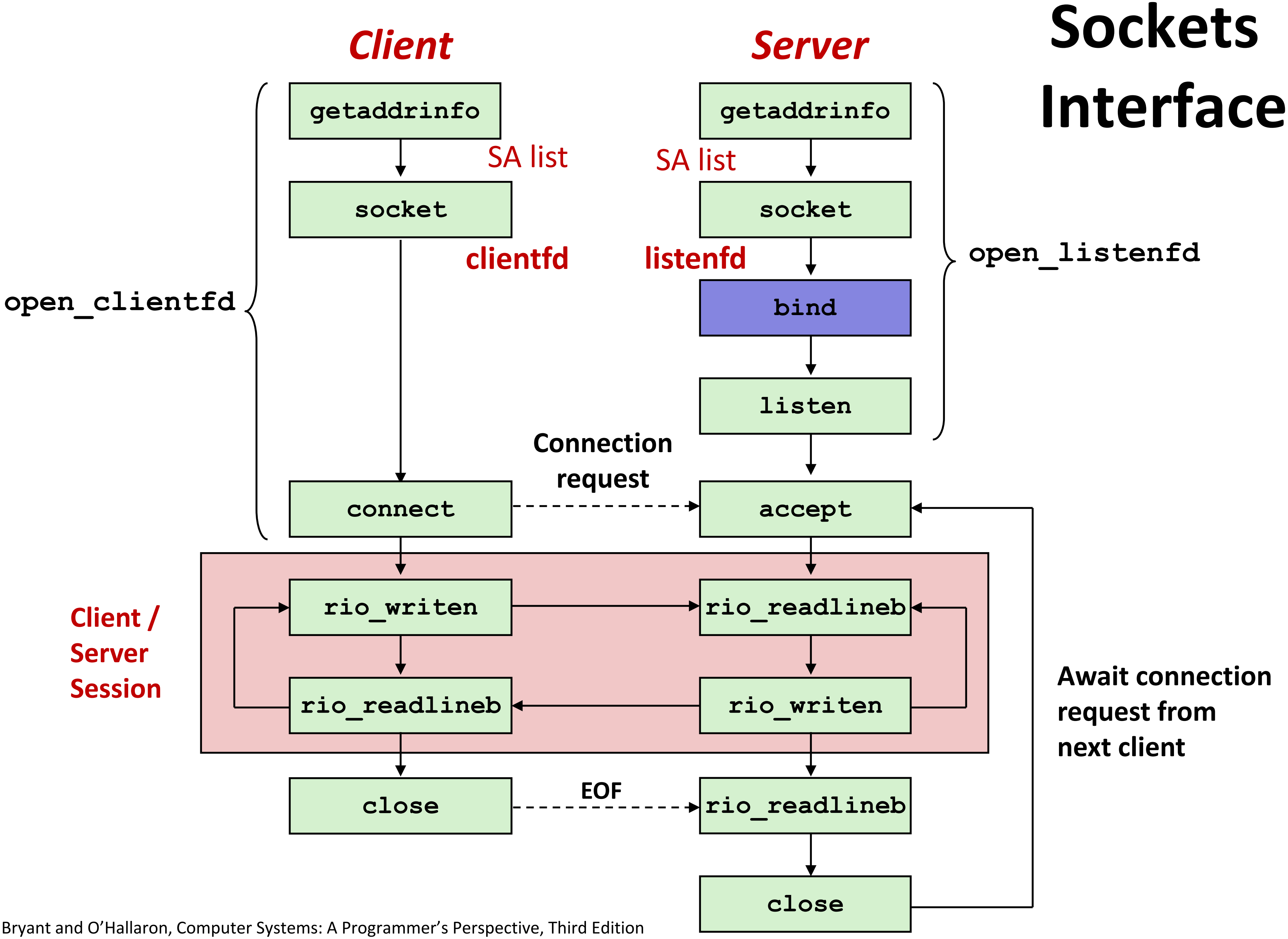
Indicates that we are using 32-bit IPV4 addresses

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

- Better Example: (protocol independent)

```
struct addrinfo *infop = …;
int sockfd = socket(p->ai_family, p-> ai_socktype, p->ai_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);
```

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`

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 result from `getaddrinfo`

```
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 */
```

Sockets Interface: bind

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

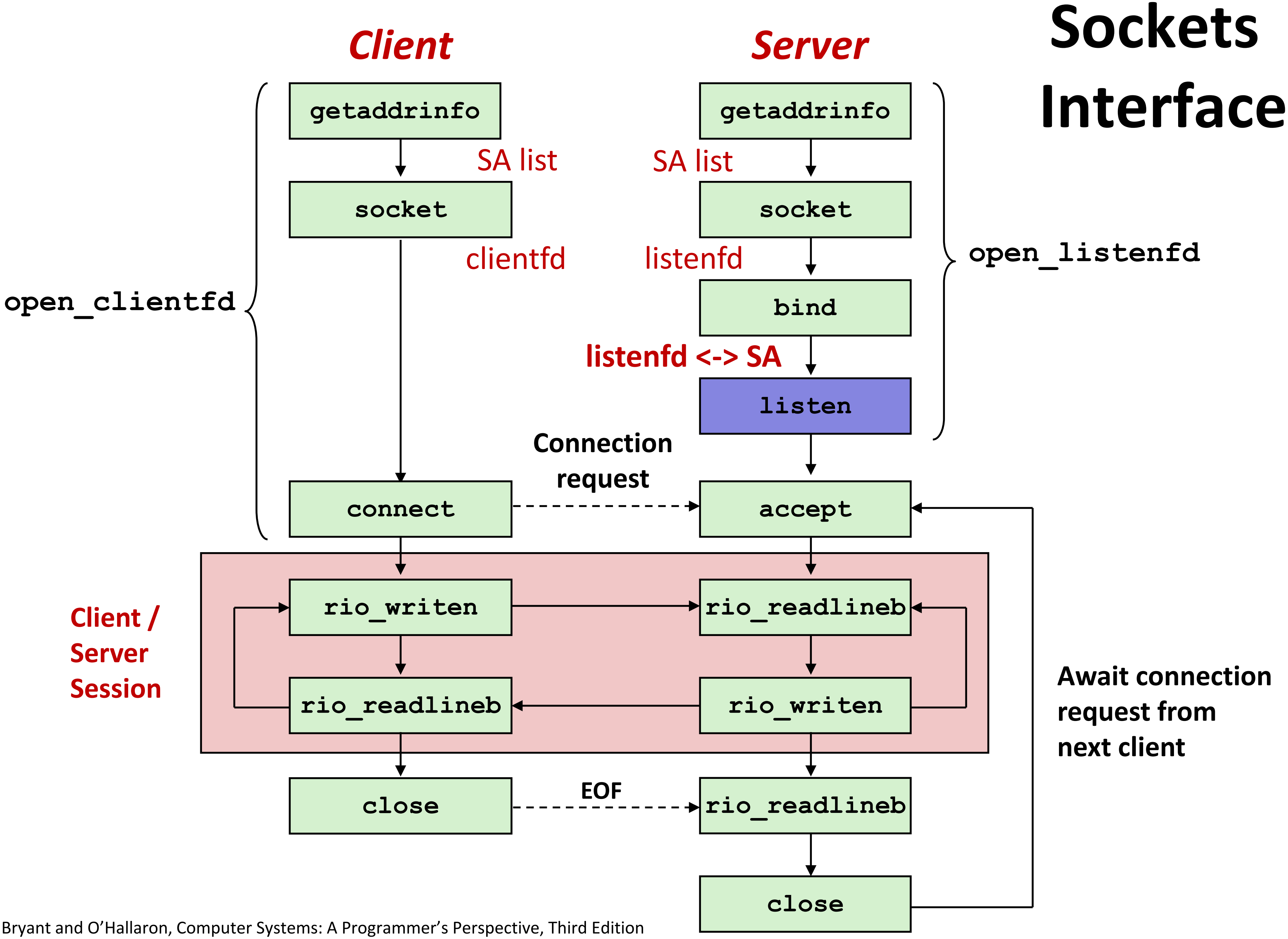
```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

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 result from `getaddrinfo`

```
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 */
```

Sockets Interface



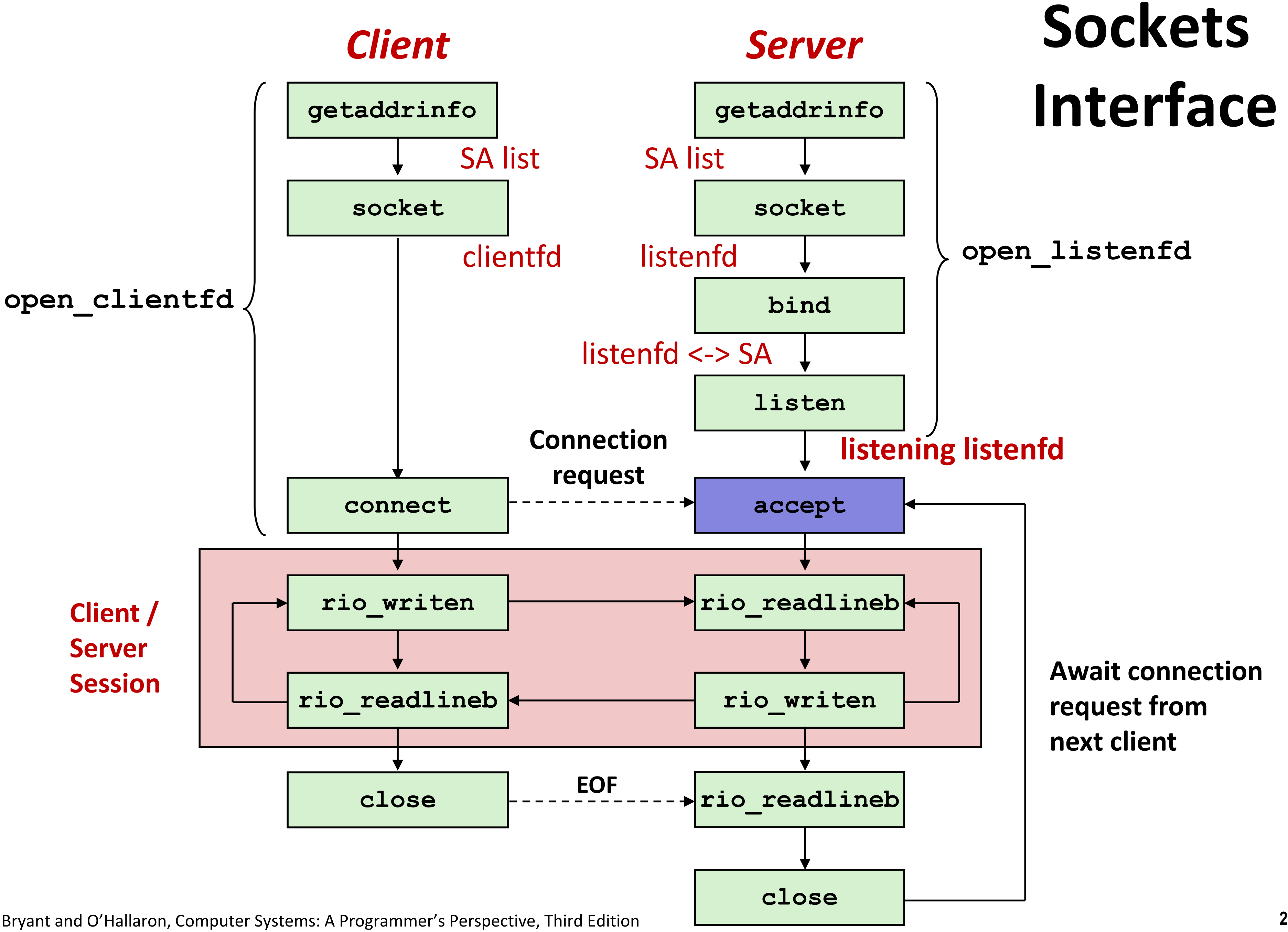
Sockets Interface: `listen`

- 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`

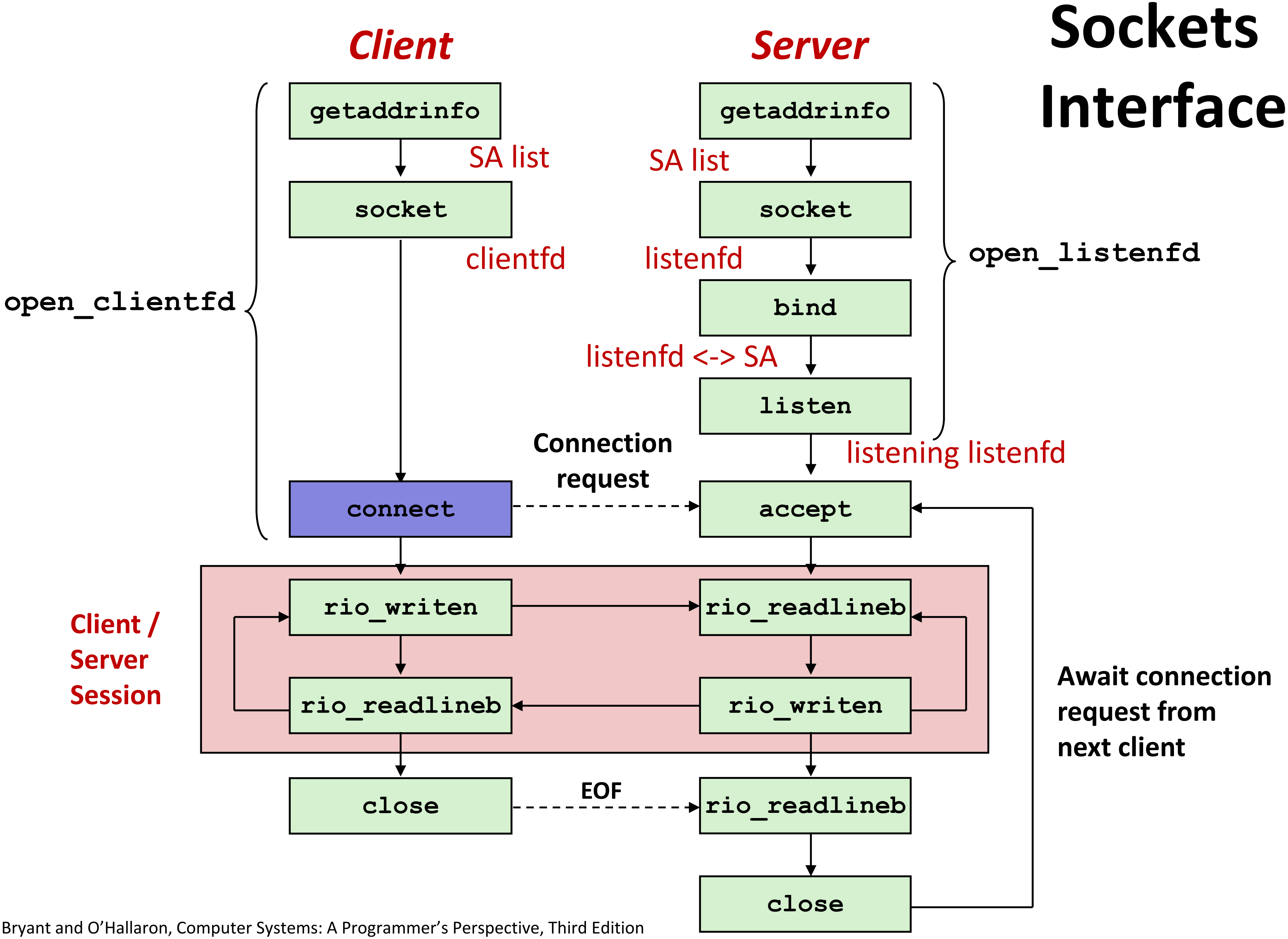
- Servers wait for connection requests from clients by calling `accept`:

```
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* 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);
```

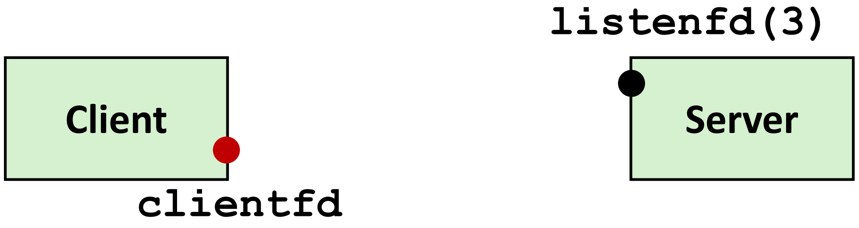
- **Attempts to establish a connection with server at socket address `addr`**
  - If successful, then **`clientfd`** is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

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

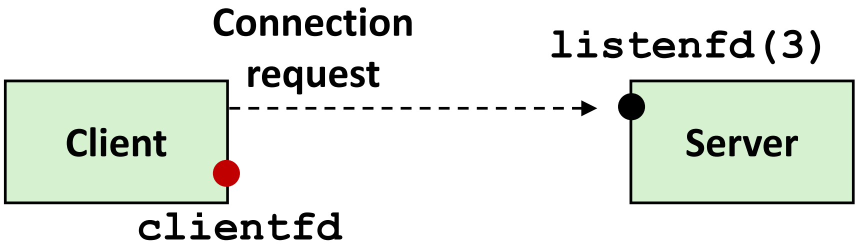
    - **`x`** is client address
    - **`y`** is ephemeral port that uniquely identifies client process on client host

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

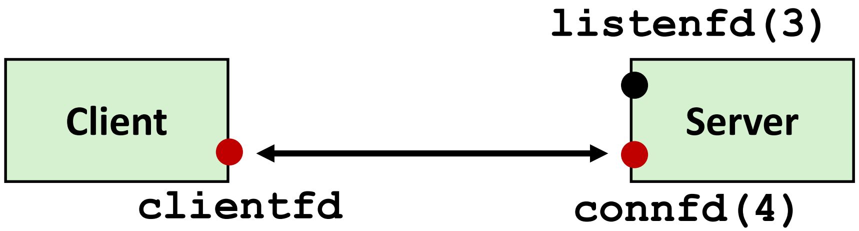
`connect`/`accept` Illustrated



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`

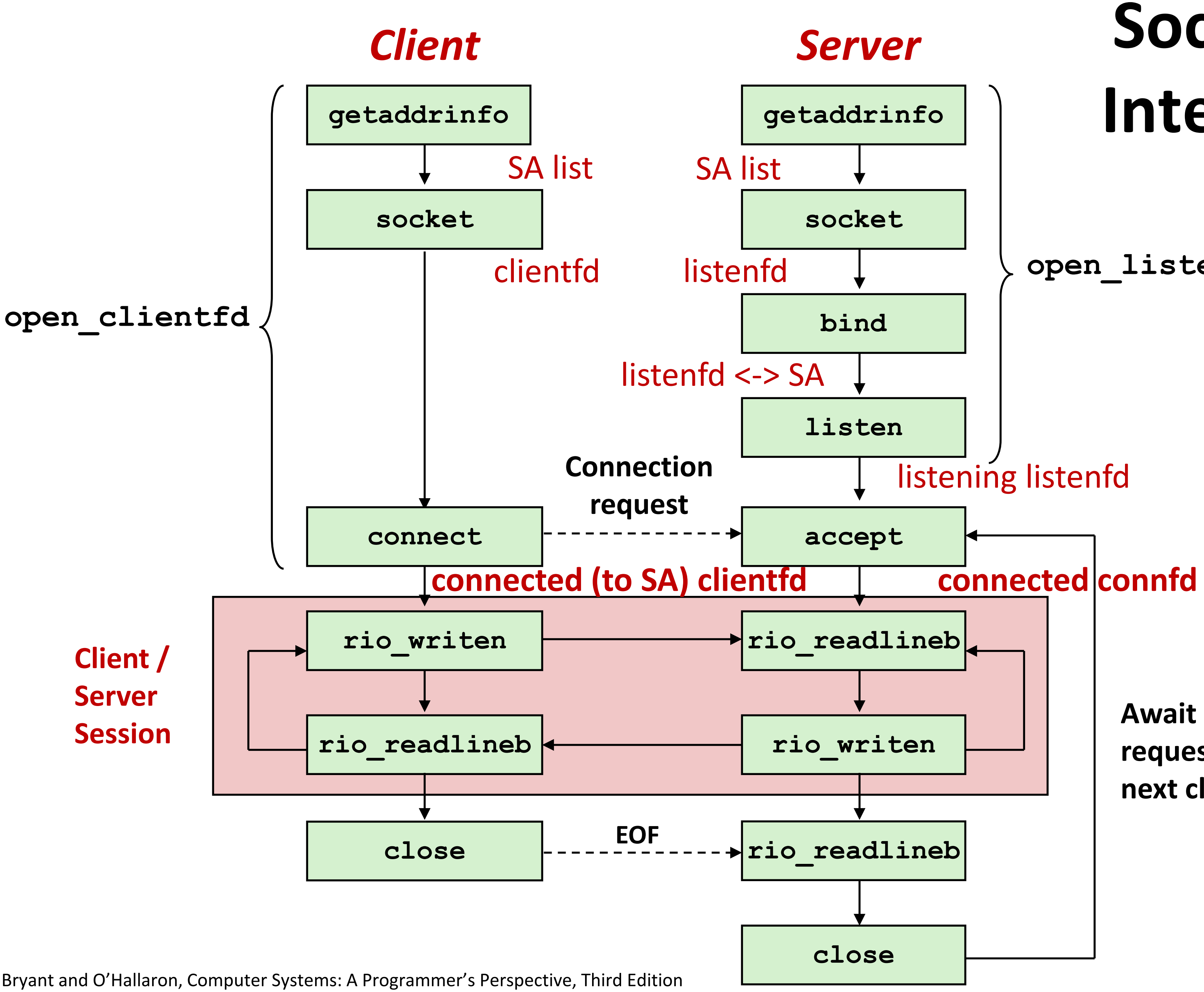
Connected vs. Listening Descriptors

- **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

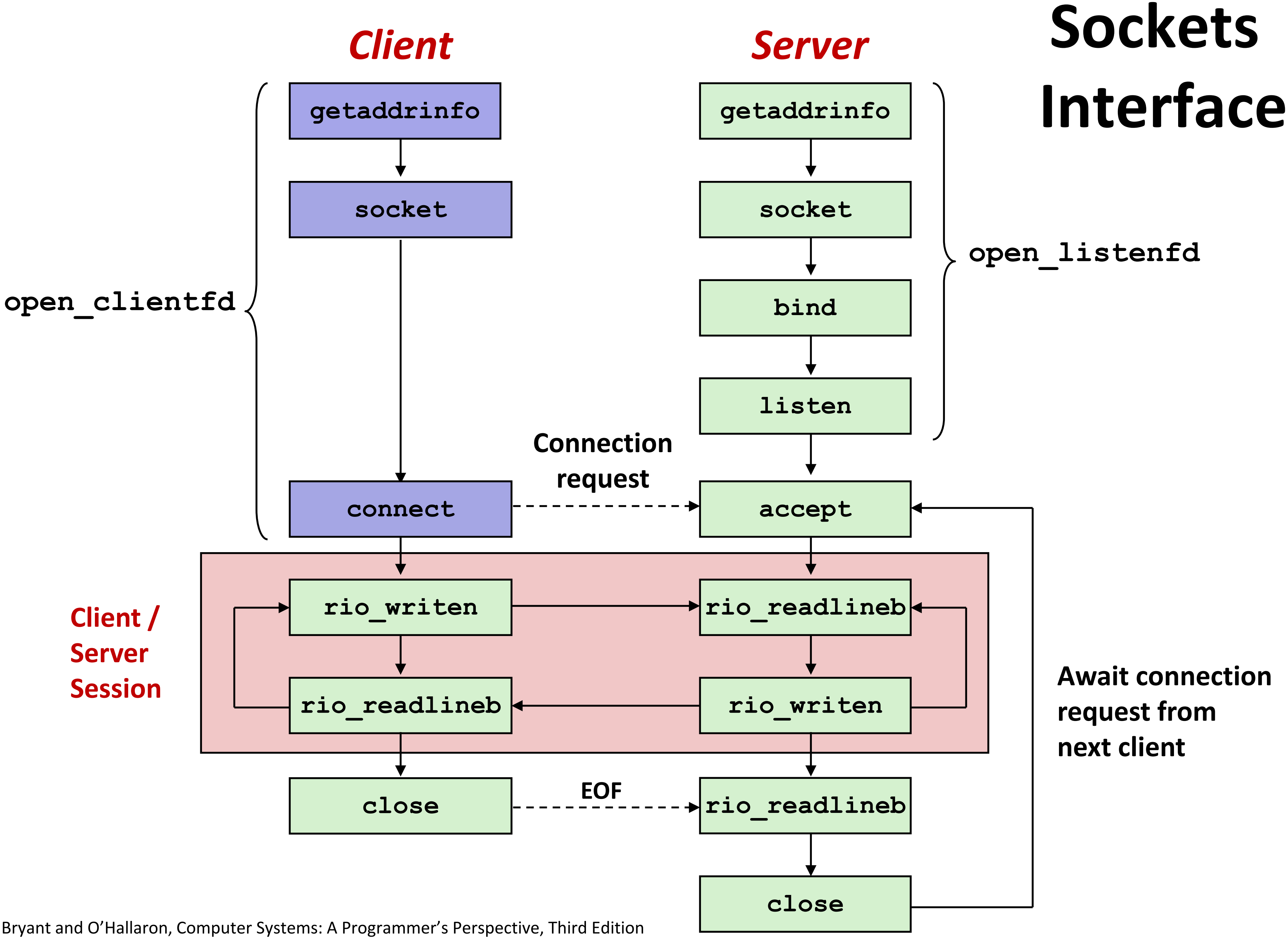
- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

Sockets Interface



Await connection request from next client

Sockets Interface



Sockets Helper: `open_clientfd`

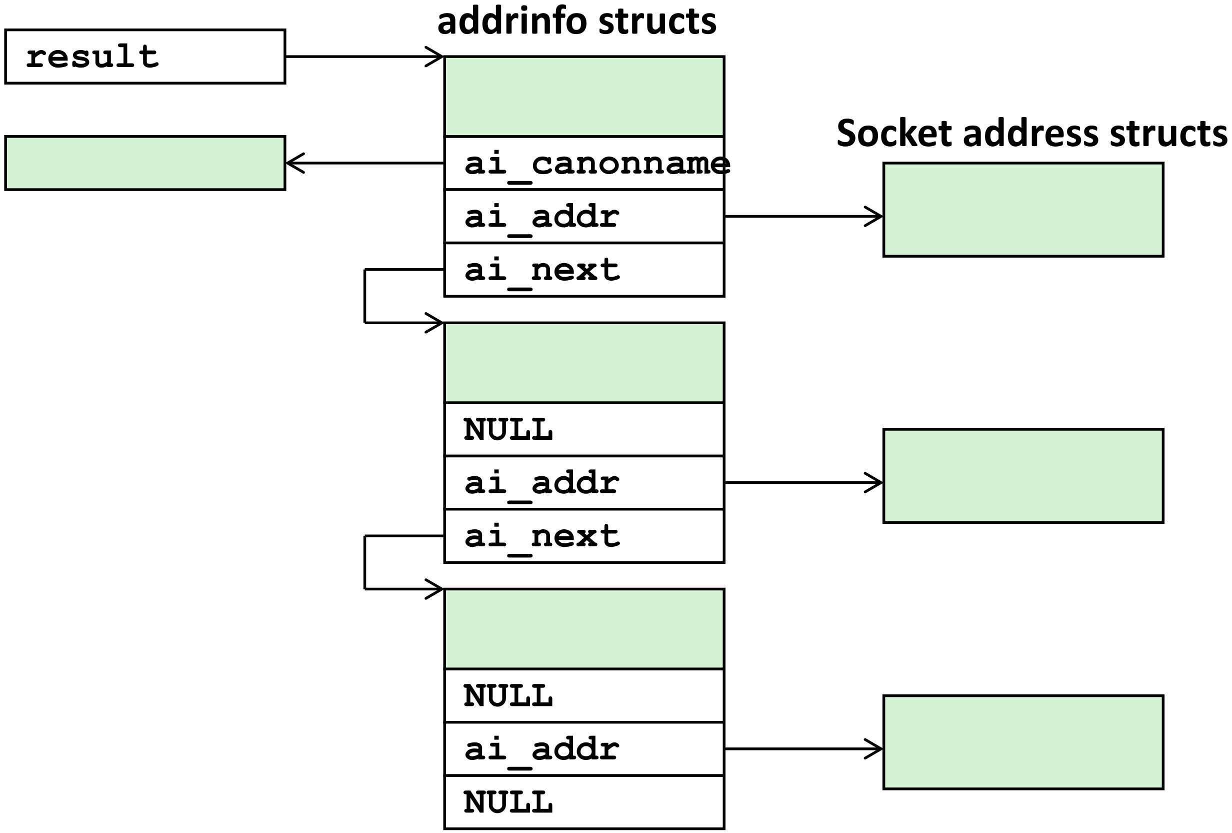
- Establish a connection with a server

```
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

`AI_ADDRCONFIG` – uses your system's address type.
You have at least one IPV4 iface? IPV4. At least one IPV6? IPV6.

Review: getaddrinfo Linked List

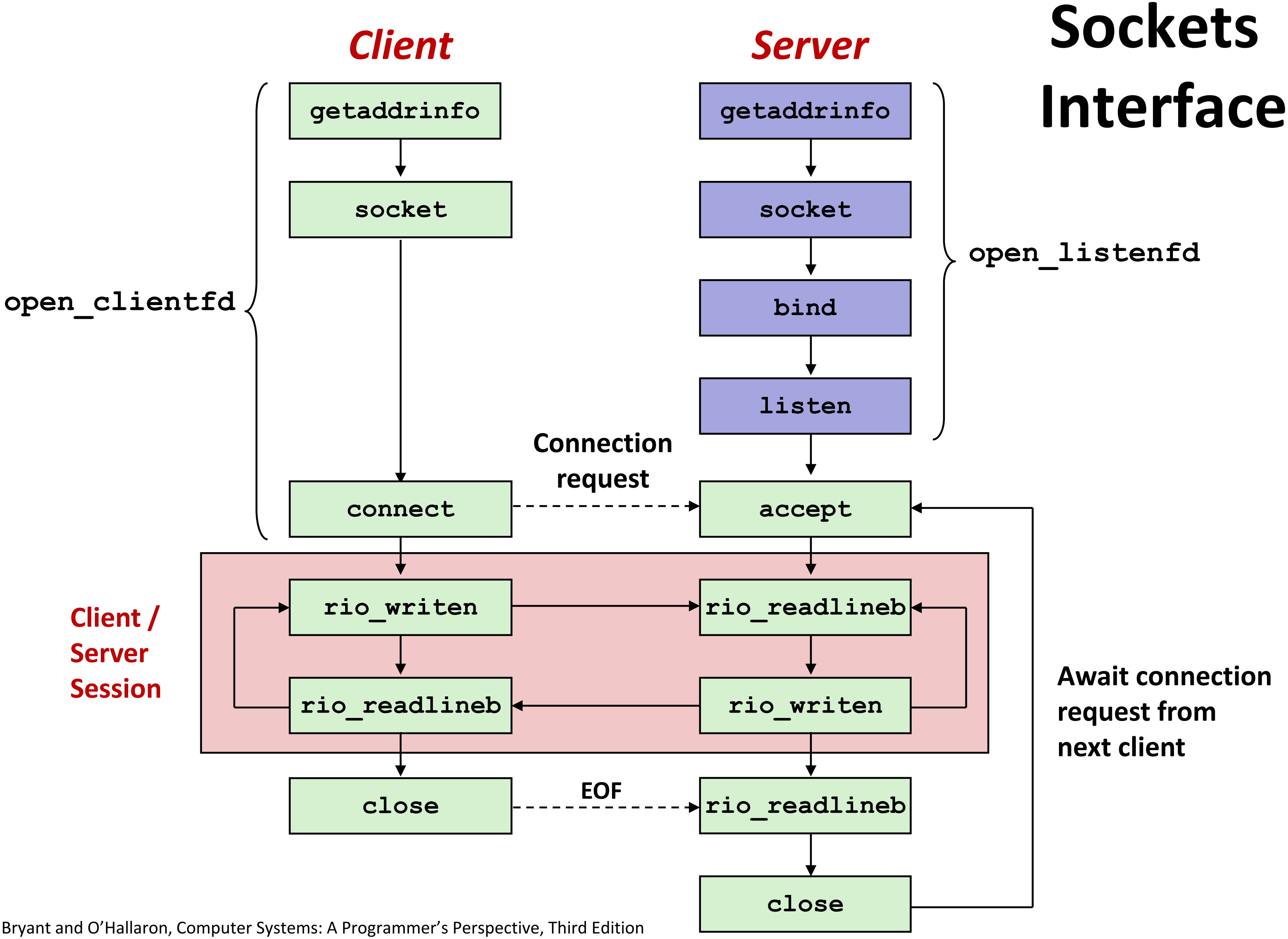


- Clients: walk this list, trying each socket address in turn, until the calls to socket and connect succeed.
- Servers: walk the list until calls to socket and bind succeed.

```
    /* 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  */
    hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for ection*/
    Getaddrinfo(hostname, port, &hints, &listp);

    /* 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.

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

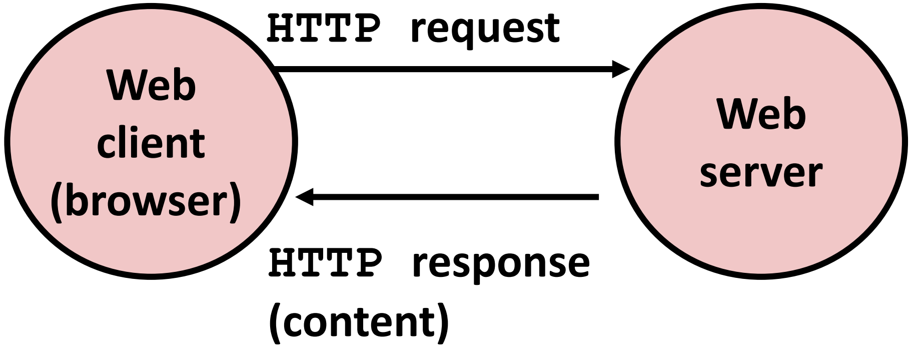
- **Usage:**
  - **`linux>`** ***`telnet <host> <portnumber>`***
  - Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

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>
```

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.

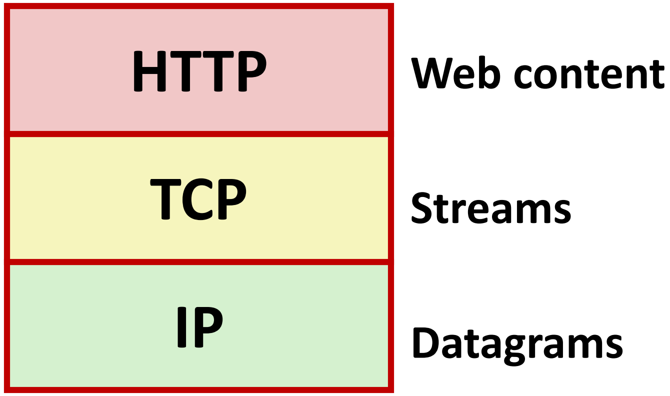


Web client (browser)

`HTTP` request

Web server

`HTTP` response (content)



HTTP — **Web content**

TCP — **Streams**

IP — **Datagrams**

`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 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
```

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 out:

https://canvas.cmu.edu/courses/13182

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

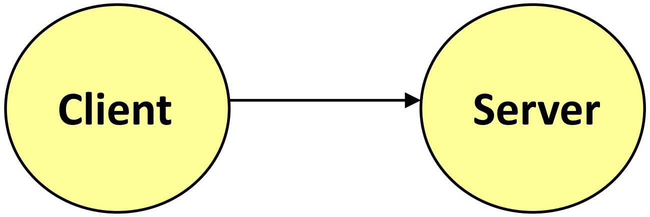
```c
void serve_static(int fd, char *filename, int filesize)
{
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    /* 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

Serving Dynamic Content

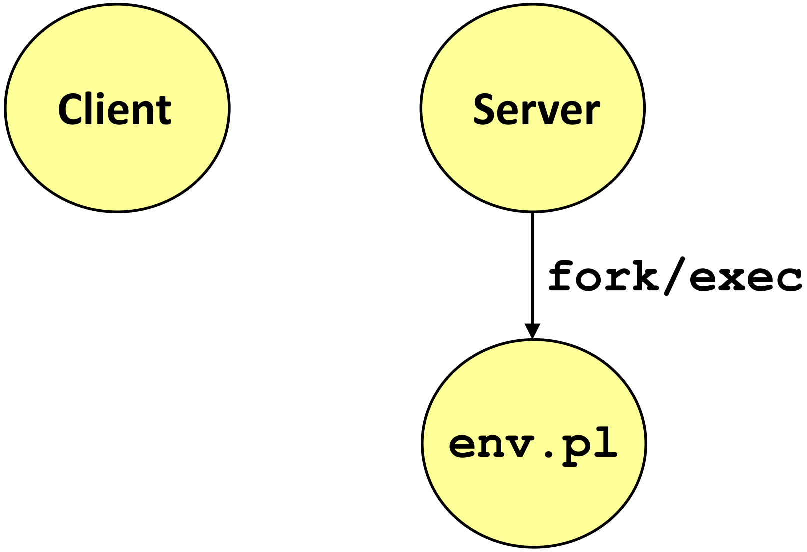
- Client sends request to server
- If request URI contains the string "`/cgi-bin`", the Tiny server assumes that the request is for dynamic content

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



Serving Dynamic Content (cont)

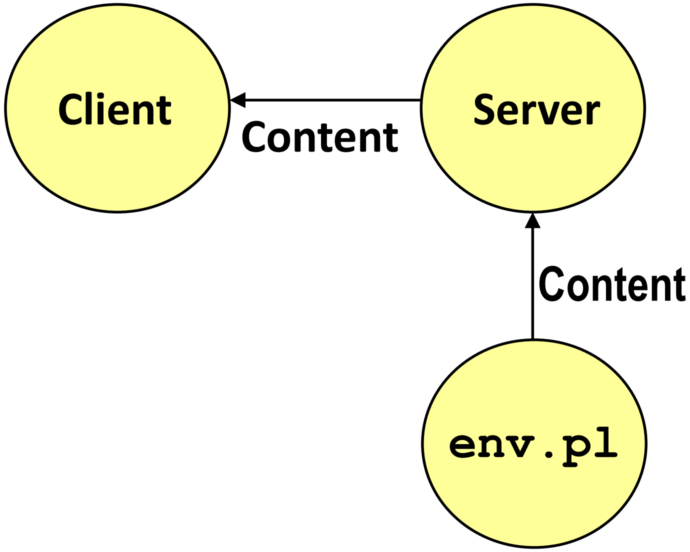
- The server creates a child process and runs the program identified by the URI in that process



Serving Dynamic Content (cont)

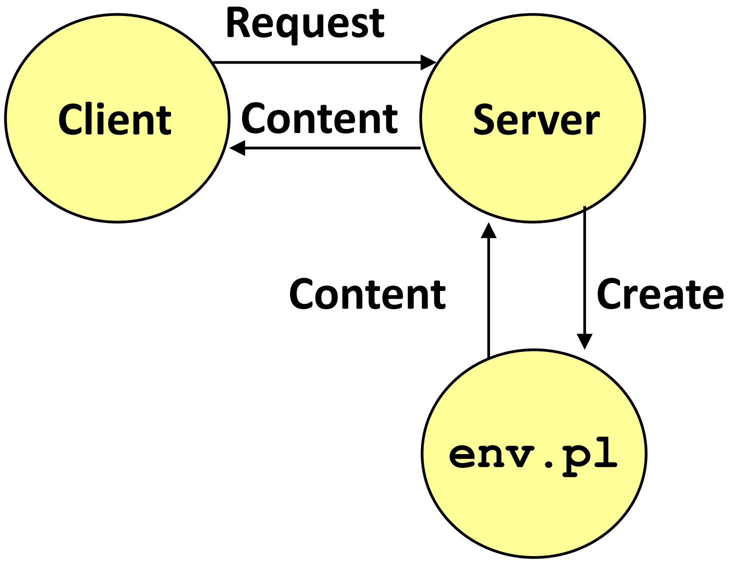
- The child runs and generates the dynamic content

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



Issues in Serving Dynamic Content

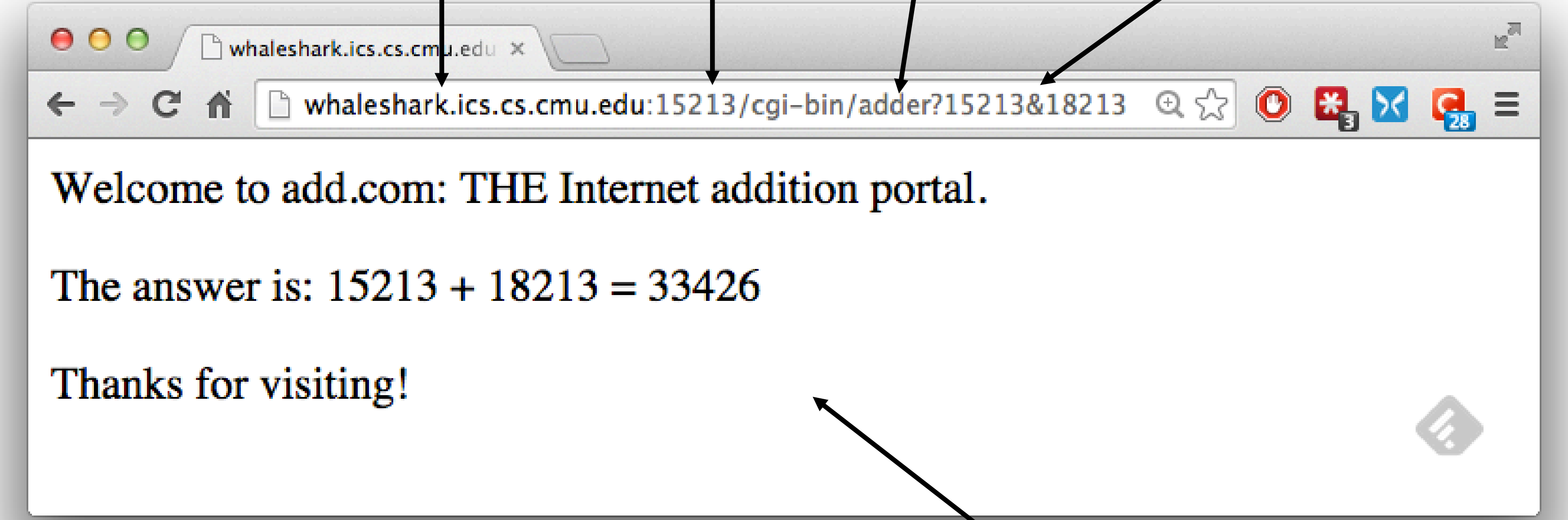
- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the Common 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



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!

host

port

CGI program

arguments

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

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.
- **Complete versions of all code in this lecture is available from the 213 schedule page.**
  - `http://www.cs.cmu.edu/~213/schedule.html`
  - csapp.{.c,h}, hostinfo.c, echoclient.c, echoserveri.c, tiny.c, adder.c
  - You can use any of this code in your assignments.

Additional slides

Web History

- **1989:**
  - Tim Berners-Lee (CERN) writes internal proposal to develop a distributed hypertext system
    - Connects "a web of notes with links"
    - Intended to help CERN physicists in large projects share and manage information
- **1990:**
  - Tim BL writes a graphical browser for Next machines

Web History (cont)

- **1992**
  - NCSA server released
  - 26 WWW servers worldwide
- **1993**
  - Marc Andreessen releases first version of NCSA Mosaic browser
  - Mosaic version released for (Windows, Mac, Unix)
  - Web (port 80) traffic at 1% of NSFNET backbone traffic
  - Over 200 WWW servers worldwide
- **1994**
  - Andreessen and colleagues leave NCSA to form "Mosaic Communications Corp" (predecessor to Netscape)

HTTP Versions

- **Major differences between HTTP/1.1 and HTTP/1.0**
  - HTTP/1.0 uses a new connection for each transaction
  - HTTP/1.1 also supports *persistent connections*
    - multiple transactions over the same connection
    - `Connection: Keep-Alive`
  - HTTP/1.1 requires `HOST` header
    - `Host: www.cmu.edu`
    - Makes it possible to host multiple websites at single Internet host
  - HTTP/1.1 supports *chunked encoding*
    - `Transfer-Encoding: chunked`
  - HTTP/1.1 adds additional support for caching

GET Request to Apache Server From Firefox Browser

URI is just the suffix, not the entire URL

```
GET /~bryant/test.html HTTP/1.1
Host: www.cs.cmu.edu
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US;
rv:1.9.2.11) Gecko/20101012 Firefox/3.6.11
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
CRLF (\r\n)
```

GET Response From Apache Server

```
HTTP/1.1 200 OK
Date: Fri, 29 Oct 2010 19:48:32 GMT
Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.7m
mod_pubcookie/3.3.2b PHP/5.3.1
Accept-Ranges: bytes
Content-Length: 479
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
<html>
<head><title>Some Tests</title></head>

<body>
<h1>Some Tests</h1>
 . . .
</body>
</html>
```

Data Transfer Mechanisms

- **Standard**
  - Specify total length with content-length
  - Requires that program buffer entire message
- **Chunked**
  - Break into blocks
  - Prefix each block with number of bytes (Hex coded)

Chunked Encoding Example

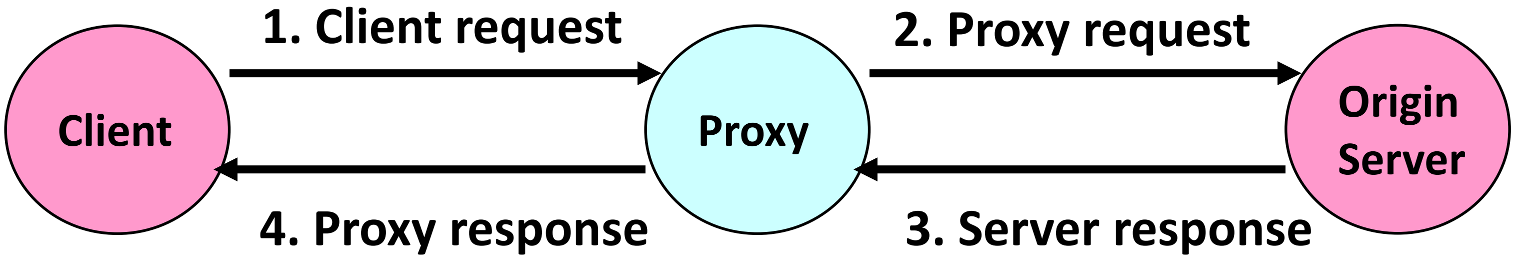
```
HTTP/1.1 200 OK\n
Date: Sun, 31 Oct 2010 20:47:48 GMT\n
Server: Apache/1.3.41 (Unix)\n
Keep-Alive: timeout=15, max=100\n
Connection: Keep-Alive\n
Transfer-Encoding: chunked\n
Content-Type: text/html\n
\r\n
d75\r\n
<html>
<head>
.<link href="http://www.cs.cmu.edu/style/calendar.css" rel="stylesheet"
type="text/css">
</head>
<body id="calendar_body">

<div id='calendar'><table width='100%'  border='0' cellpadding='0'
cellspacing='1' id='cal'>


 . . .
</body>
</html>
\r\n
0\r\n
\r\n
```

First Chunk: 0xd75 = 3445 bytes

Second Chunk: 0 bytes (indicates last chunk)

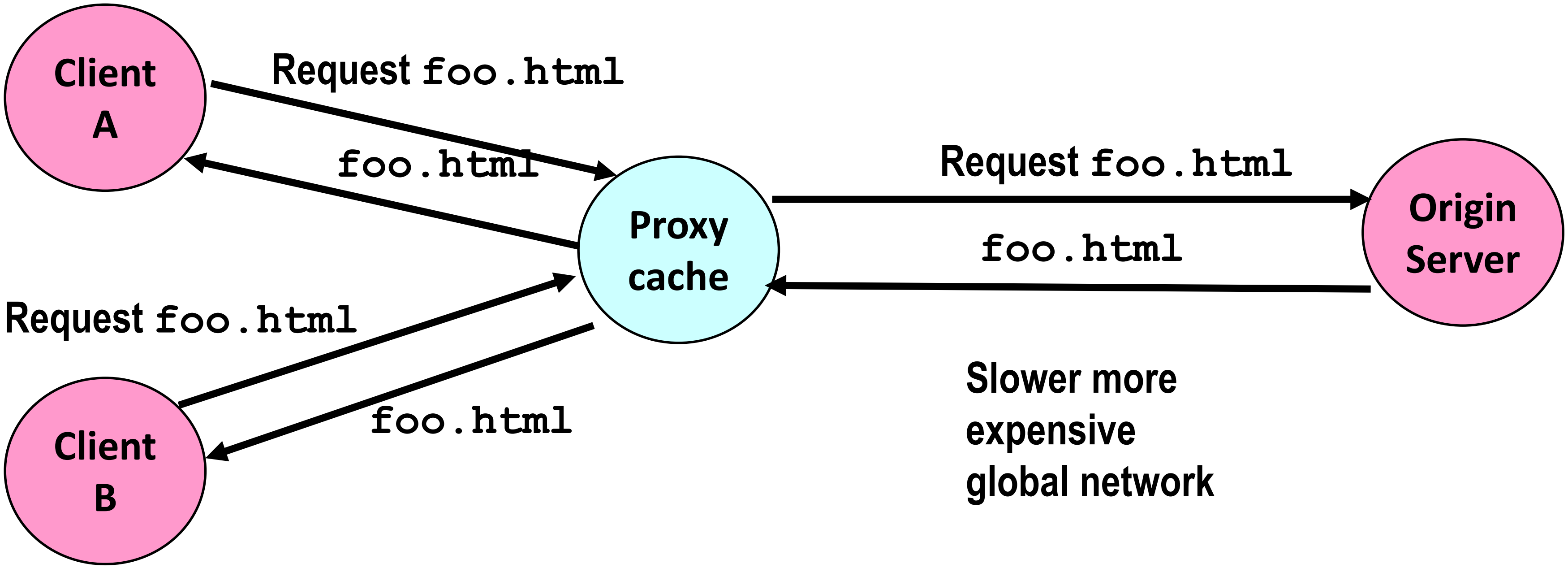
Proxies

- 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



Fast inexpensive local network