COMP2310/COMP6310
Systems, Networks, & Concurrency

Convener: Prof John Taylor
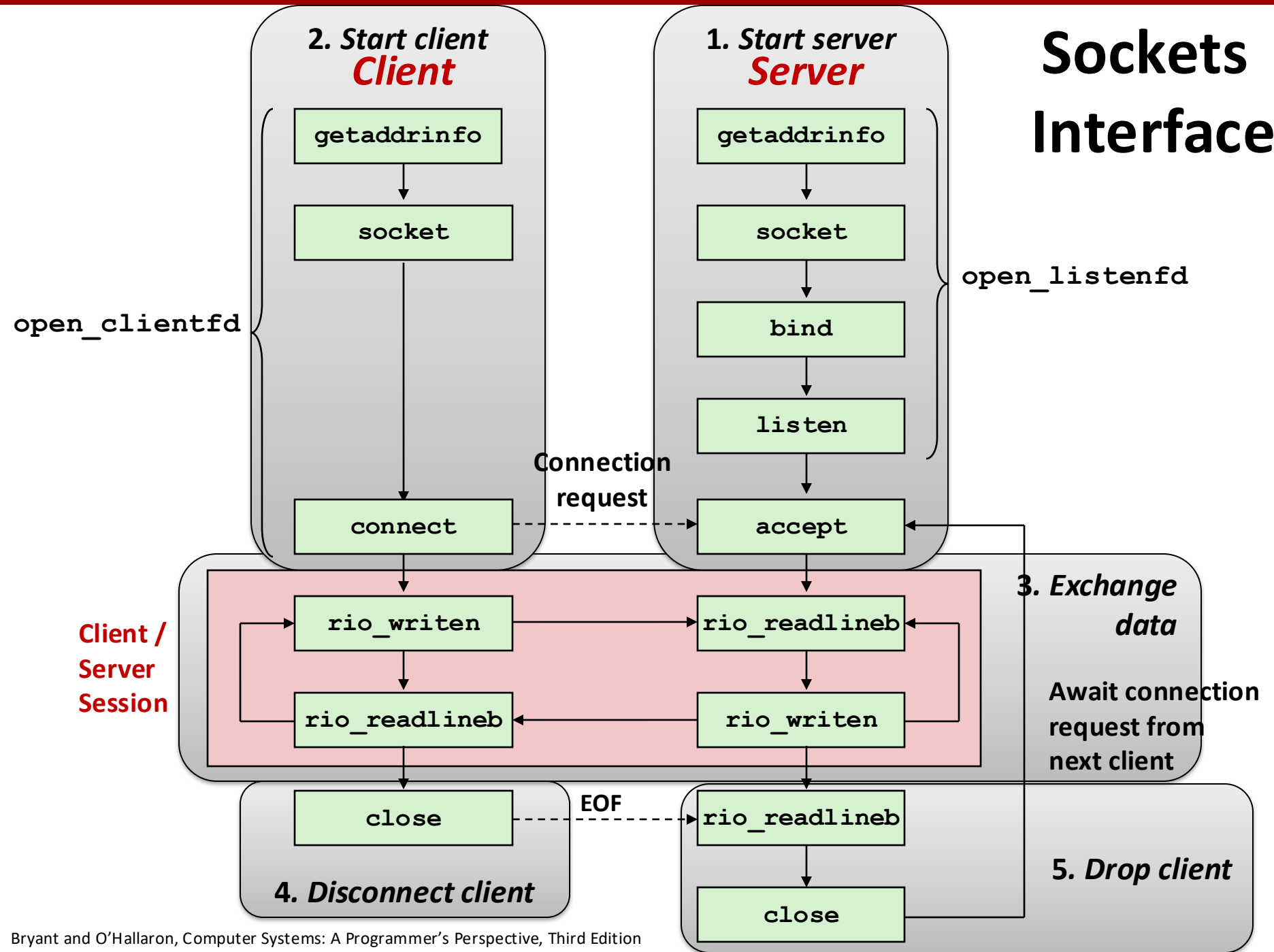
Course Update

➢ **Assignment 1 – Marking now, due in about 2 weeks...**

➢ **Quiz 2 - Due today!**
  - ➢ Cover weeks 3-6 of lectures
  - ➢ 30 questions in 30 mins

➢ **Checkpoint 2 - Released Friday 27 September**
  - ➢ Due Thursday 10 October

Network Programming: Part II

Acknowledgement of material: With changes suited to ANU needs, the slides are obtained from Carnegie Mellon University: https://www.cs.cmu.edu/~213/

Sockets Interface

2. *Start client*
Client
- getaddrinfo
- socket
- connect

open_clientfd

1. *Start server*
Server
- getaddrinfo
- socket
- bind
- listen
- accept

open_listenfd

Connection request

3. *Exchange data*

Client / Server Session
- rio_writen → rio_readlineb
- rio_readlineb ← rio_writen

Await connection request from next client

4. *Disconnect client*
- close

EOF

5. *Drop client*
- rio_readlineb
- close

Recall: Socket Address Structures

- **Generic socket address:**
  - For address arguments to `connect`, `bind`, and `accept`
  - Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed
  - For casting convenience, we adopt the Stevens convention:
    `typedef struct sockaddr SA;`

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

`sa_family`
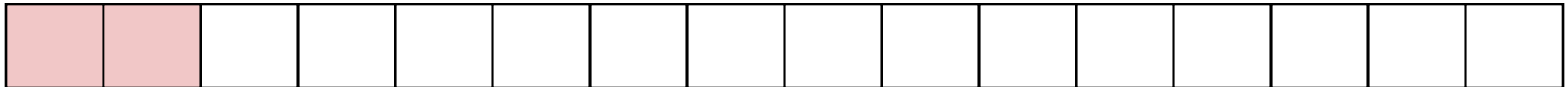
Family Specific

Recall: Socket Address Structures

- **Internet-specific socket address:**
  - Must cast `(struct sockaddr_in *)` to `(struct sockaddr *)` for functions that take socket address arguments.
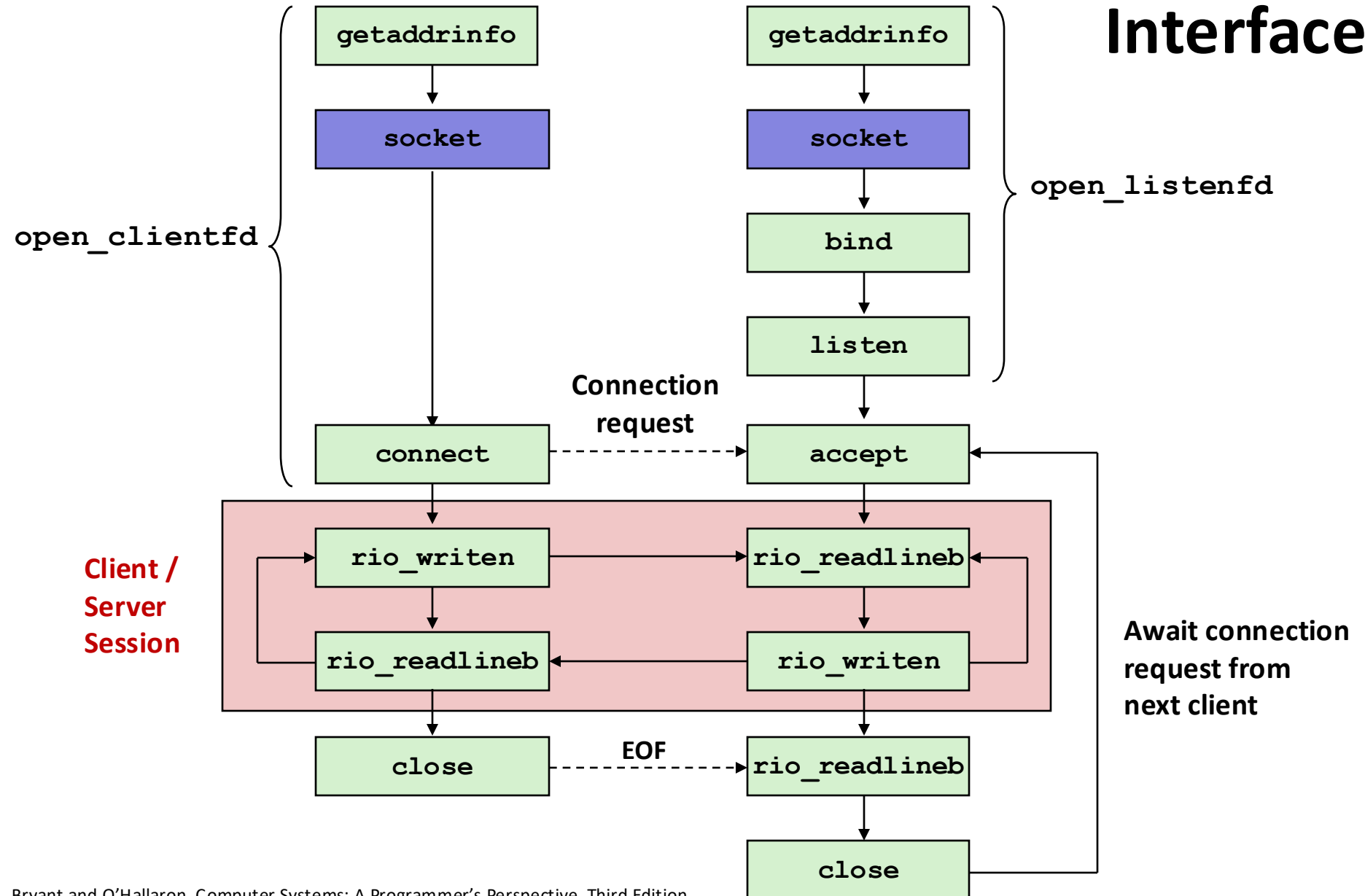
```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```


sin_port **sin_addr**

| AF_INET | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

sa_family

sin_family

Family Specific
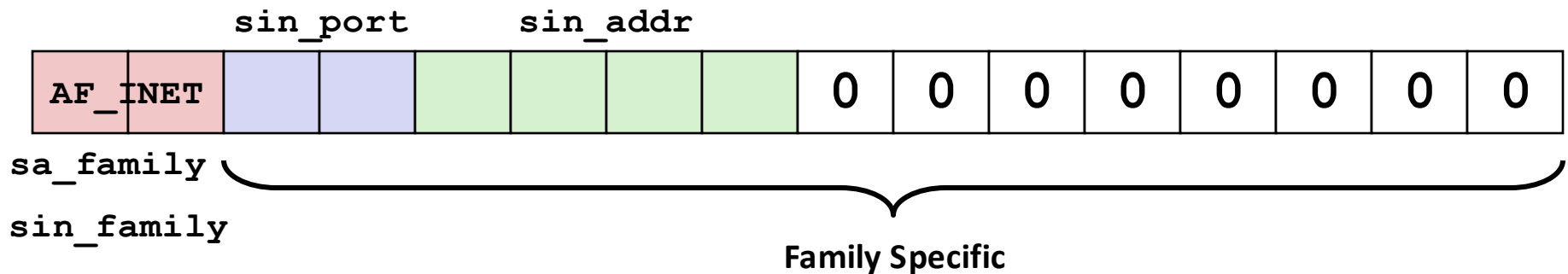
Sockets Interface

Client

Server

Sockets Interface: `socket`

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

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

- **Example:**

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

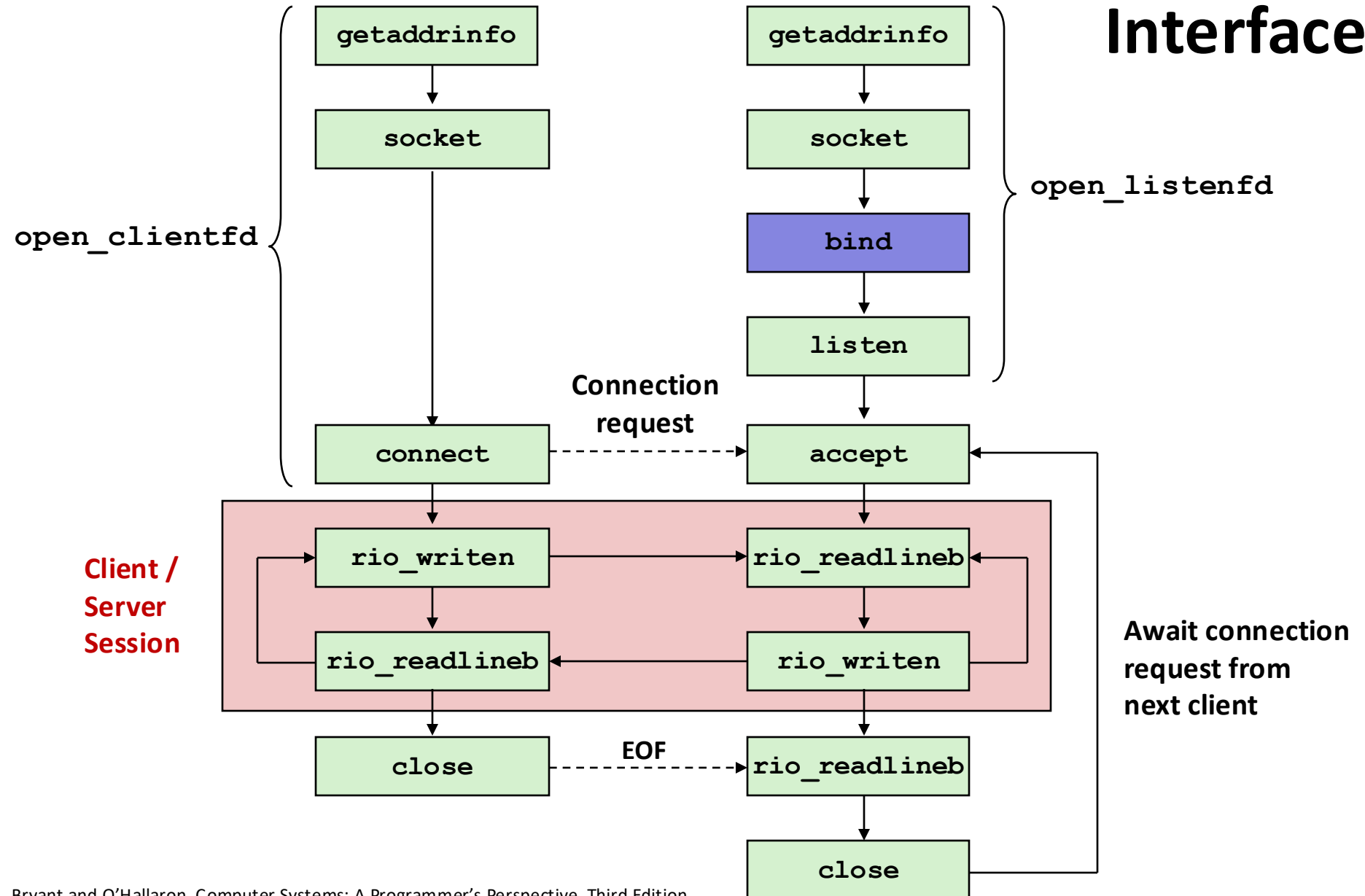**Indicates that we are using 32-bit IPV4 addresses**

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

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

Sockets Interface

Client — *Server*

Client:
- getaddrinfo
- socket
- connect
- rio_writen
- rio_readlineb
- close

open_clientfd (covers getaddrinfo, socket, connect)

Server:
- getaddrinfo
- socket
- bind
- listen
- accept
- rio_readlineb
- rio_writen
- rio_readlineb
- close

open_listenfd (covers getaddrinfo, socket, bind, listen)

Connection request (connect → accept)

Client / Server Session (rio_writen ↔ rio_readlineb, rio_readlineb ↔ rio_writen)

Await connection request from next client

EOF (close → rio_readlineb)
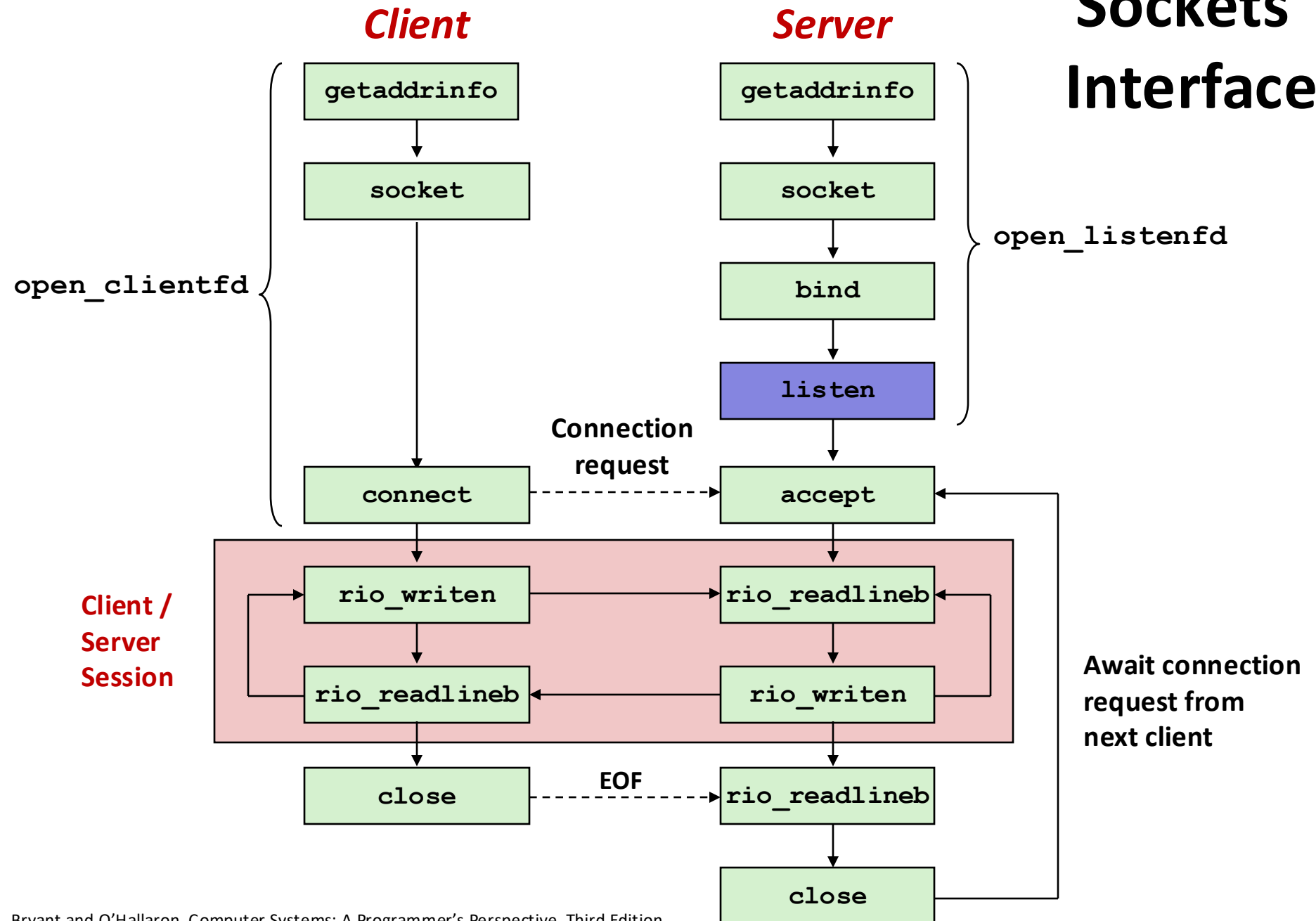
Sockets Interface: `bind`

- **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);
```

- **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

Client

Server

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

11

Sockets Interface: `listen`

- **By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection.**

- **A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:**
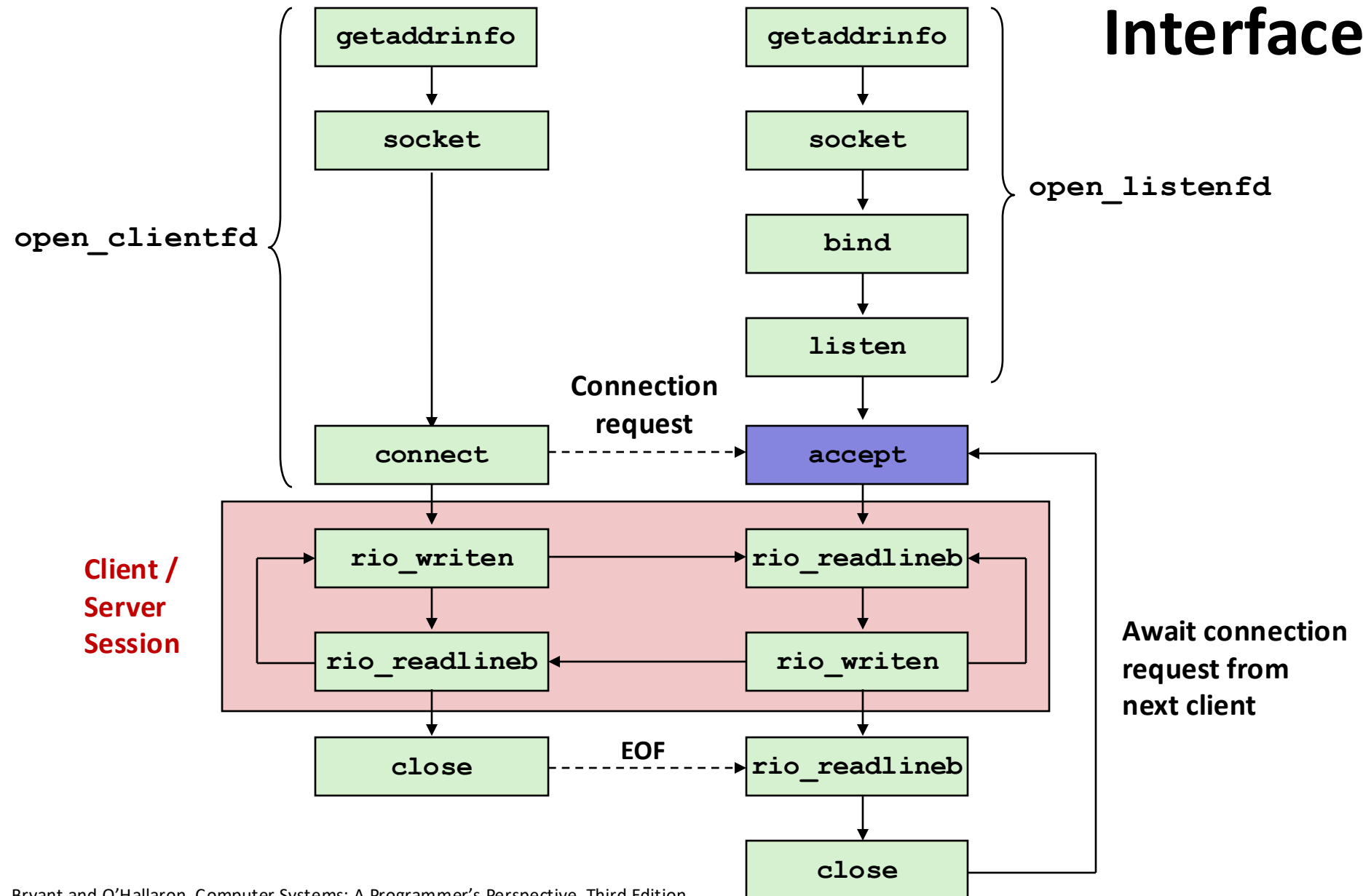
```
int listen(int sockfd, int backlog);
```

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

- **`backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.**

Sockets Interface

Client

Server

```
getaddrinfo
```
↓
```
socket
```
↓
```
connect
```

```
getaddrinfo
```
↓
```
socket
```
↓
```
bind
```
↓
```
listen
```
↓
```
accept
```

open_clientfd

open_listenfd

Connection request

Client / Server Session

```
rio_writen
```
↓
```
rio_readlineb
```
↓
```
close
```

```
rio_readlineb
```
↓
```
rio_writen
```
↓
```
rio_readlineb
```
↓
```
close
```

EOF

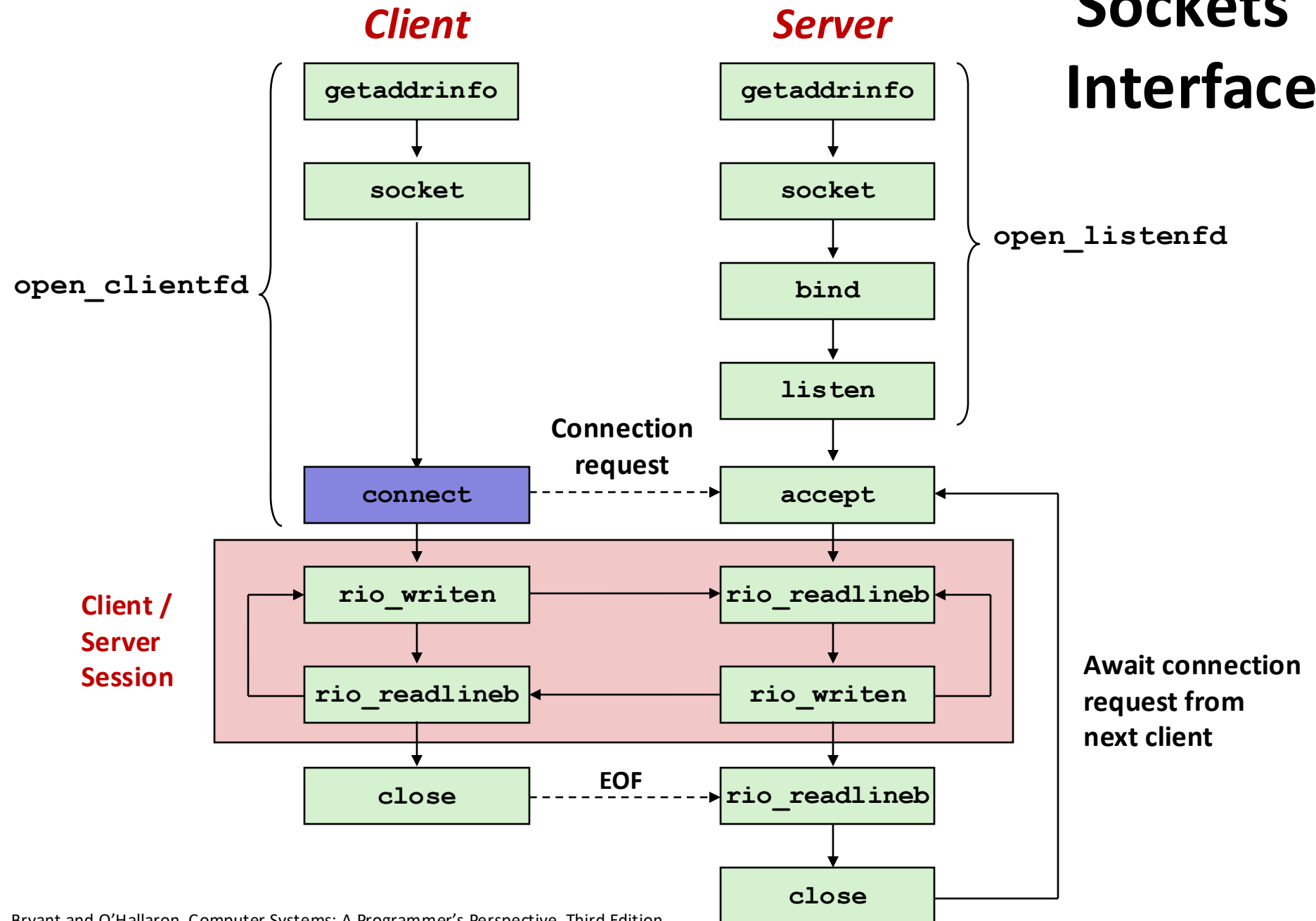**Await connection request from next client**

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

Client

Server

open_clientfd

open_listenfd

- getaddrinfo → socket → connect
- getaddrinfo → socket → bind → listen → accept

Connection request

connect - - - - - → accept

Client / Server Session

- rio_writen → rio_readlineb
- rio_readlineb ← rio_writen

Await connection request from next client

close - - - - - **EOF** - - - - → rio_readlineb → close

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

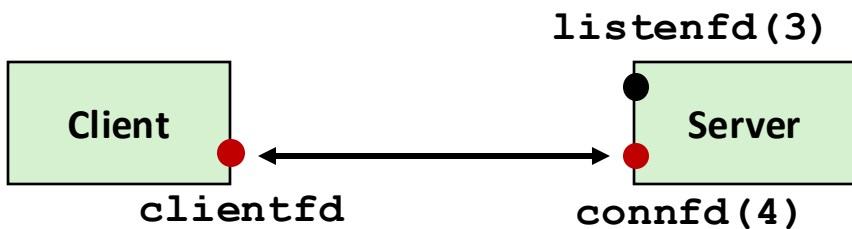**Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.**

`accept` Illustrated

listenfd(3)

Client

clientfd

Server

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

Connection request

listenfd(3)

Client
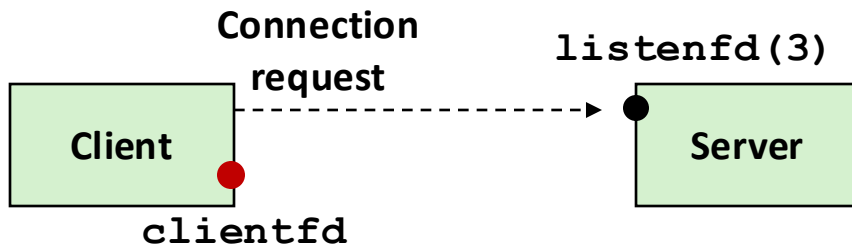
clientfd

Server

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

---

listenfd(3)

Client

clientfd   **connfd(4)**

Server

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
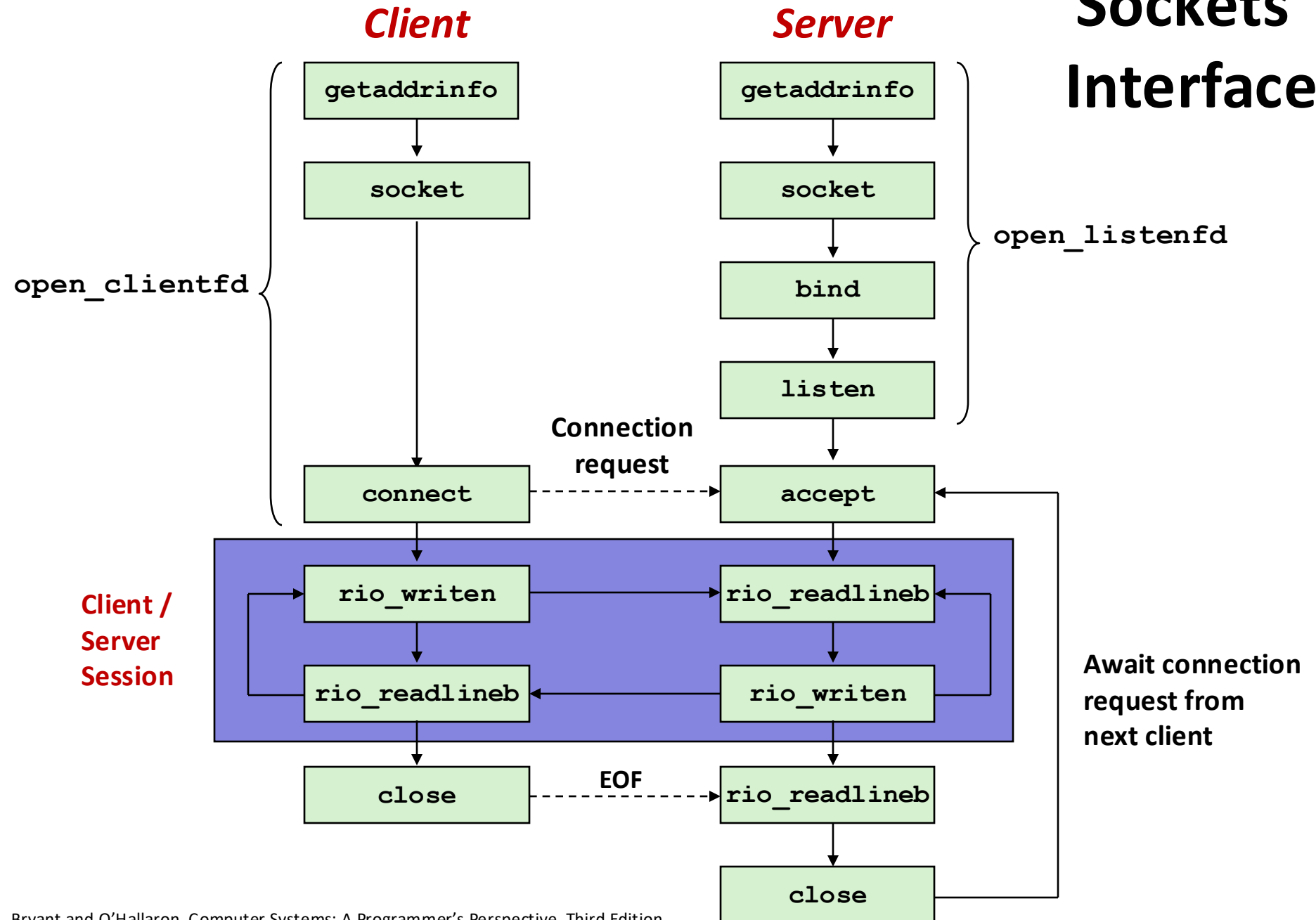 - Created once and exists for lifetime of the server

- **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
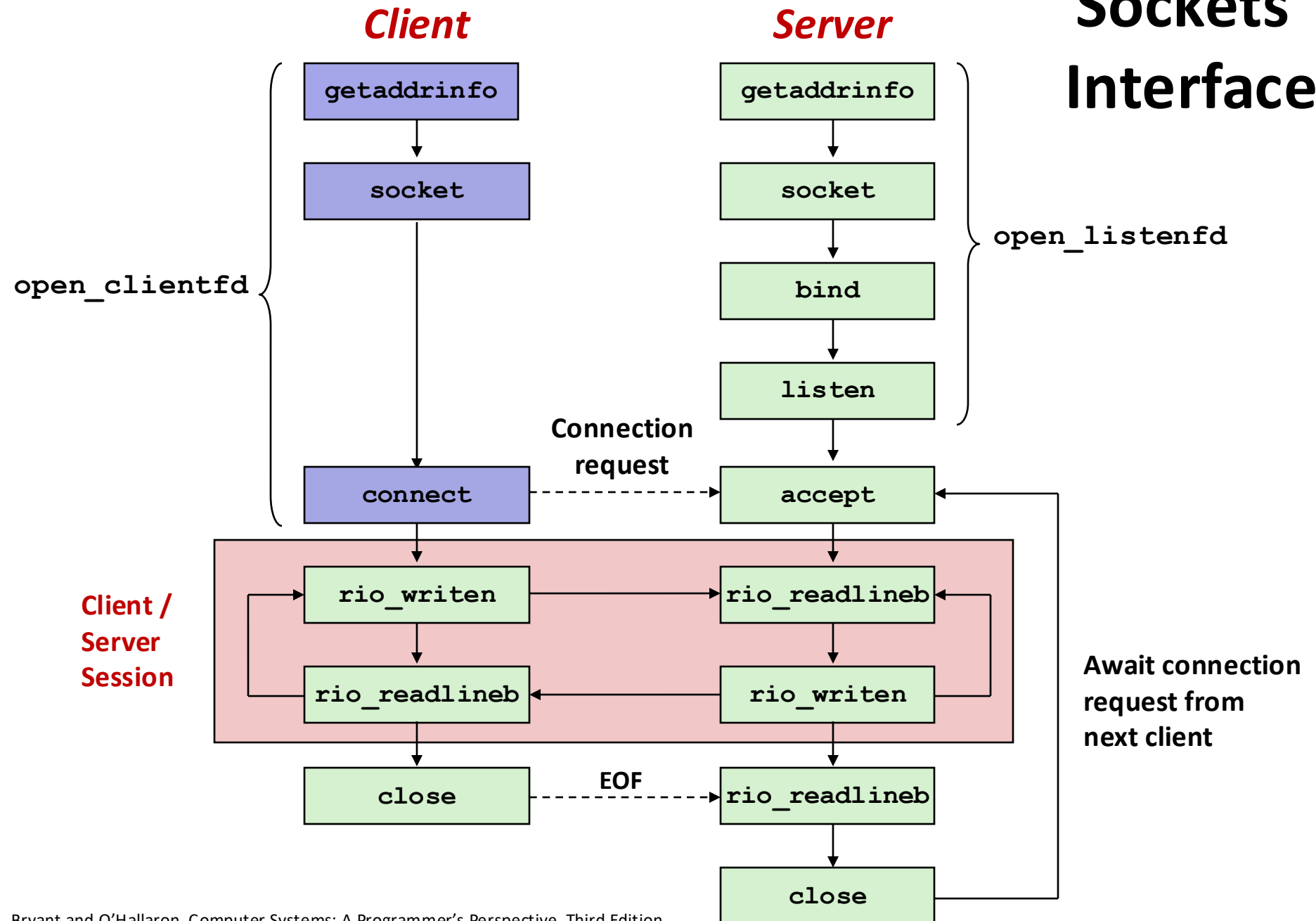 - Exists only as long as it takes to service client

- **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

Client

Server

open_clientfd

open_listenfd

Connection request

Client / Server Session

Await connection request from next client

EOF

getaddrinfo → socket → connect → rio_writen → rio_readlineb → close

getaddrinfo → socket → bind → listen → accept → rio_readlineb → rio_writen → rio_readlineb → close

Sockets Interface

Client

```
getaddrinfo
```
↓
```
socket
```
↓

open_clientfd

```
connect
```

Server

```
getaddrinfo
```
↓
```
socket
```
↓
```
bind
```
↓
```
listen
```

open_listenfd

Connection request

```
accept
```

Client / Server Session

```
rio_writen
```
→
```
rio_readlineb
```
↓

```
rio_readlineb
```
←
```
rio_writen
```

Await connection request from next client

```
close
```
- - - **EOF** - - →
```
rio_readlineb
```
↓
```
close
```

Sockets Helper: `open_clientfd`

- **Establish a connection with a server**

```c
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

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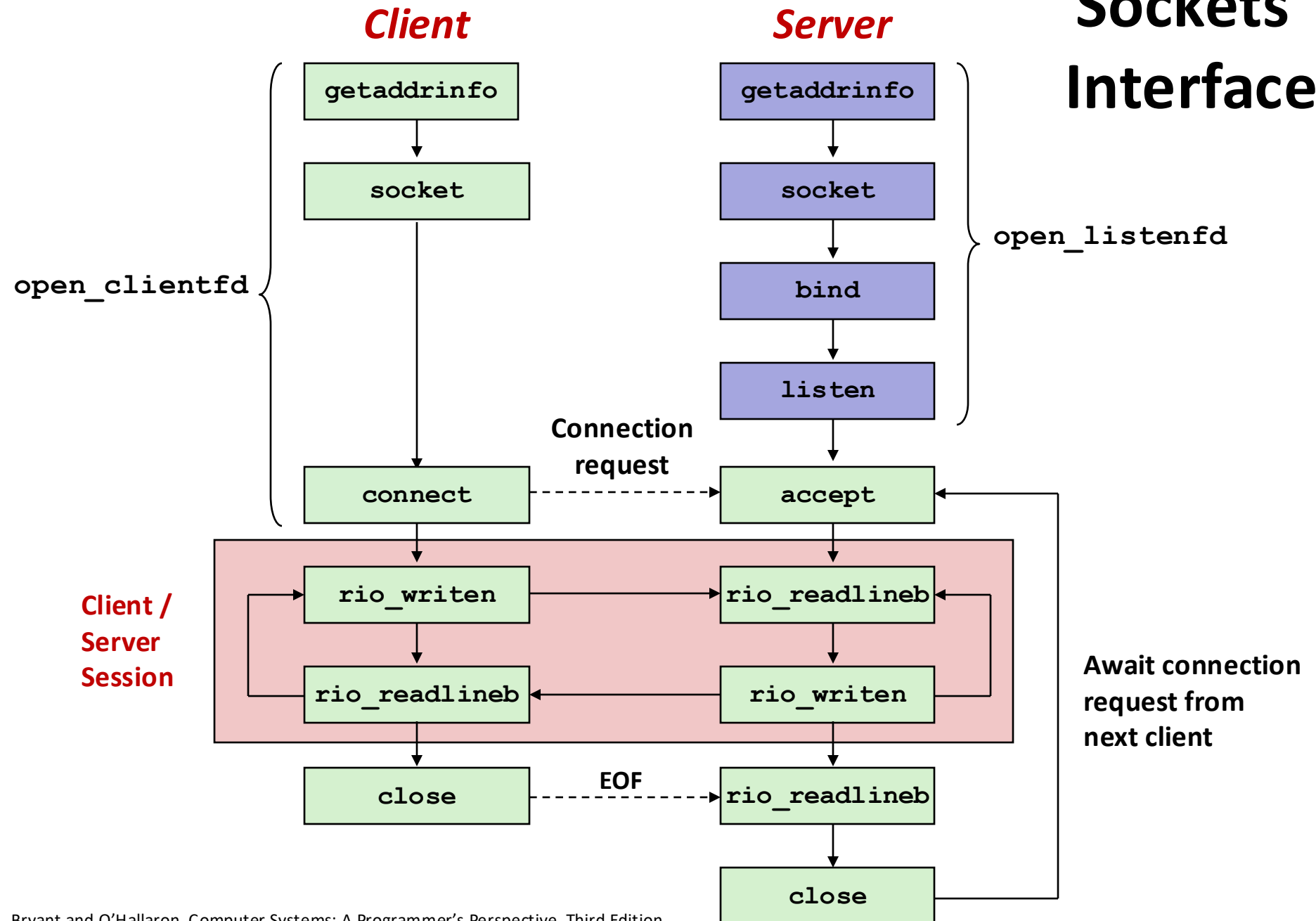
```c
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket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

Client

```
getaddrinfo
     │
     ▼
   socket
     │
     ▼
  connect
```

open_clientfd

Server

```
getaddrinfo
     │
     ▼
   socket
     │
     ▼
    bind
     │
     ▼
   listen
```

open_listenfd

Connection request

connect ┄┄┄┄┄┄┄┄┄► accept

Client / Server Session

rio_writen ──────►	rio_readlineb
rio_readlineb ◄──────	rio_writen

close ┄┄┄ **EOF** ┄┄┄► rio_readlineb

close

Await connection request from next client

Sockets Helper: `open_listenfd`

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

```c
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)

```c
    /* 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)

```c
    /* 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;
}
```
<div align="right">csapp.c</div>

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

Echo Client: Main Routine

```c
#include "csapp.h"

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

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

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```
echoclient.c

Iterative Echo Server: Main Routine

```c
#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);
}
```

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)`

```c
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
```

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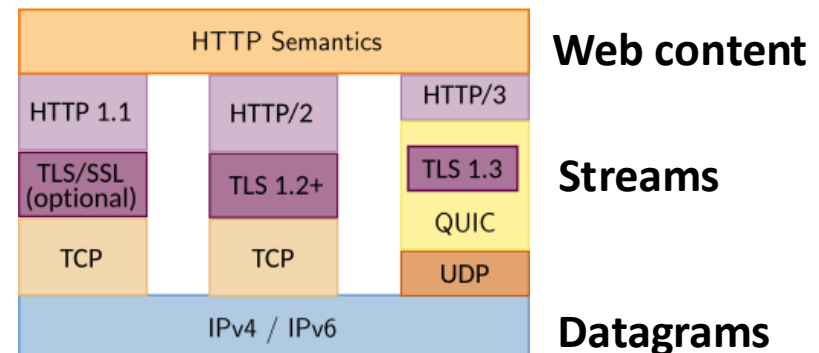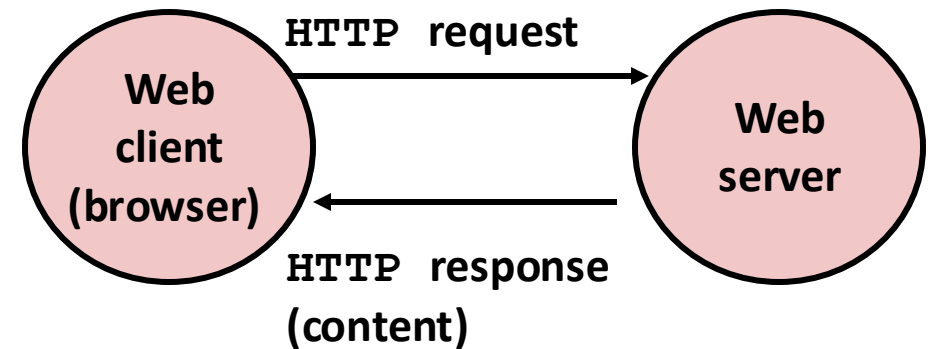 - Creates a connection with a server running on **`<host>`** and listening on port **`<portnumber>`**

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/QUIC connection
  - Client requests content
  - Server responds with requested content
  - Client and server close connection (eventually)
- **Current version is HTTP/3**
  - RFC 9114 in 2022. HTTP/3
  - HTTP semantics are consistent across versions

HTTP request → Web server

Web client (browser) ← HTTP response (content)

Web content

Streams

Datagrams

https://en.wikipedia.org/wiki/HTTP/3

`https://www.rfc-editor.org/rfc/rfc9114`

Web Content

- **Web servers return *content* to clients**
  - *content:* a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

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

You can find the complete list of MIME types at:
`http://www.iana.org/assignments/media-types/media-types.xhtml`

Static and Dynamic Content

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

URLs and how clients and servers use them

- **Unique name for a file: URL (Universal Resource Locator)**
- **Example URL: `https://www.anu.edu:443/index.html`**
- **Clients use *prefix* (`https://www.anu.edu:443`) to infer:**
  - What kind (protocol) of server to contact (HTTPS)
  - Where the server is (`www.anu.edu`)
  - What port it is listening on (443)
- **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 (e.g `HTTP/3.0` 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
```

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);
}
```
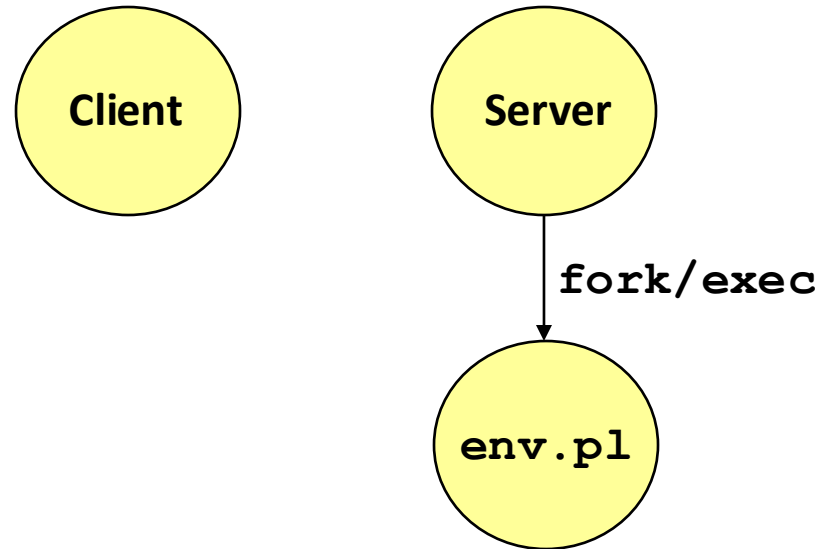
tiny.c

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`

Client → Server
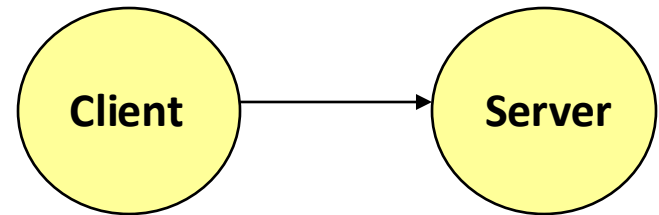
Serving Dynamic Content (cont)

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

Client

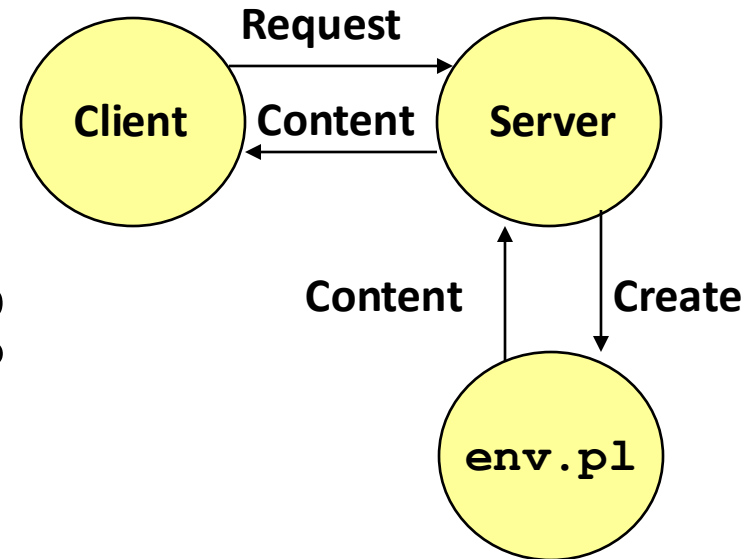**Server**

fork/exec

env.pl

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**
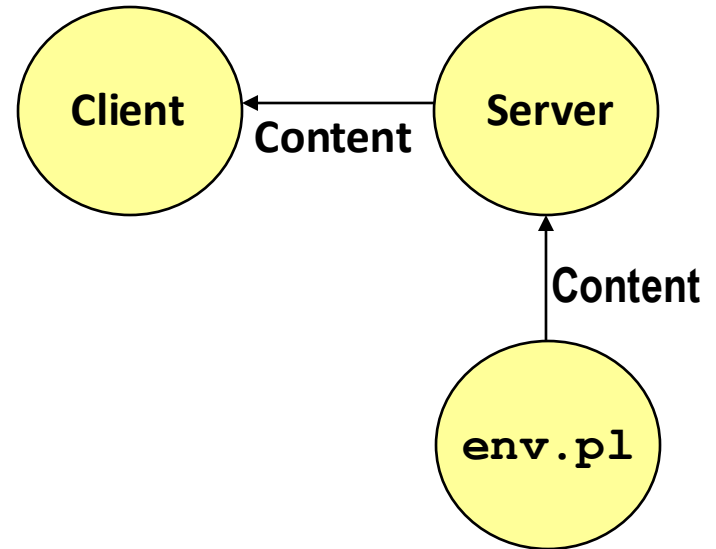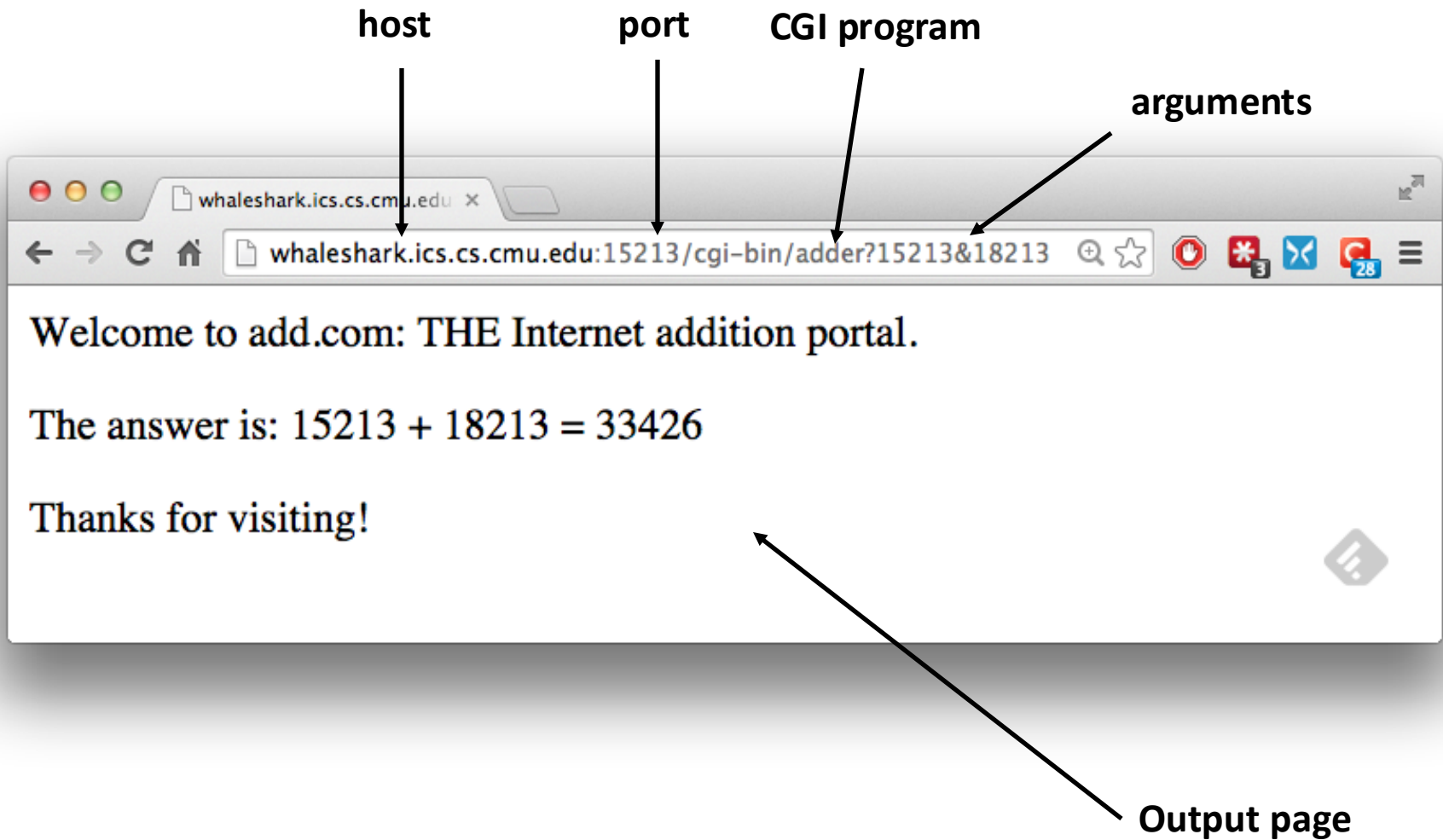
Issues in Serving Dynamic Content

- **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.**

```
                Request
    Client  --------->  Server
            <---------
                Content
                            Content |  | Create
                                    |  v
                         env.pl
```

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).

A CGI Program

host   port   CGI program

arguments

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

- **<u>Question:</u> How does the client pass arguments to the server?**

- **<u>Answer:</u> The arguments are appended to the URI**

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

Serving Dynamic Content With GET

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

- **Result displayed on browser:**

> **Welcome to add.com: THE Internet addition portal.**
>
> **The answer is: 15213 + 18213 = 33426**
>
> **Thanks for visiting!**

Serving Dynamic Content With GET

- **Question: How does the server pass these arguments to the child?**

- **Answer: In environment variable QUERY_STRING**
  - A single string containing everything after the "?"
  - For add: `QUERY_STRING` = "`15213&18213`"

```c
/* 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.**

```c
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

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       First Chunk: 0xd75 = 3445 bytes
<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          Second Chunk: 0 bytes (indicates last chunk)
\r\n
```
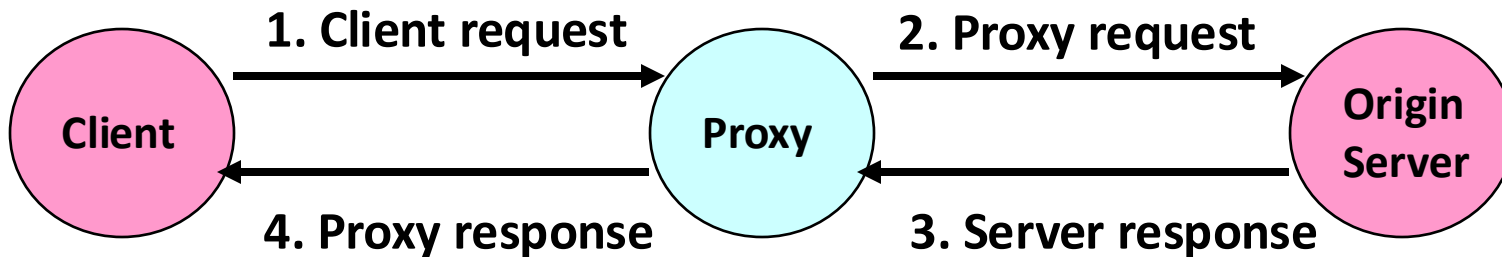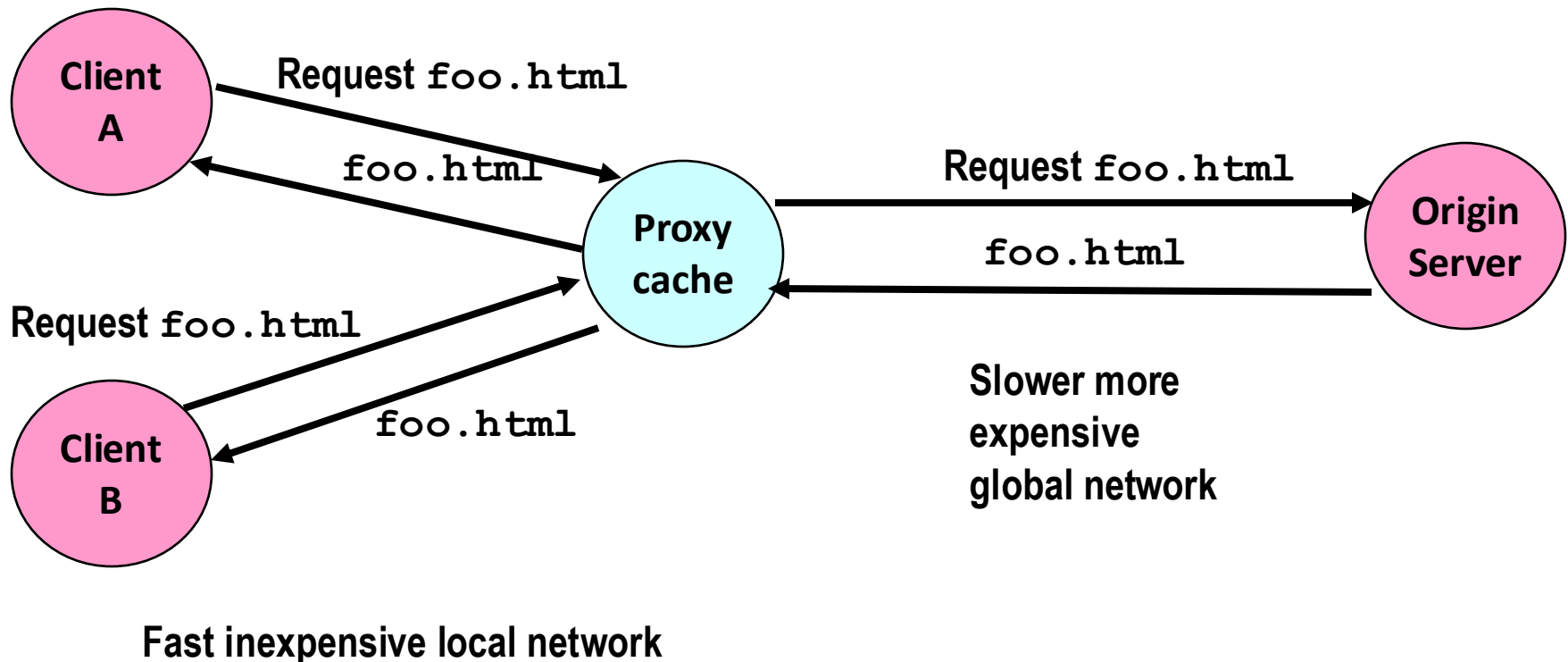
Proxies

- **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

1. Client request   **2. Proxy request**

Client   **Proxy**   **Origin Server**

4. Proxy response   **3. Server response**

Why Proxies?

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

Fast inexpensive local network

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, 2017**
  - 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.