

# Network Programming: Part II

15-213 / 18-213: Introduction to Computer Systems  
“22<sup>nd</sup>” Lecture, July 24, 2019

**Instructor:**

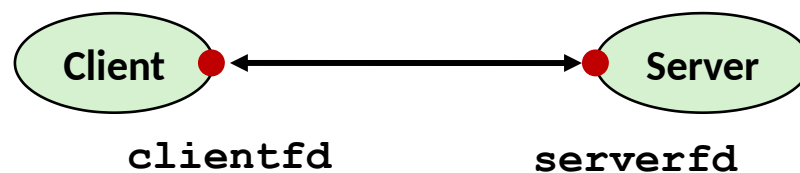
Sol Boucher

# Sockets Interface

- **Set of system-level functions used in conjunction with Unix I/O to build network applications.**
- **Created in the early 80s 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

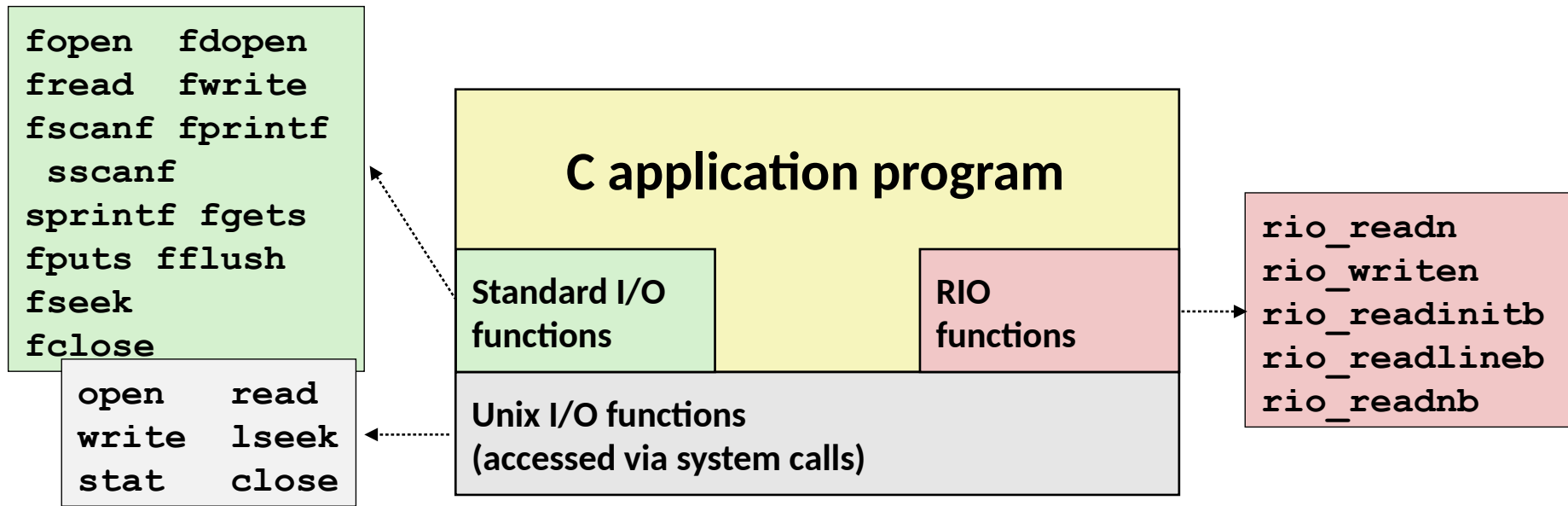
- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - **Remember:** All Unix I/O devices, including networks, are modeled as files
- **Clients and servers 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**

# Recall: C Standard I/O, Unix I/O and RIO

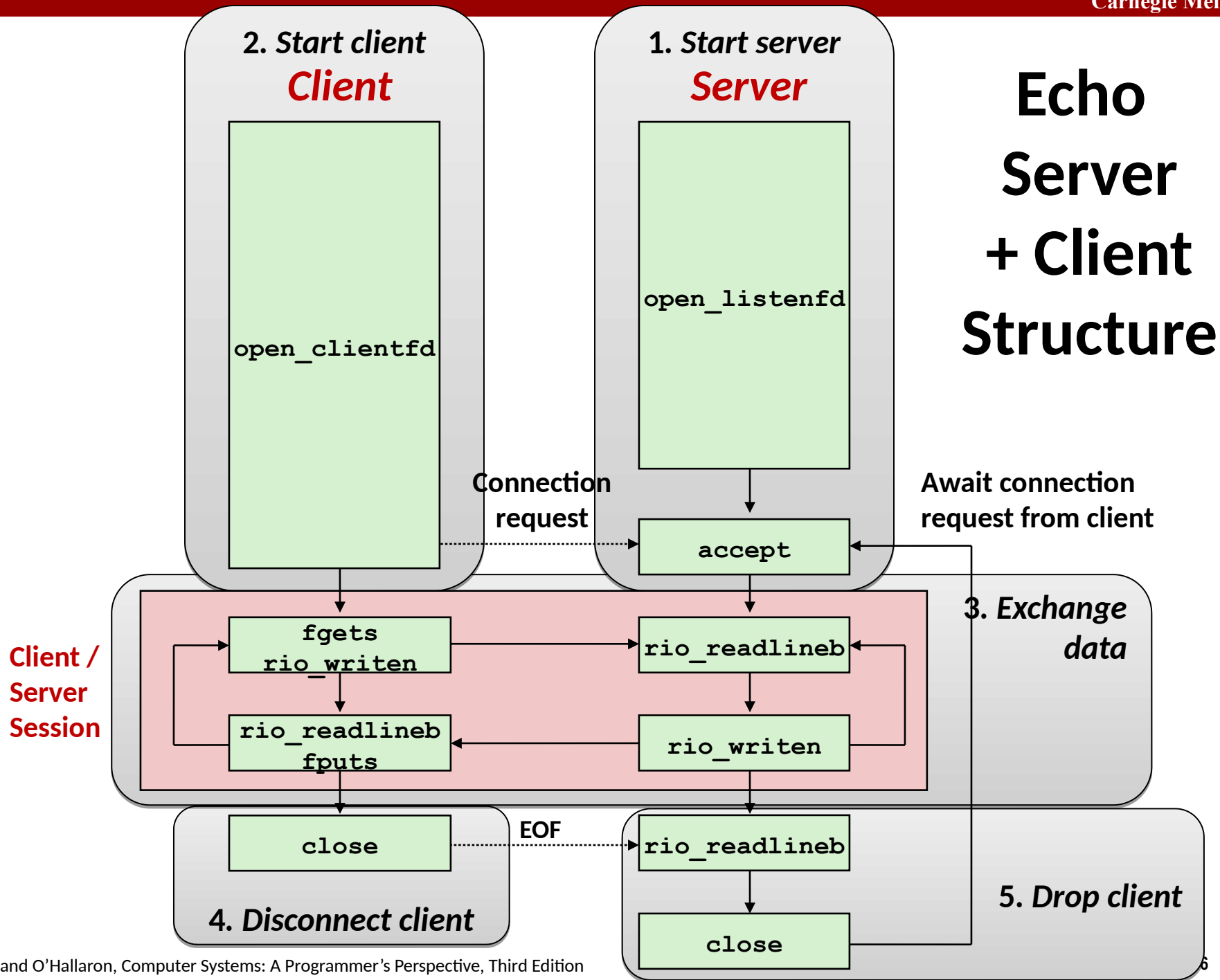
- Robust I/O (RIO): 15-213 special wrappers**  
**good coding practice:** handles error checking, signals, and  
 “short counts”



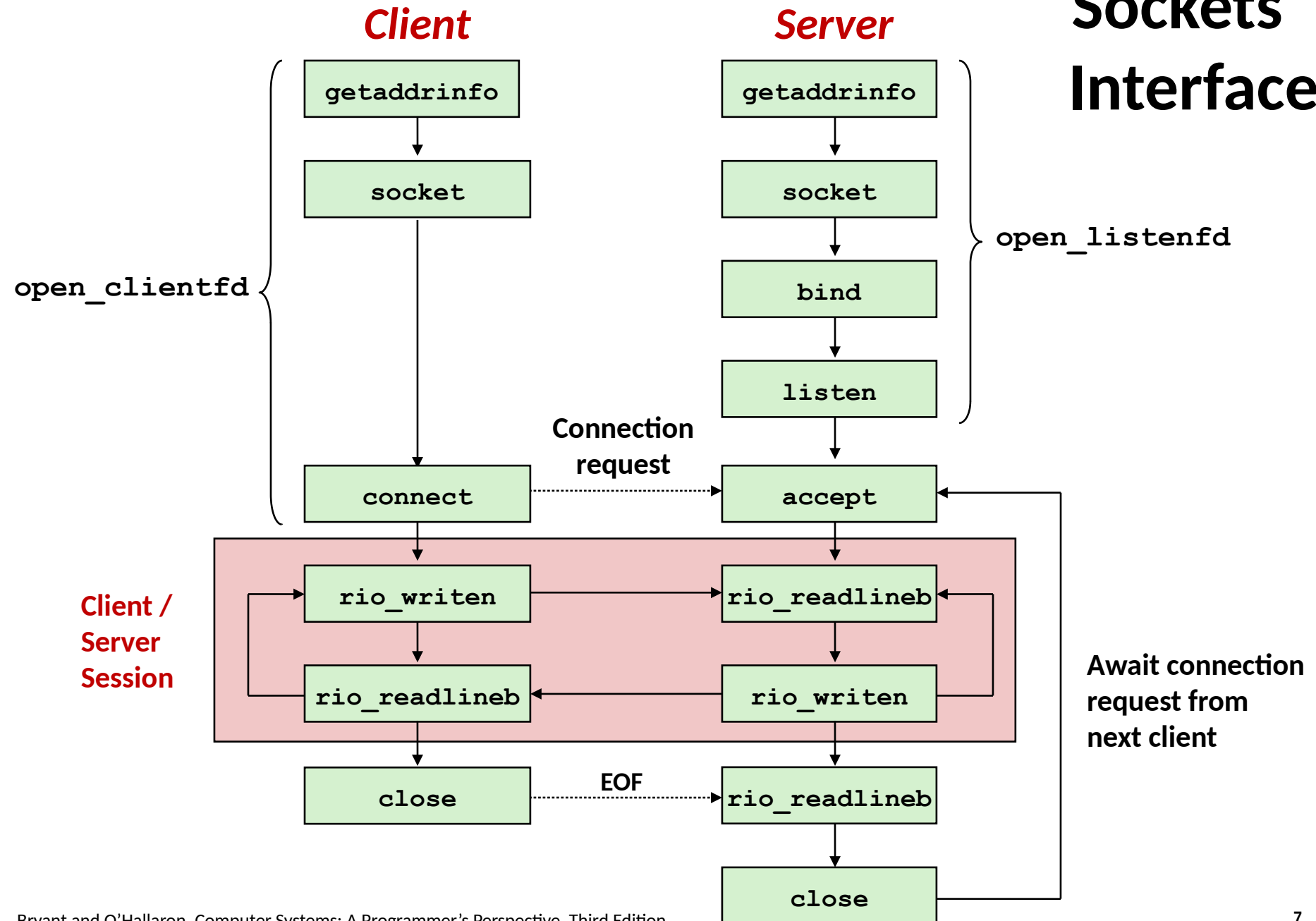
# Socket Programming Example

**Coding demo**

- **Echo server and client**
- **Server**
  - Accepts connection request
  - Repeats back lines as they are typed
- **Client**
  - Requests connection to server
  - Repeatedly:
    - Read line from terminal
    - Send to server
    - Read reply from server
    - Print line to terminal



# Sockets Interface



# Today

## ■ Addresses

- Structures
- String conversions
- DNS

## ■ Sockets and ports

- Creating and associating sockets
- Opening ports

## ■ Connections



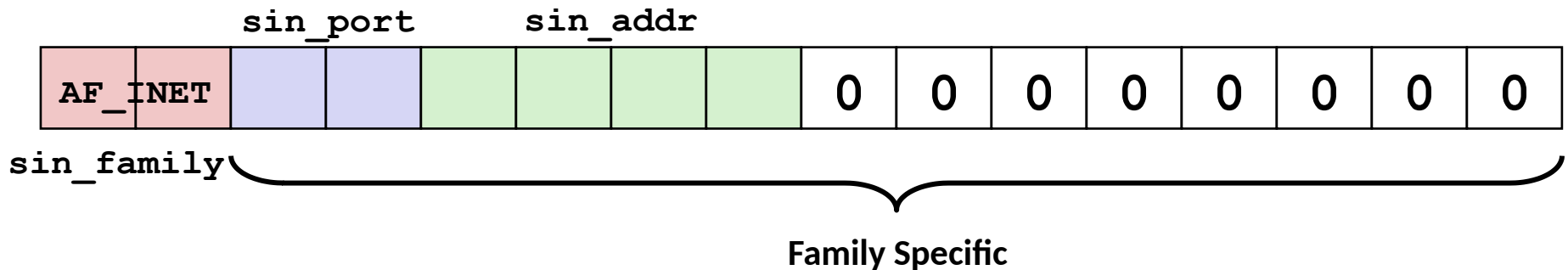
# Socket Address Structures

- 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) */
};

```



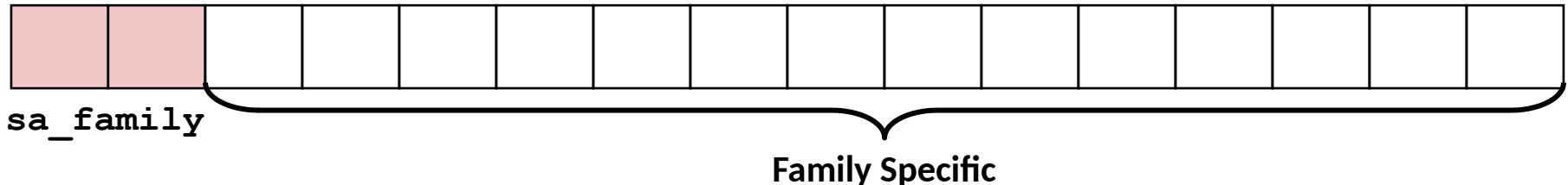
# Socket Address Structures & `getaddrinfo`

## ■ 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:

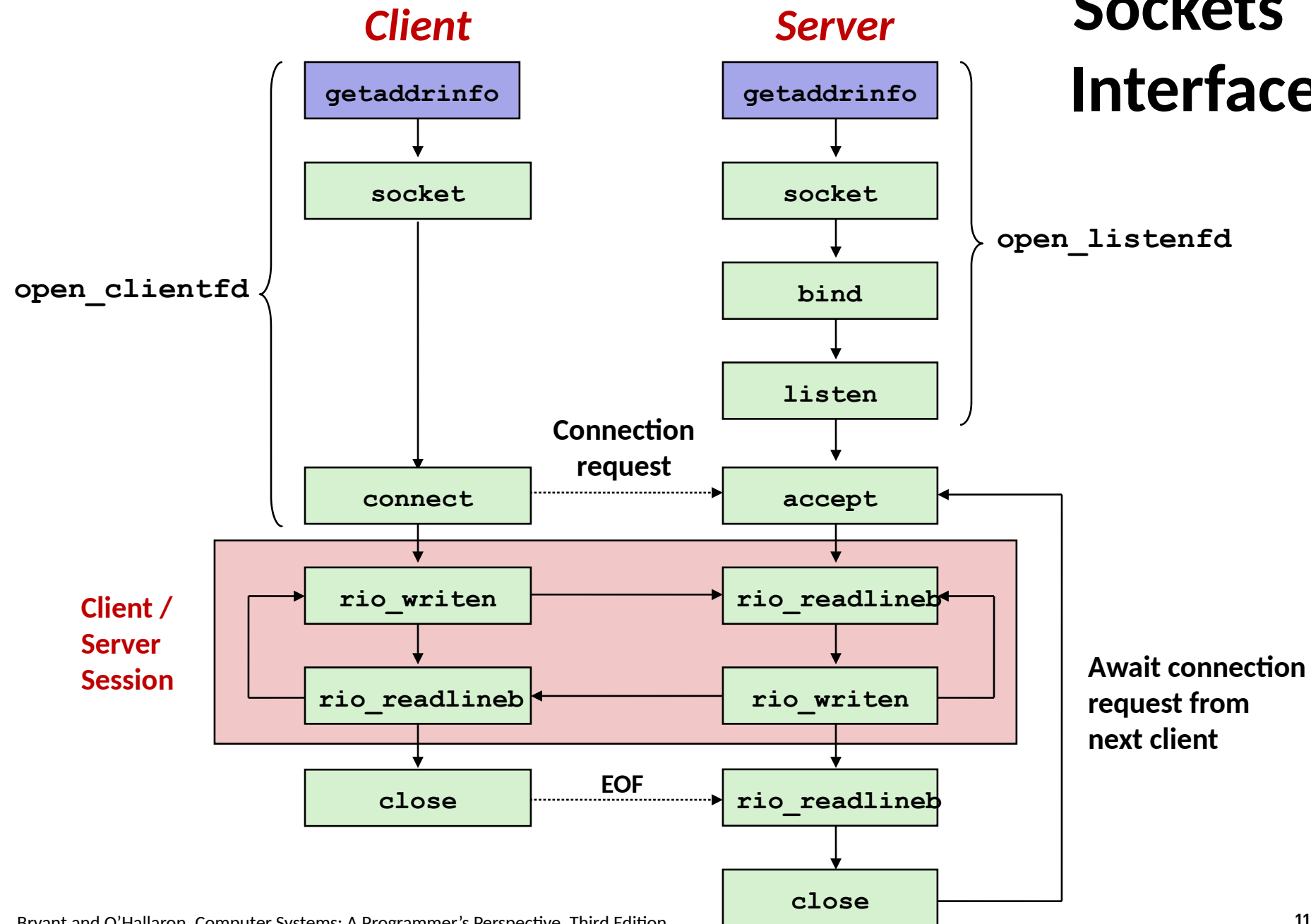
```
typedef struct sockaddr SA;
```

```
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

# Sockets Interface



# Host and Service Conversion: `getaddrinfo`

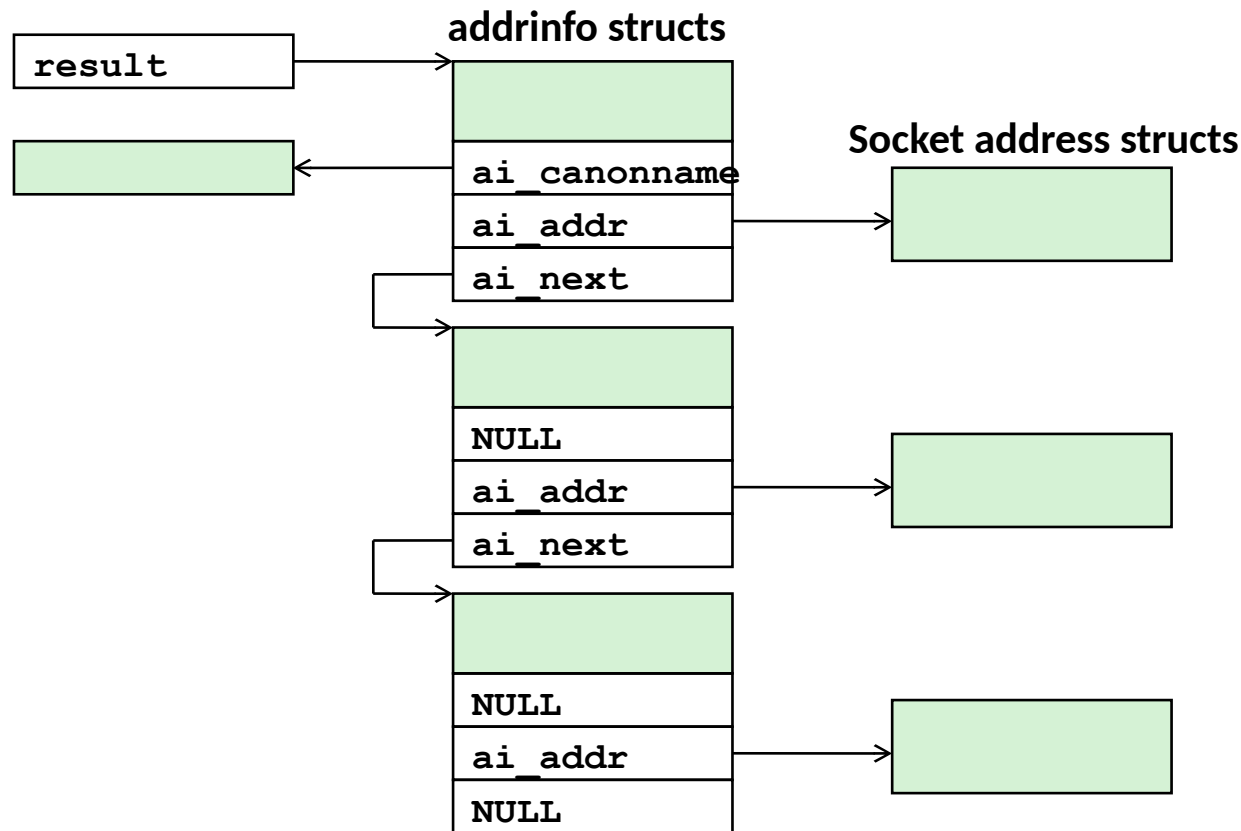
```
int getaddrinfo(const char *host,          /* Hostname or address */
               const char *service,      /* Port or service name */
               const struct addrinfo *hints, /* Input parameters */
               struct addrinfo **result); /* Output linked list */

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:**
  - `freeaddrinfo` frees the entire linked list.
  - `gai_strerror` converts error code to an error message.

# 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.

# Host and Service Conversion: `getnameinfo`

- `getnameinfo` is the inverse of `getaddrinfo`, converting a socket address to the corresponding host and service.
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.
  - Reentrant and protocol independent.

```
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
               char *host, size_t hostlen, /* Out: host */
               char *serv, size_t servlen, /* Out: service */
               int flags); /* optional flags */
```

# Conversion Example

```
#include "csapp.h"

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

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof hints);
    // hints.ai_family = AF_INET;          /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM;     /* TCP only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
}
```

hostinfo.c



# Conversion Example (cont)

```
/* Walk the list and display each IP address */
flags = NI_NUMERICHOST; /* Display address instead of name */
for (p = listp; p; p = p->ai_next) {
    Getnameinfo(p->ai_addr, p->ai_addrlen,
                buf, MAXLINE, NULL, 0, flags);
    printf("%s\n", buf);
}

/* Clean up */
Freeaddrinfo(listp);

exit(0);
}
```

hostinfo.c

# Running hostinfo

```
whaleshark> ./hostinfo localhost  
127.0.0.1
```

```
whaleshark> ./hostinfo whaleshark.ics.cs.cmu.edu  
128.2.210.175
```

```
whaleshark> ./hostinfo twitter.com  
199.16.156.230  
199.16.156.38  
199.16.156.102  
199.16.156.198
```

```
whaleshark> ./hostinfo google.com  
172.217.15.110  
2607:f8b0:4004:802::200e
```

# Today

## ■ Addresses

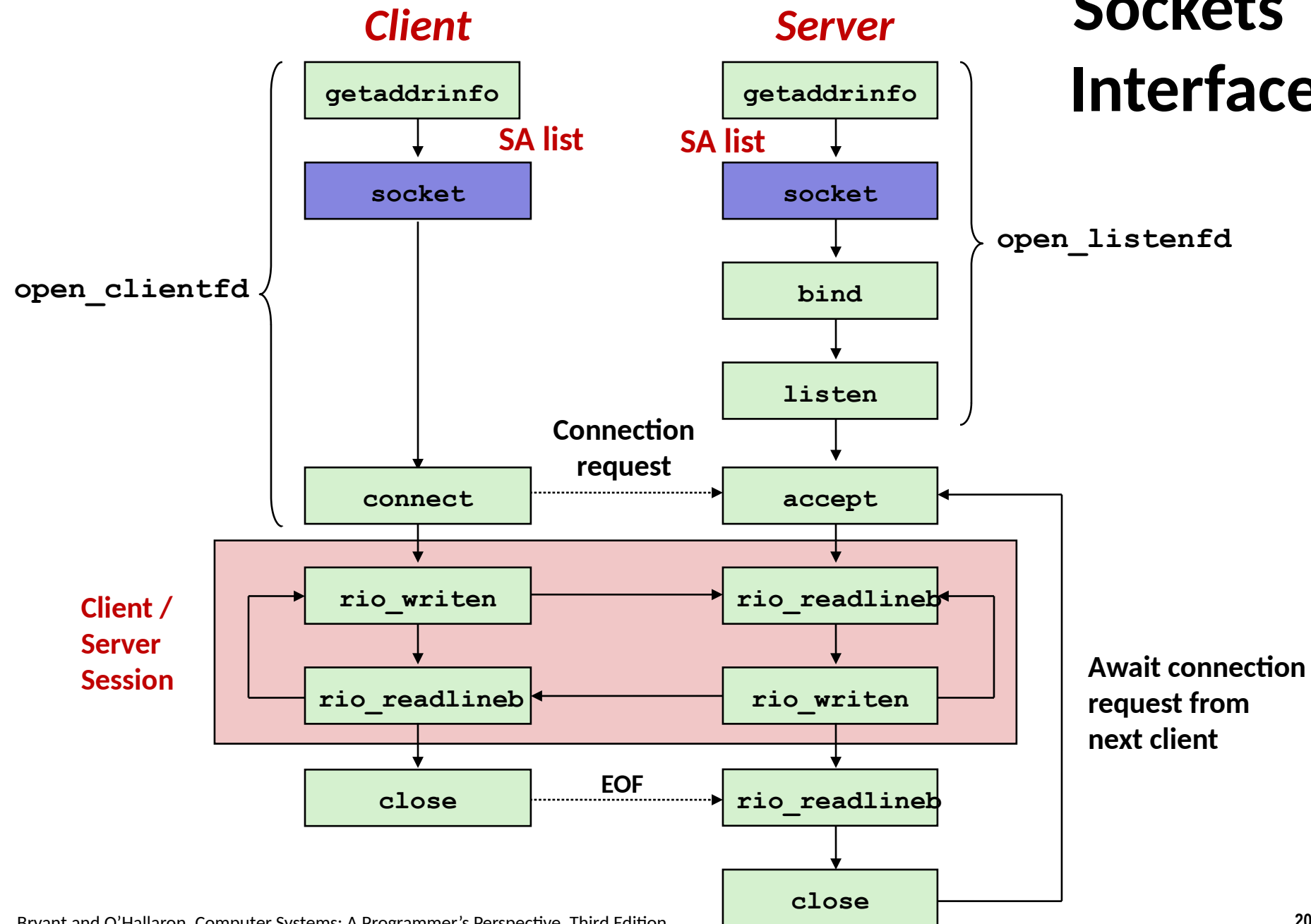
- Structures
- String conversions
- DNS

## ■ Sockets and ports

- Creating and associating sockets
- Opening ports

## ■ Connections

# Sockets Interface



# Sockets Interface: `socket`

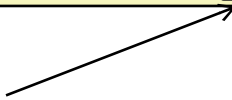
- Clients and servers use the `socket` function to create a *socket descriptor*:

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

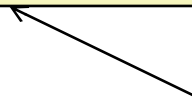
- **Example:**

```
int sockfd = 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



# 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 sockfd = 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

- **Better Example: (It is protocol Independent)**

```
struct addrinfo *p = ...;  
int clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
```

# Sockets Interface: socket

```

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 */
};

```

connection

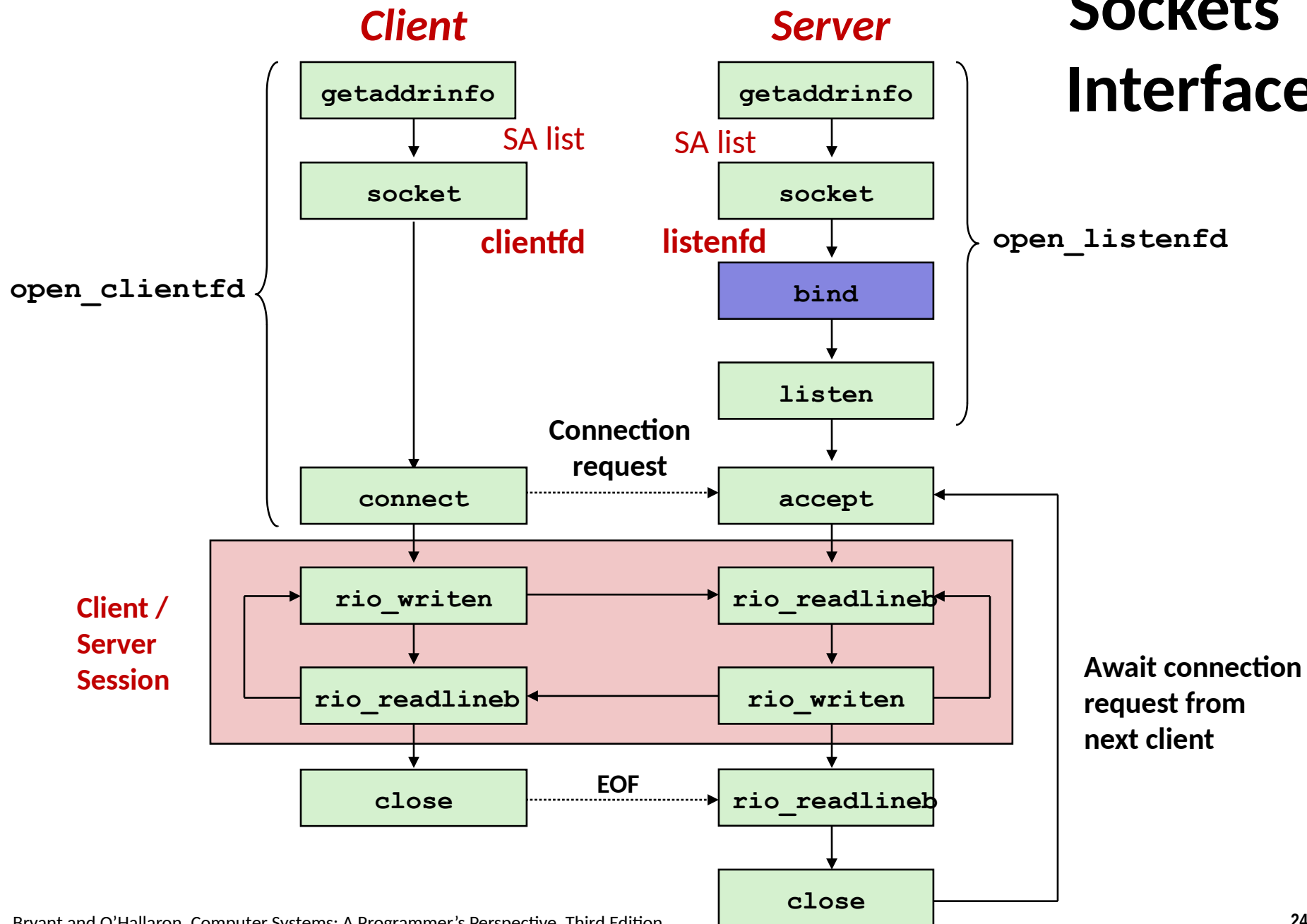
## ■ Better Example: (It is protocol Independent)

```

struct addrinfo *p = ...;
int clientfd = 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`

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

# 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);
```

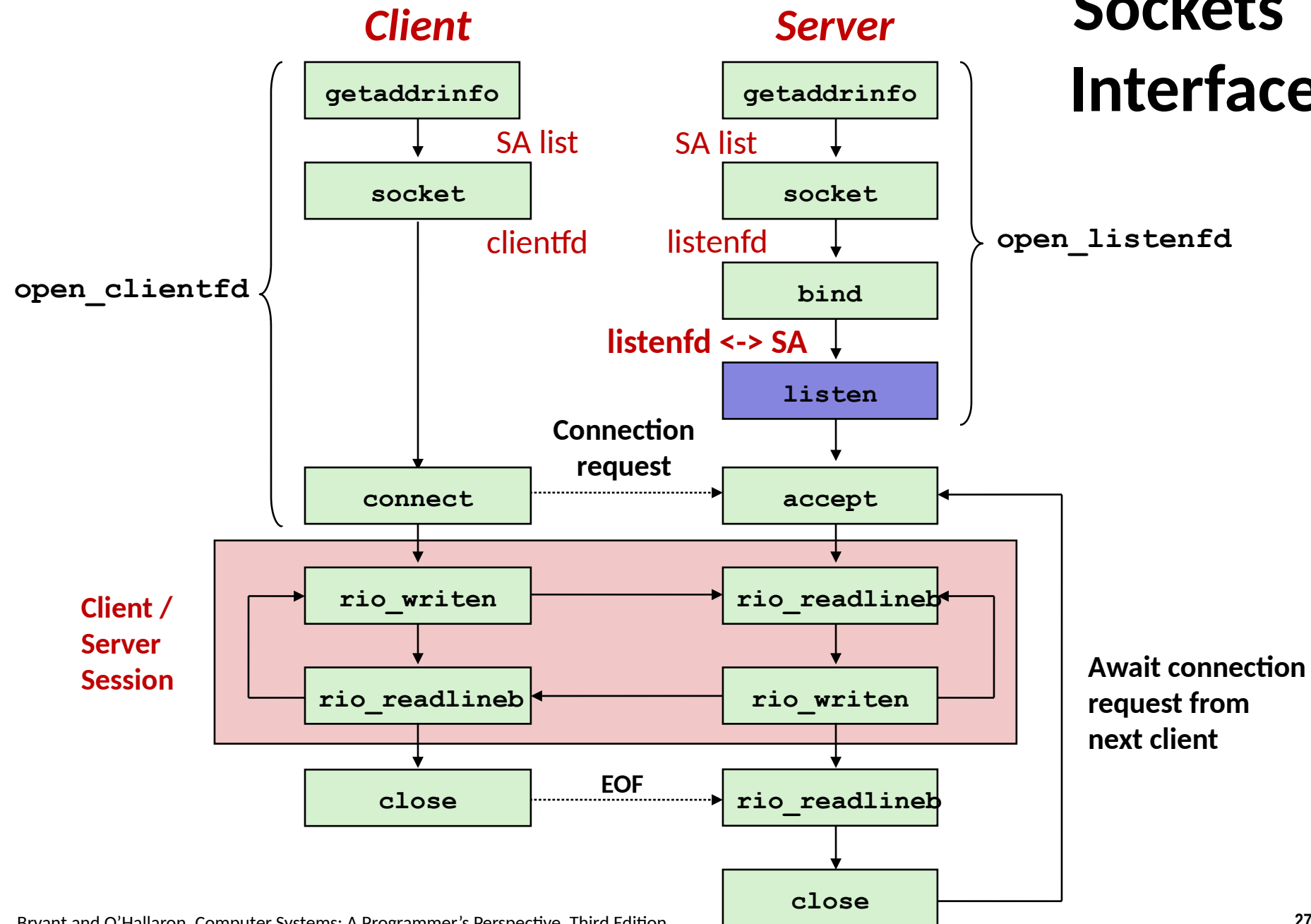
Recall: `typedef struct sockaddr SA;`

- 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

B  
a

```
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 */
};
```

# Sockets Interface



# 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.

# Today

## ■ Addresses

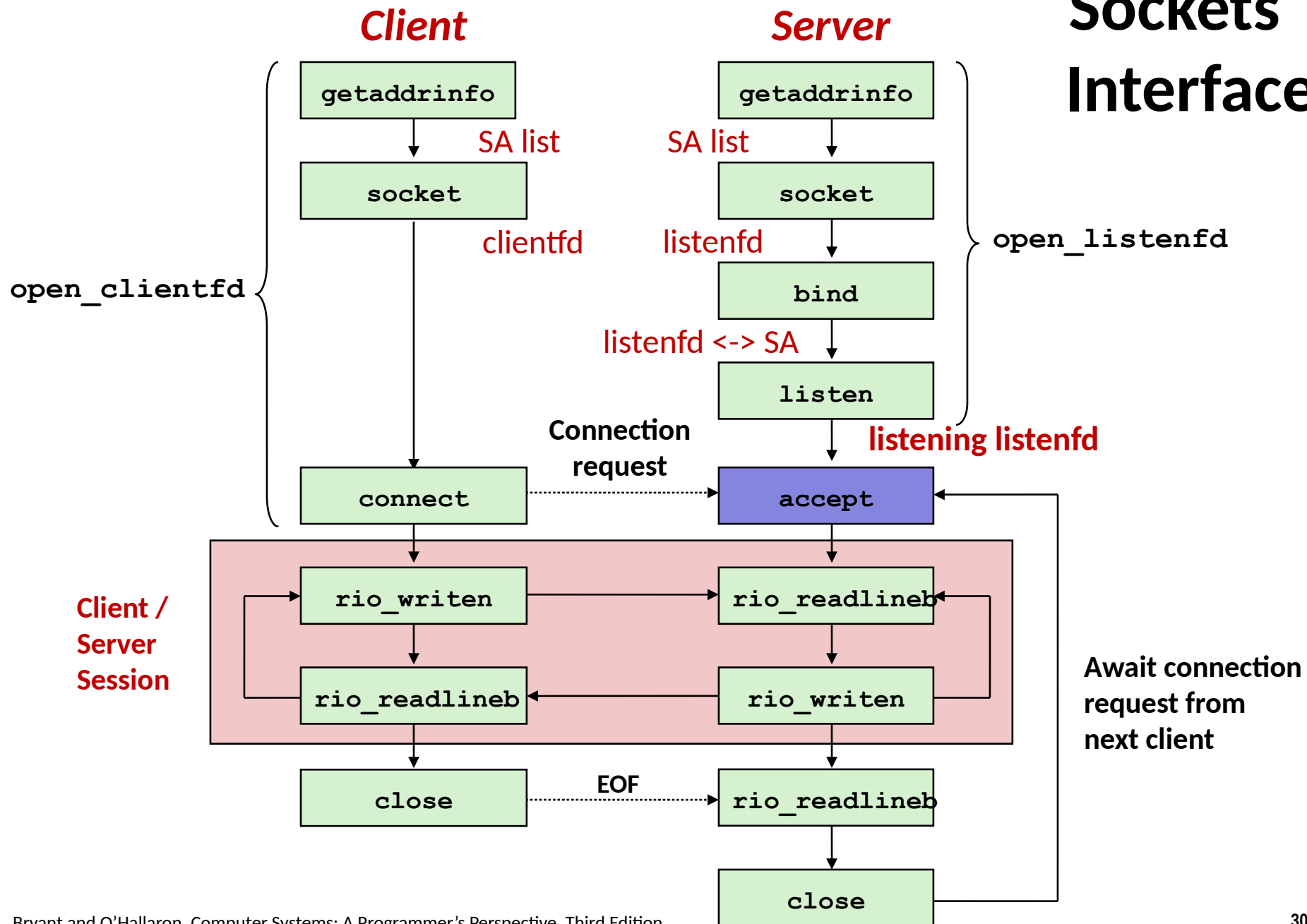
- Structures
- String conversions
- DNS

## ■ Sockets and ports

- Creating and associating sockets
- Opening ports

## ■ Connections

# Sockets Interface



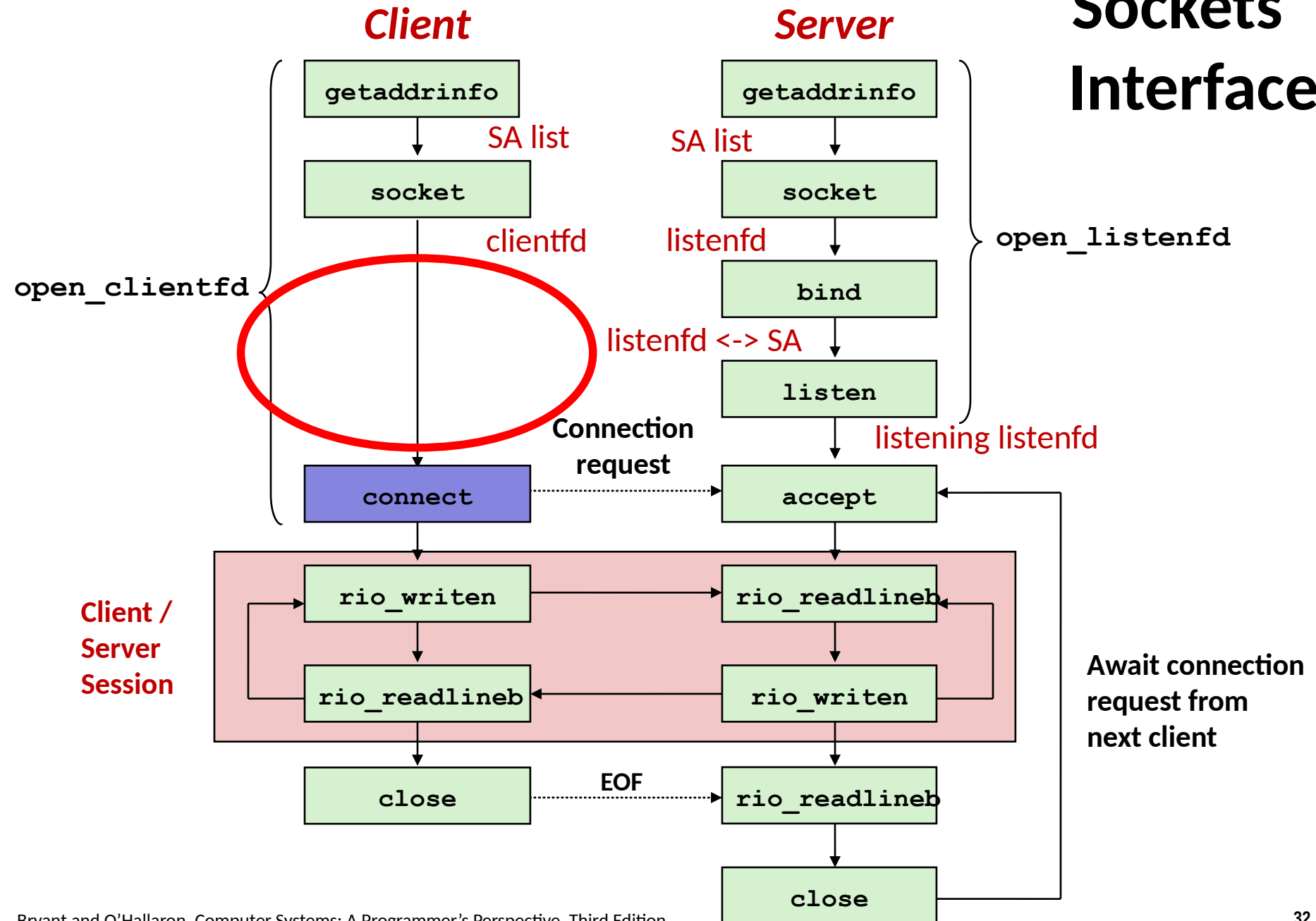
# Sockets Interface: `accept`

- 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);
```

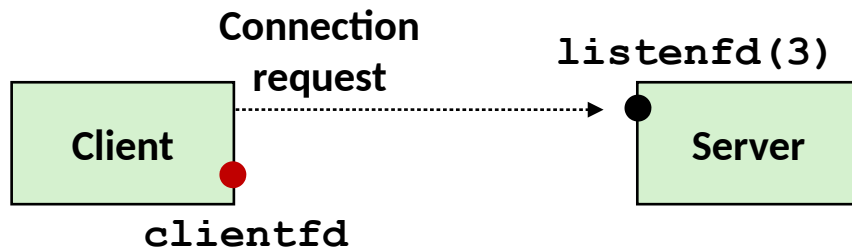
- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair `(x:y, addr.sin_addr:addr.sin_port)`
    - `x` is client address
    - `y` is ephemeral port that 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?

**Demo**

- 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

# Today

## ■ Addresses

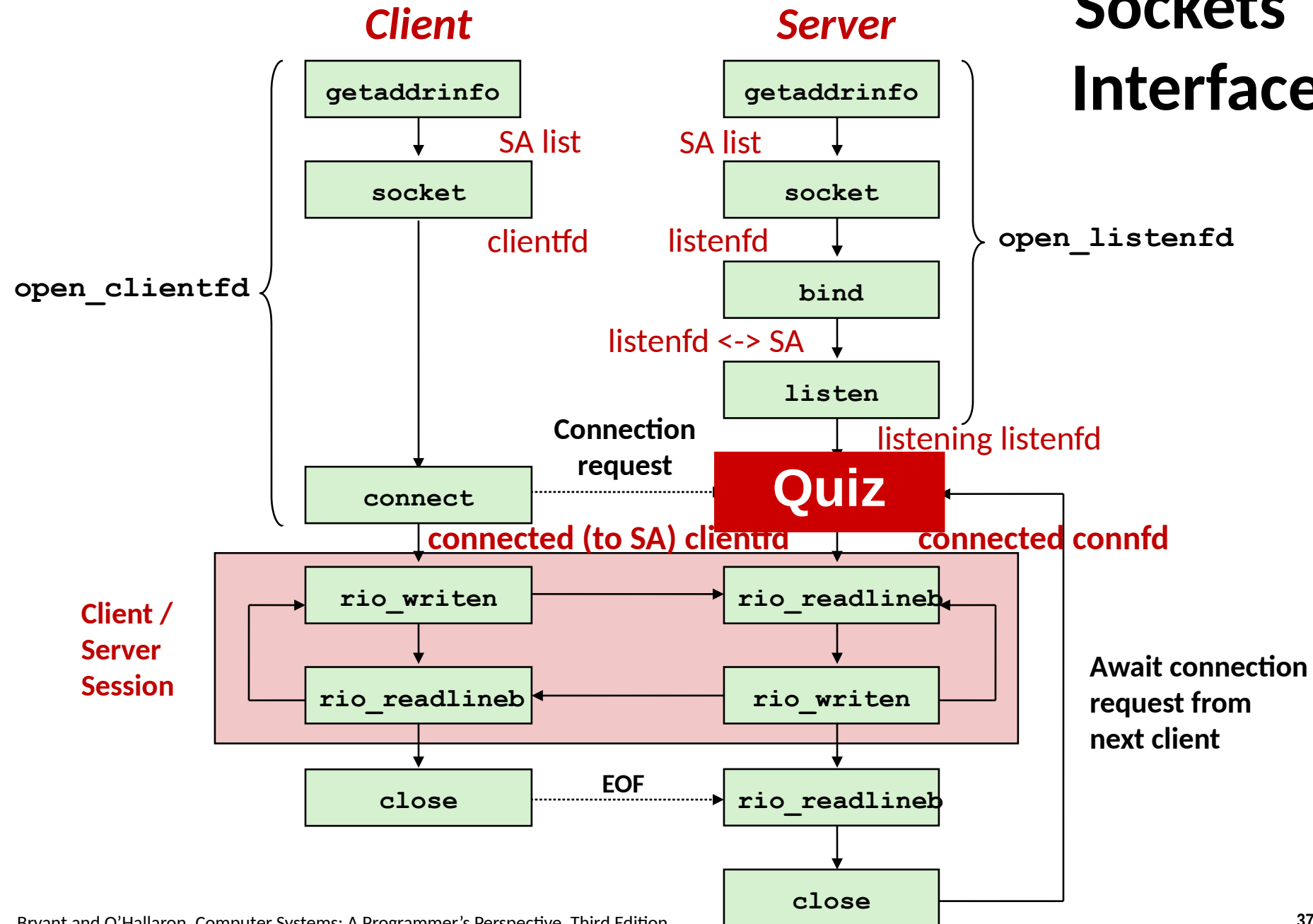
- Structures
- String conversions
- DNS

## ■ Sockets and ports

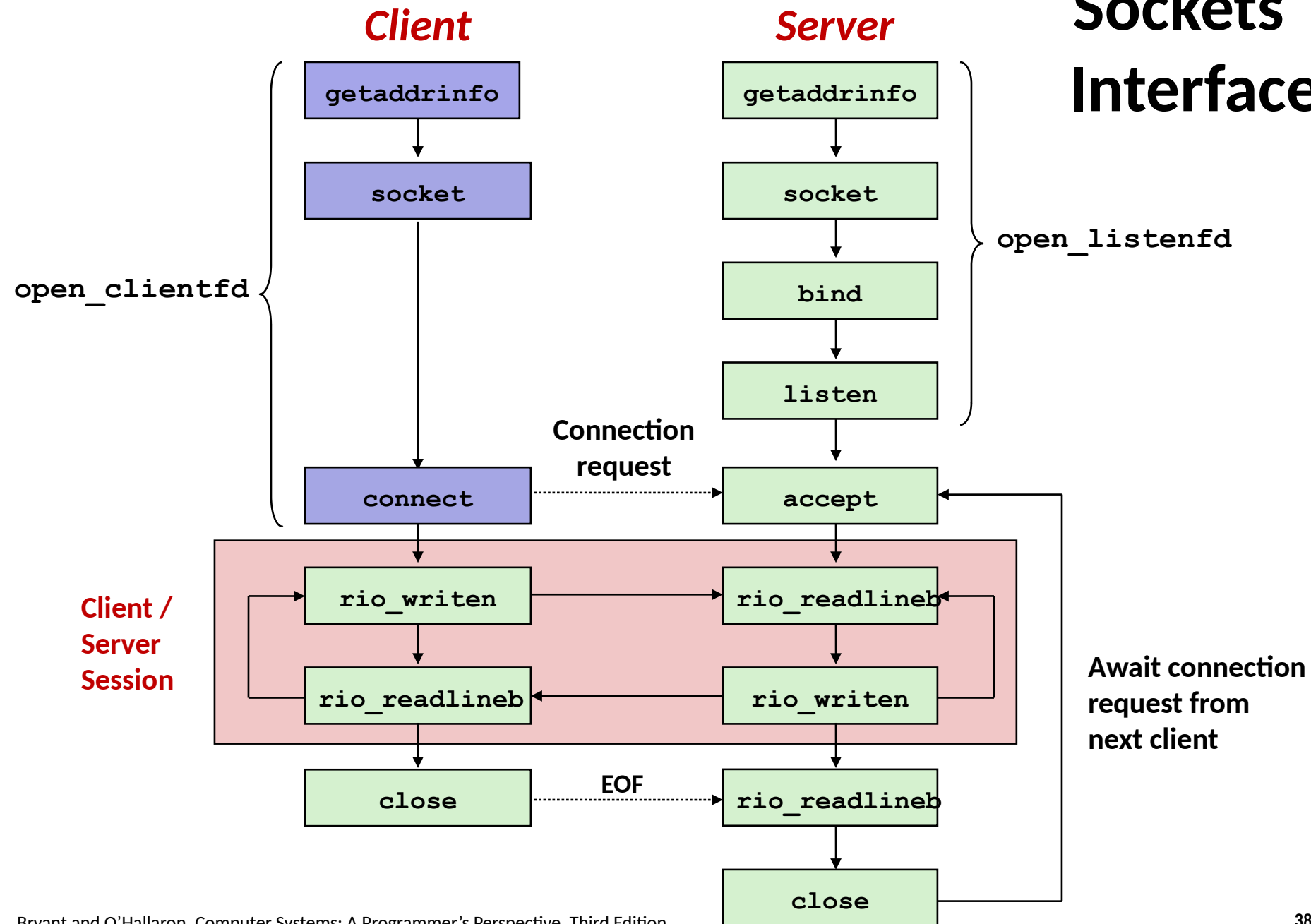
- Creating and associating sockets
- Opening ports

## ■ Connections

# Sockets Interface



# Sockets Interface



# Sockets Helper: `open_clientfd`

- Establish a connection with a server

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

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof hints);
    hints.ai_socktype = SOCK_STREAM; /* Open a TCP 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

```
/* Get a list of potential server addresses */
memset(&hints, 0, sizeof hints);
hints.ai_socktype = SOCK_STREAM; /* Open a connection */
hints.ai_flags = AI_NUMERICSERV; /* ...using numeric port. */
hints.ai_flags |= AI_ADDRCONFIG; /* Recommended */
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 */

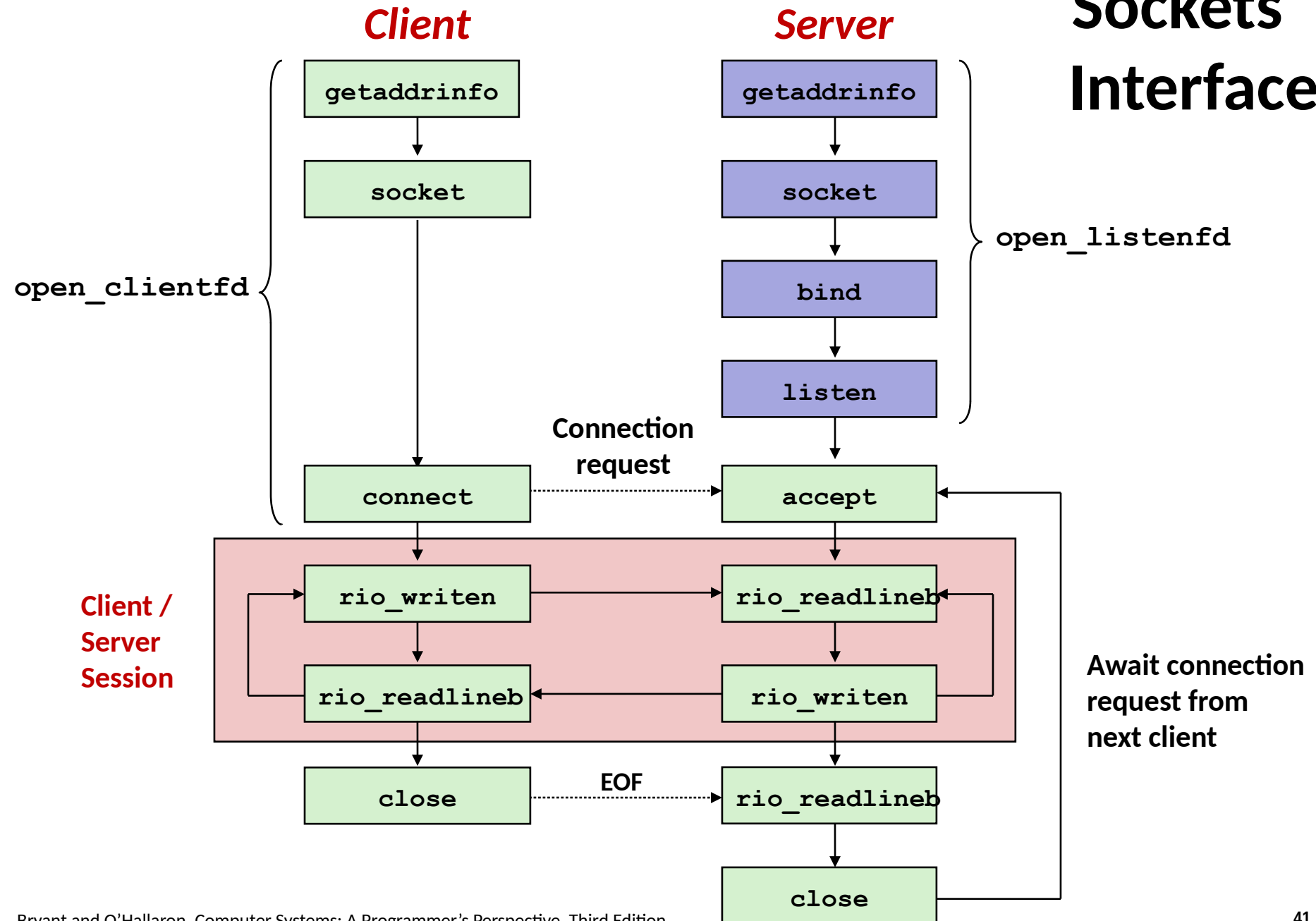
    /* Connect to the server */
    if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
        break; /* Success */
    Close(clientfd); /* Connect failed, try another */
}

/* 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 hints);
    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.

# Additional slides

# 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.

# Echo Server/Client Session Example

## Client

```

bambooshark: ./echoclient whaleshark.ics.cs.cmu.edu 6616      (A)
This line is being echoed                                     (B)
This line is being echoed
This one is, too                                           (C)
This one is, too
^D
bambooshark: ./echoclient whaleshark.ics.cs.cmu.edu 6616      (D)
This one is a new connection                               (E)
This one is a new connection
^D

```

## Server

```

whaleshark: ./echoserver 6616
Connected to (BAMBOOSHARK.ICS.CS.CMU.EDU, 33707)          (A)
server received 26 bytes                                    (B)
server received 17 bytes                                    (C)
Connected to (BAMBOOSHARK.ICS.CS.CMU.EDU, 33708)          (D)
server received 29 bytes                                    (E)

```

# Recall: Unbuffered RIO Input/Output

- Same interface as Unix `read` and `write`
- Especially useful for transferring data on network sockets

```
#include "csapp.h"
```

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n);  
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Return: num. bytes transferred if OK, 0 on EOF (`rio_readn` only), -1 on error

- `rio_readn` returns short count only if it encounters EOF
  - Only use it when you know how many bytes to read
- `rio_writen` never returns a short count
- Calls to `rio_readn` and `rio_writen` can be interleaved arbitrarily on the same descriptor



# Recall: Buffered RIO Input Functions

- Efficiently read text lines and binary data from a file partially cached in an internal memory buffer

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

Return: num. bytes read if OK, 0 on EOF, -1 on error

- `rio_readlineb` reads a *text line* of up to `maxlen` bytes from file `fd` and stores the line in `usrbuf`
  - Especially useful for reading text lines from network sockets
- Stopping conditions
  - `maxlen` bytes read
  - EOF encountered
  - Newline (`'\n'`) encountered

# Echo Client: Main Routine

```
#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

```
#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)`

```
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

# 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

# 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>
```

# 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));

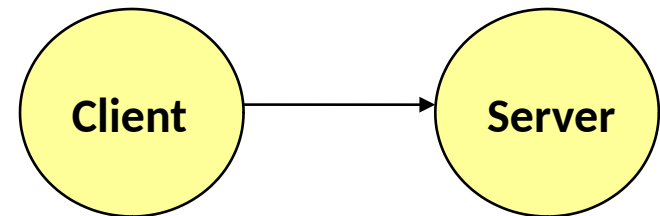
    /* Send response body to client */
    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```

tiny.c

# Serving Dynamic Content

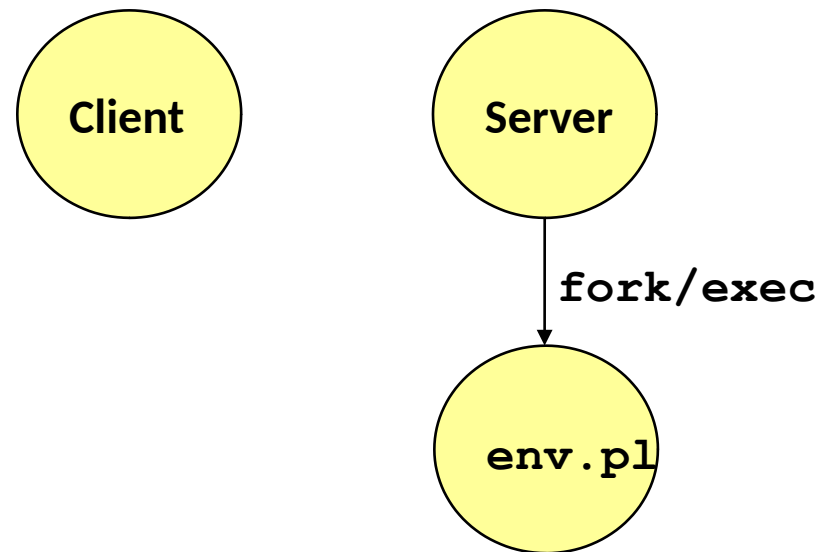
- Client sends request to server
- If request URI contains the string `"/cgi-bin"`, the Tiny server assumes that the request is for dynamic content

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



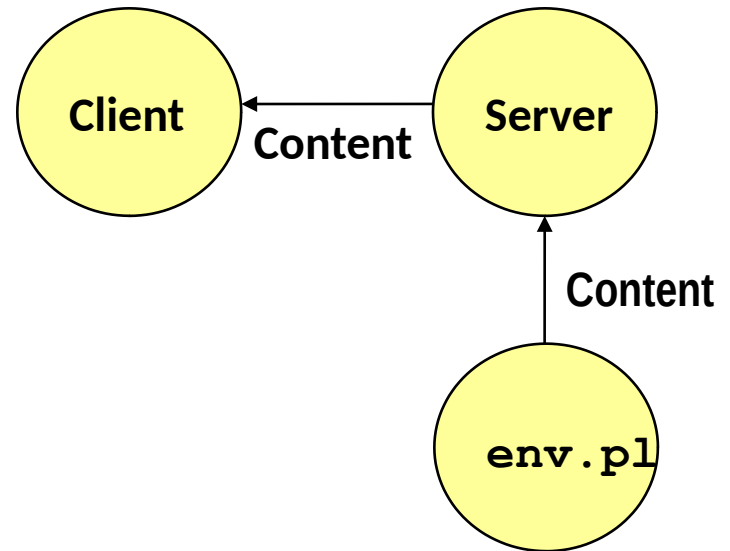
# Serving Dynamic Content (cont)

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



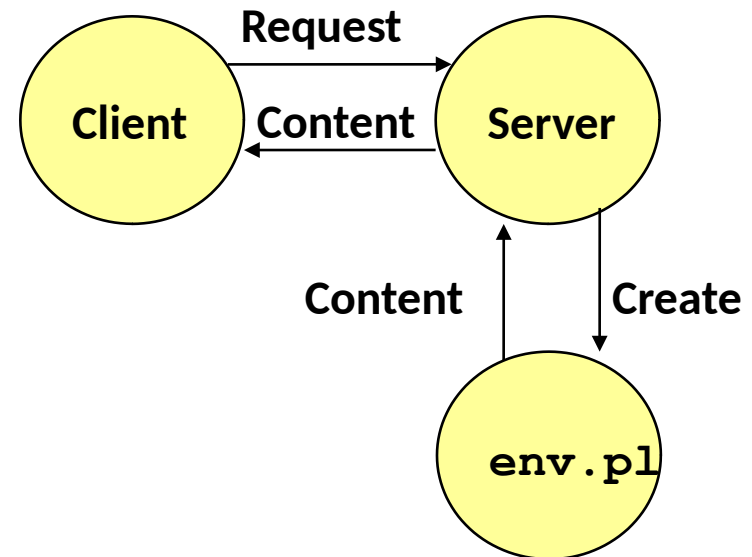
# Serving Dynamic Content (cont)

- 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

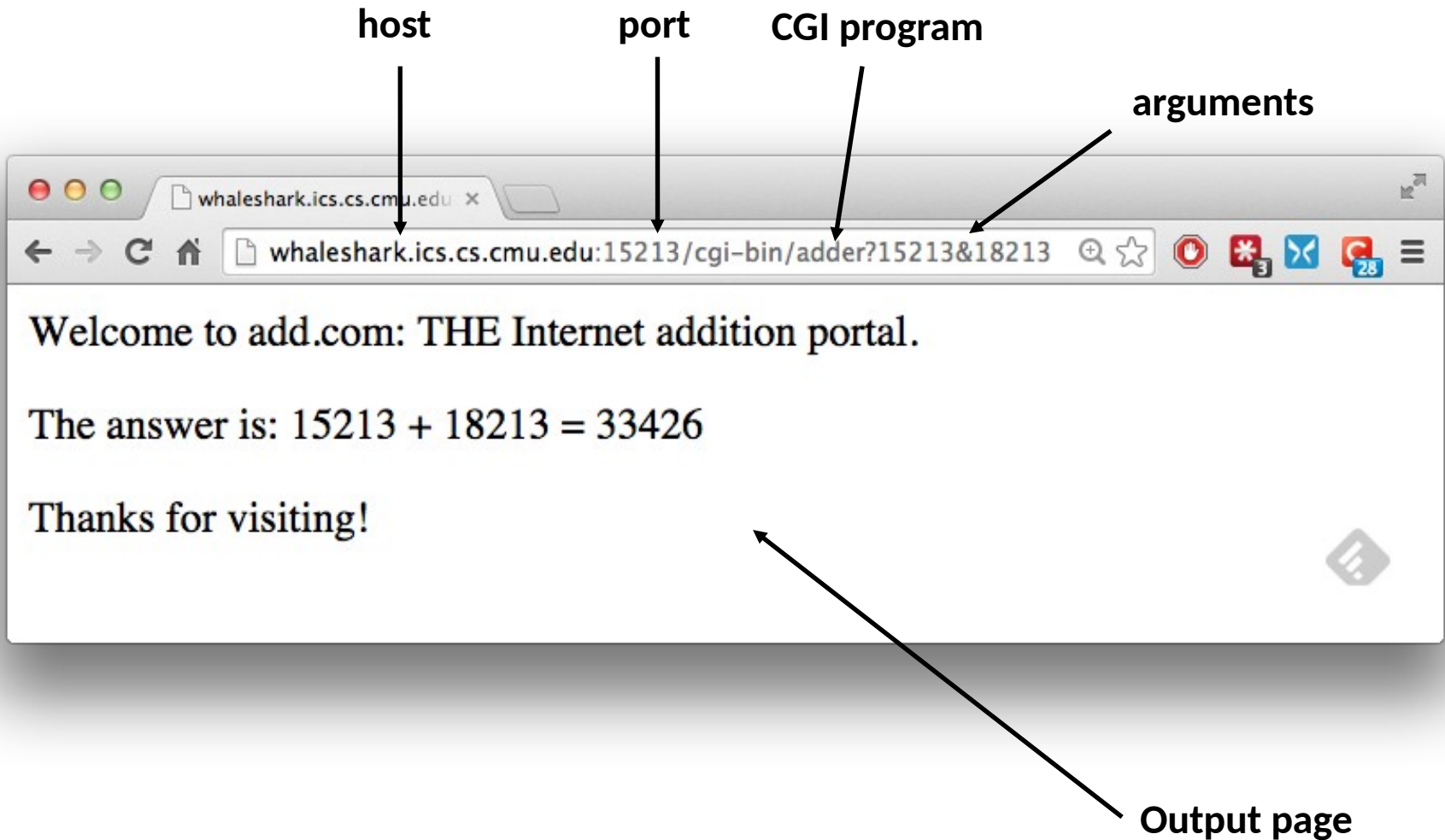
- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the **Common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





# 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 */
}
```

# 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 request sent by client*

```
HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close

```

*HTTP response generated  
by the server*

```
Content-length: 117
Content-type: text/html

```

*HTTP response generated  
by the CGI program*

```
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>
```

# 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.

# 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
```

**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
```

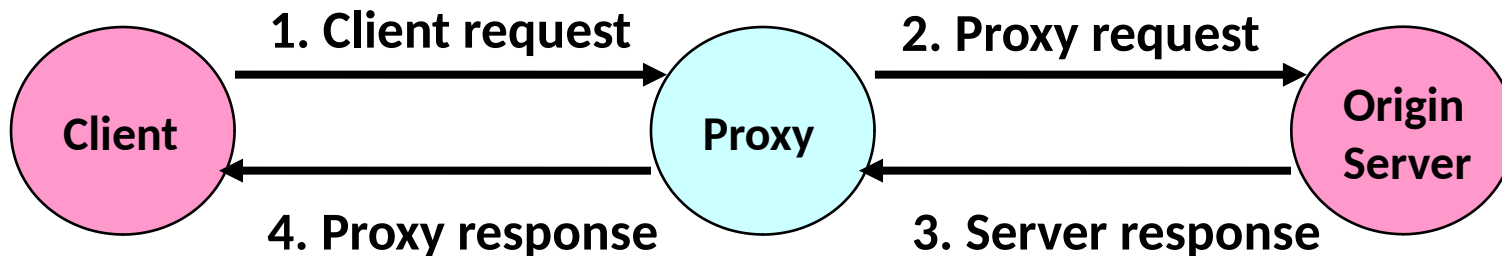
```
0\r\n
```

**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



# Why Proxies?

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

