

# Socket Programming

Introduction to Computer Systems  
23<sup>th</sup> Lecture, Dec. 9, 2019

**Instructors:**

**Class 1: Chen Xiangqun, Sun Guangyu**

**Class 2: Guan Xuetao**

**Class 3: Lu Junlin**

# Outline

- **A Programmer's View of the Internet**
- Socket programming example
- Socket detail: socket address
- Socket detail: related functions
- Put it together, and test it

# A Programmer's View of the Internet

## 1. Hosts are mapped to a set of 32-bit *IP addresses*

- 128.2.203.179
- 127.0.0.1 (always *localhost*)

## 2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*

- 128.2.217.3 is mapped to `www.cs.cmu.edu`

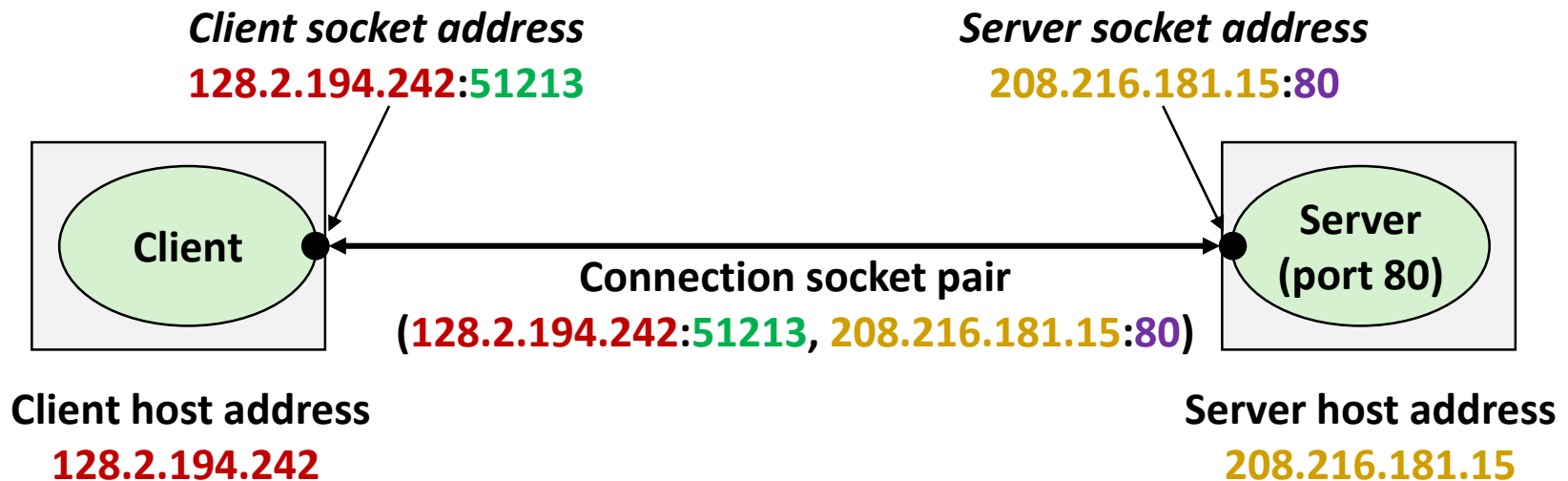
## 3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

# Internet Connections

- Clients and servers communicate by sending streams of bytes over **connections**. Each connection is:
  - *Point-to-point*: connects a pair of processes.
  - *Full-duplex*: data can flow in both directions at the same time,
  - *Reliable*: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.
- A **socket** is an endpoint of a connection
  - *Socket address* is an `IPAddress:port` pair
- A **port** is a 16-bit integer that identifies a process:
  - **Ephemeral port**: Assigned automatically by client kernel when client makes a connection request.
  - **Well-known port**: Associated with some **service** provided by a server (e.g., port 80 is associated with Web servers)

# Anatomy of a Connection

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



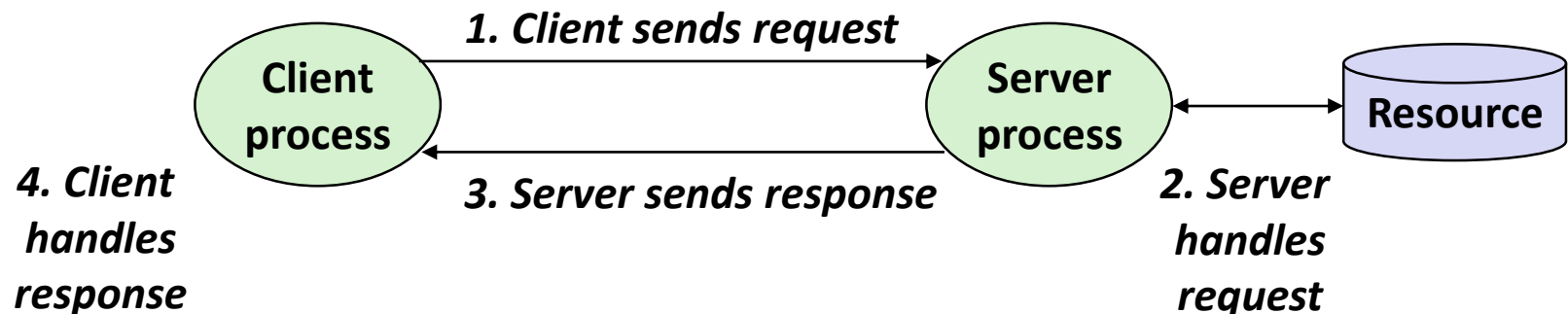
51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

# A Client-Server Transaction

## ■ Most network applications are based on the client-server model:

- A **server** process and one or more **client** processes
- Server manages some **resource**
- Server provides **service** by manipulating resource for clients
- Server activated by request from client (vending machine analogy)



*Note: clients and servers are processes running on hosts (can be the same or different hosts)*

# Clients

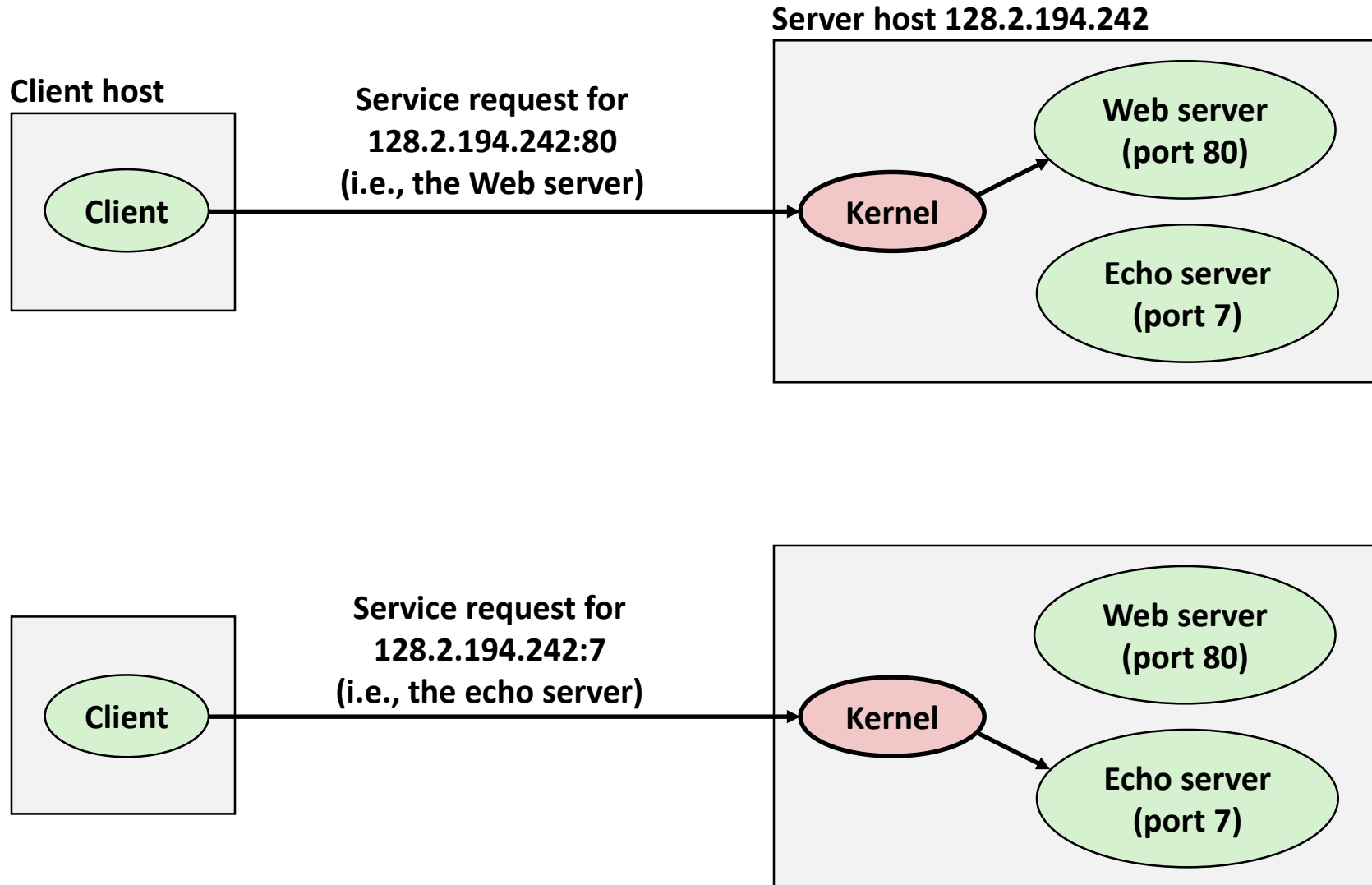
## ■ Examples of client programs

- Web browsers, `ftp`, `telnet`, `ssh`

## ■ How does a client find the server? **IP addr : port #**

- The IP address in the server socket address *identifies the host* (more precisely, an adapter on the host)
- The (well-known) port in the server socket address *identifies the service*, and thus implicitly identifies the server process that performs that service.
- Examples of well know ports
  - **Port 7: Echo server**
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server

# Using Ports to Identify Services





# Servers

- **Servers are long-running processes (daemons)**
  - Created at boot-time (typically) by the init process (process 1)
  - Run continuously until the machine is turned off
- **Each server waits for requests to arrive on a well-known port associated with a particular service**
  - Port 7: echo server
  - Port 23: telnet server
  - Port 25: mail server
  - Port 80: HTTP server
- **A machine that runs a server process is also often referred to as a “server”**

# Server Examples

## ■ Web server (port 80)

- Resource: files/compute cycles (CGI programs)
- Service: retrieves files and runs CGI programs on behalf of the client

## ■ FTP server (20, 21)

- Resource: files
- Service: stores and retrieve files

See `/etc/services` for a comprehensive list of the port mappings on a Linux machine

## ■ Telnet server (23)

- Resource: terminal
- Service: proxies a terminal on the server machine

## ■ Mail server (25)

- Resource: email “spool” file
- Service: stores mail messages in spool file

# Outline

- A Programmer's View of the Internet
- **Socket programming example**
- Socket detail: socket address
- Socket detail: related functions
- Put it together, and test it

# Socket Programming Example

- **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

# Echo Server/Client Session Example

## Client

```
linux> ./echoclient ics12 15213 (A)
This line is being echoed (B)
This line is being echoed
This one is, too (C)
This one is, too
^D
linux> ./echoclient ics12 15213 (D)
This one is a new connection (E)
This one is a new connection
^D
```

## Server

```
ics12> ./echoserveri 15213
Connected to (ics12.pku.edu.cn, 52069) (A)
server received 26 bytes (B)
server received 17 bytes (C)
Connected to (ics12.pku.edu.cn, 52070) (D)
server received 29 bytes (E)
```

```
ics> ./echoserveri 15213
```

```
linux> ./echoclient ics 15213 (A)
```

```
Connected to (ics.pku.edu.cn, 52069) (A)
```

```
This line is being echoed (B)
```

```
server received 26 bytes (B)
```

```
This line is being echoed (echo)
```

```
This one is, too (C)
```

```
server received 17 bytes (C)
```

```
This one is, too (echo)
```

```
^D
```

```
Connection closed
```

```
linux> ./echoclient ics 15213 (D)
```

```
Connected to (ics.pku.edu.cn, 52070) (D)
```

```
This one is a new connection (E)
```

```
server received 29 bytes (E)
```

```
This one is a new connection (echo)
```

```
^D
```

```
Connection closed
```

# Sockets Interface

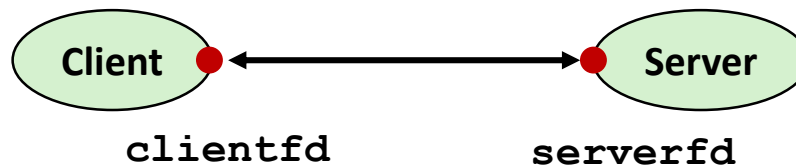
- Set of system-level functions used in conjunction with Unix I/O to build network applications.
- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.
- Available on all modern systems
  - Unix variants, Windows, OS X, IOS, Android, ARM

# Sockets

## ■ What is a socket?

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

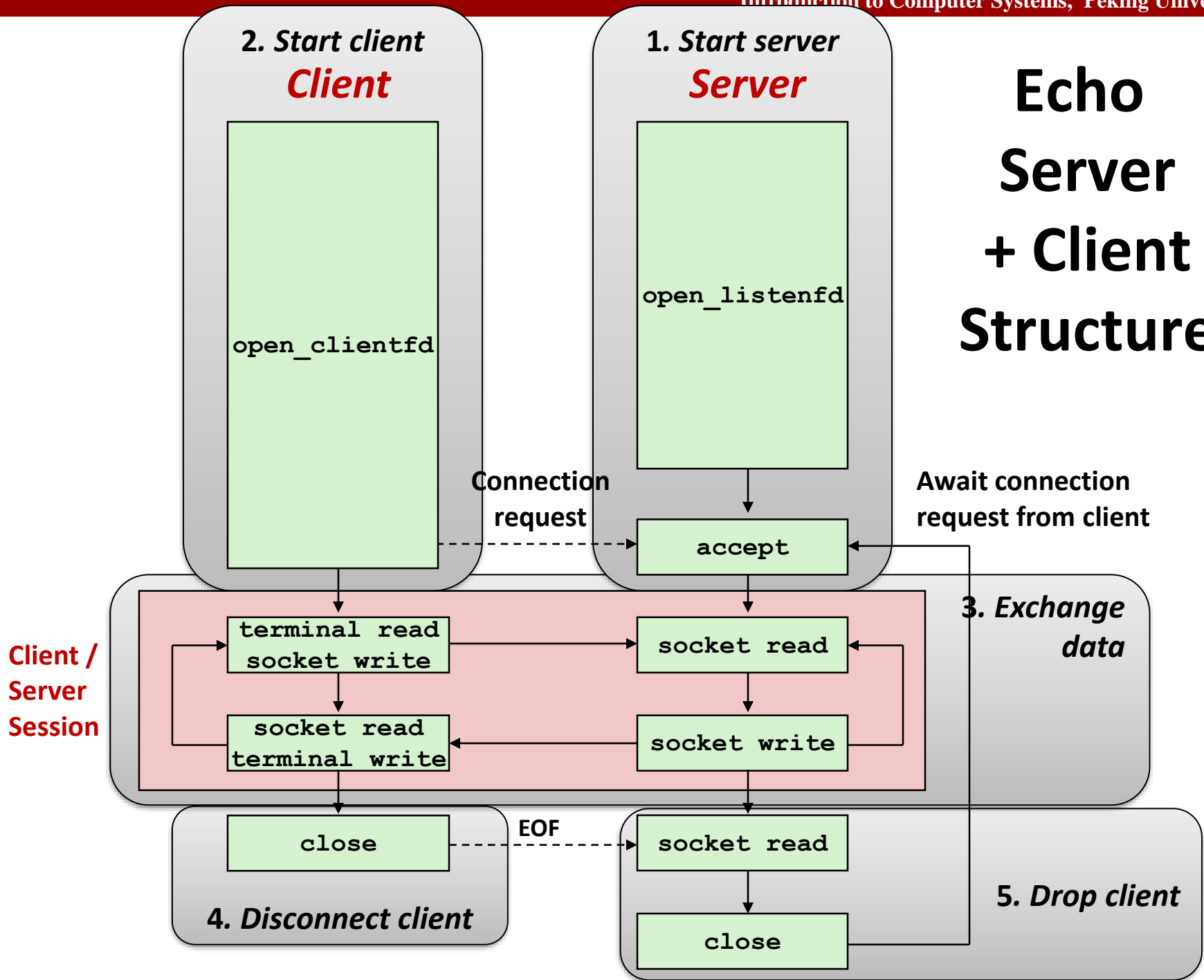
## ■ Clients and servers communicate with each other by reading from and writing to socket descriptors



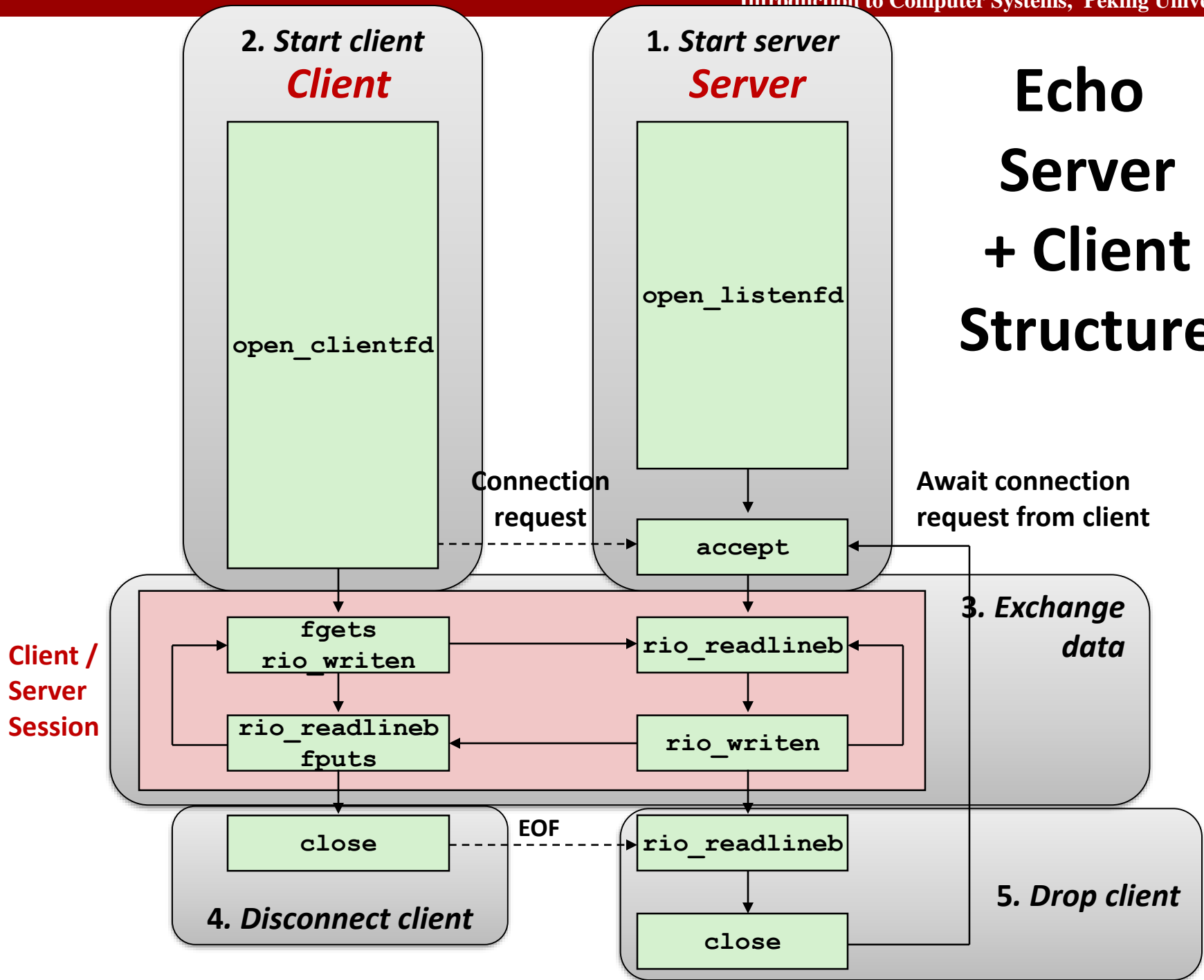
- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors



# Echo Server + Client Structure

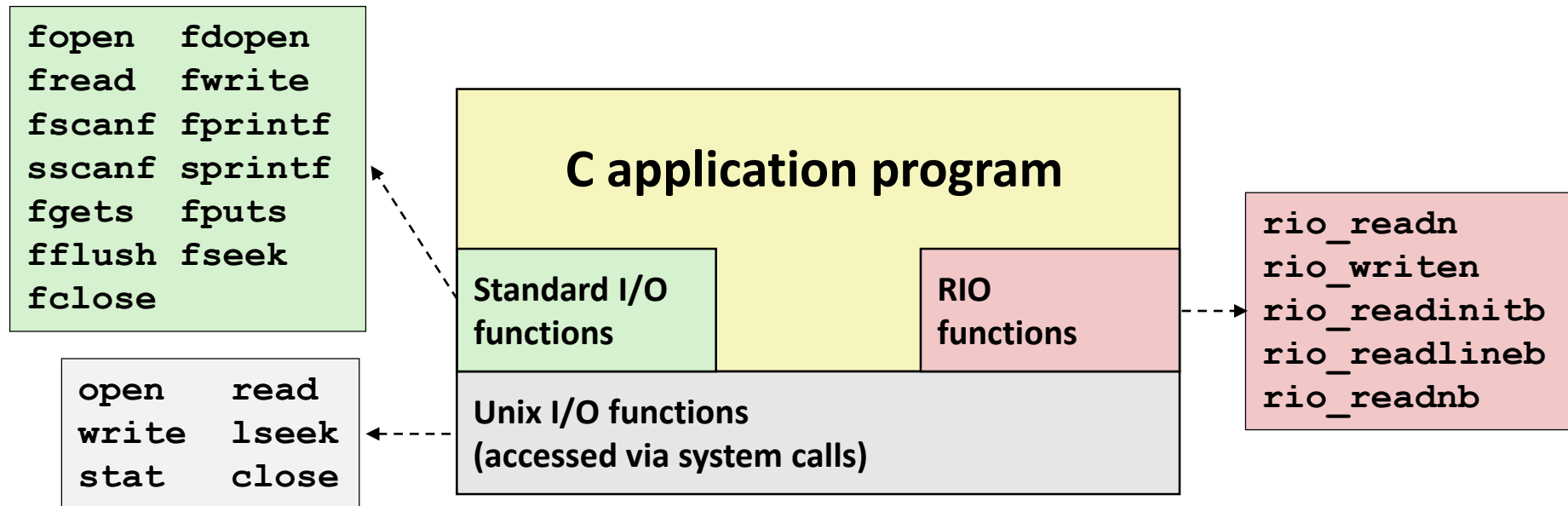


# Echo Server + Client Structure



# 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”



# 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)`

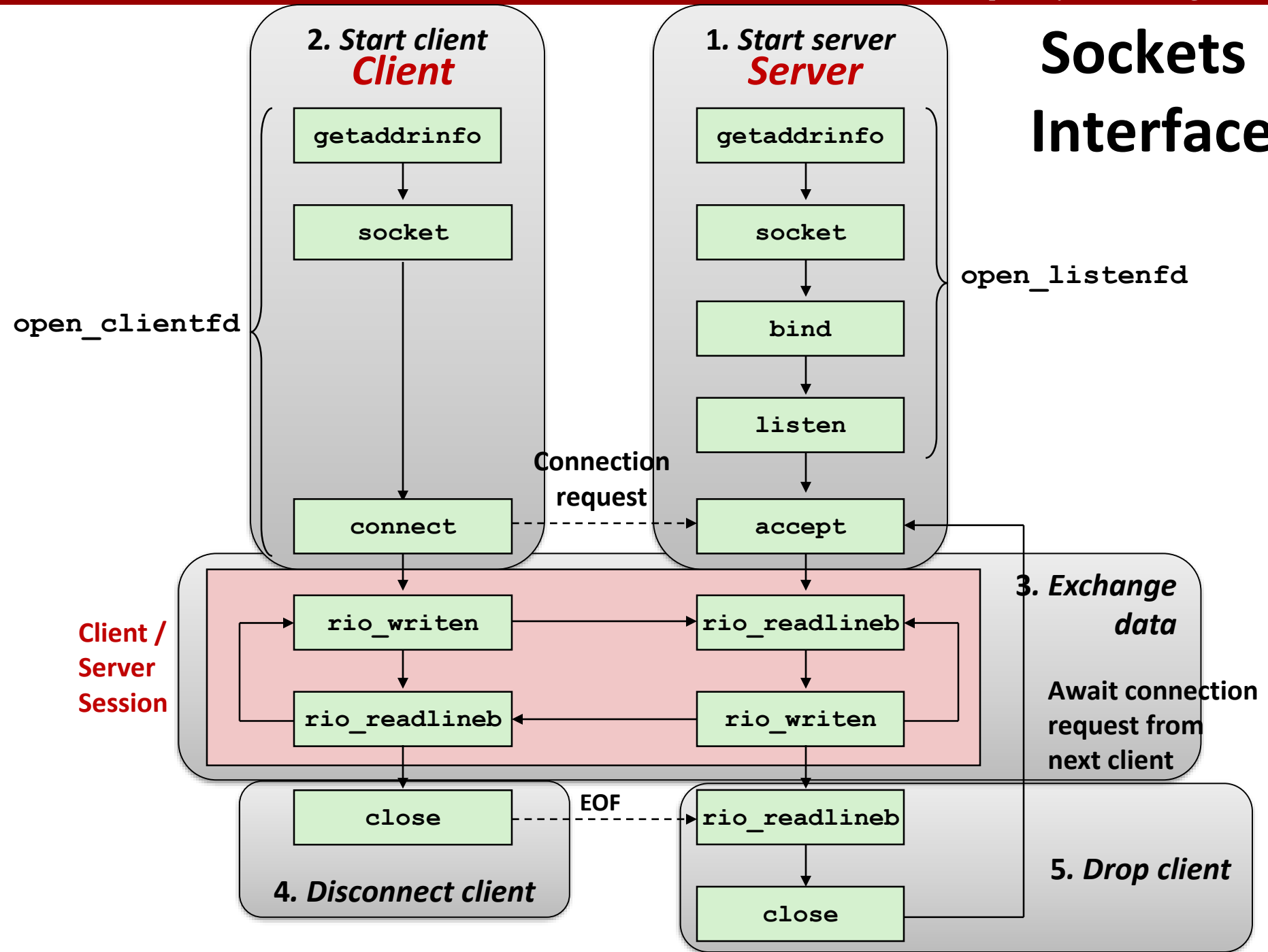
```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    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



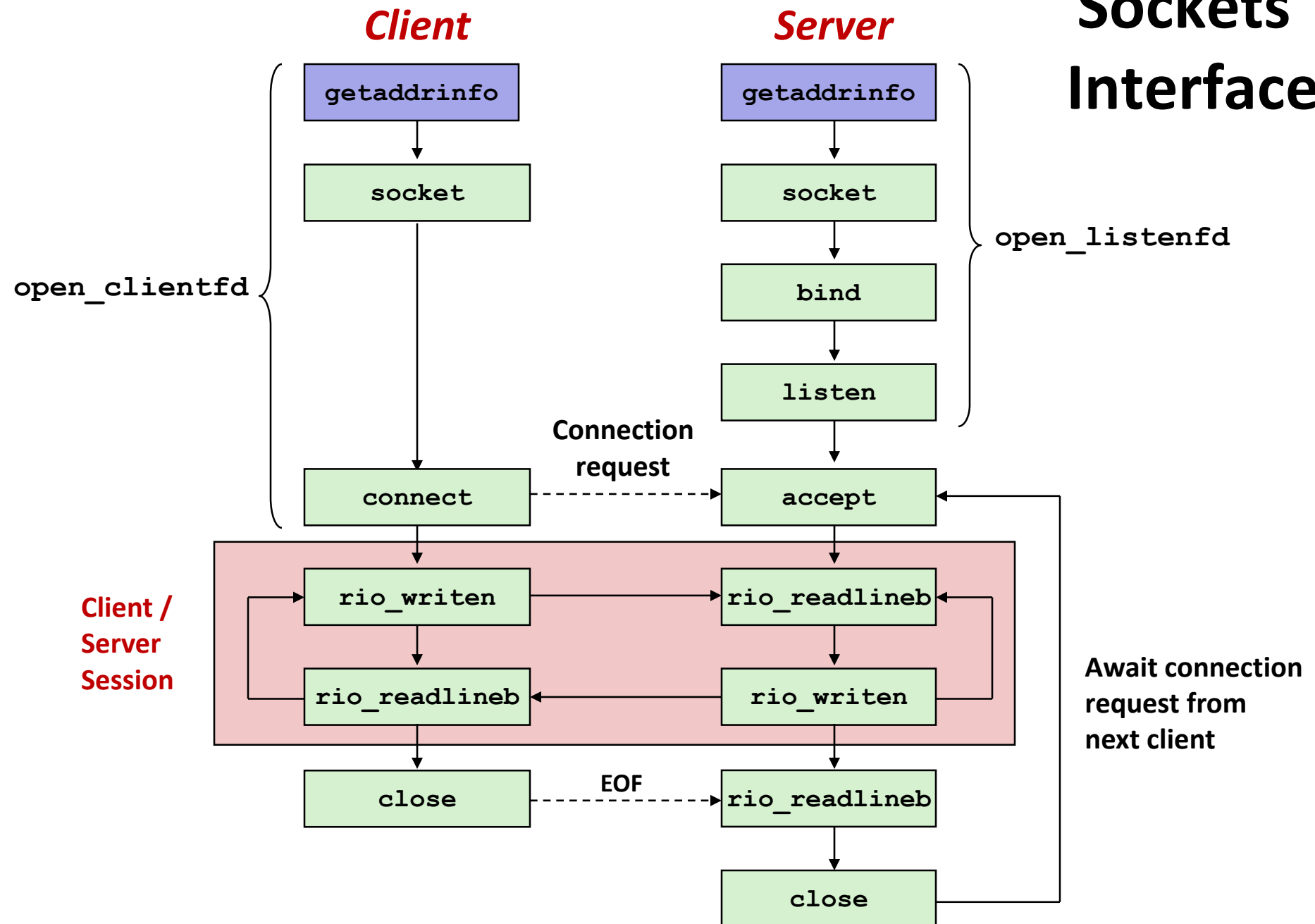
# Sockets Interface



# Outline

- A Programmer's View of the Internet
- Socket programming example
- **Socket detail: socket address**
- Socket detail: related functions
- Put it together, and test it

# Sockets Interface



# Socket Address Structures

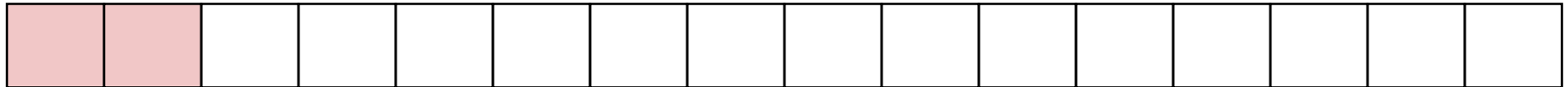
## ■ Generic socket address:

- 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



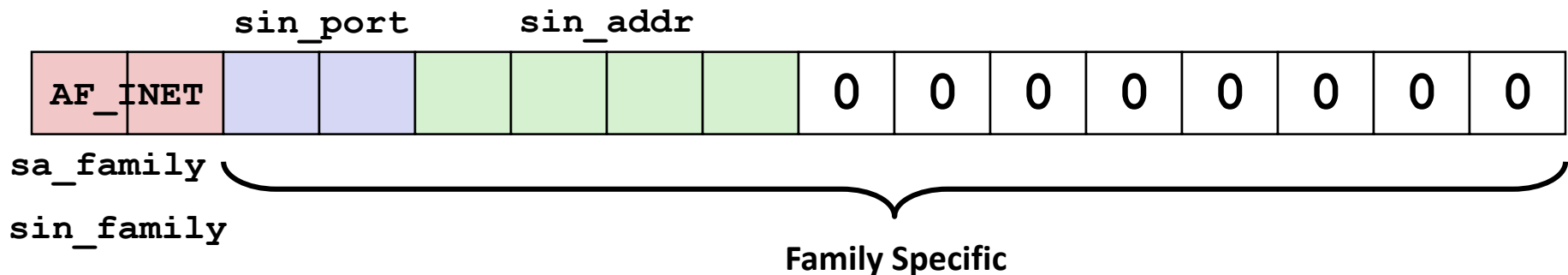
Family Specific

# 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) */
};
```



# Host and Service Conversion: `getaddrinfo`

- `getaddrinfo` is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.
  - Replaces obsolete `gethostbyname` and `getservbyname` funcs.
- **Advantages:**
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6
- **Disadvantages**
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

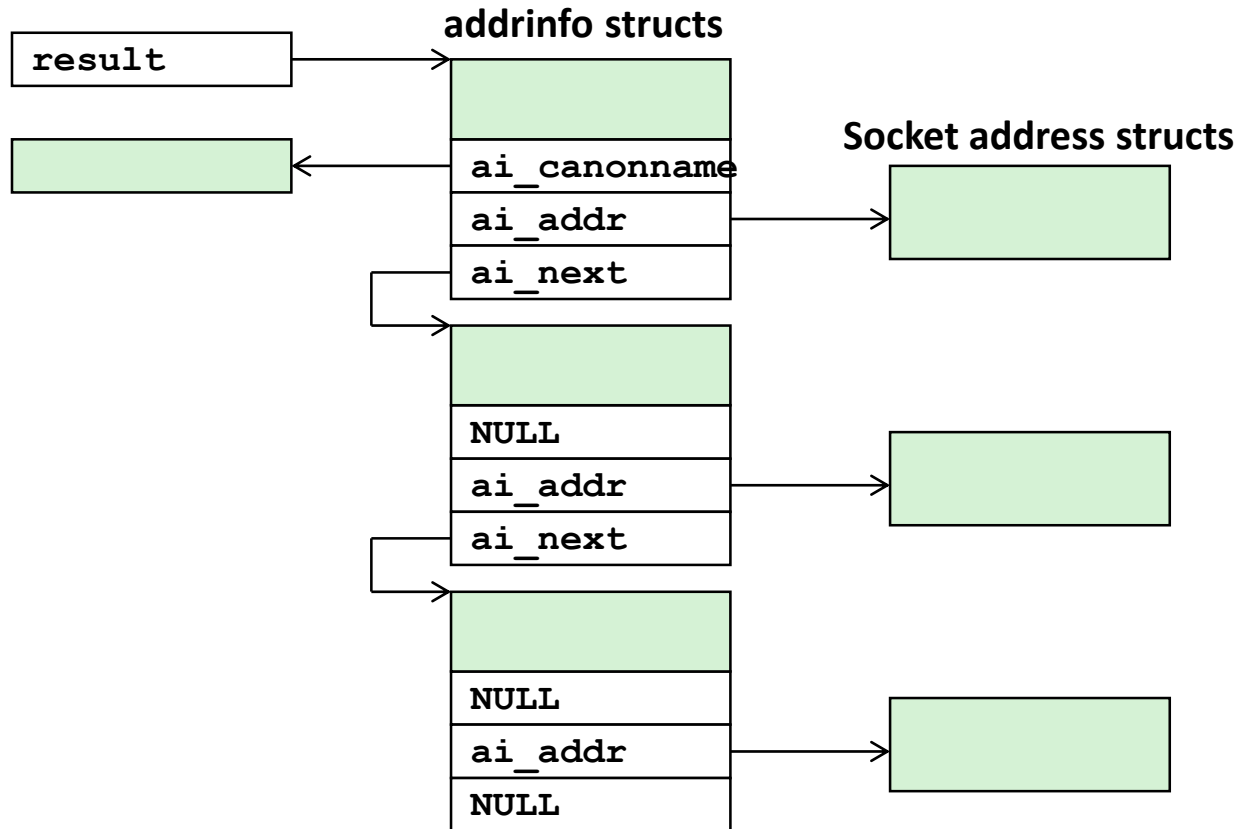
# Host and Service Conversion: `getaddrinfo`

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

void freeaddrinfo(struct addrinfo *result); /* Free linked list */
const char *gai_strerror(int errcode);     /* Return error msg */
```

- Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of **`addrinfo`** structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.
- **Helper functions:**
  - `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(struct addrinfo));
    // hints.ai_family = AF_INET;          /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections 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

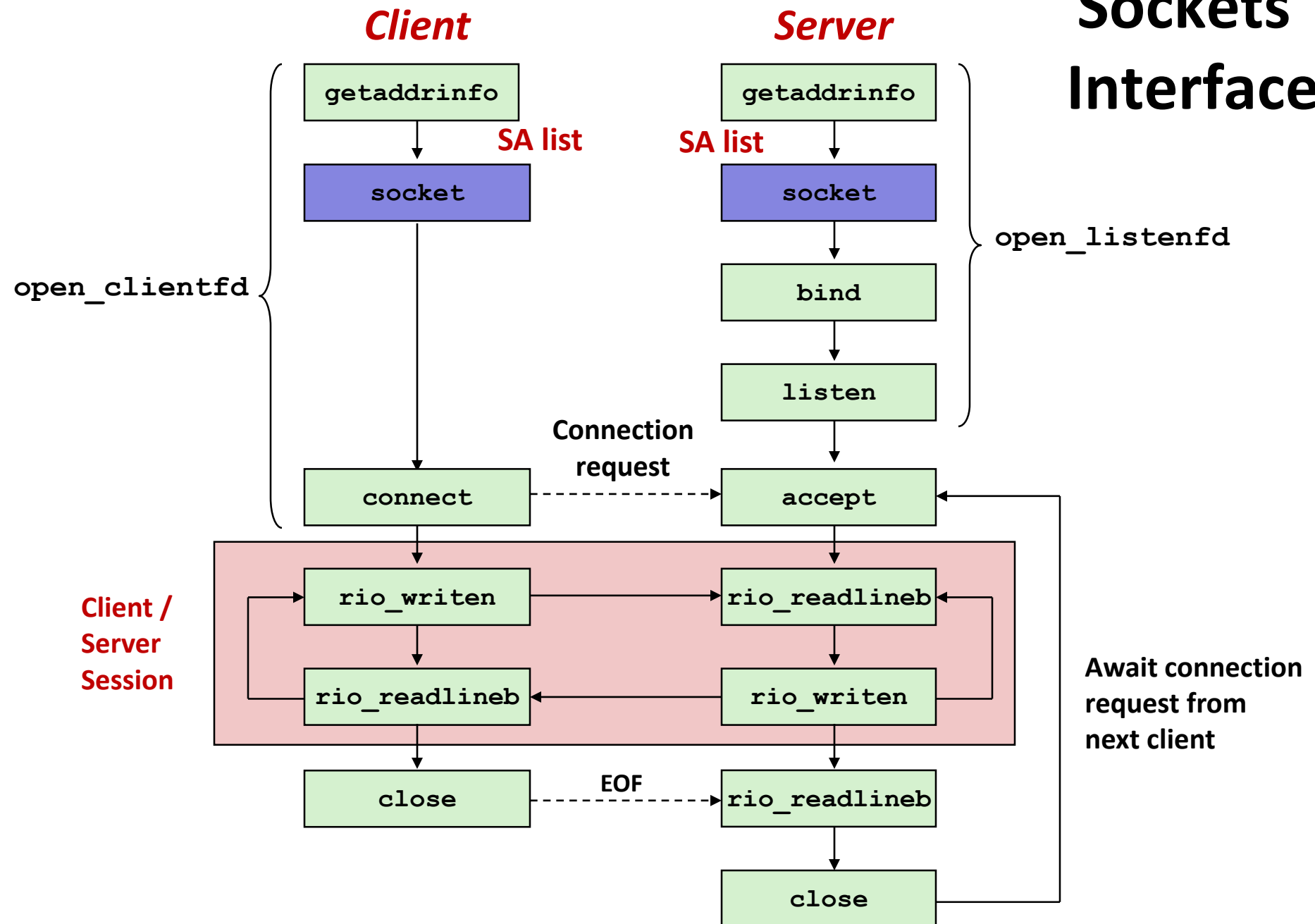
```
whaleshark> ./hostinfo localhost  
127.0.0.1
```

```
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
```

# 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  
connection

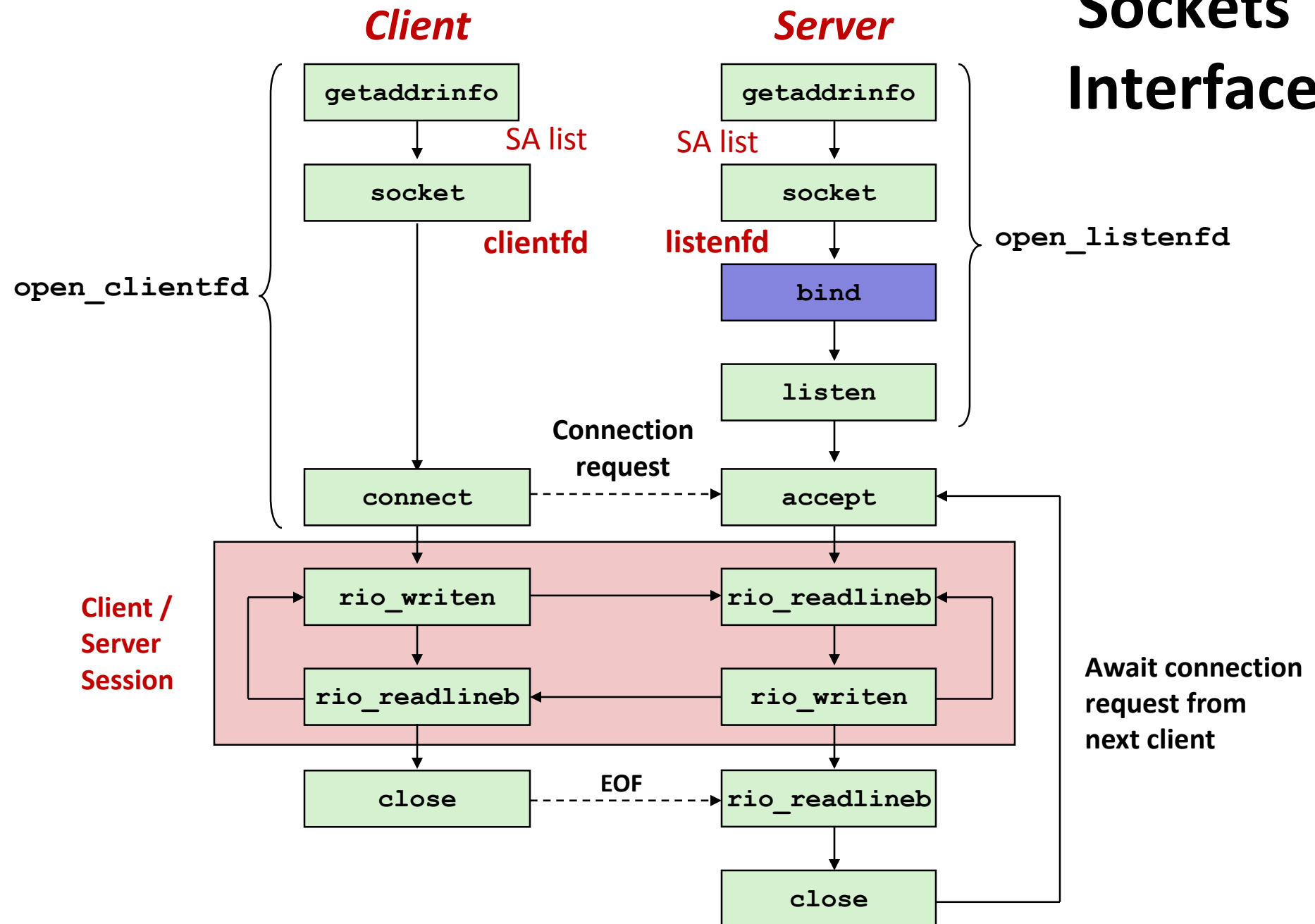
Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

# Outline

- A Programmer's View of the Internet
- Socket programming example
- Socket detail: socket address
- **Socket detail: related functions**
- Put it together, and test it



# Sockets Interface



# Sockets Interface: `bind`

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

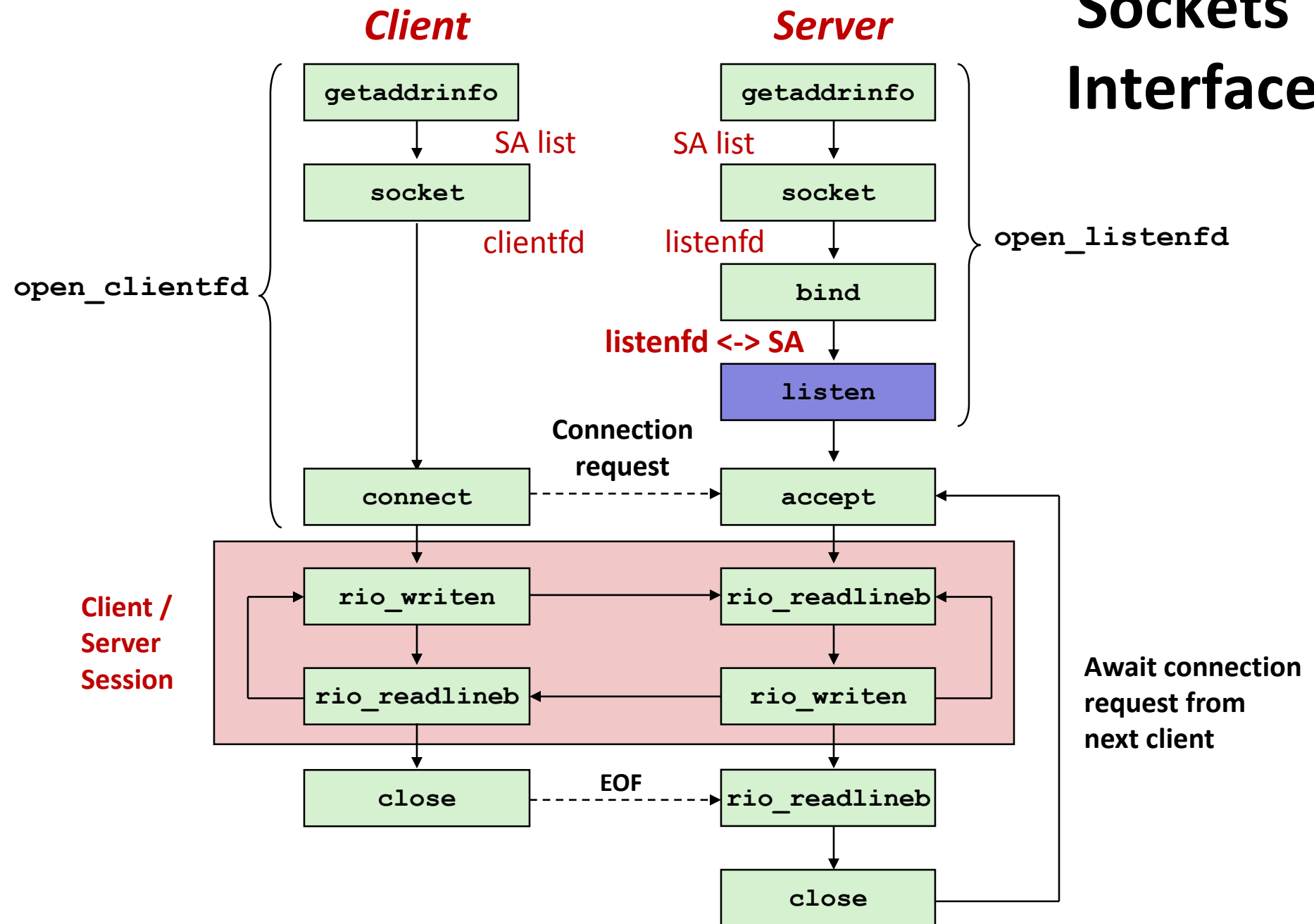
```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

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



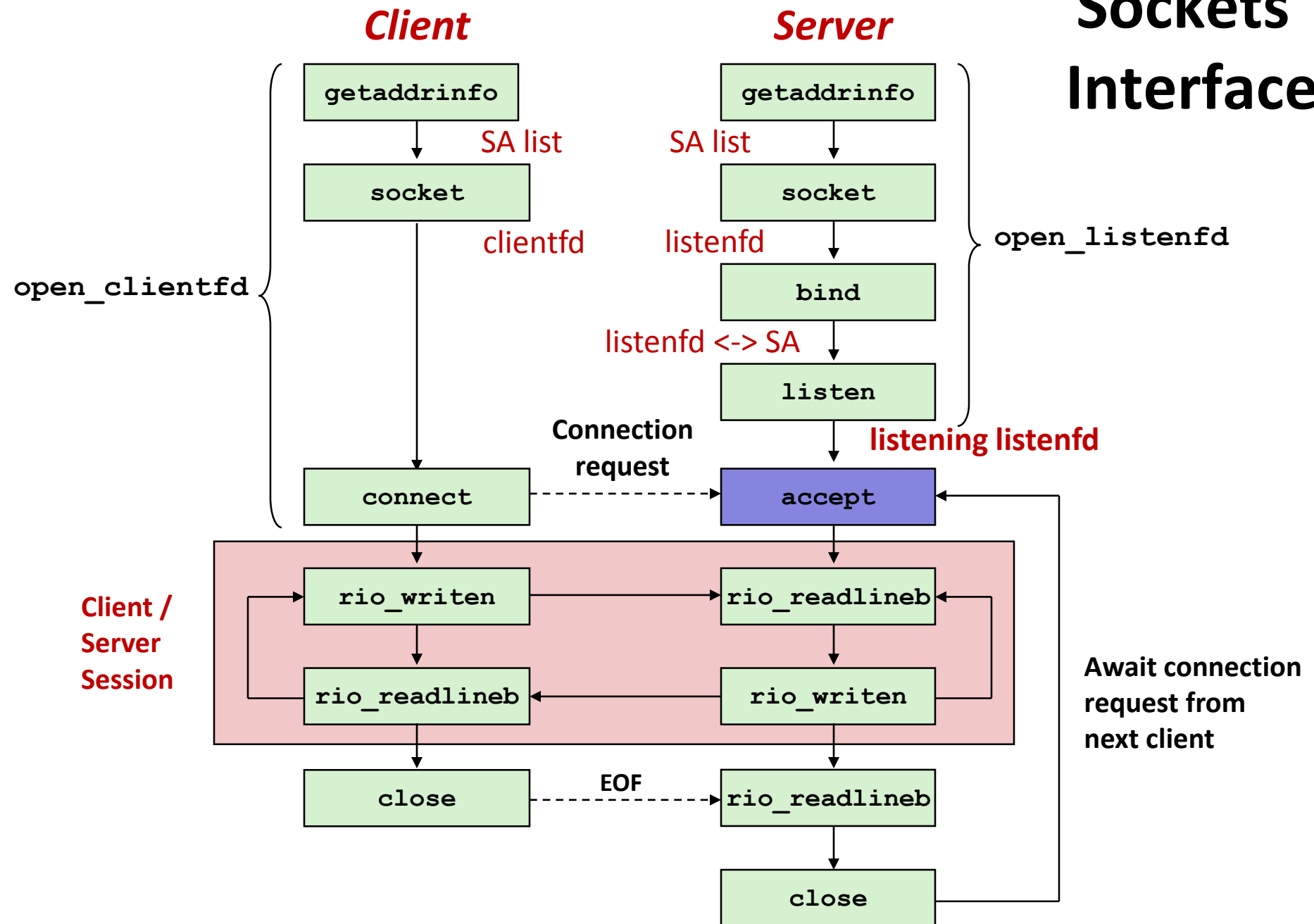
# Sockets Interface: `listen`

- 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



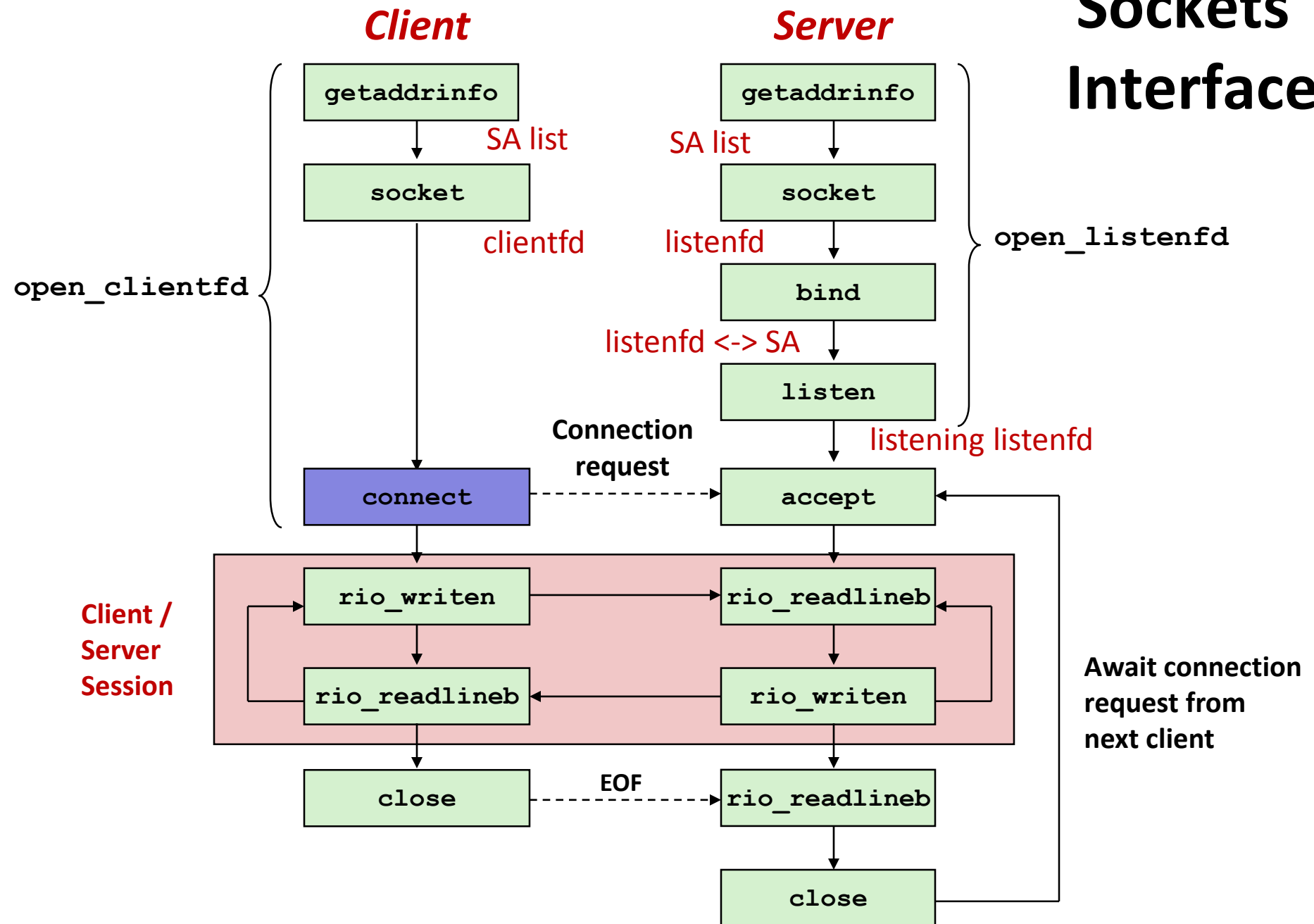
# Sockets Interface: `accept`

- 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

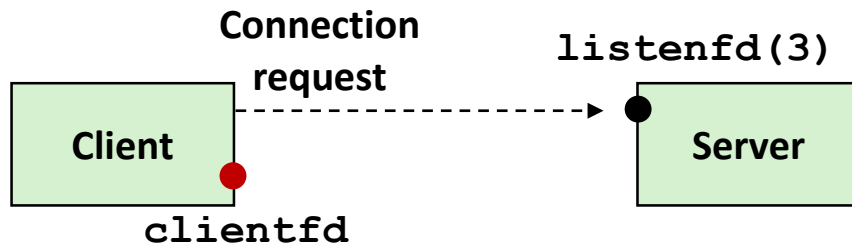
Best practice is to use **getaddrinfo** to supply the arguments **addr** and **addrlen**.



# 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

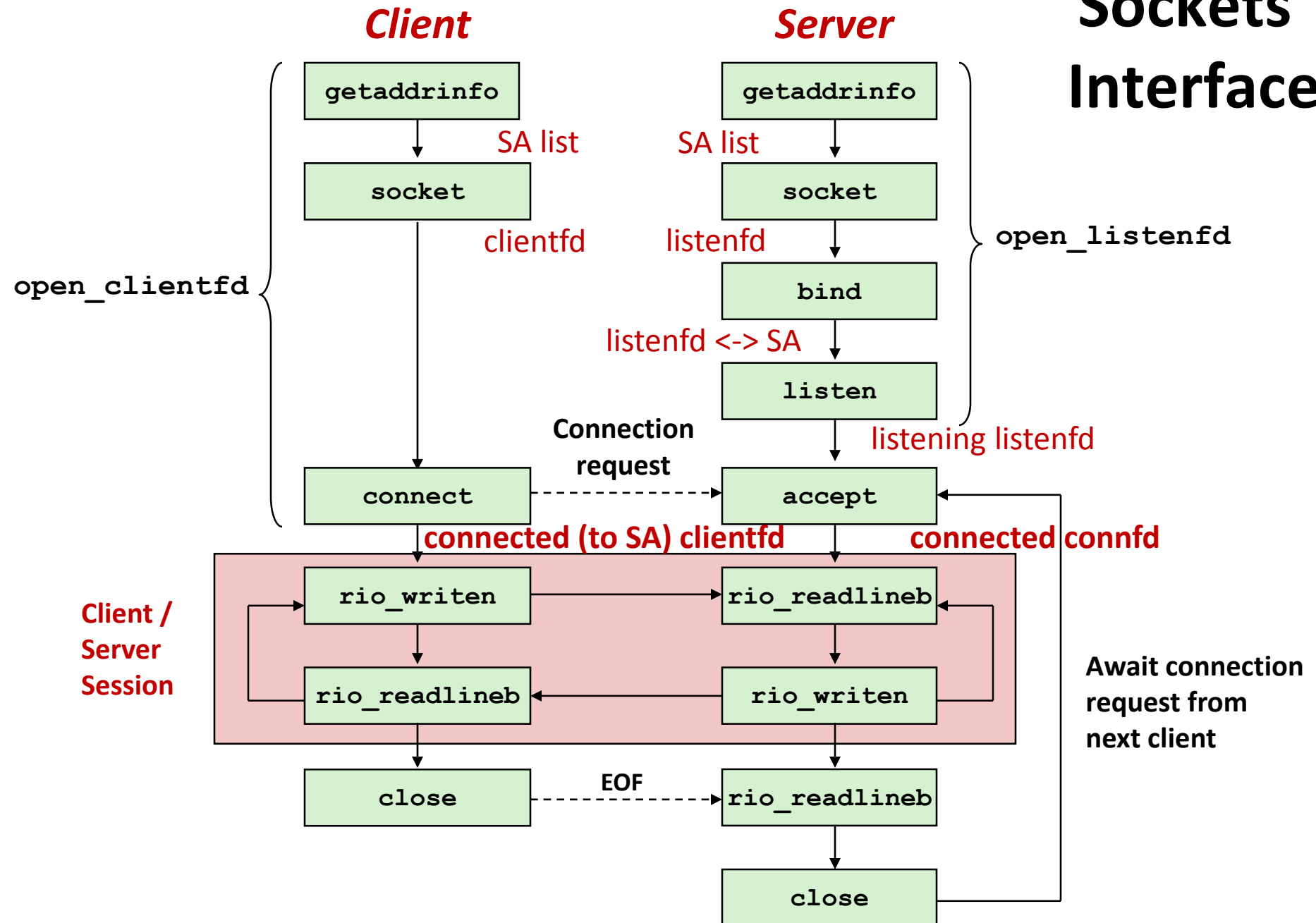
## ■ Connected descriptor

- 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

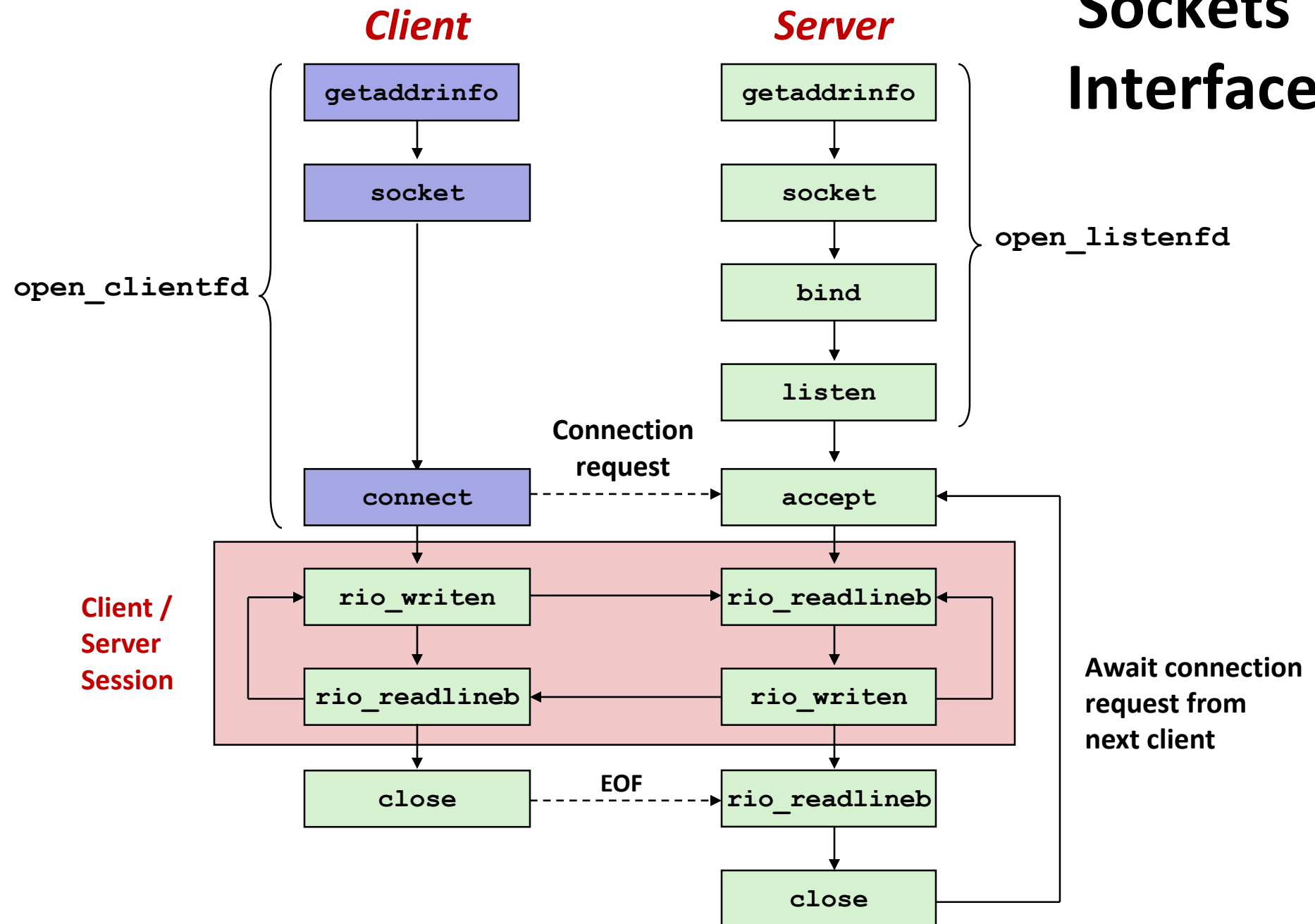
# Sockets Interface



# Outline

- A Programmer's View of the Internet
- Socket programming example
- Socket detail: socket address
- Socket detail: related functions
- **Put it together, and test it**

# 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

# Sockets Helper: open\_clientfd (cont)

```

/* Walk the list for one that we can successfully connect to */
for (p = listp; p; p = p->ai_next) {
    /* Create a socket descriptor */
    if ((clientfd = socket(p->ai_family, p->ai_socktype,
                          p->ai_protocol)) < 0)
        continue; /* Socket failed, try the next */

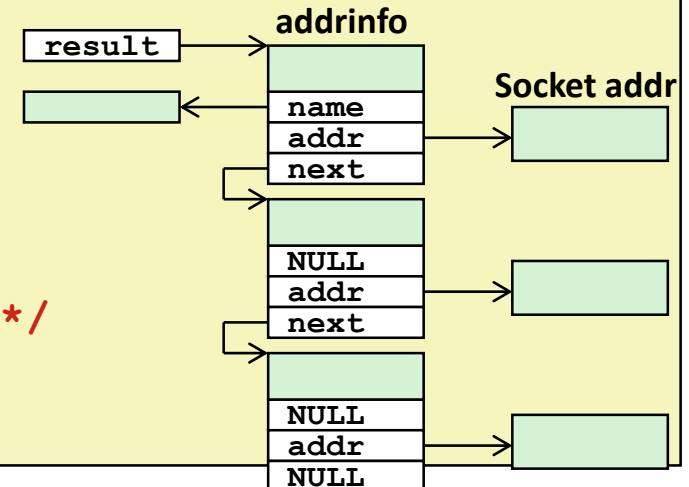
    /* Connect to the server */
    if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
        break; /* Success */
    Close(clientfd); /* Connect failed, try another */
}

```

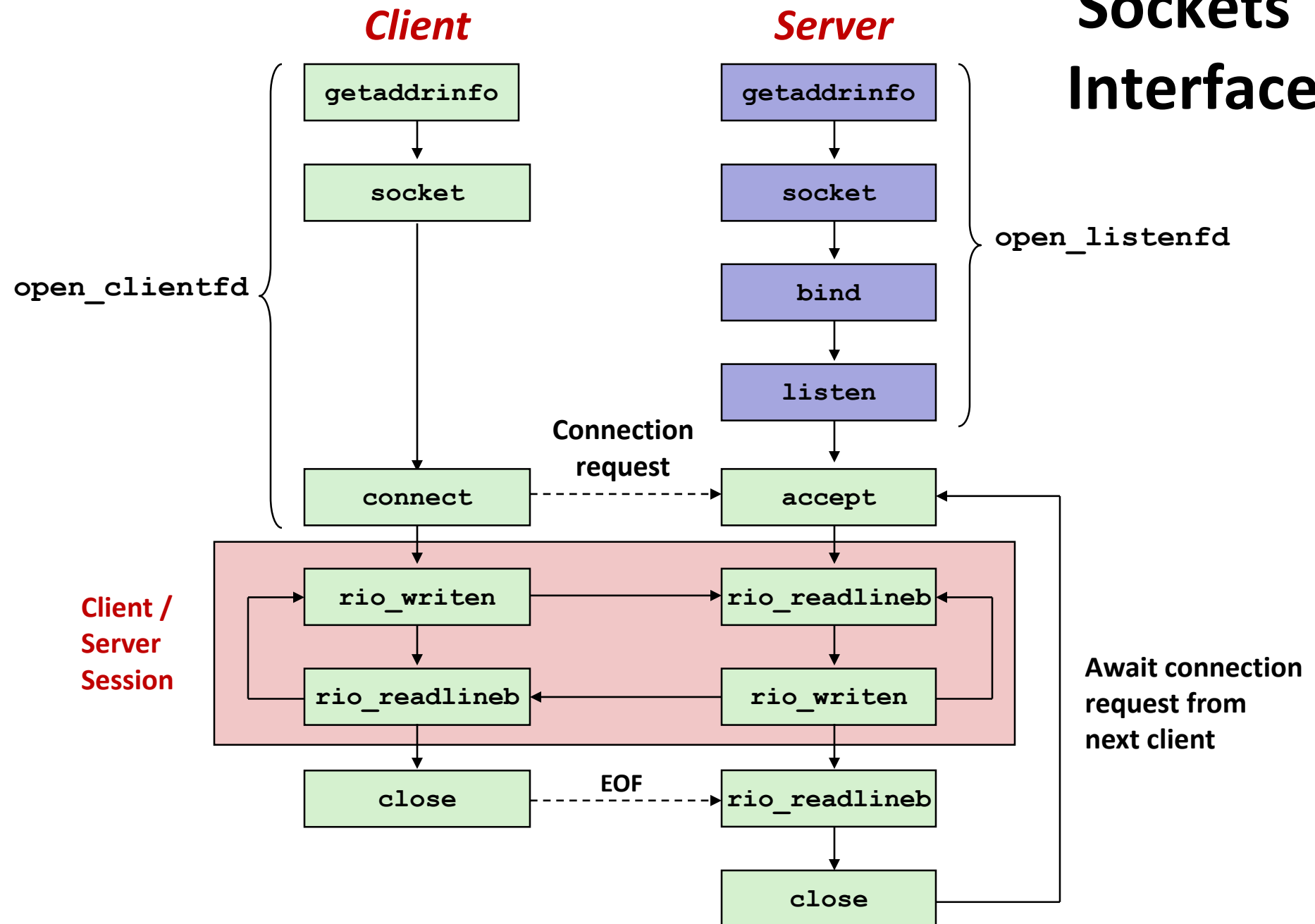
```

/* Clean up */
Freeaddrinfo(listp);
if (!p) /* All connects failed */
    return -1;
else /* The last connect succeeded */
    return clientfd;
}

```



# 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.

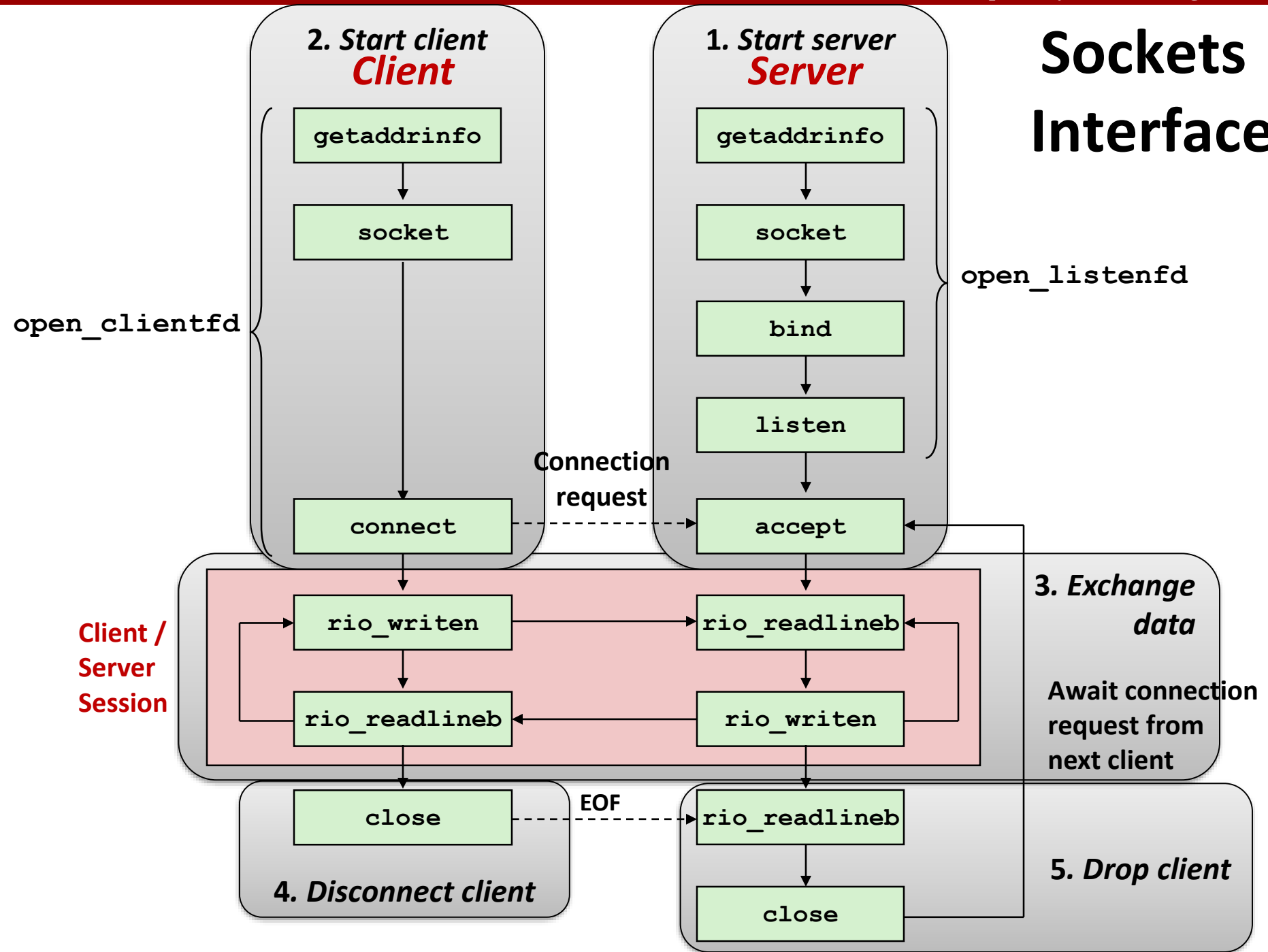
# Sockets Attributes: `setsockopt`

- The socket can be given some attributes

```
...  
/* Eliminates "Address already in use" error from bind(). */  
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,  
              (const void *)&optval , sizeof(int)) < 0)  
    return -1;
```

- Handy trick that allows us to rerun the server immediately after we kill it
  - Otherwise we would have to wait about 15 seconds
  - Eliminates “Address already in use” error from `bind()`
- Strongly suggest you do this for all your servers to simplify debugging

# Sockets Interface



# Testing Servers Using `telnet`

- The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections
  - Our simple echo server
  - Web servers
  - Mail servers
- Usage:
  - `unix> telnet <host> <portnumber>`
  - Creates a connection with a server running on `<host>` and listening on port `<portnumber>`

# Testing the Echo Server With `telnet`

```
ics12> ./echoserveri 15213
```

```
Connected to (ics12, 50280)
```

```
server received 11 bytes
```

```
server received 8 bytes
```

```
ics11> telnet ics12 15213
```

```
Trying 192.168.168.112...
```

```
Connected to ics12 (192.168.168.112).
```

```
Escape character is '^['.
```

```
Hi there!
```

```
Hi there!
```

```
Howdy!
```

```
Howdy!
```

```
^]
```

```
telnet> quit
```

```
Connection closed.
```

```
ics11>
```

# For More Information

- **W. Richard Stevens, “Unix Network Programming: Networking APIs: Sockets and XTI”, Volume 1, Second Edition, Prentice Hall, 1998**
  - THE network programming bible
- **Unix Man Pages**
  - Good for detailed information about specific functions
- **Complete versions of the echo client and server are developed in the text**
  - Updated versions linked to course website
  - Feel free to use this code in your assignments