

Computer Systems: Network Programming (Sockets)

David Marchant

Based on slides by Randal E. Bryant and David R. O'Halloran, with alterations by Vivek Shah

A Programmer's View of the Internet

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

- 128.2.203.179

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*

Global IP Internet (upper case)

- Most famous example of an internet
- Based on the TCP/IP protocol family
 - IP (Internet Protocol) :
 - Provides *basic naming scheme* and unreliable *delivery capability* of packets (datagrams) from *host-to-host*
 - UDP (Unreliable Datagram Protocol)
 - Uses IP to provide *unreliable* datagram delivery from *process-to-process*
 - TCP (Transmission Control Protocol)
 - Uses IP to provide *reliable* byte streams from *process-to-process* over *connections*
- Accessed via a mix of Unix file I/O and functions from the *sockets interface*

IP Addresses

- 32-bit IP addresses are stored in an *IP address struct*
 - IP addresses are always stored in memory in *network byte order* (big-endian byte order)
 - True in general for any integer transferred in a packet header from one machine to another.
 - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    uint32_t  s_addr; /* network byte order (big-endian) */
};
```

Dotted Decimal Notation

- By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period
 - IP address: `0x8002C2F2` = `128.2.194.242`
- Use `inet_ntop`, `inet_pton` functions for converting between dotted decimal notation and IP addresses
 - Use `htonl`, `htons`, `ntohl` and `ntohs` functions for network byte order conversions
- Use `getaddrinfo` and `getnameinfo` functions (described later) to convert between IP addresses and dotted decimal format.

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)

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**

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



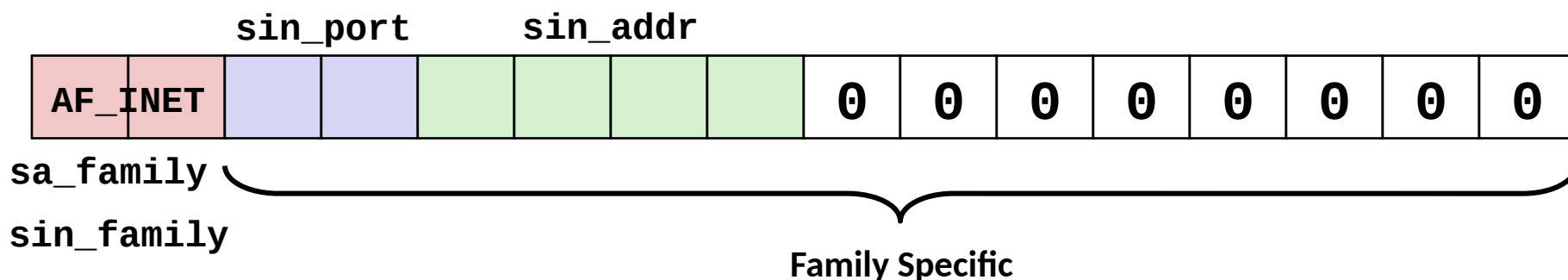
Family Specific

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.

Do-it-yourself Recap: System-level I/O

- What is the difference between the Unix I/O and the book's robust I/O APIs?
- When can short counts be returned by I/O functions? Why?
- What did each of the following functions do?
 - `ssize_t rio_writen(int fd, void *usrbuf, size_t n);`
 - `ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);`
 - `ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);`

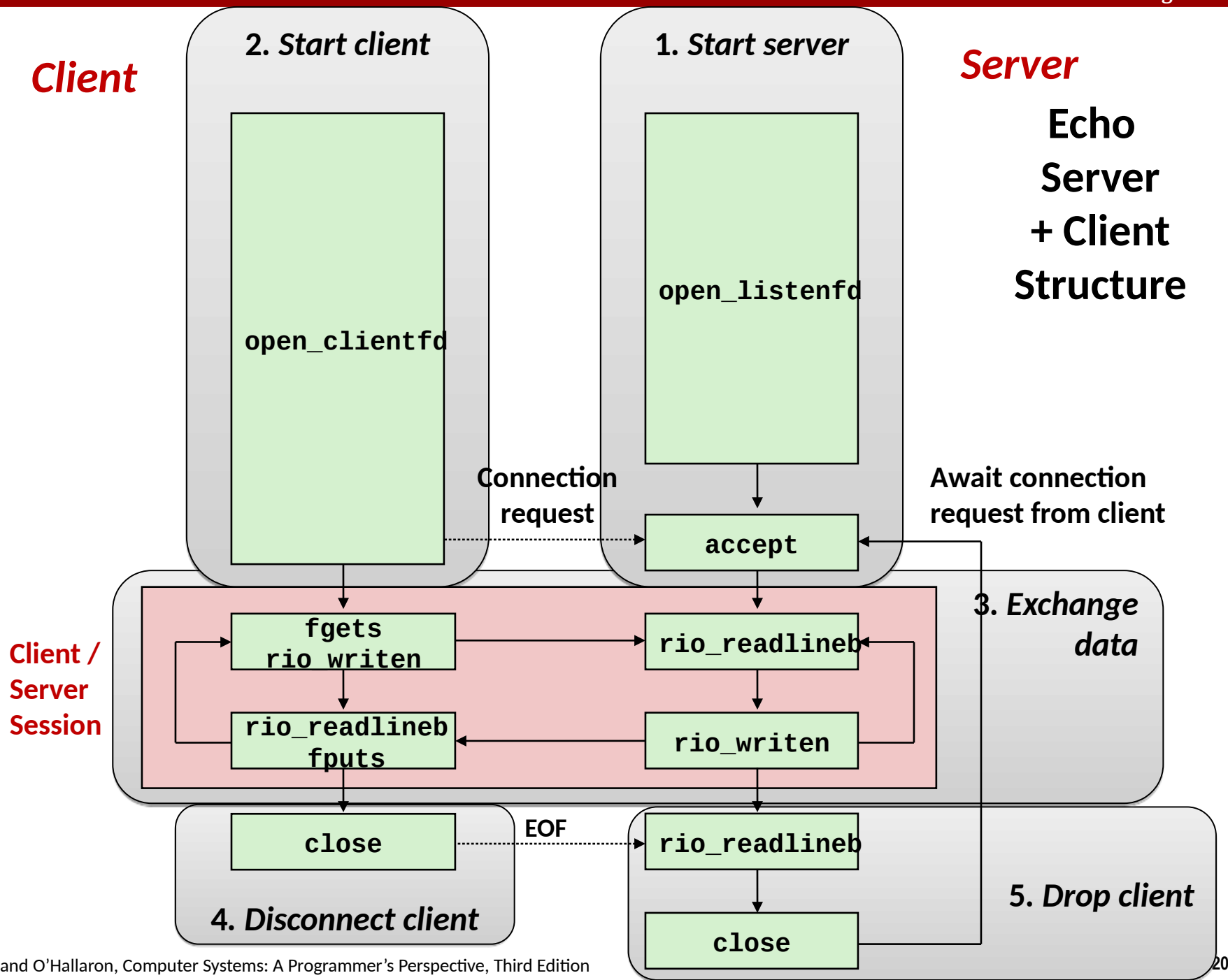
Socket Programming Example

- **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

Client**2. Start client**`open_clientfd`**1. Start server**`open_listenfd`**Server****Echo
Server
+ Client
Structure**Connection
requestAwait connection
request from client`accept`**Client /
Server
Session**`terminal read
socket write``socket read``socket read
terminal write``socket write`**3. Exchange
data**`close`

EOF

`socket read`**4. Disconnect client**`close`**5. Drop client**



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];

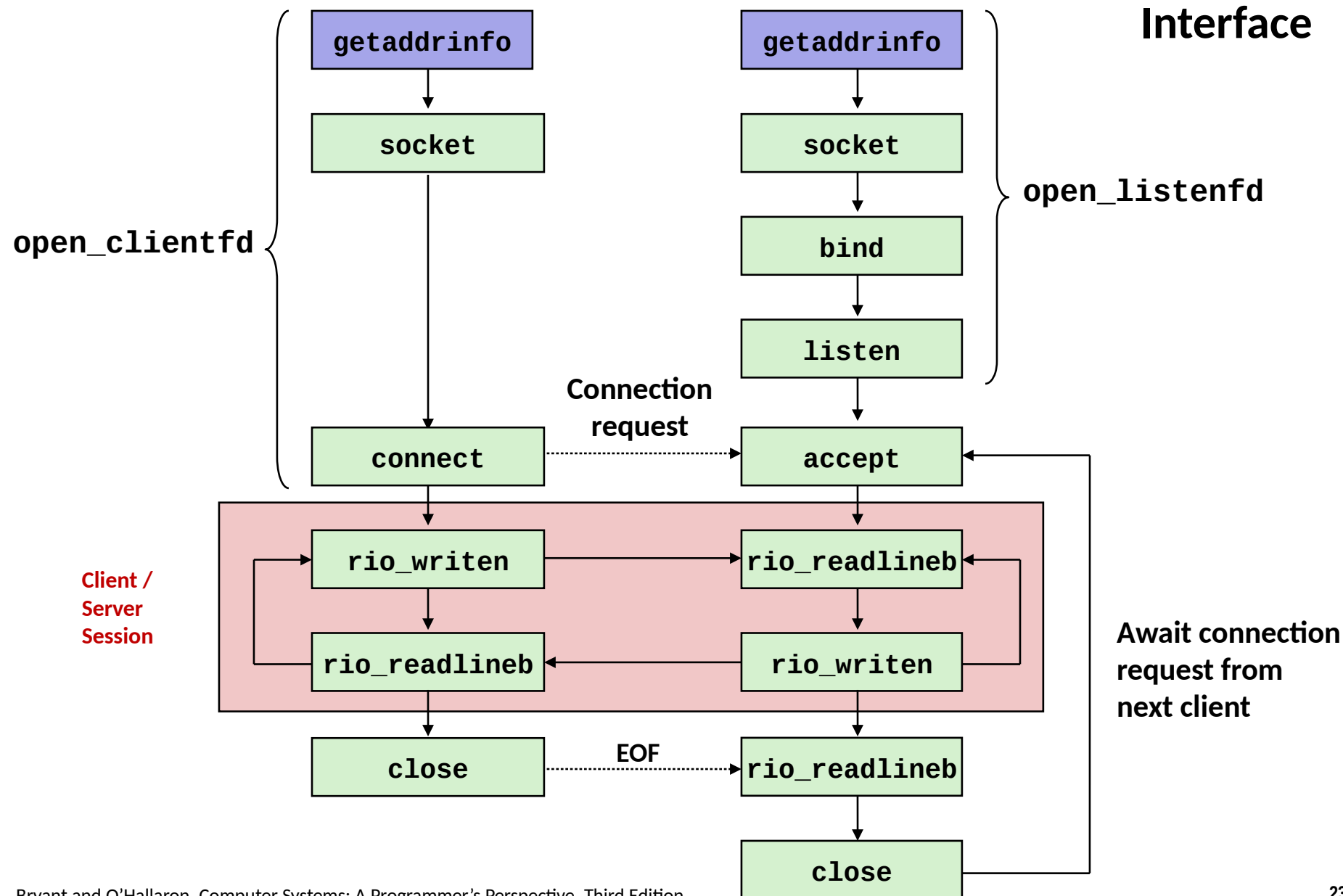
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *)&clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE,
0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

echoserveri.c

Sockets Interface

Client

Server



Sockets Interface: `socket`

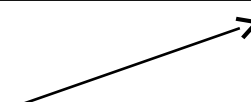
- Clients and servers use the `socket` function to create a *socket descriptor*:

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

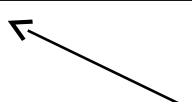
- Example:

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

Indicates that we are using
32-bit IPV4 addresses

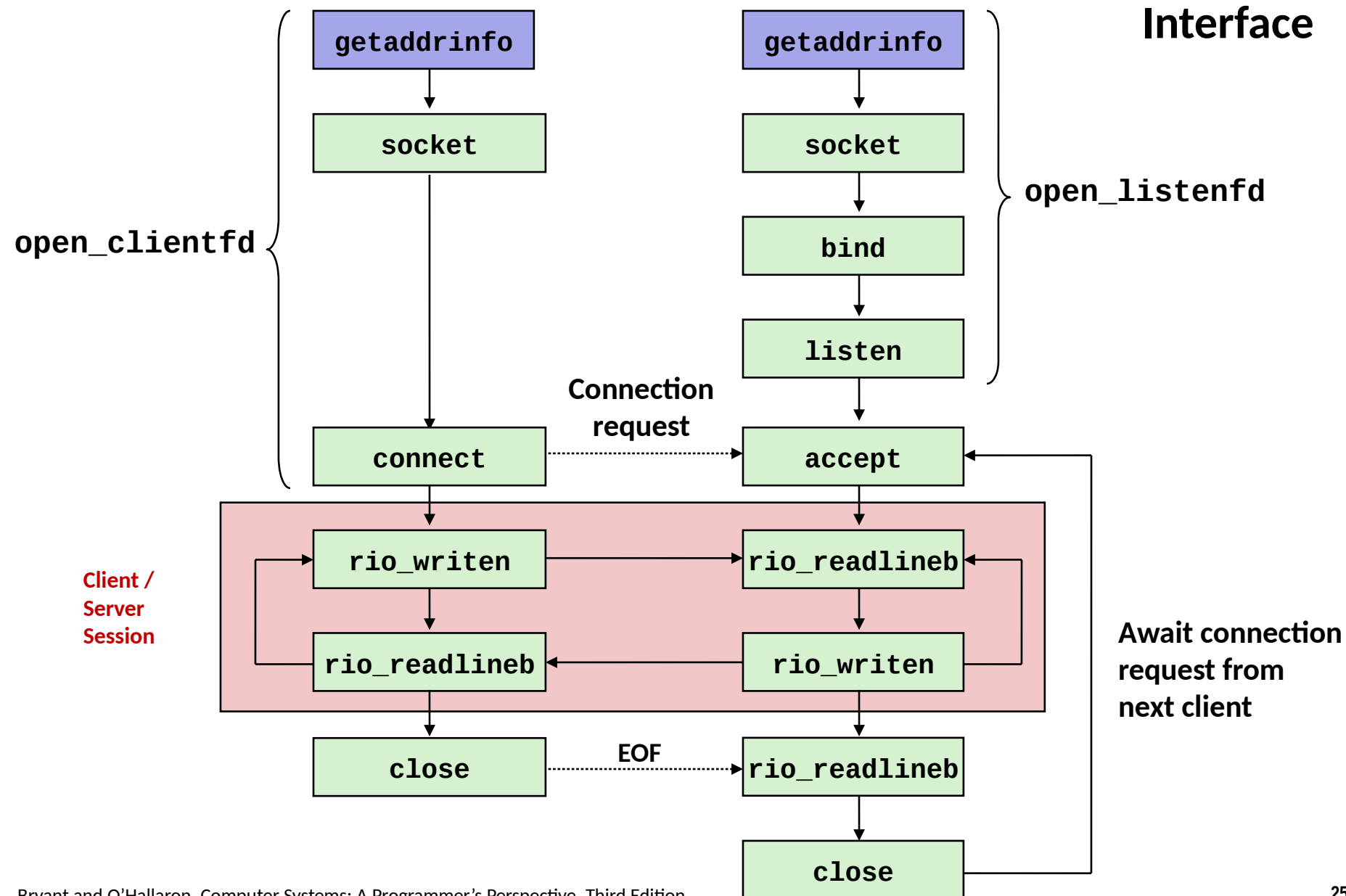


Indicates that the socket
will be the end point of a
connection



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

Sockets Interface

*Client**Server*

Sockets Interface: `bind`

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

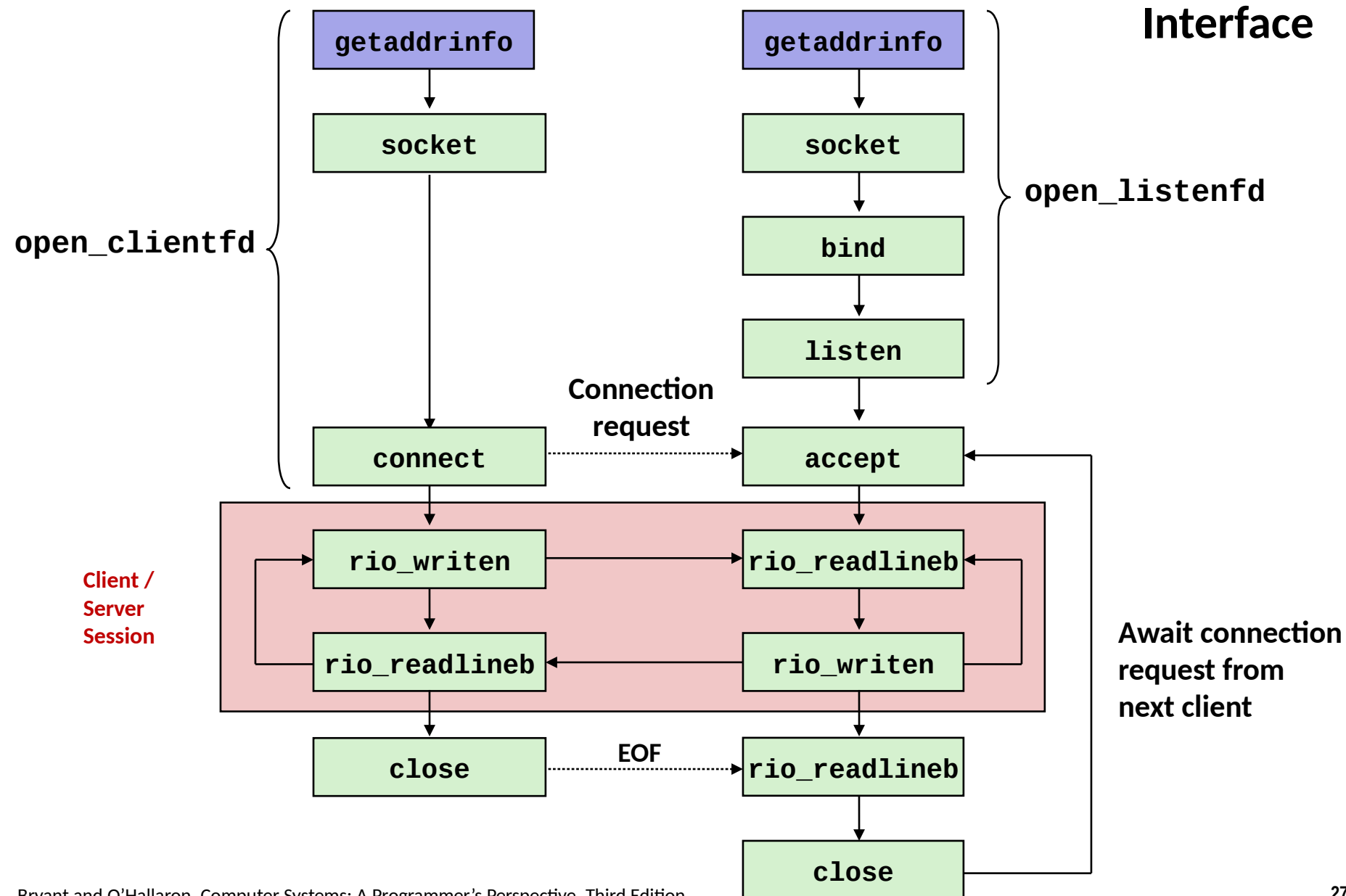
```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

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

*Client**Server*

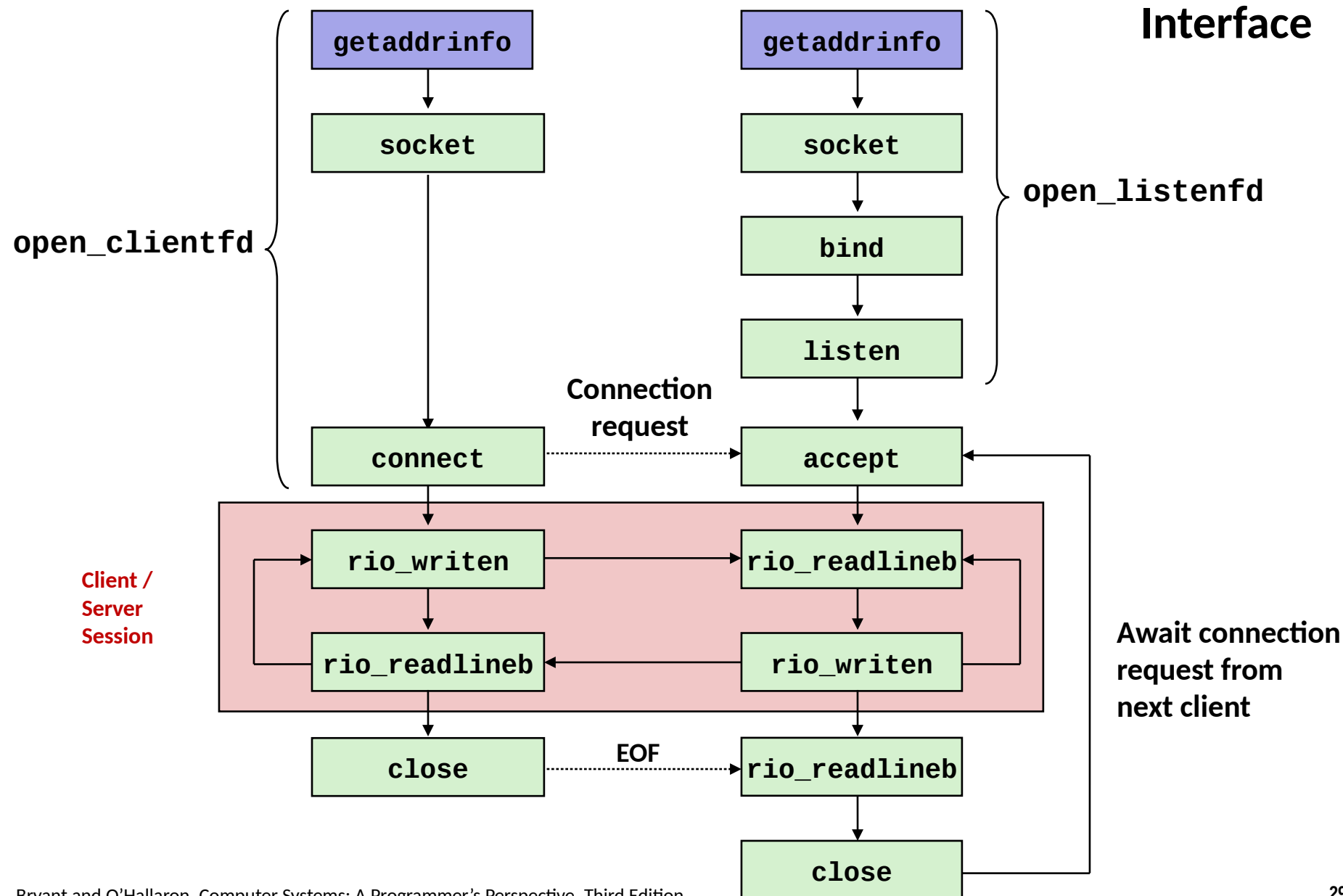
Sockets Interface: `listen`

- 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:
- Converts `int listen(int sockfd, int backlog);` *ing 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



Sockets Interface: `accept`

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

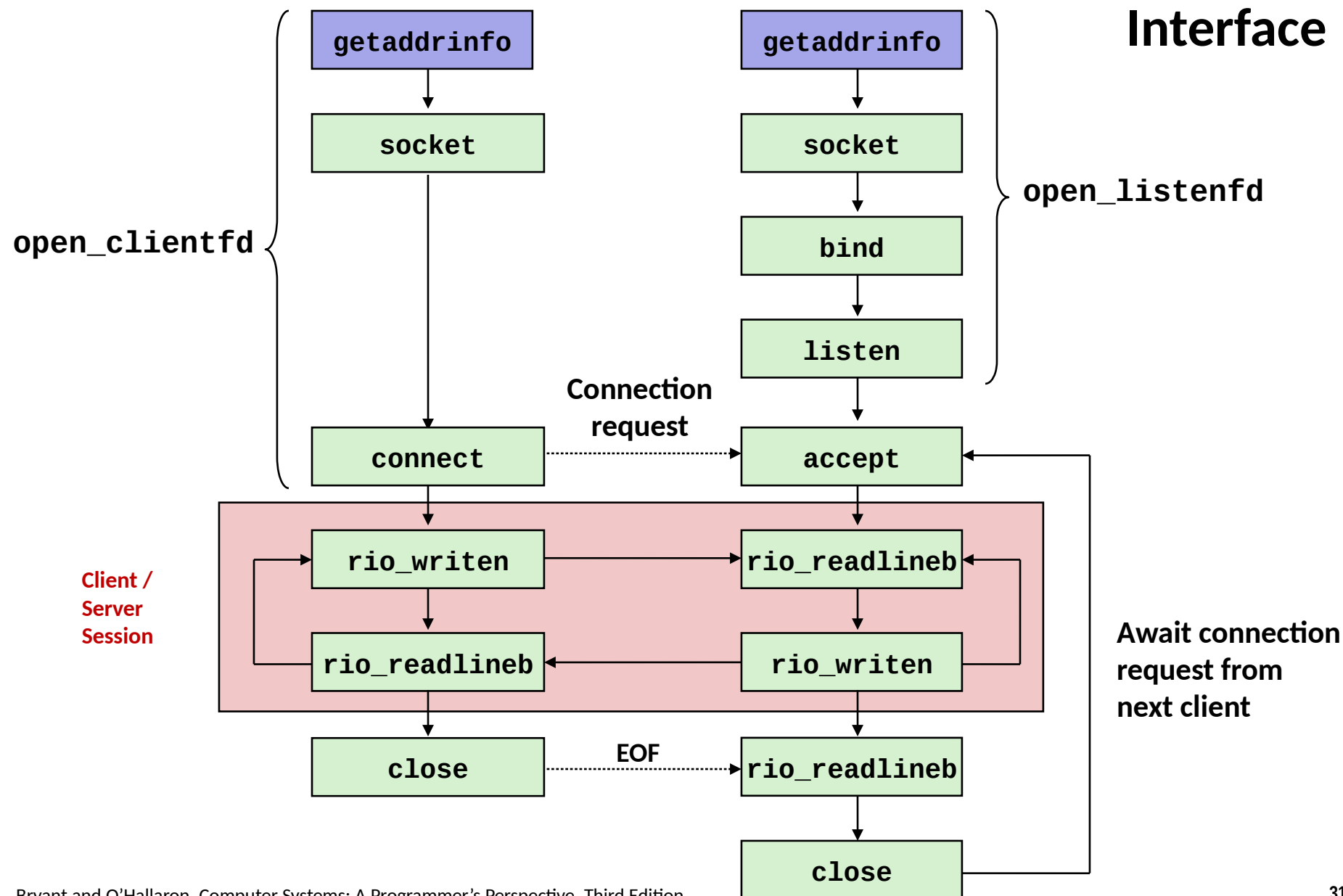
```
int accept(int listenfd, SA *addr, int *addrlen);
```

- 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



Sockets Interface: `connect`

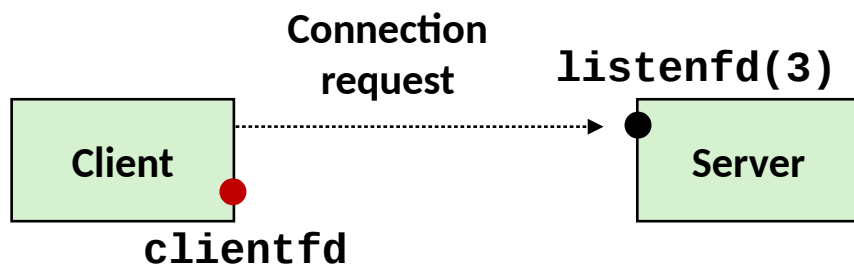
- A client establishes a connection with a server by calling `connect`:
- At `int connect(int clientfd, SA *addr, socklen_t addrlen);`
 - If successful, then `clientfd` is now ready for reading and writing.
 - Resulting connection is characterized by socket pair
(`x:y`, `addr.sin_addr:addr.sin_port`)
 - `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



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

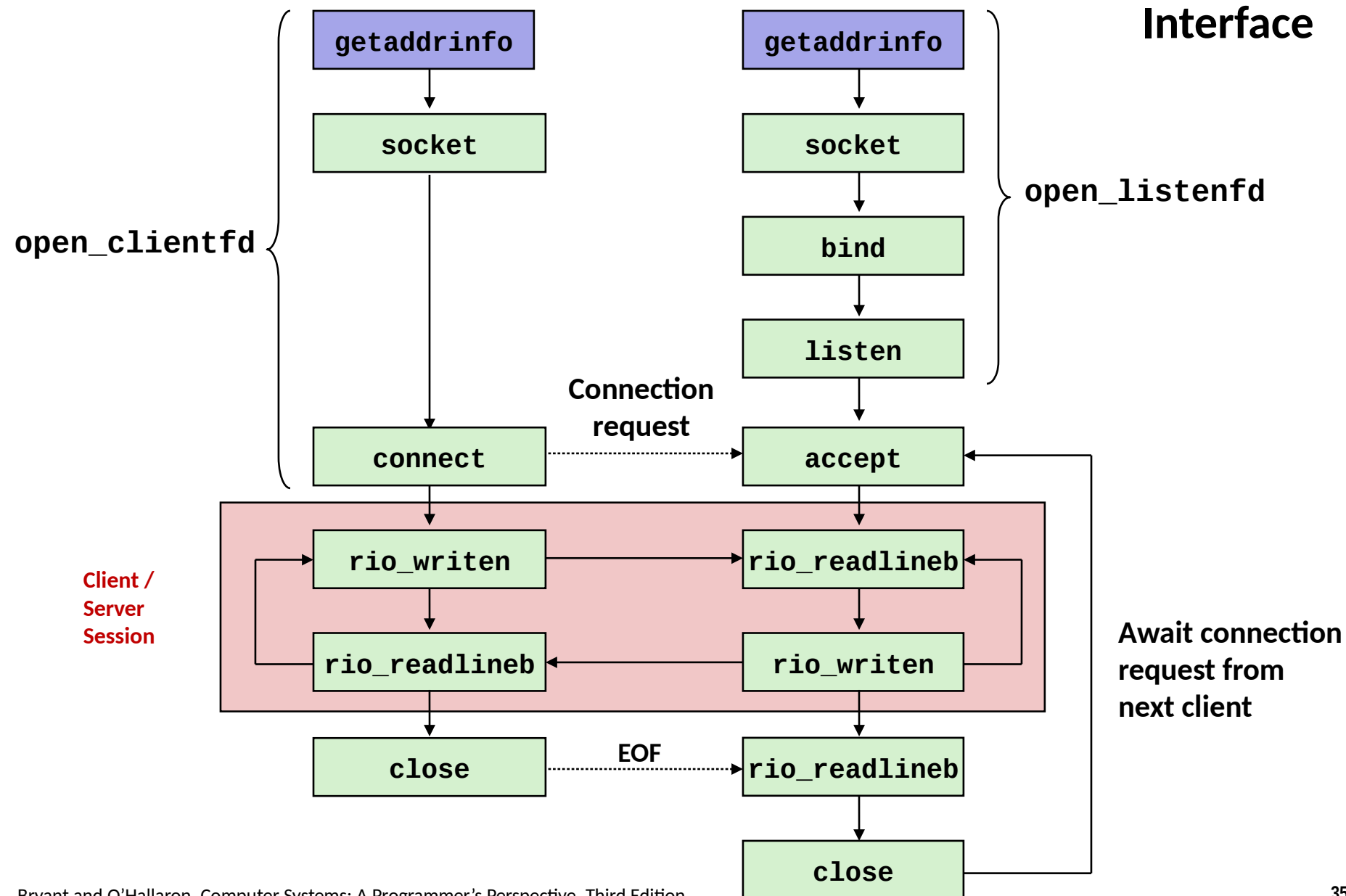
■ Connected descriptor

- 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

*Client**Server*

Summary

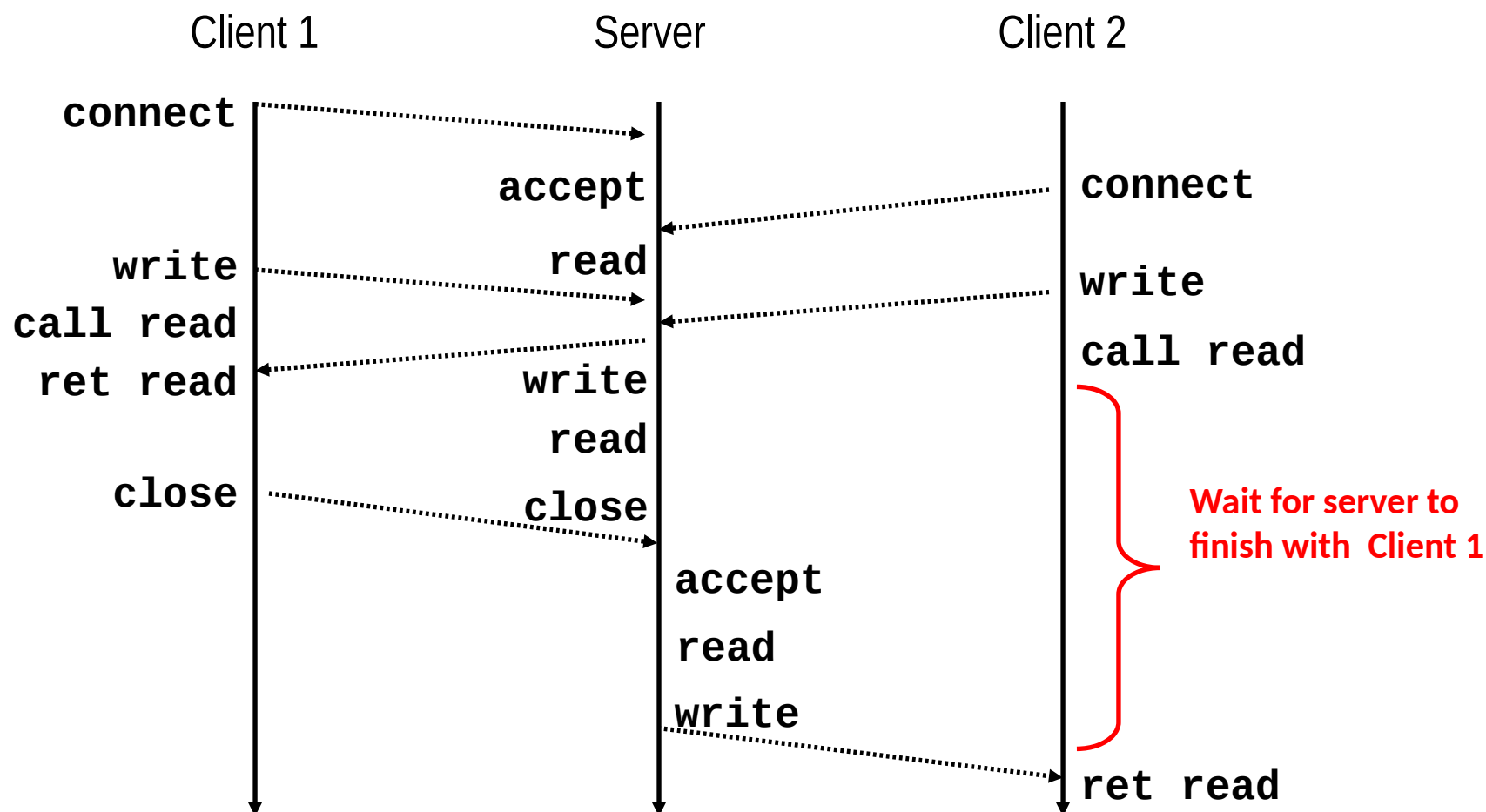
- **Sockets used to communicate across processes over a network (even same network card)**
 - TCP sockets – Listening vs connecting sockets
 - Quirks in structs representing network addresses.
 - Use `getaddrinfo()` or fill up the struct yourself.
 - Usage of `rio` library for buffered I/O.

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>`

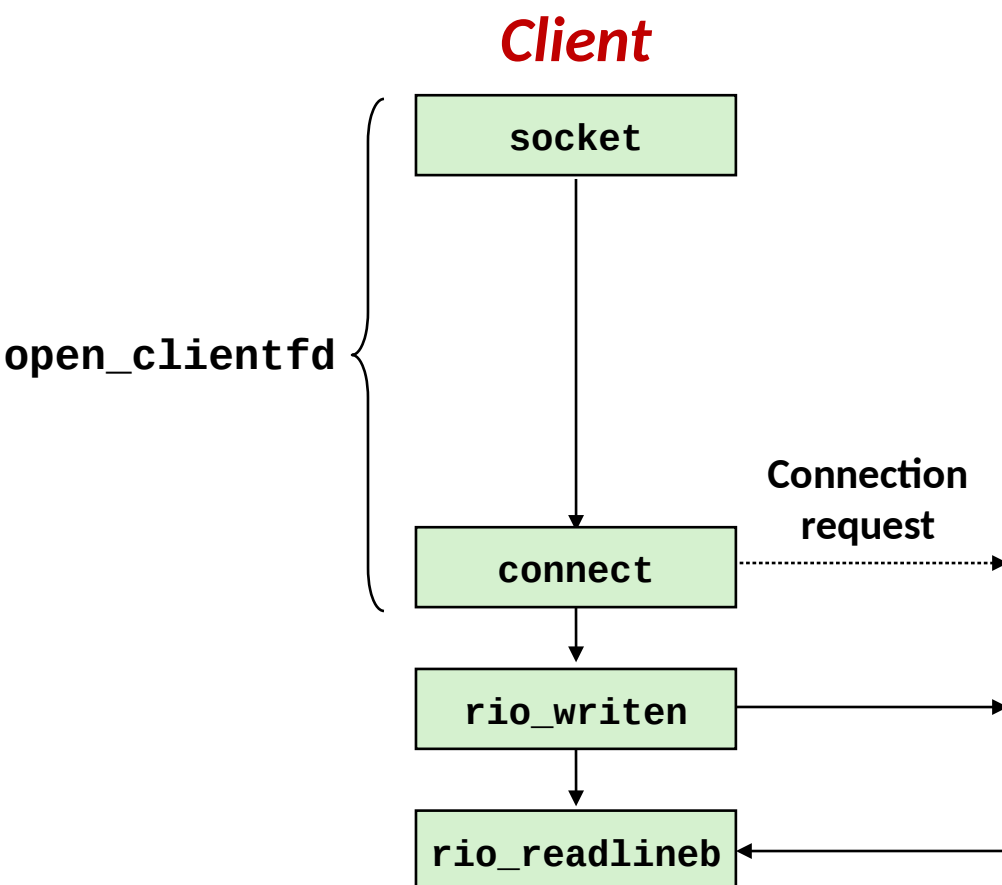
Iterative Servers

- Iterative servers process one request at a time



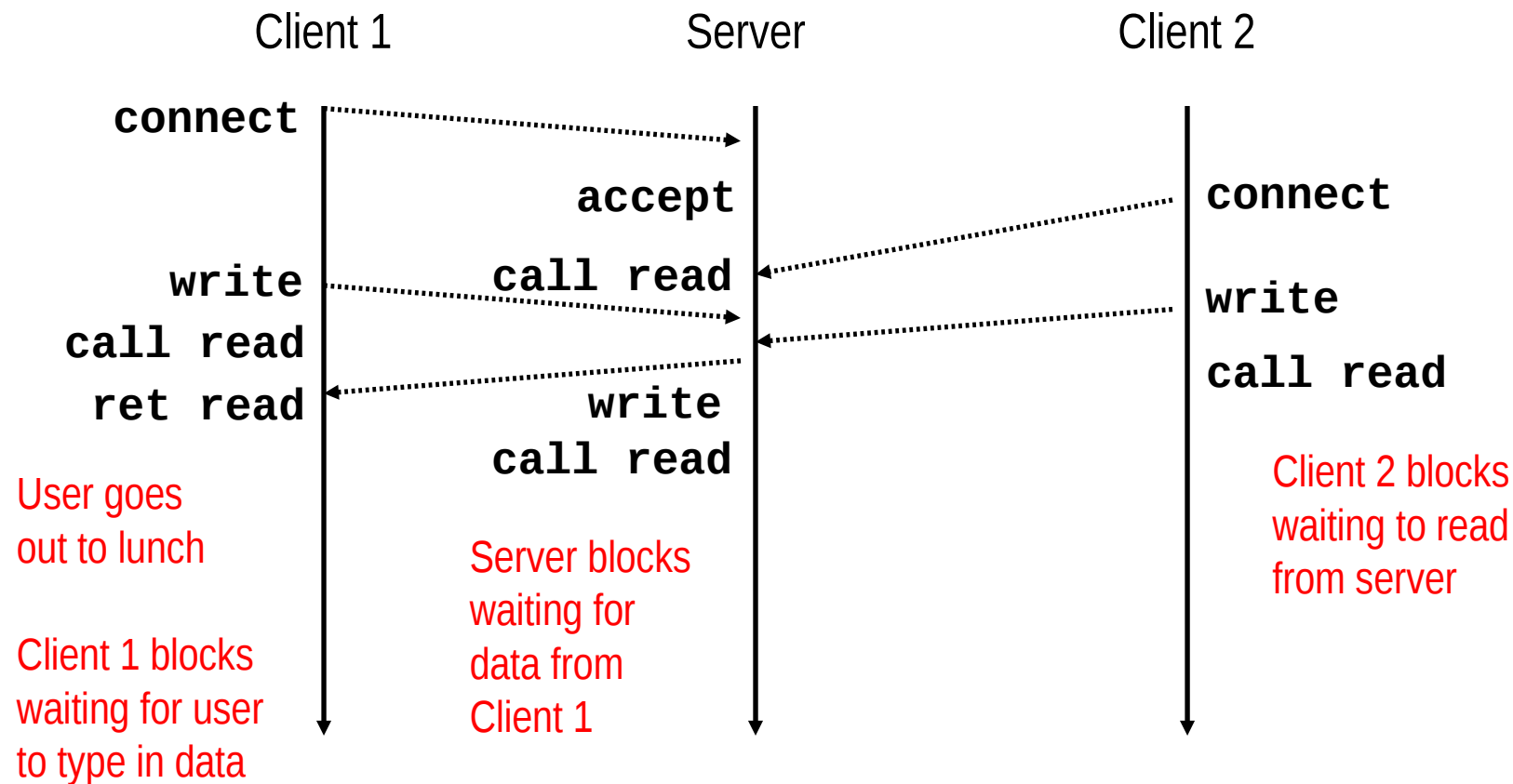
Where Does Second Client Block?

- Second client attempts to connect to iterative server



- Call to connect returns
 - Even though connection not yet accepted
 - Server side TCP manager queues request
 - Feature known as “TCP listen backlog”
- Call to rio_writen returns
 - Server side TCP manager buffers input data
- Call to rio_readlineb blocks
 - Server hasn't written anything for it to read yet.

Fundamental Flaw of Iterative Servers



■ Solution: use *concurrent servers* instead

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

1. Process-based

- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

2. Event-based

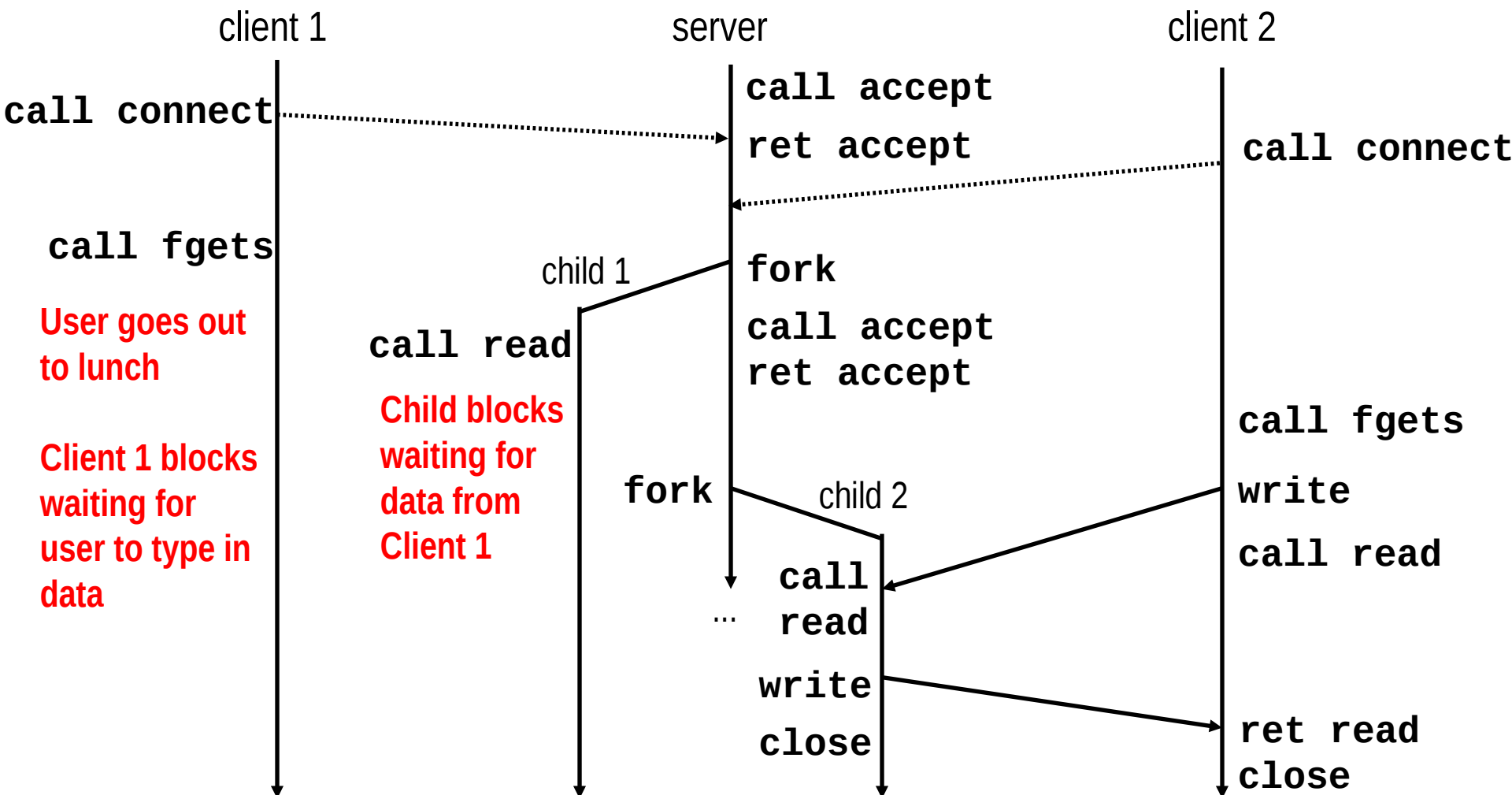
- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

3. Thread-based

- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of of process-based and event-based.

Approach #1: Process-based Servers

- Spawn separate process for each client



Process-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);  /* Child closes connection with client */
            exit(0);        /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

Process-Based Concurrent Echo Server (cont)

```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

echoserverp.c

- Reap all zombie children

Issues with Process-based Servers

- **Listening server process must reap zombie children**
 - to avoid fatal memory leak

- **Parent process must `close` its copy of `connfd`**
 - Kernel keeps reference count for each socket/open file
 - After fork, `refcnt(connfd) = 2`
 - Connection will not be closed until `refcnt(connfd) = 0`

Pros and Cons of Process-based Servers

- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
 - descriptors (no)
 - file tables (yes)
 - global variables (no)
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
 - Requires IPC (interprocess communication) mechanisms
 - FIFO's (named pipes), System V shared memory and semaphores

Approach #2: Event-based Servers

- **Server maintains set of active connections**
 - Array of `connfd`'s
- **Repeat:**
 - Determine which descriptors (`connfd`'s or `listenfd`) have pending inputs
 - e.g., using `select` or `epoll` functions
 - arrival of pending input is an *event*
 - If `listenfd` has input, then `accept` connection
 - and add new `connfd` to array
 - Service all `connfd`'s with pending inputs
- **Details for select-based server in book**

Pros and Cons of Event-based Servers

- + One logical control flow and address space.
- + Can single-step with a debugger.
- + No process or thread control overhead.
 - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- – Significantly more complex to code than process- or thread-based designs.
- – Hard to provide fine-grained concurrency
 - E.g., how to deal with partial HTTP request headers
- – Cannot take advantage of multi-core
 - Single thread of control

Approach #3: Thread-based Servers

- **Very similar to approach #1 (process-based)**
 - ...but using threads instead of processes

Threads vs. Processes

■ How threads and processes are similar

- Each has its own logical control flow
- Each can run concurrently with others (possibly on different cores)
- Each is context switched

■ How threads and processes are different

- Threads share all code and data (except local stacks)
 - Processes (typically) do not
- Threads are somewhat less expensive than processes
 - Process control (creating and reaping) twice as expensive as thread control
 - Linux numbers:
 - ~20K cycles to create and reap a process
 - ~10K cycles (or less) to create and reap a thread

Posix Threads (Pthreads) Interface

- ***Pthreads***: Standard interface for ~60 functions that manipulate threads from C programs
 - Creating and reaping threads
 - `pthread_create()`
 - `pthread_join()`
 - Determining your thread ID
 - `pthread_self()`
 - Terminating threads
 - `pthread_cancel()`
 - `pthread_exit()`
 - `exit()` [terminates all threads] , `RET` [terminates current thread]
 - Synchronizing access to shared variables
 - `pthread_mutex_init`
 - `pthread_mutex_[un]lock`

Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                          (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

echoserv.c

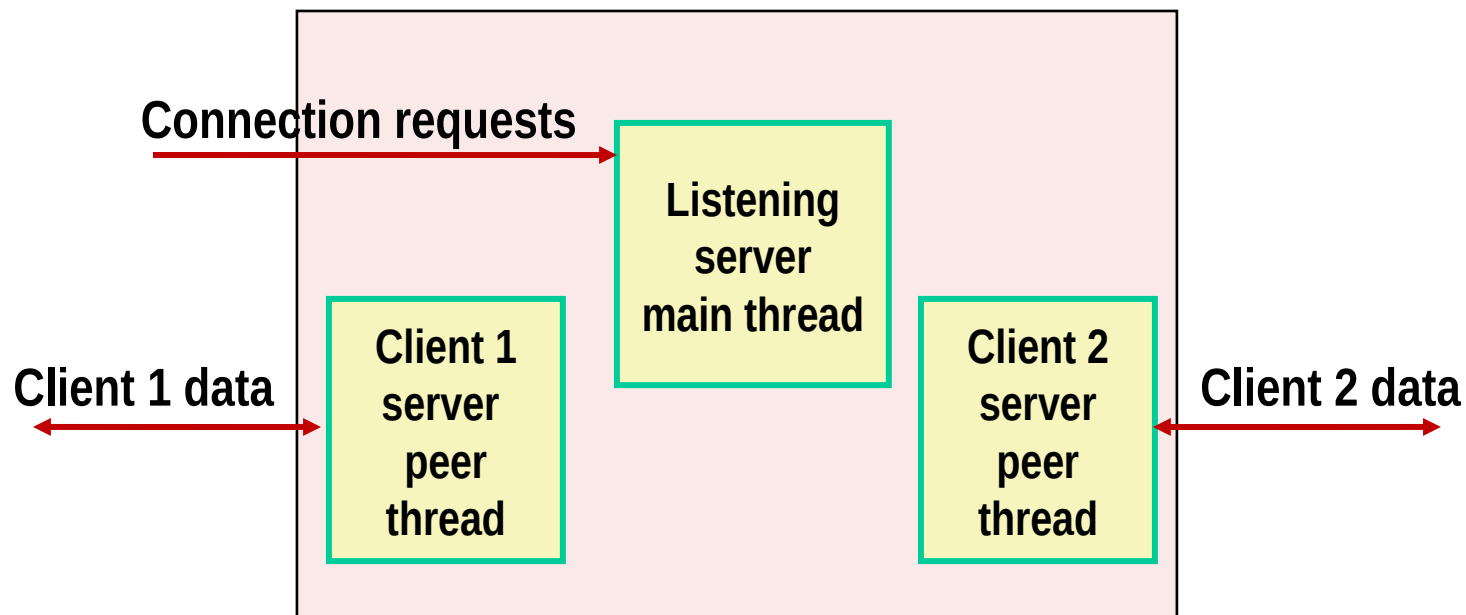
- malloc of connected descriptor necessary to avoid deadly race

Thread-Based Concurrent Server (cont)

```
/* Thread routine */  
void *thread(void *vargp)  
{  
    int connfd = *((int *)vargp);  
    Pthread_detach(pthread_self());  
    Free(vargp);  
    echo(connfd);  
    Close(connfd);  
    return NULL;  
} echoserver.c
```

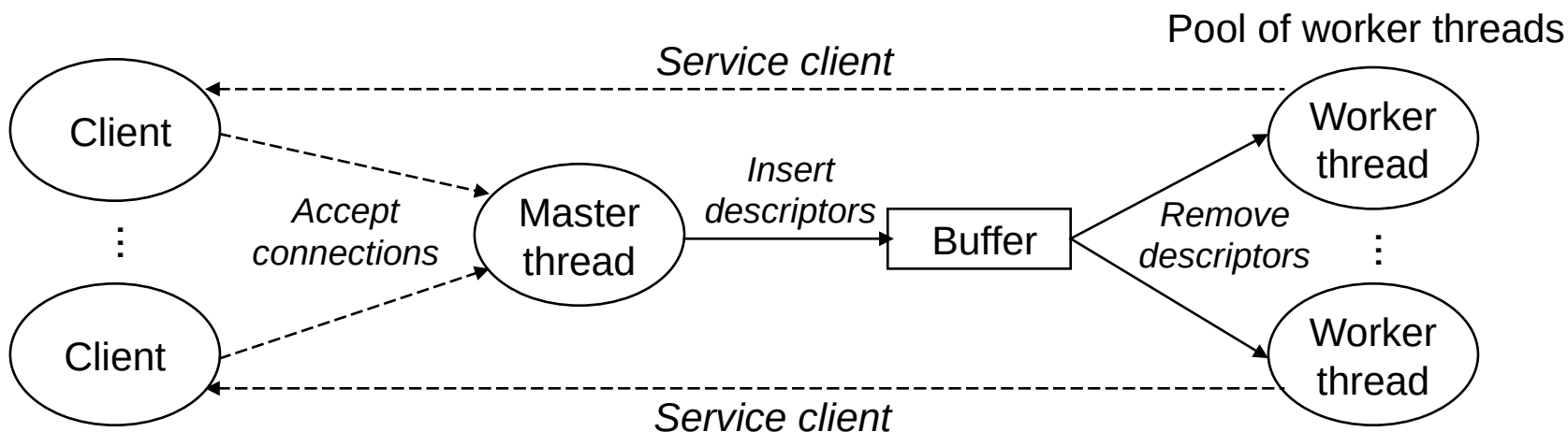
- Run thread in “detached” mode.
 - Runs independently of other threads
 - Reaped automatically (by kernel) when it terminates
- Free storage allocated to hold connfd.
- Close connfd (important!)

Thread-based Server Execution Model



- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables

Pre-threaded Server Model



- Clients handled using a thread-pool architecture
- Bounded/Unbounded buffer can be used for synchronization

Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
 - e.g., logging information, file cache
- **+ Threads are more efficient than processes**
- **- Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
 - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
 - Hard to know which data shared & which private
 - Hard to detect by testing
 - Probability of bad race outcome very low
 - But nonzero!

Summary: Approaches to Concurrency

■ Process-based

- Hard to share resources: Easy to avoid unintended sharing
- High overhead in adding/removing clients

■ Event-based

- Tedious and low level
- Total control over scheduling
- Very low overhead
- Cannot create as fine grained a level of concurrency
- Does not make use of multi-core

■ Thread-based

- Easy to share resources: Perhaps too easy
- Medium overhead
- Not much control over scheduling policies
- Difficult to debug
 - Event orderings not repeatable

Code references can be found at
<http://csapp.cs.cmu.edu/3e/code.html>

Assignment A6

- Build client server portion of a distributed chat service

