What is a File System?

15-213 / 15-513 / 14-513: Introduction to Computer Systems
Lecture 20, Nov 12, 2024

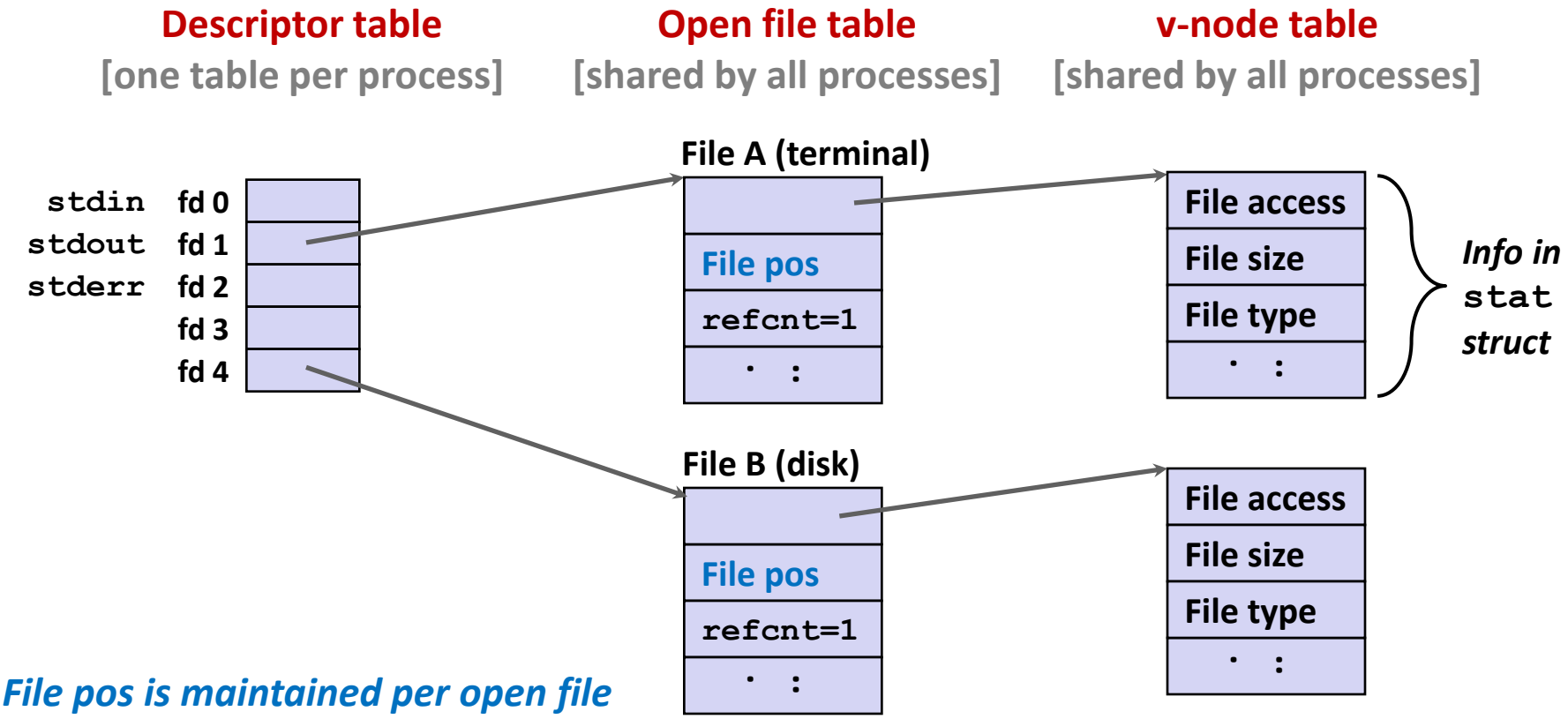
Instructors:

Brian Railing

David Varodayan

How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



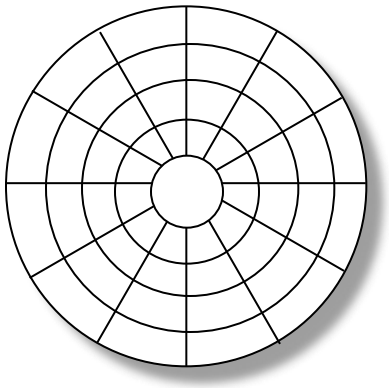
File pos is maintained per open file

Today

- What is a File System?
- Managing a file system
- Common operations

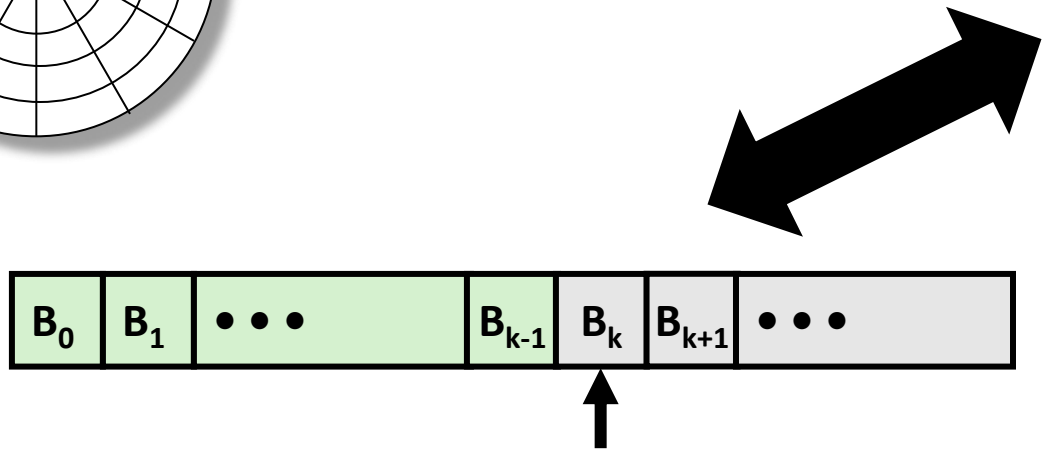
File System

- Manages disk blocks to provide a file abstraction



Surface* organized into tracks

Tracks divided into sectors



*Durable storage has many architectures, but ultimately they expose “blocks”

Making a File System

- **File systems start by formatting raw disk blocks**
  - Designate one (or more) blocks as “super”
  - Record the rest of the blocks as free

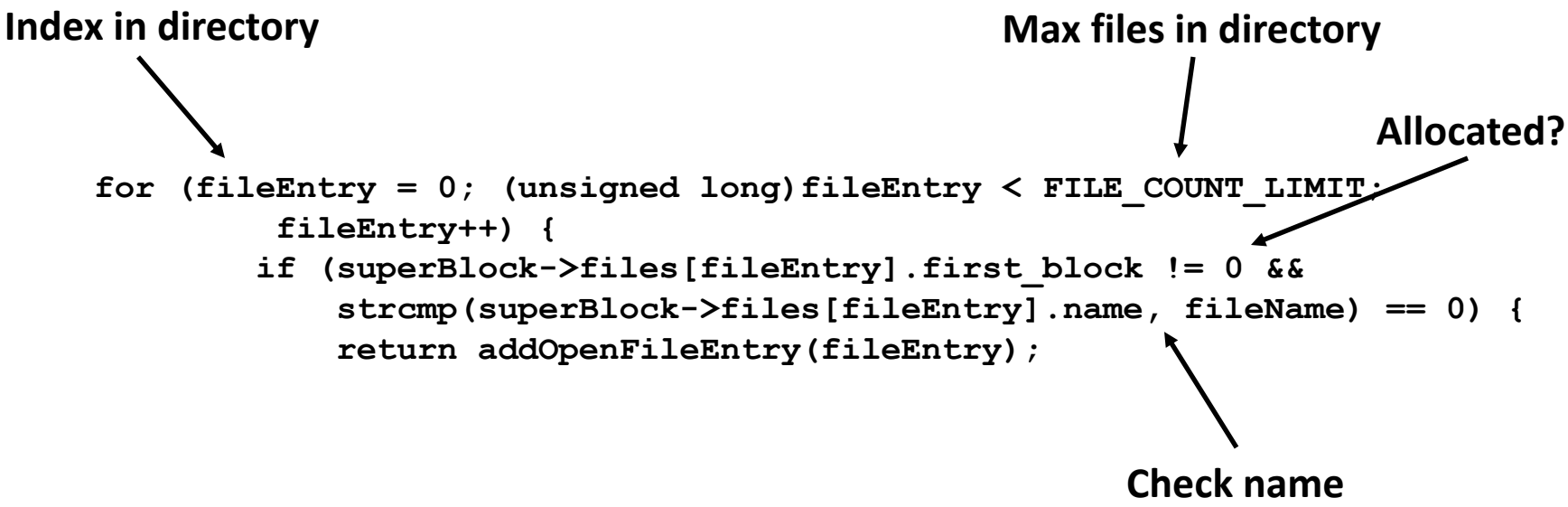
Managing a File System

- **"Super" block is the master block with information**
  - Type information
  - Size
  - Root directory
  - Free blocks

- **SFS has a flat directory structure, so the root directory is part of the superblock**

Finding a File

- **A directory is a special file**
  - Maps strings to files
  - Those files could also be directories



Files and Names

- File contents are separate from the file's "name"
- File extensions do not set the "type" of the data
- Names are just a "key" to locate the data

Opening a File

- **Find the file**
- **Create the three table entries**
  - Find an available file descriptor
  - Allocate an open file table entry
    - Pos, permissions, etc
  - Load file info into memory
    - *SFS is always in-memory, so this is implicit

Reading a File

- **The file system will map file pos to disk blocks**
- **Lots of ways to map**
  - Contiguous
  - Linked / FAT ← **SFS**
  - Indexed

Writing a File

- **Like reading, but the file could grow**
  - SFS preallocates space
  - Interesting synchronization

Deleting a File

- **Like free(), but …**
  - Can open files be deleted?

- **Two steps:**
  - Removing the mapping
  - Putting the blocks into the free list

SFS Specific Notes

- **“Shark” File System**
  - Uses mmap to bring the entire “disk” file into memory
  - Treats the disk as an array of 512-byte blocks
  - Block 0 is the superblock, other references to 0 are NULLs
  - Flat directory structure

Network Programming: Part II

14-513: Introduction to Computer Systems
21st Lecture, November 14, 2024

Instructors:

David Varodayan

Today

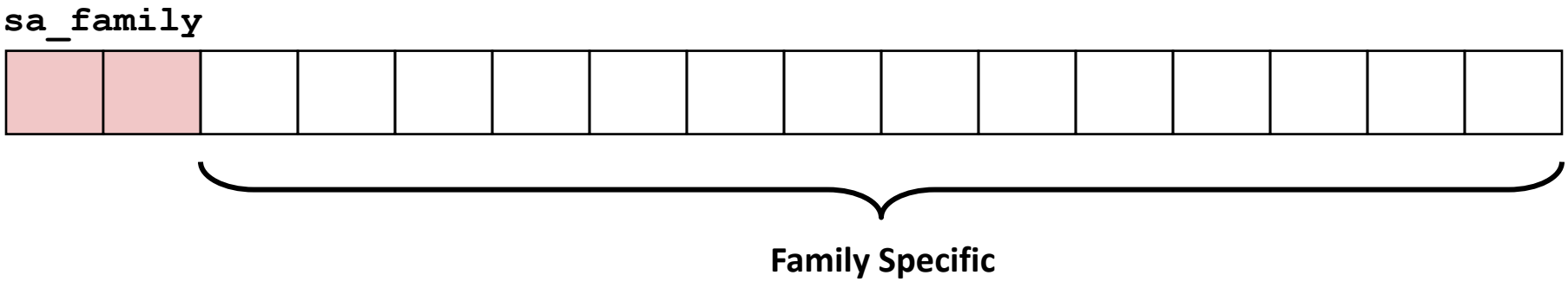
- Getaddrinfo and getnameinfo
- The echo server example in full

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;`

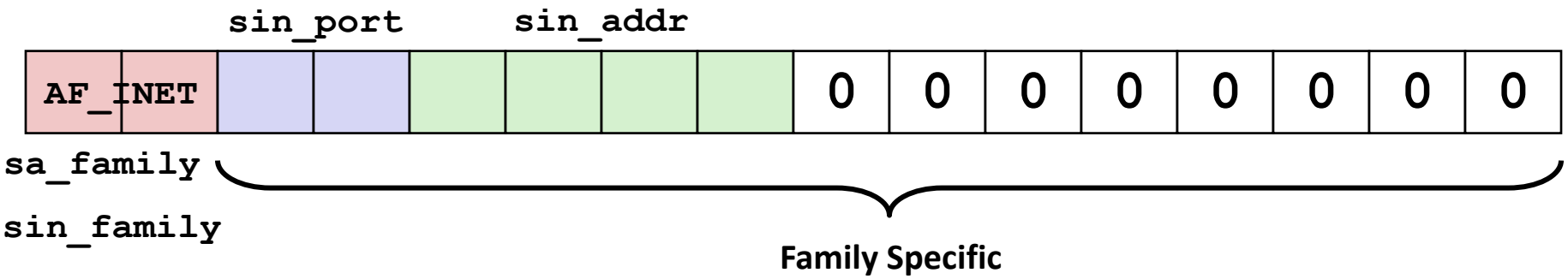
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data  */
};
```



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`

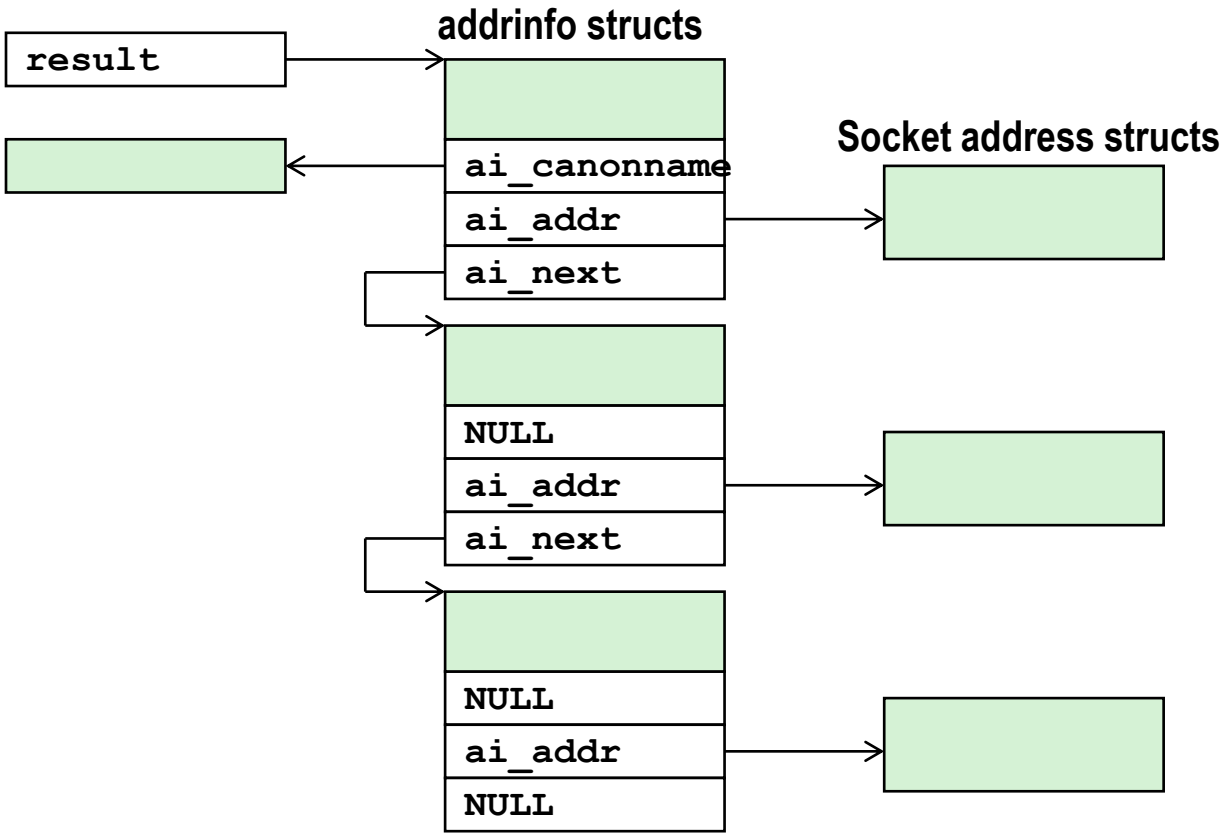
```
int getaddrinfo(const char *host,              /* Hostname or address */
                const char *service,           /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);    /* Output linked list */

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

Linked List Returned by `getaddrinfo`



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

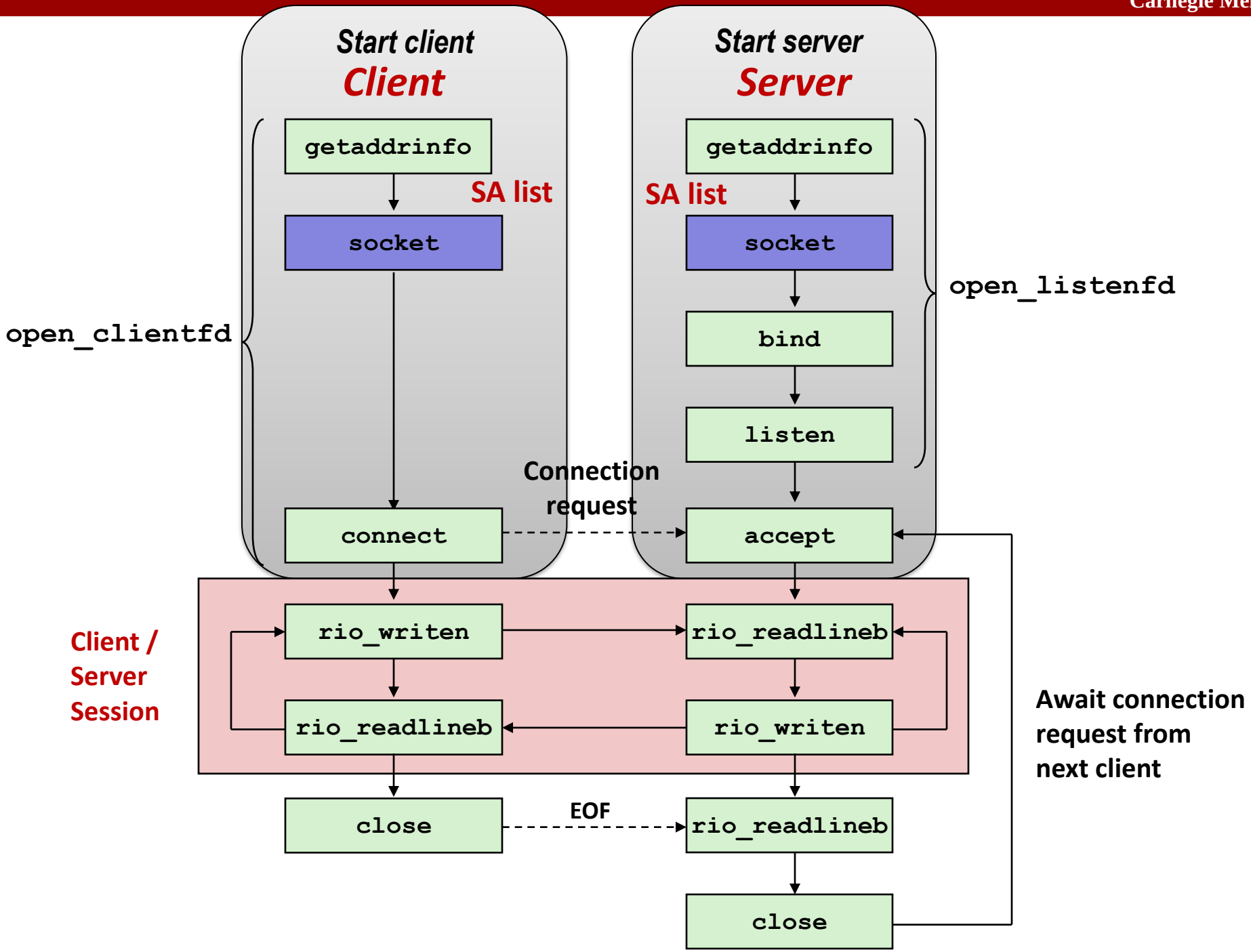
Running hostinfo

```
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
```


Start client
Client
Start server
Server
getaddrinfo
SA list
socket
open_clientfd
getaddrinfo
SA list
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 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);
```

Protocol specific!

Indicates that we are using 32-bit IPV4 addresses

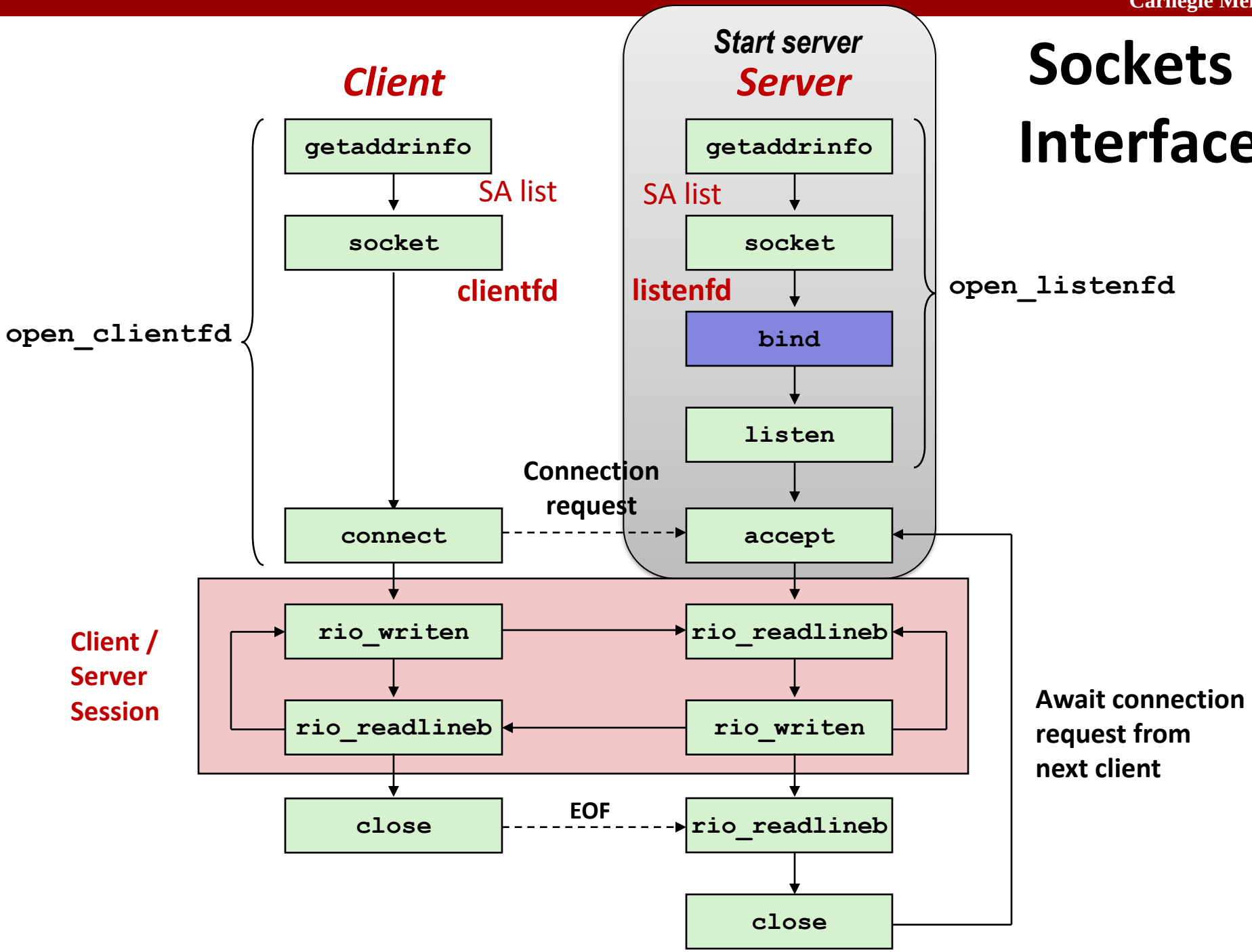
Indicates that the socket will be the end point of a reliable (TCP) connection

- Example:

```
int clientfd = socket(ai->ai_family, ai->ai_socktype,
                      ai->ai_protocol);
```

Use getaddrinfo and you don't have to know or care which protocol!

Sockets Interface



Sockets Interface: `bind`

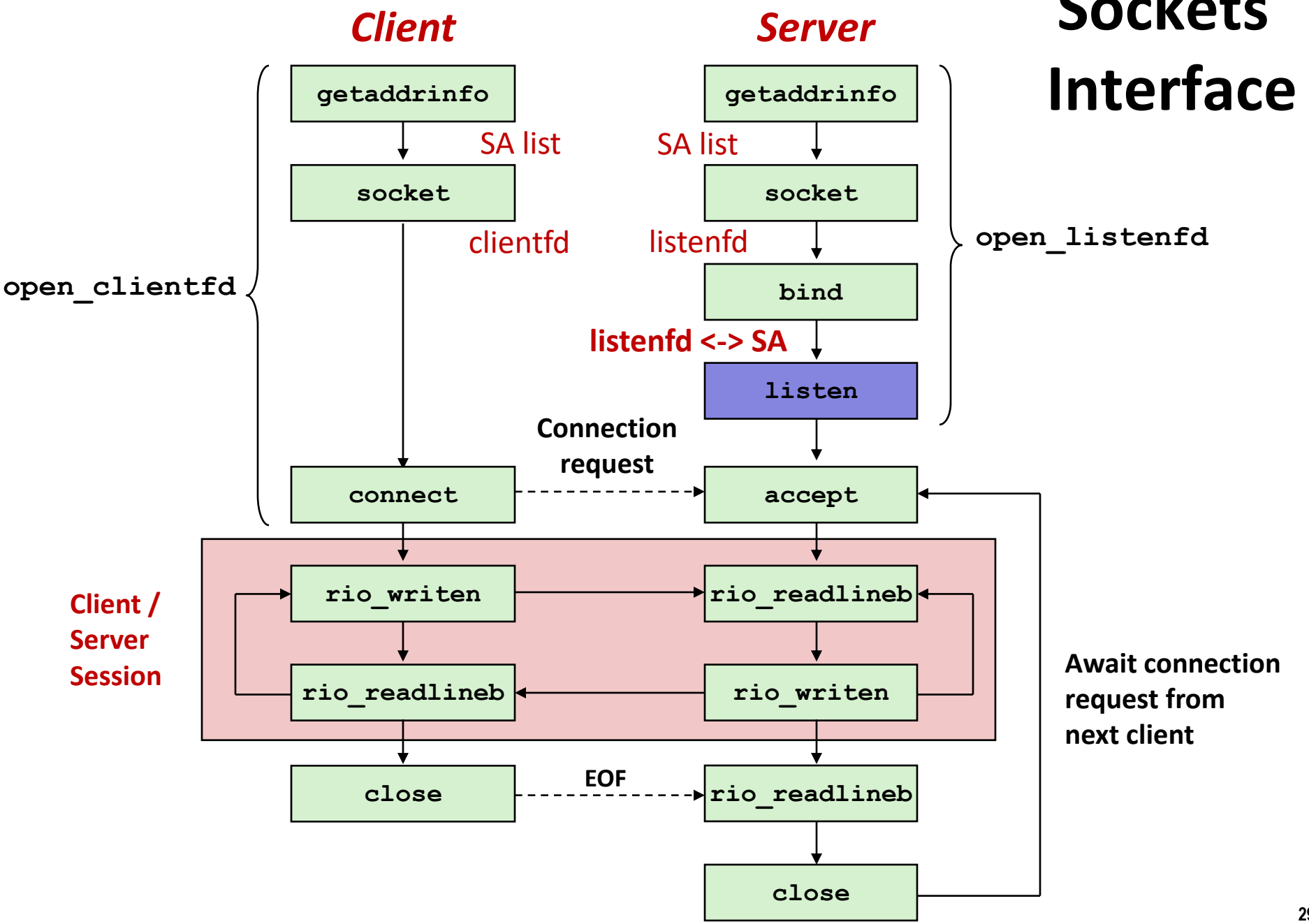
- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

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

Our convention: `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`

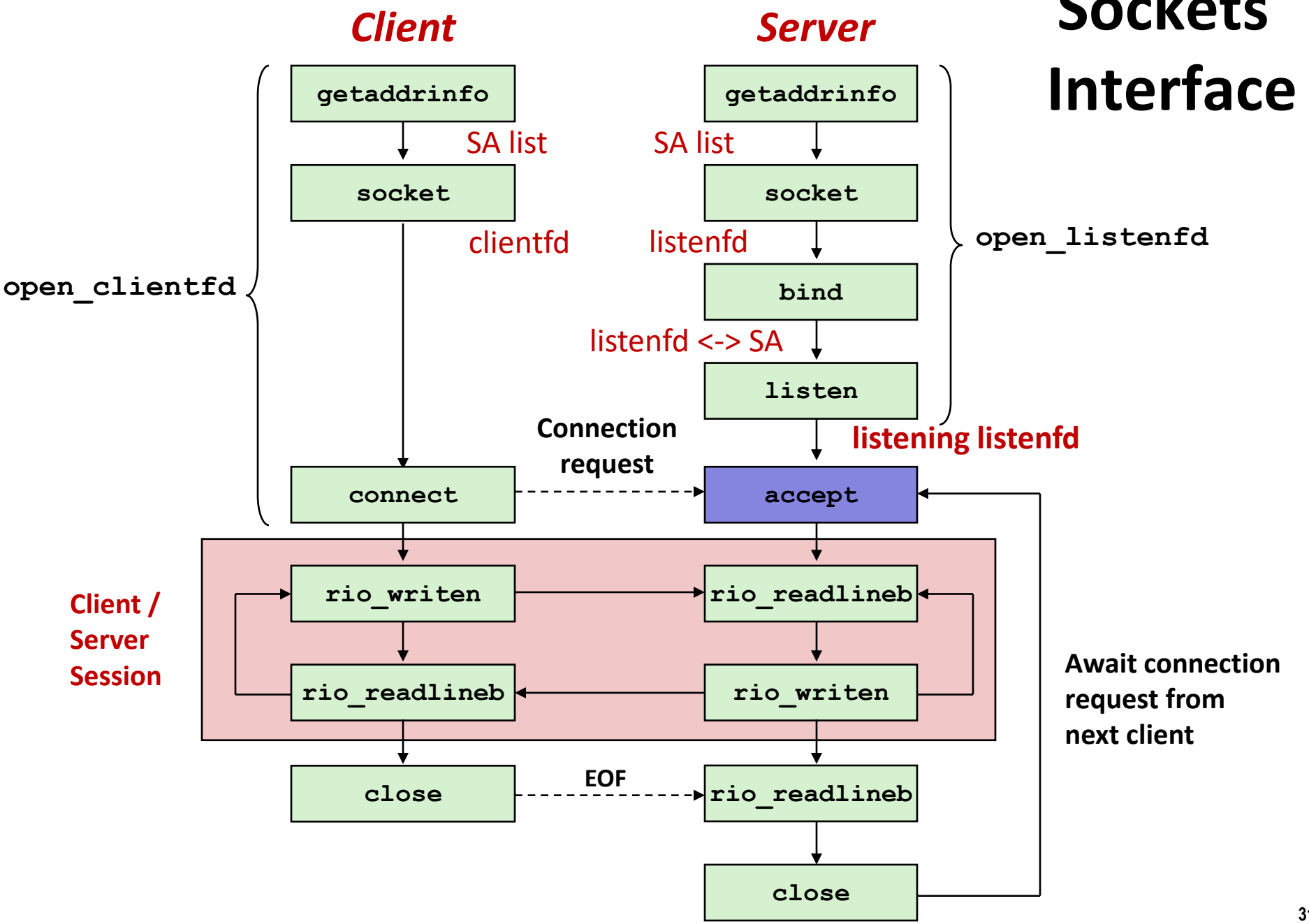
- Kernel assumes that descriptor from socket function is an *active socket* that will be on the client end
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

```
int listen(int sockfd, int backlog);
```

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

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests (128-ish by default)

Sockets Interface



Sockets Interface: `accept`

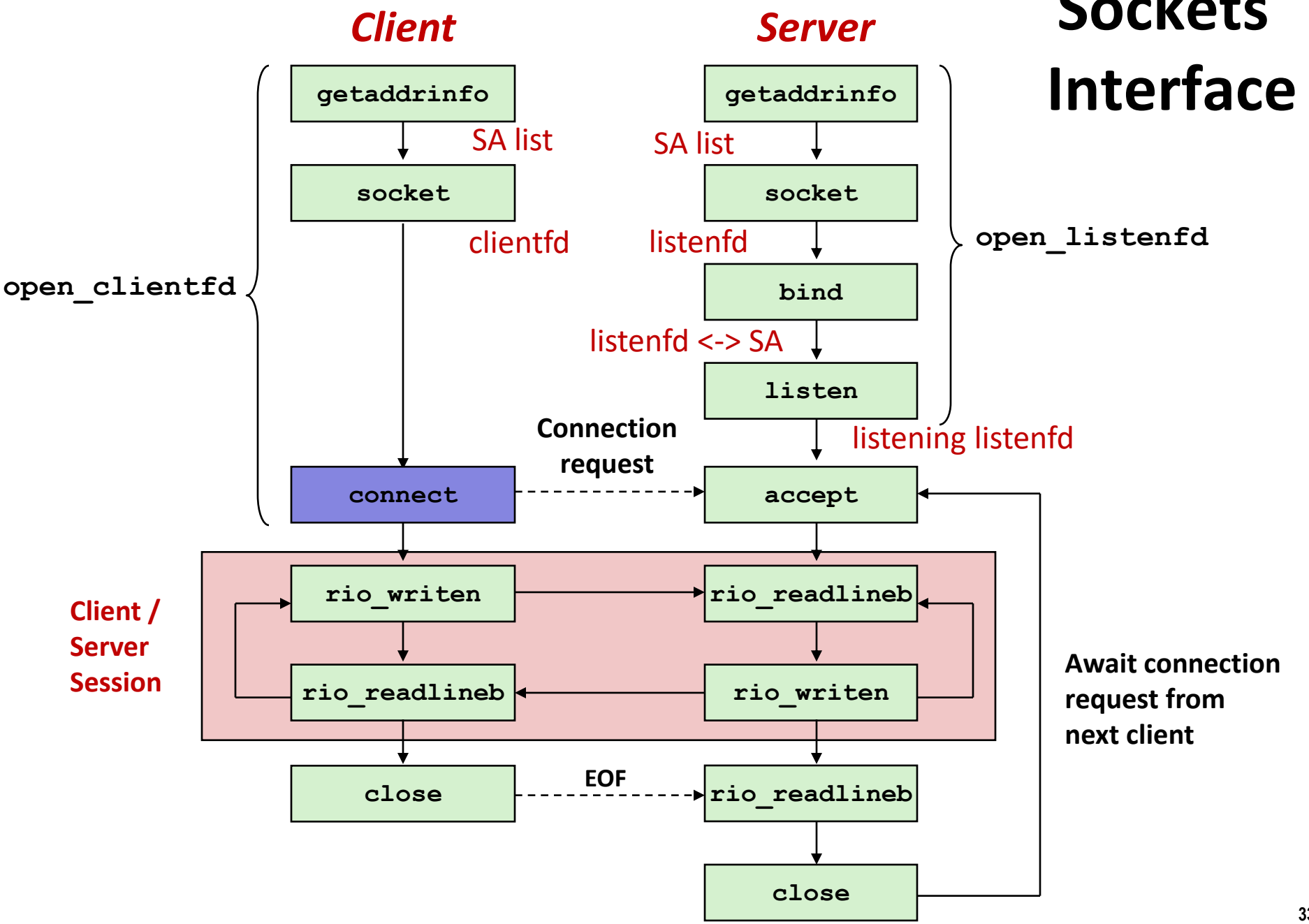
- Servers wait for connection requests from clients by calling `accept`:

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

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.

- Returns a *connected descriptor* `connfd` 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`:**

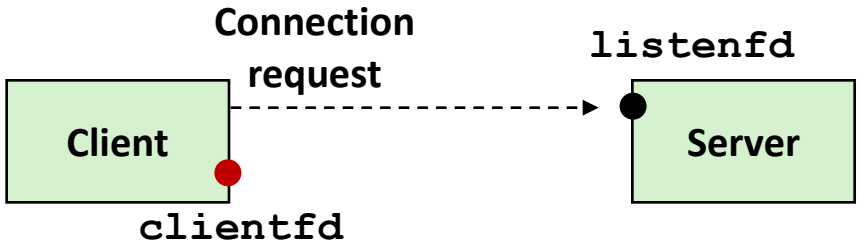
```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- **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`

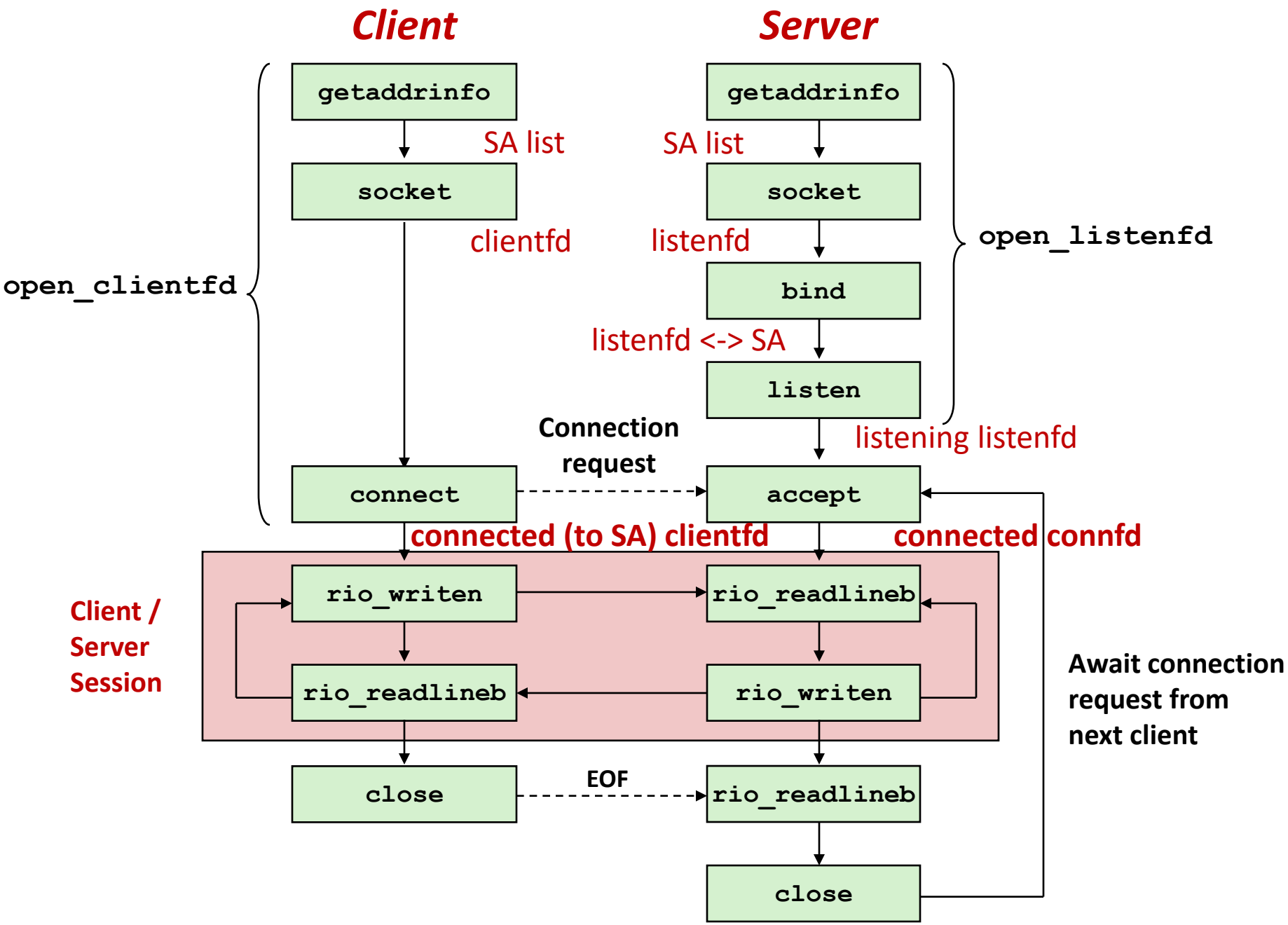
Connected vs. Listening Descriptors

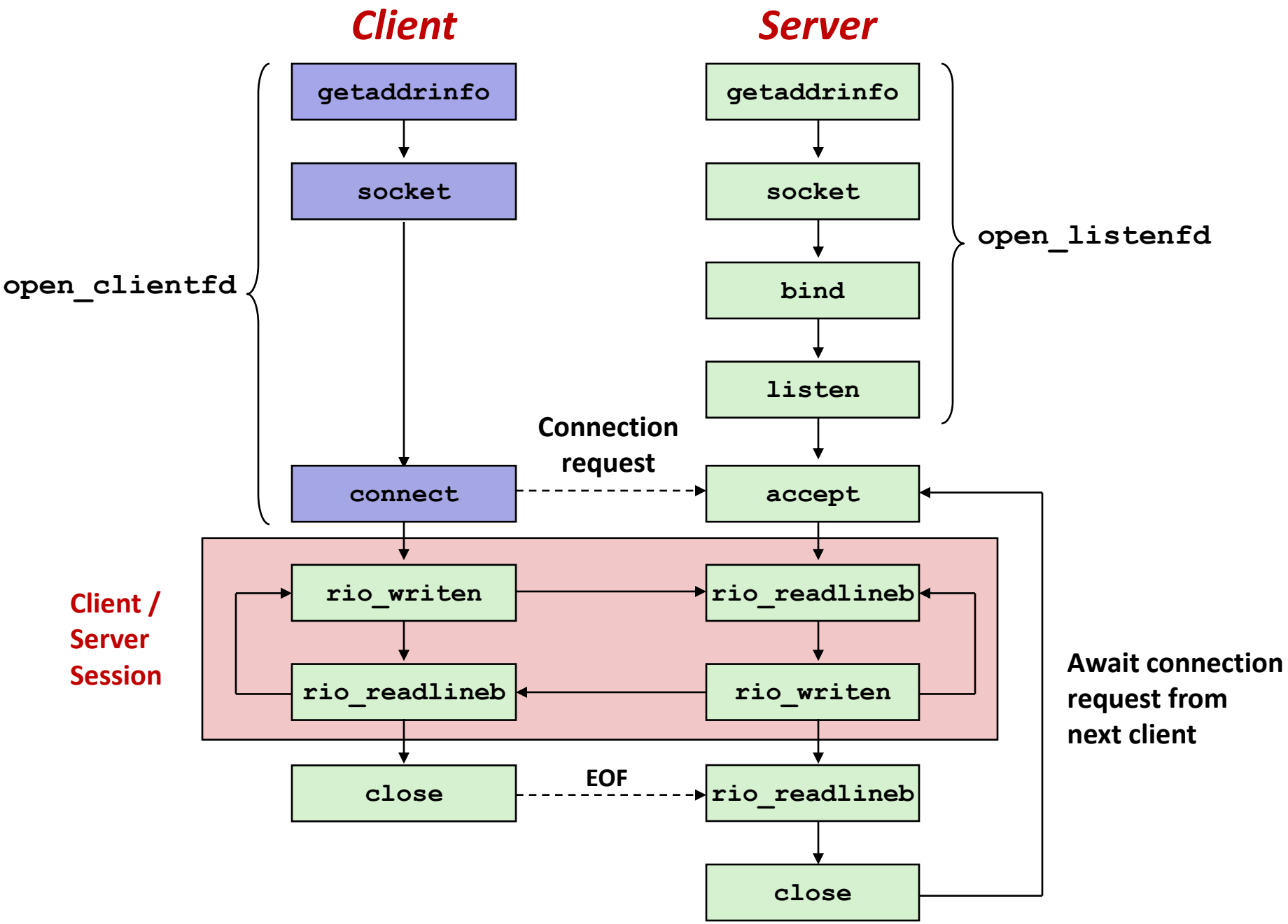
- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **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

Quiz


Client
Server
getaddrinfo
getaddrinfo
SA list
SA list
socket
socket
clientfd
listenfd
open_clientfd
open_listenfd
bind
listenfd <-> SA
listen
listening listenfd
Connection request
connect
accept
connected (to SA) clientfd
connected connfd
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
Await connection request from next client
close
EOF
rio_readlineb
close


Client
Server
getaddrinfo
socket
connect
getaddrinfo
socket
bind
listen
accept
open_clientfd
open_listenfd
Connection request
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_clientfd`

- Establish a connection with a server

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

  /* Get a list of potential server addresses */
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
  hints.ai_flags = AI_NUMERICSERV;  /* …using numeric port arg. */
  hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
  Getaddrinfo(hostname, port, &hints, &listp);
```

csapp.c

`AI_ADDRCONFIG` means "use whichever of IPv4 and IPv6 works on this computer". Good practice for clients, not for servers.

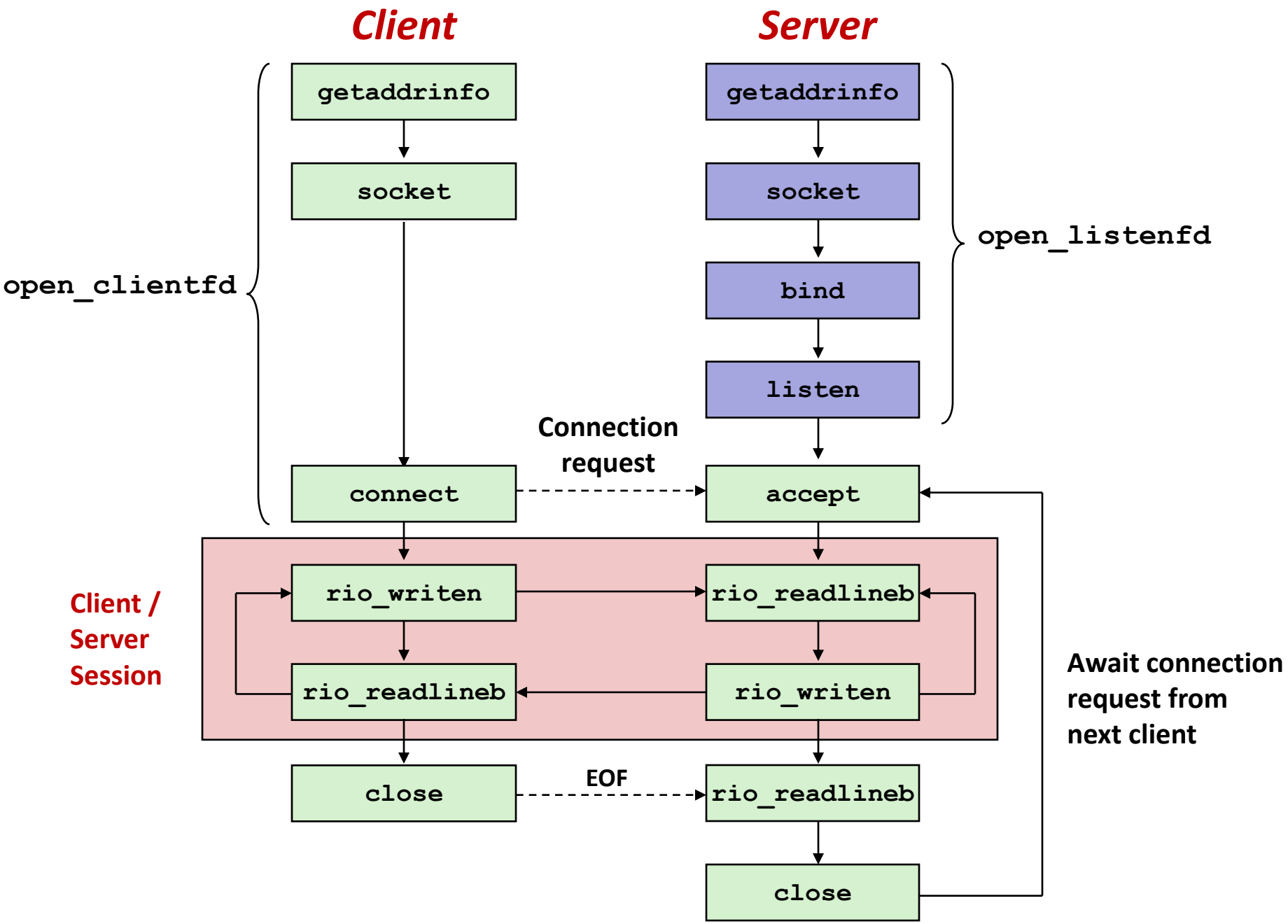
Sockets Helper: `open_clientfd` (cont)

```
    /* 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;
}
```

csapp.c


Client
Server
getaddrinfo
socket
open_clientfd
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.

```
 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

`AI_PASSIVE` means "I plan to listen on this socket."
`AI_ADDRCONFIG` normally not used for servers, but we use it for convenience

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

A production server would not break out of the loop on the first success.
We do that for simplicity only.

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.