Computer Systems: Network Programming (Sockets)

David Marchant

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

But first, Unix I/O

- A Linux *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots., B_k, \ldots., B_{m-1}$

- Cool fact: All I/O devices are represented as files:
  - `/dev/tty` (the current terminal)
  - `/dev/sda2` (a disk partition)
  - `/dev/tty2` (some other terminal)

- Even the kernel is represented as a file:
  - `/boot/vmlinuz-3.13.0-55-generic` (kernel image)
  - `/proc` (process information)
  - `/sys` (kernel data structures)

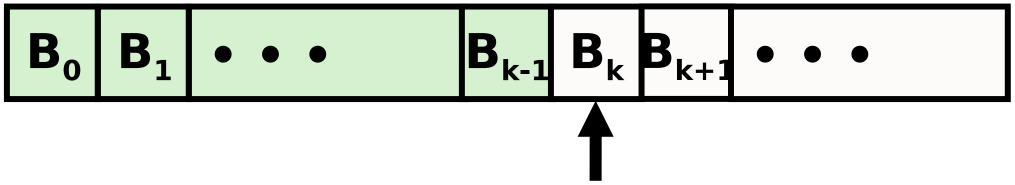
File Types

- Each file has a *type* indicating its role in the system
  - *Regular file:* Contains arbitrary data
  - *Directory:* Index for a related group of files
  - *Socket:* For communicating with a process on another machine

- Other file types beyond our core scope
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

Unix I/O Overview

- Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:*
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**
    - Not all files support seeking (e.g. pipes, sockets)



Current file position = k

Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer ***file descriptor***
  - **`fd == -1`** indicates that an error occurred
- Each process created by a Linux shell begins life with three open files associated with a terminal:
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs, because the file descriptor number may have been re-used
- Always check return codes, even for seemingly benign functions such as `close()`

Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read at least 1 byte and
   up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
  - Return type **`ssize_t`** is signed integer
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;        /* file descriptor */
int nbytes;    /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

Simple Unix I/O example

- Copying stdin to stdout, one byte at a time

```
int main(void)
{
    char c;

    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

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*

4. This is **very similar** to the pipes we briefly introduced in the concurrency lectures.

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 *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

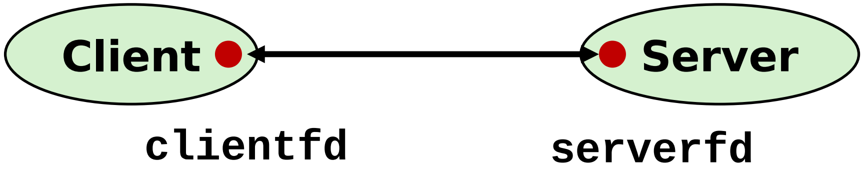
- Clients and servers communicate by sending streams of bytes over *connections*. Each connection is:
  - *Point-to-point*: connects a pair of processes.
  - *Full-duplex*: data can flow in both directions at the same time,
  - *Reliable*: stream of bytes sent by the source is eventually received by 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



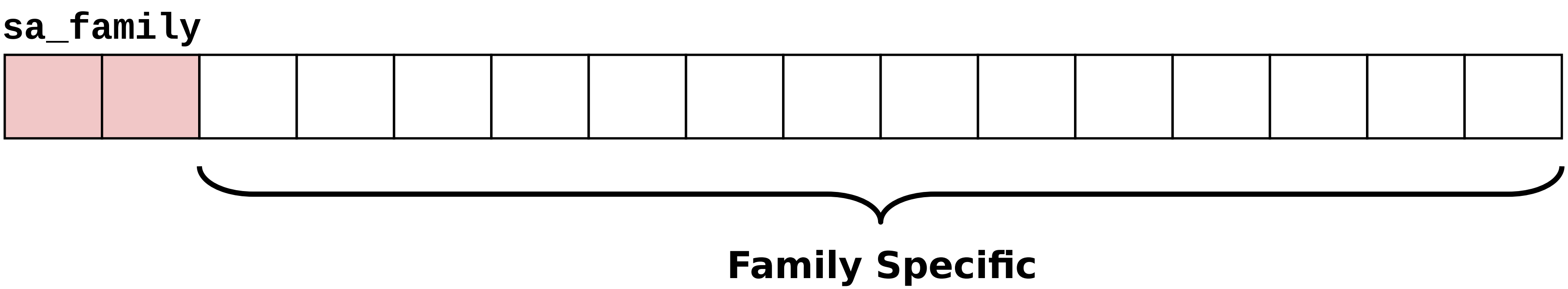
- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors

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.  */
};
```



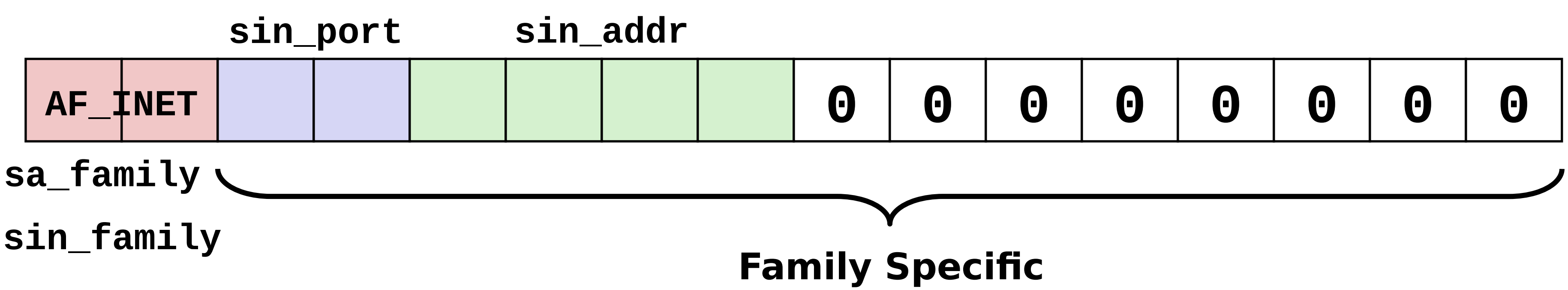
Socket Addresses Domains

- There are many different address domains, each with a corresponding struct.
- See link for complete list: https://man7.org/linux/man-pages/man2/socket.2.html
- In this course we will stick with IPv4 (explained in the following slides)

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) */
};
```



Setting up an address example

```
struct sockaddr_in serv_addr;

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(12345);

if (inet_pton(AF_INET, “123.123.123.123”, &serv_addr.sin_addr) <= 0) {
    printf("Invalid address\n");
    return -1;
}
```

- Note that we’re already validating results. This is important in any application, but in networking its even more necessary than that
- Also note byte order on the port, and notation for the host

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`

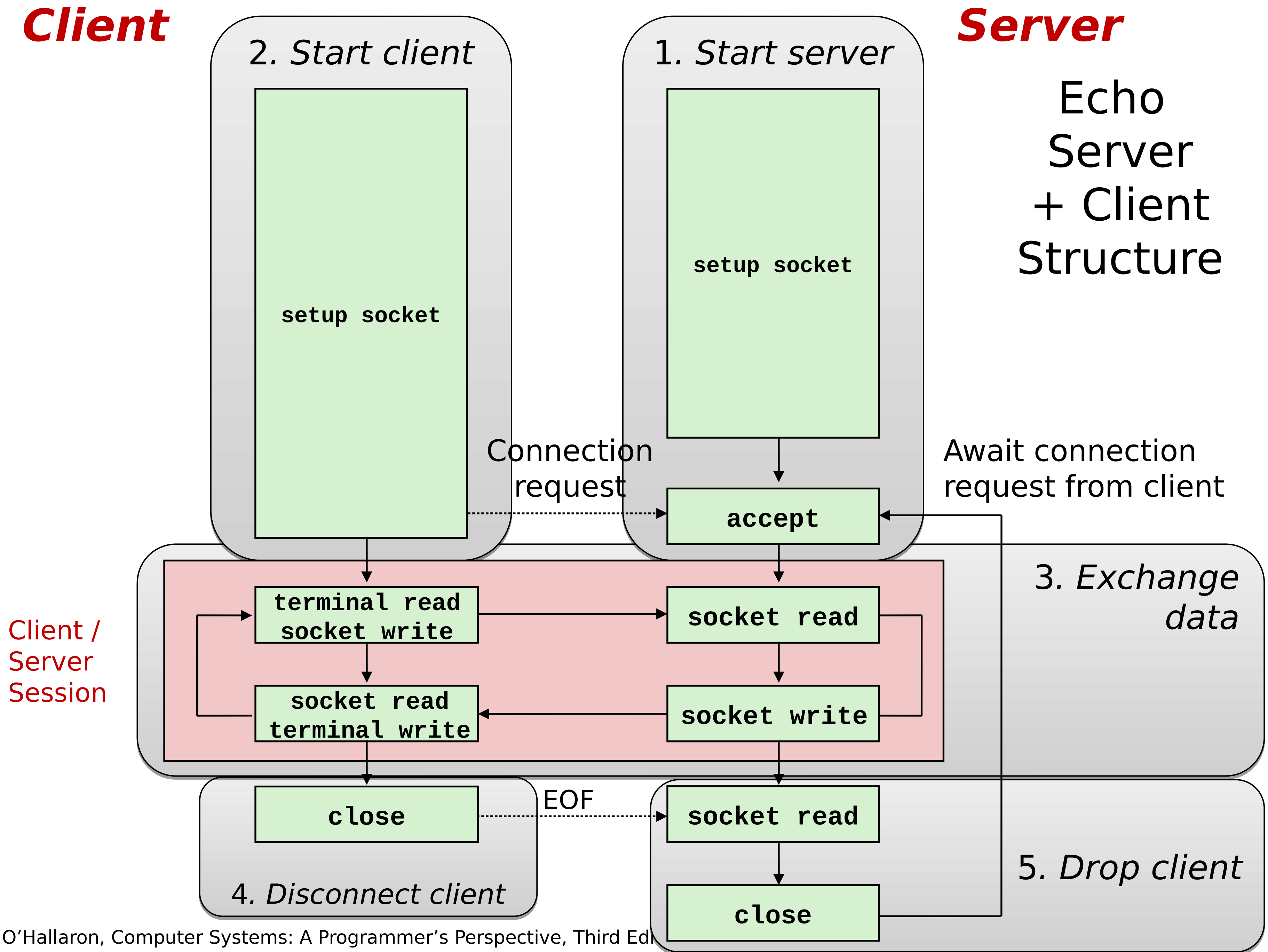
```
int getaddrinfo(const char *host,            /* Hostname or address */
                const char *service,         /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);   /* Output linked list */

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

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 Structure



Echo Client: Main Routine

```
#include "compsys_helpers.h"

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

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

    clientfd = compsys_helper_open_clientfd(host, port);
    compsys_helper_readinitb(&state, clientfd);

    while (fgets(buf, MAXLINE, stdin) != NULL) {
      compsys_helper_writen(clientfd, buf, strlen(buf));
      compsys_helper_readlineb(&state, buf, MAXLINE);
      fputs(buf, stdout);
    }
    close(clientfd);
    exit(0);
}
```

echoclient.c

On Short Counts

- Short counts often occurs in these situations:
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets
- Short counts rarely occurs in these situations:
  - Reading from disk files (except for EOF)
    - …but may happen for huge reads, depending on file system.
  - Writing to disk files
    - …similarly.
- Best practice is to always allow for short counts.

The compsys_helpers package

- A set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts
- Provides two different kinds of functions
  - Unbuffered input and output of binary data
    - **compsys_helper_readn** and **compsys_helper_writen**
  - Buffered input of text lines and binary data
    - **compsys_helper_readlineb** and **compsys_helper_readnb**
    - Buffered functions are thread-safe and can be interleaved arbitrarily on the same descriptor
- Part of **compsys_helpers.c/compsys_helpers.h**
- For those that are resitting the course these are the same as the RIO functions. Feel free to continue using those but they are no longer supported

Unbuffered Compsys_helper Input and Output

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

```
#include "compsys_helpers.h"

ssize_t compsys_helpers_readn(int fd, void *usrbuf, size_t n);
ssize_t compsys_helpers_writen(int fd, void *usrbuf, size_t n);

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

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

compsys_helper_readn

```
/*
 * compsys_helper_readn - Robustly read n bytes (unbuffered)
 */
ssize_t compsys_helper_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* Interrupted by sig handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);     /* Return >= 0 */
}
```

Buffered Input Helpers

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

```
#include "compsys_helpers.h"

void compsys_helpers_readinitb(compsys_helper_state_t *rp, int fd);

ssize_t compsys_helpers_readlineb(
    compsys_helper_state_t *rp, void *usrbuf, size_t maxlen);
ssize_t compsys_helpers_readnb(
    compsys_helper_state_t *rp, void *usrbuf, size_t n);

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

- **`compsys_helpers_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

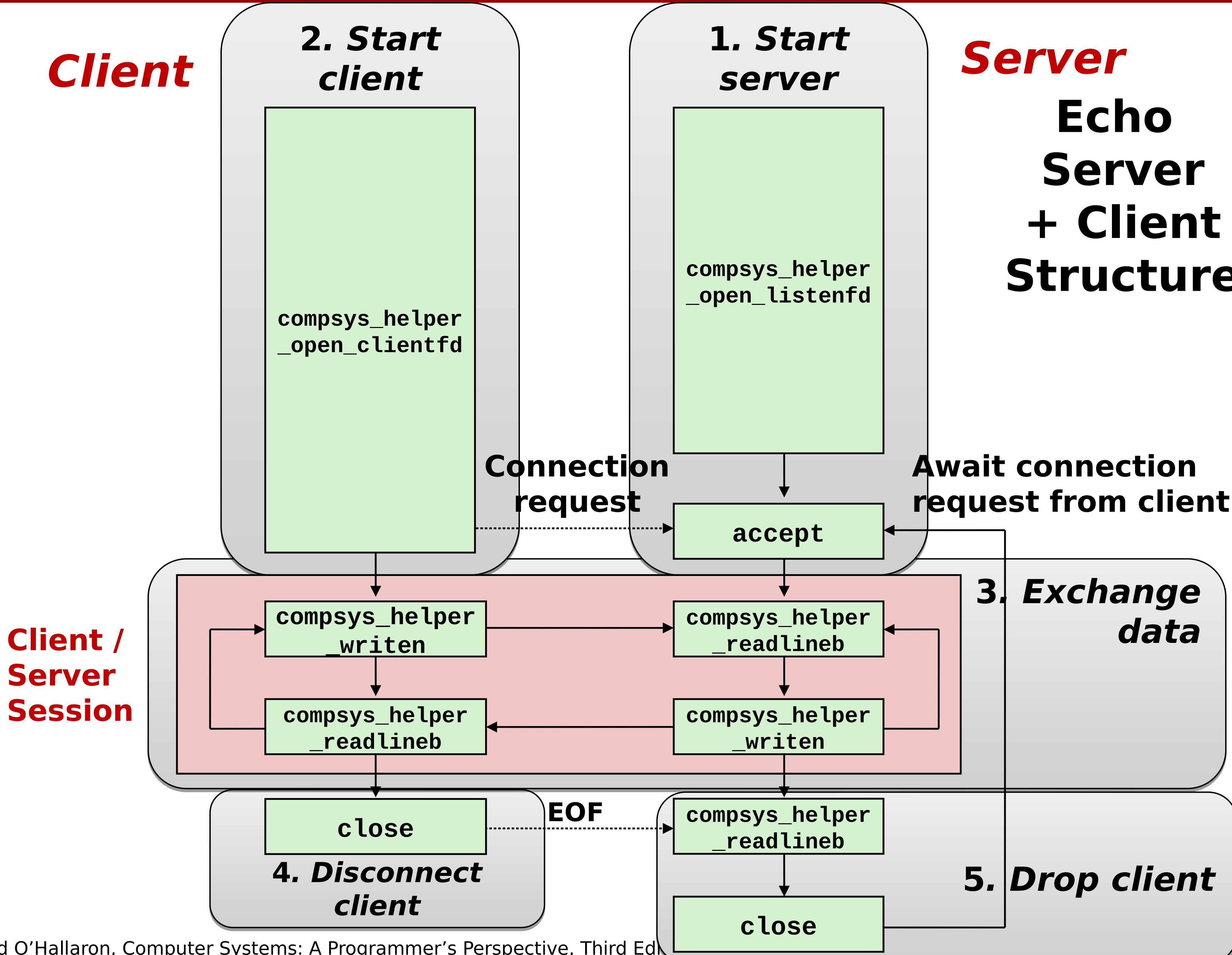
Buffered Input Helpers

```
#include "compsys_helpers.h"

void compsys_helper_readinitb(compsys_helper_state_t *rp, int fd);

ssize_t compsys_helper_readlineb(
    compsys_helper_state_t *rp, void *usrbuf, size_t maxlen);
ssize_t compsys_helper_readnb(
    compsys_helper_state_t *rp, void *usrbuf, size_t n);
```

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

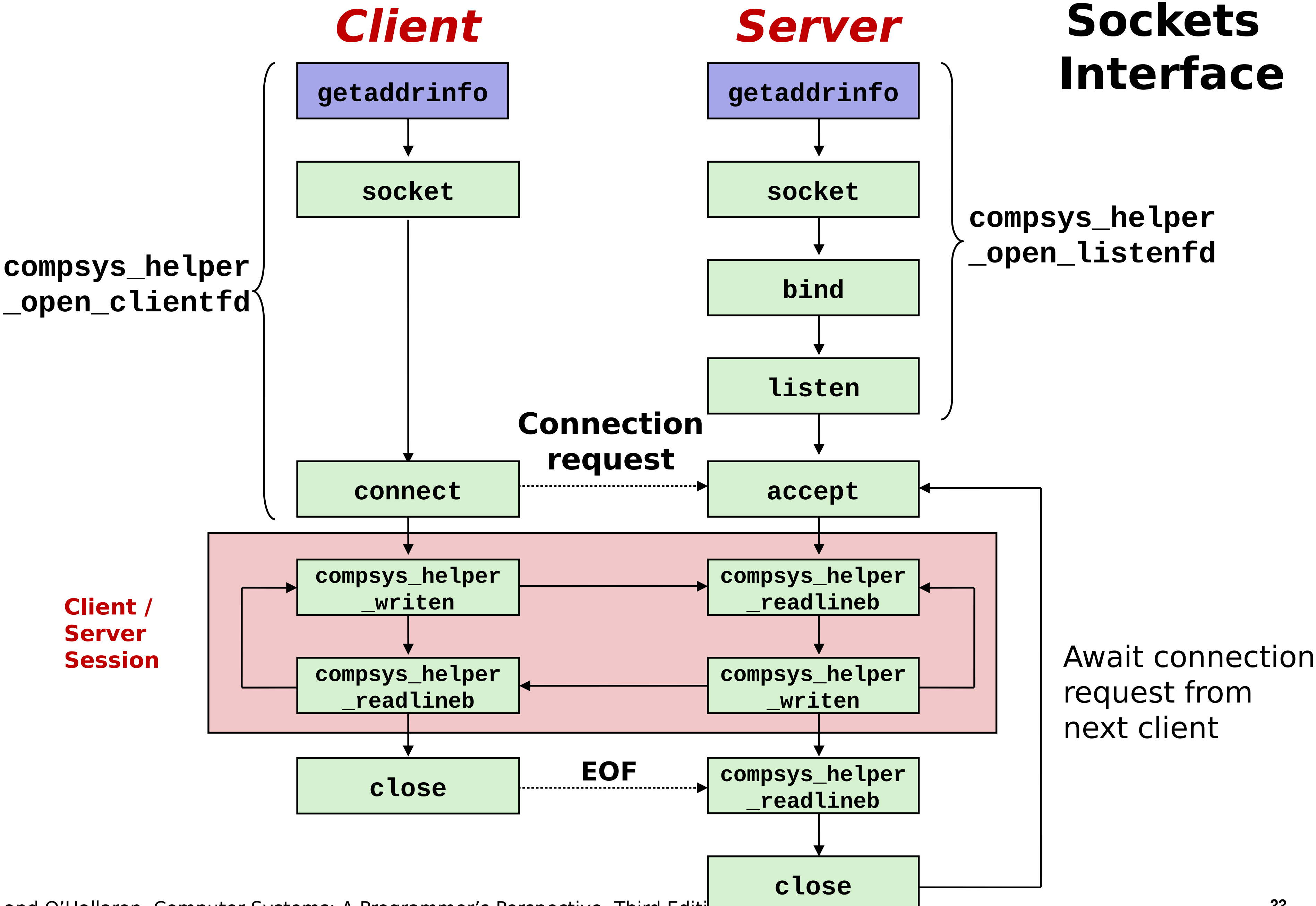
- **compsys_helper_readnb** reads up to **n** bytes from file **fd**
- Stopping conditions
  - **maxlen** bytes read
  - EOF encountered
- Calls to **compsys_helper_readlineb** and **compsys_helper_readnb** can be interleaved arbitrarily on the same descriptor
  - Warning: Don't interleave with calls to **compsys_helper_readn**

Echo Server + Client Structure



See lecture code

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:

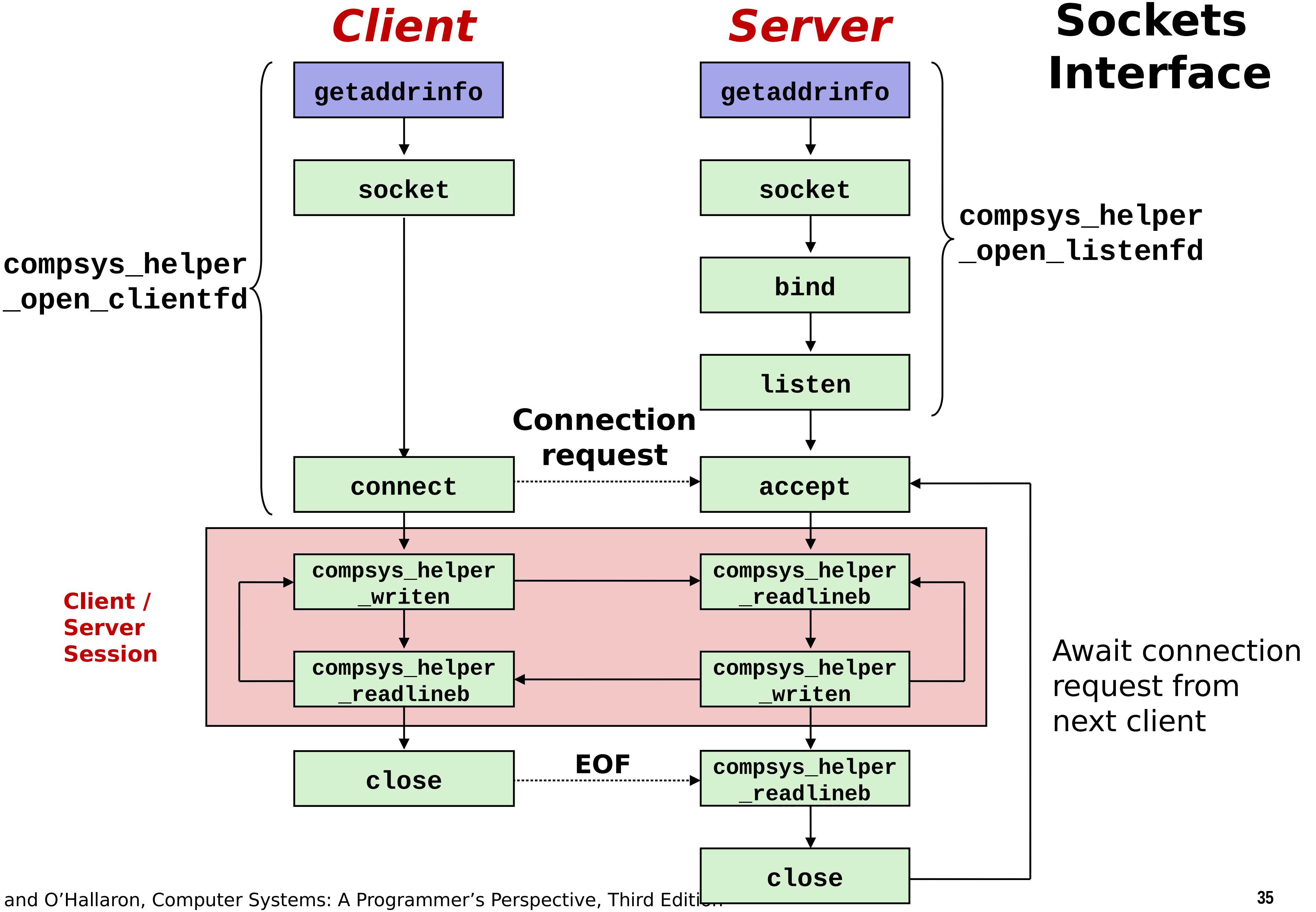
```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

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.

Sockets Interface



Sockets Interface: `bind`

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

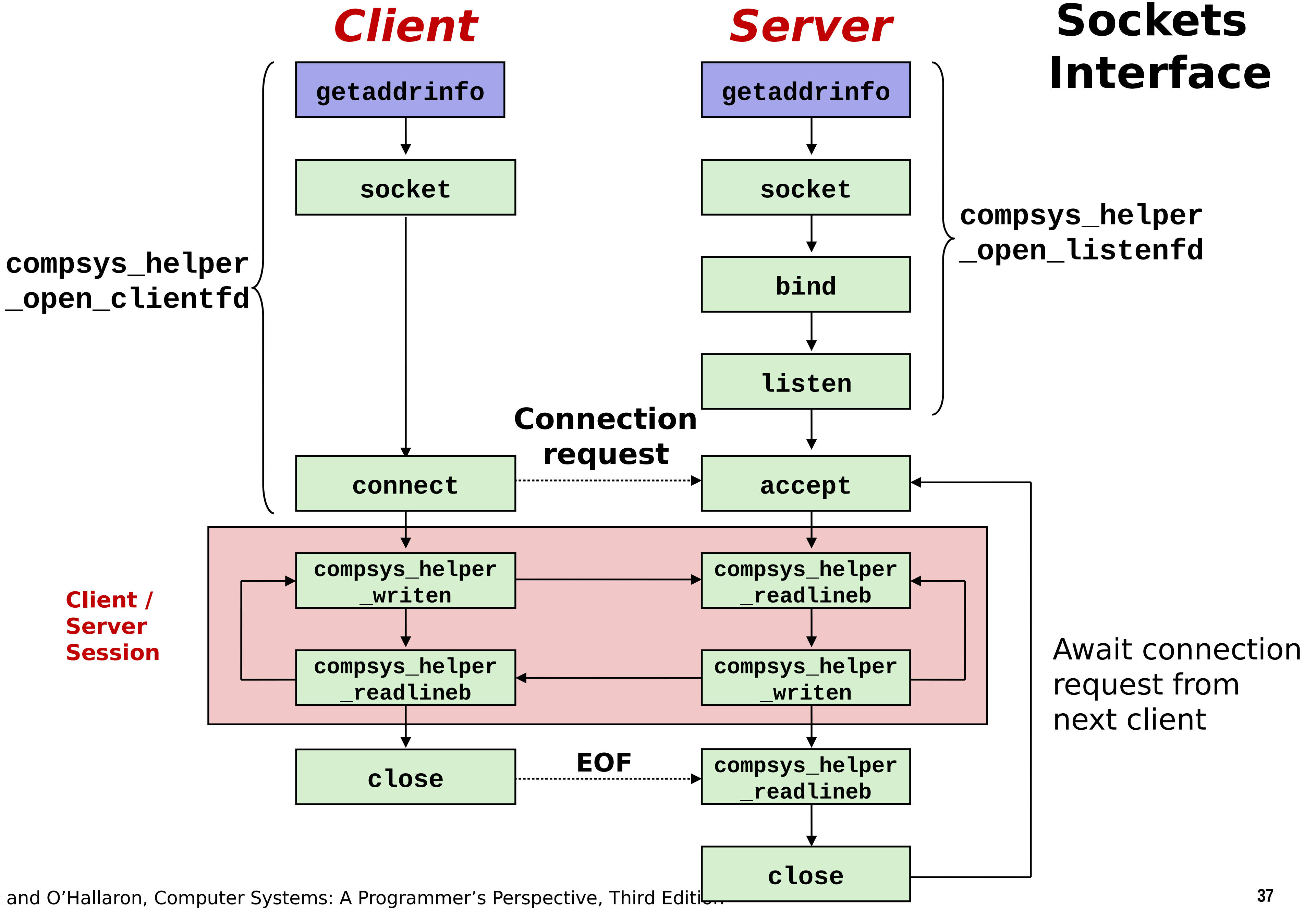
```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

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



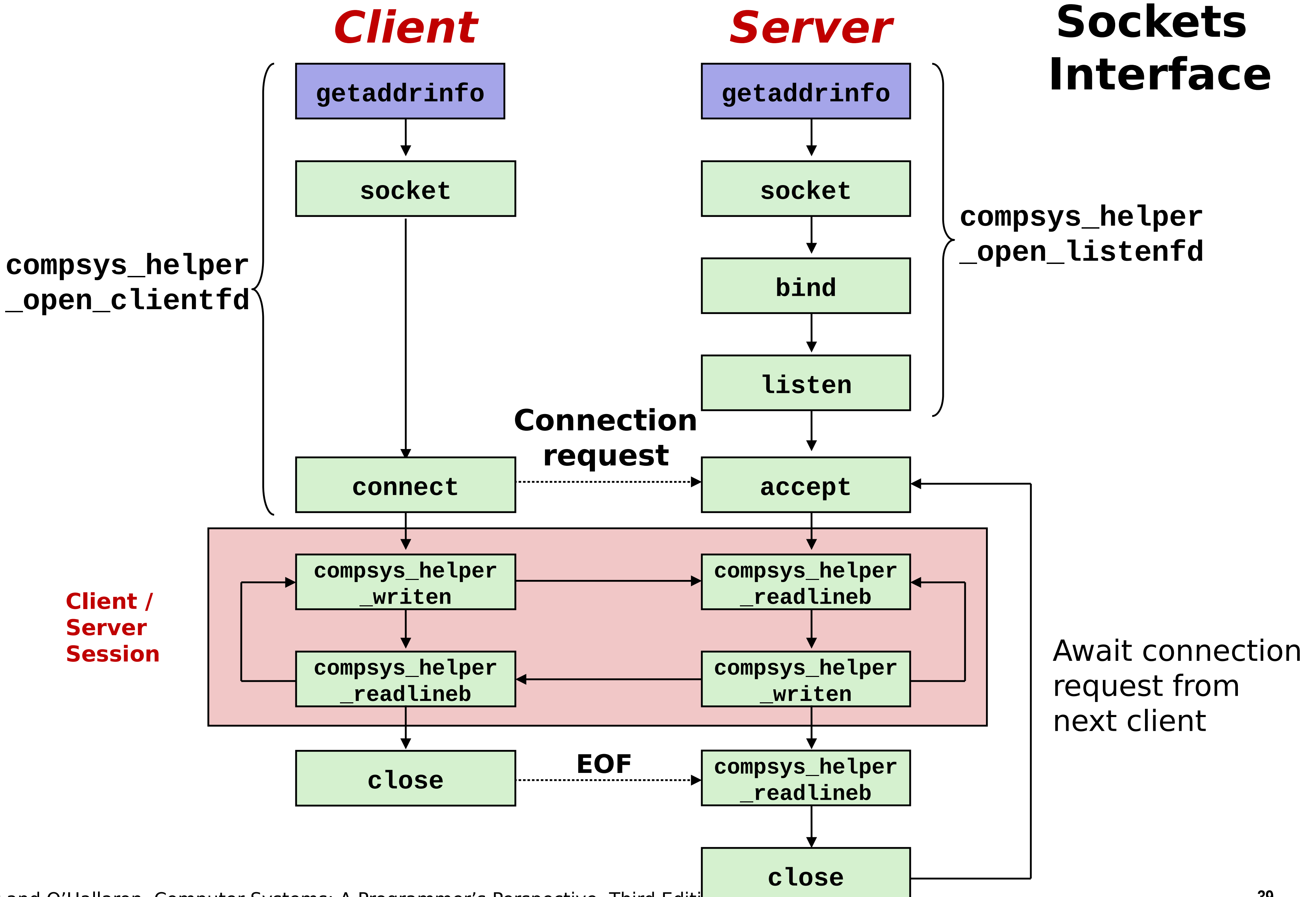
Sockets Interface: `listen`

- 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`

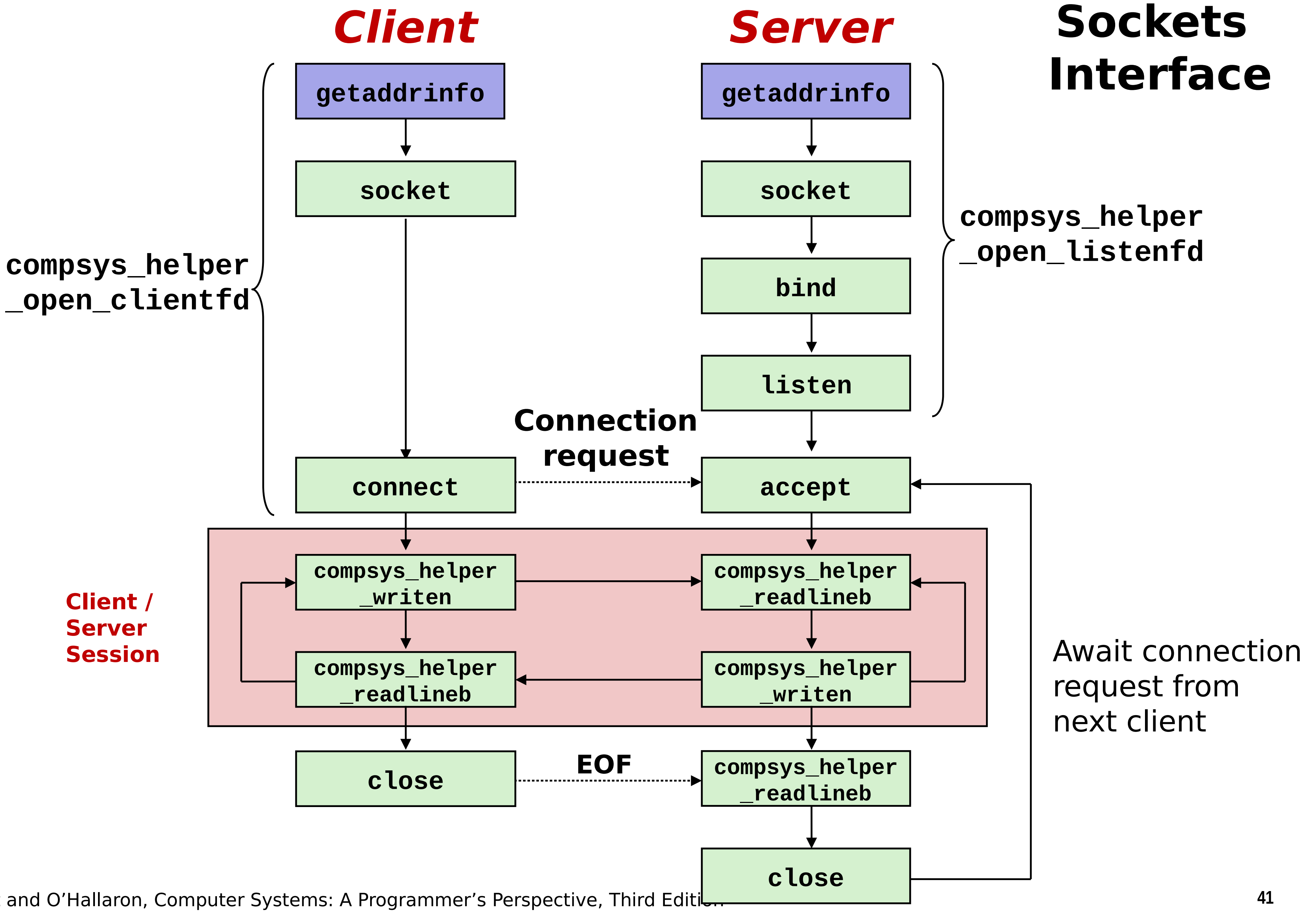
- Servers wait for connection requests from clients by calling **`accept`:**

```
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

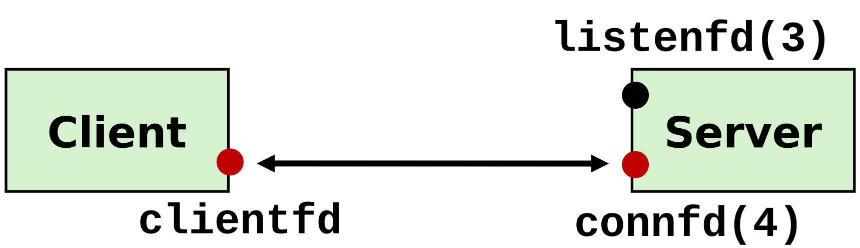
    `(x:y, addr.sin_addr:addr.sin_port)`
    - x is client address
    - y is ephemeral port that uniquely identifies client process on client host

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

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***

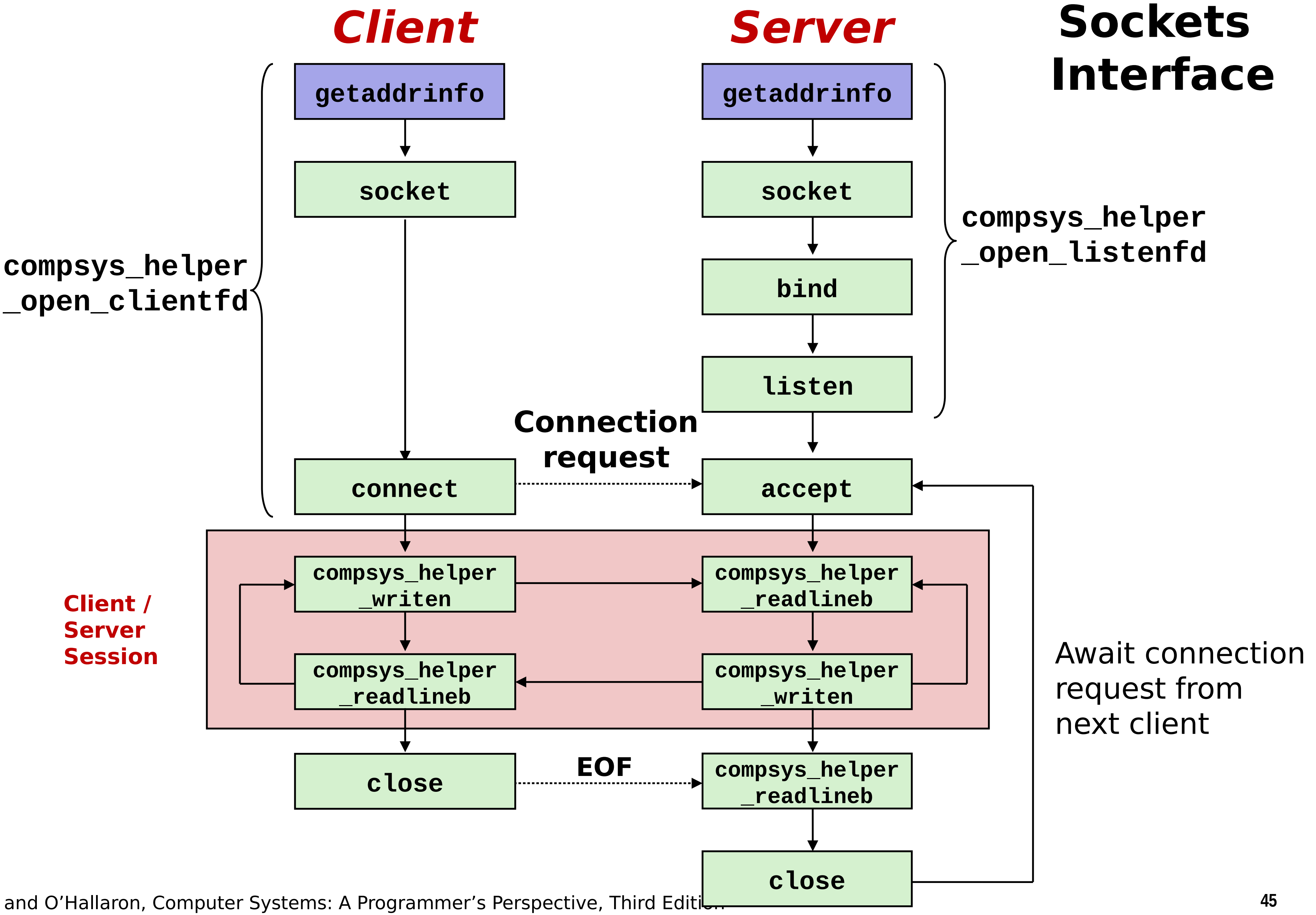


3. *Server returns* ***connfd*** *from* ***accept.*** *Client returns from* ***connect.*** *Connection is now established between* ***clientfd*** *and* ***connfd***

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



Python for Networks

- **Higher level than C, so should be easier to follow and understand**
- **More abstractions, so quicker to get a working networked application, but runs slower**
- **Typically, you are more likely to use it yourselves so its worth introduction. Assignments will still be in C (sorry not sorry)**

```
def function(num):
    for i in [1, 2, 3, 4]:
        print(num + i)

    return num * 2

print(function(10))
```

Sockets: socket

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

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

- Example:

C:

```
#include <sys/socket.h>

int socket_fd = 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

Python:

```
from socket import *

with socket(AF_INET, SOCK_STREAM) as sock:
    ...
```

Sockets: 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.

C:
```
listen(socket_fd, 10);
```

Python:
```
sock.listen(10)
```

Sockets: 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.

C:

```c
socklen_t clientlen;
struct sockaddr_storage clientaddr;
conn_fd = accept(socket_fd, (SA *) &clientaddr, &clientlen);
```

Python:

```python
Conn, conn_addr = sock.accept()
```

Sockets: 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

C:

```
struct sockaddr s_addr;
connect(socket_fd, (struct sockaddr *)&s_addr, sizeof(s_addr));
```

Python:

```
client_sock.connect(“130.226.237.173”, 56)
```

Final building blocks

- Reading from Python socket:

```
socket.recv(buffsize)
```

- Writing to Python socket:

```
socket.send(bytes)
```

```
socket.sendall(bytes)
```

- Both send bytes, but send may only send some and it is your responsibility to check. Sendall manages sending until everythings sent or an error was encountered

Python Example

Client:

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
    client_socket.connect((“127.0.0.1”, 5678))
    request = bytearray(“This is a message”.encode())
    client_socket.sendall(request)
    response = client_socket.recv(1024)
    print(response)
```

Server:

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind((“127.0.0.1”, 5678))
    server_socket.listen()
    while True:
        connection, connection_address = server_socket.accept()
        with connection:
            message = connection.recv(1024)
            connection.sendall(response)
```

Bytes in Python

- In networking we need to be deliberate in what bytes we send, but Python does not like opperating at this level

- Bytearrays must be manually packed and extended:

```
import struct

payload = bytearray()
payload.extend(“Some long string.”)
payload.extend(4798.5)
payload.extend(struct.pack('!I', 4294967295))
payload.extend(struct.pack('!I', 0))
```

- Key difference:
  - Extend will simply add its input to the end of the array, usefull for message bodies
  - struct.pack takes a formatting variable defining exactly how much space a variable should take up, and the endianess of the bytes
  - formatting: https://docs.python.org/3/library/struct.html

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 compsys_helpers 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>`***