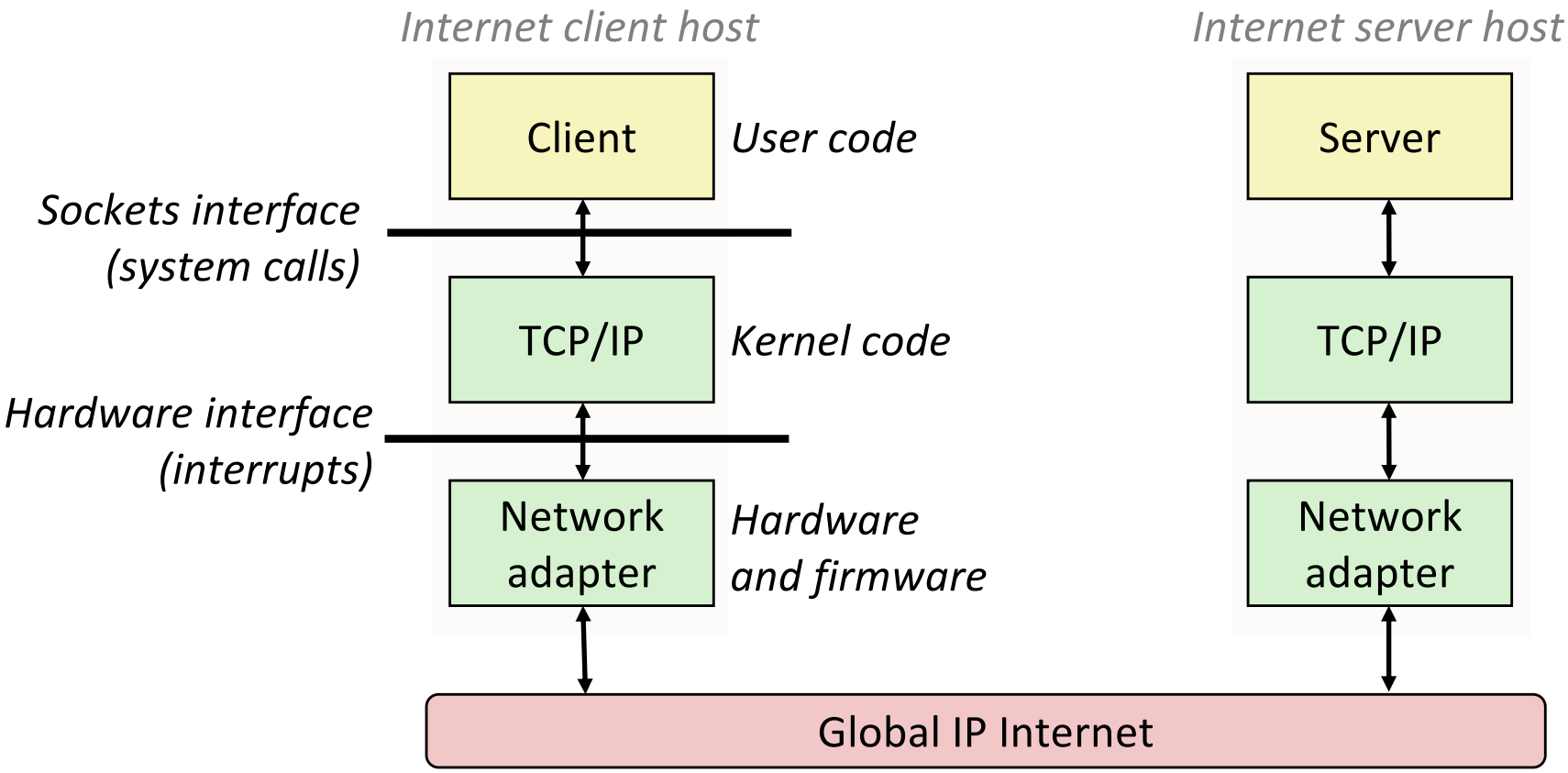
# Sockets

# Bringing It Together

- Networks are collections of boxes (computers and specialized computers) and cables that move data
  - Networks themselves are disparate and have their own protocols
    - These protocols move data within a network
  - internets smooth out those differences and allow us to communicate between the different networks
    - internet protocols move data from source machine to destination machine
  - The Internet is an example of an internet
- When data arrives at a machine, the OS is responsible for delivering it to the correct process
  - UDP and TCP/IP help with this
  - TCP provides an illusion of reliability---the OS is responsible for implementing this reliability
  - With reliability comes danger: if we aren't careful, we could overload an already congested network and cause it to collapse
  - TCP has algorithmic pieces to help with this: additive increase, multiplicative decrease; slow start; reaction to timeout events; round trip time variance (initial timer setting); and retransmit backoff

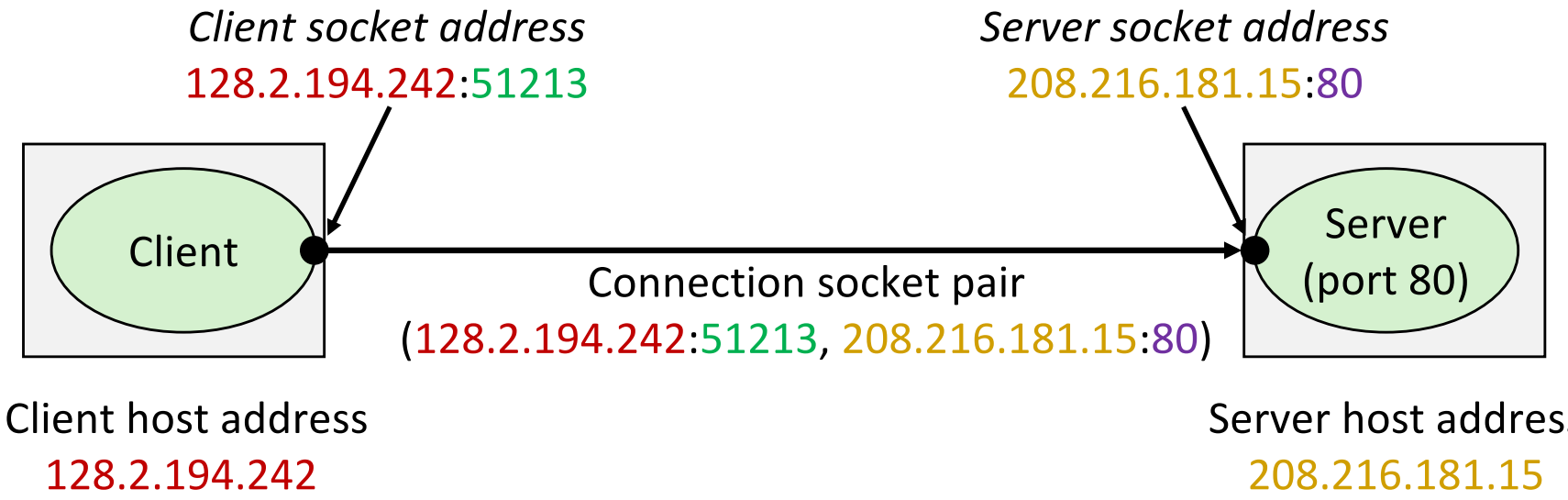
# Hardware and Software Organization of an Internet Application



# Internet Connections

- Clients and servers communicate by sending streams of bytes over connections
- Processes are identified by ports
  - A *port* is a 16-bit integer that identifies a process
  - *Well-known port*: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)
  - *Ephemeral port*: Assigned automatically by client kernel when client makes a connection request.
- A socket is an endpoint of a connection
  - Underlying basis for all Internet applications
  - Created in the early 80s as a part of the original Berkeley distribution of UNIX that contained an early version of the Internet protocols
  - Socket address is an IPaddress:port pair

# Anatomy of an Internet Connection




51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with web servers

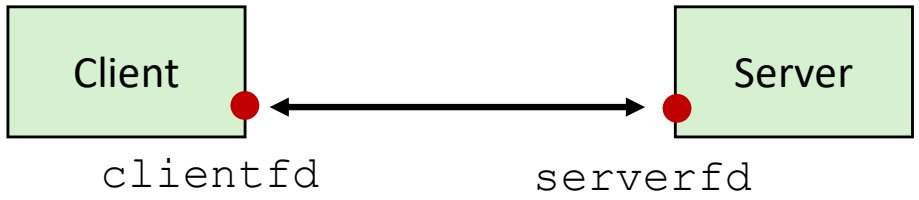
# Well-known Ports and Service Names

- Popular services have permanently assigned *well-known ports and* corresponding *well-known service names*:
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - web servers: 80/http
  - https server: 443/https
- Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.

# Sockets: Implementation Details

- 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
    - All Unix I/O devices, including networks, are modeled as files

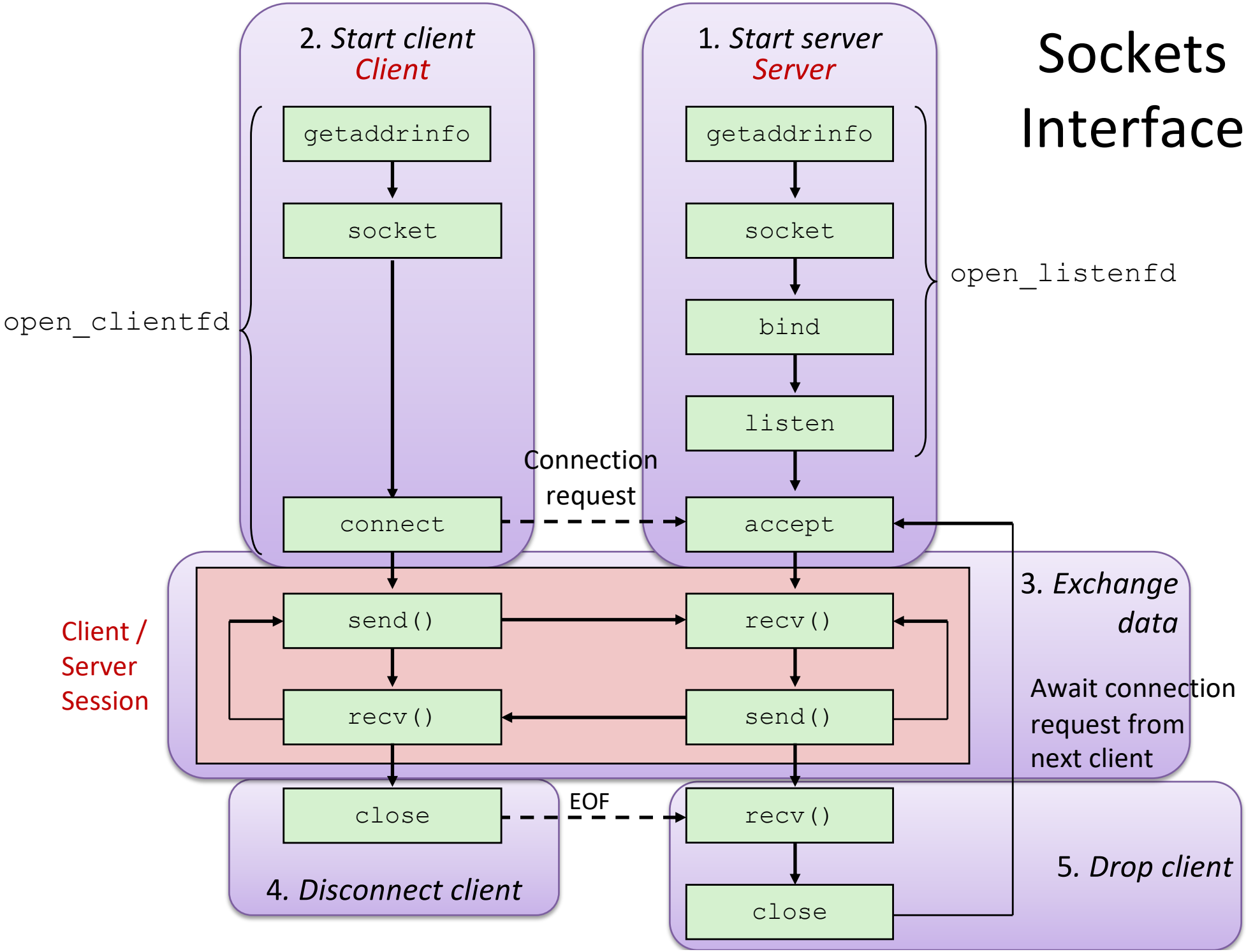
- Clients and servers communicate with each other by reading from and writing to socket descriptors



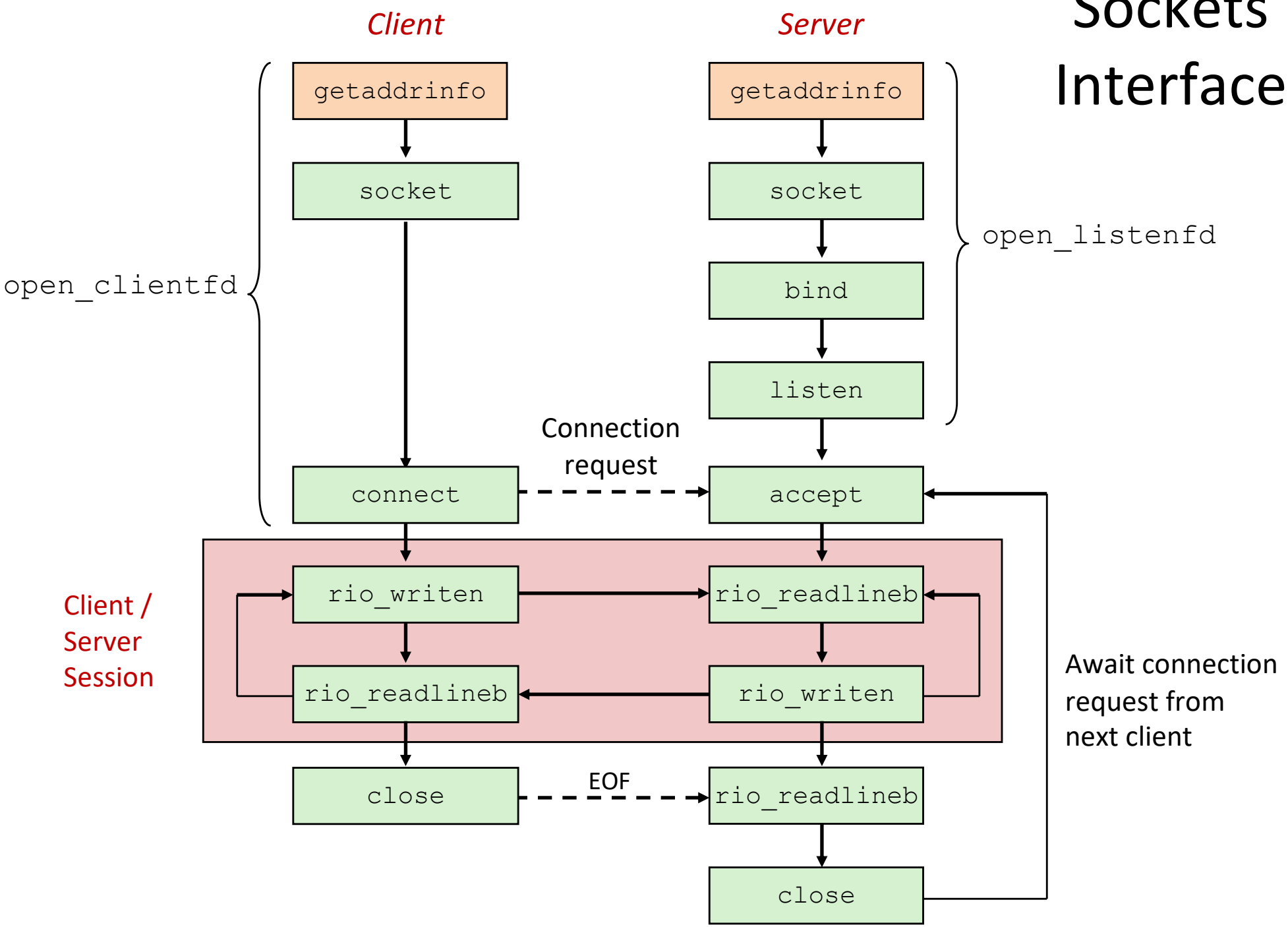

- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors

# Sockets Interface: Client

# Sockets Interface



# Sockets Interface



# Interfacing with DNS

Functions for retrieving host entries from DNS:

**`getaddrinfo()`:** query key is a DNS domain name, returns:

```
struct addrinfo {
          int              ai_flags;
          int              ai_family;     /*desired address family*/
          int              ai_socktype;   /*desired Layer 4 protocol*/
          int              ai_protocol;
          socklen_t        ai_addrlen;
          struct sockaddr *ai_addr;
          char            *ai_canonname;
          struct addrinfo *ai_next;
      };
```

**`getnameinfo()`:** query key is an IP address, returns a sockaddr struct (more soon)

# Using getaddrinfo(3)

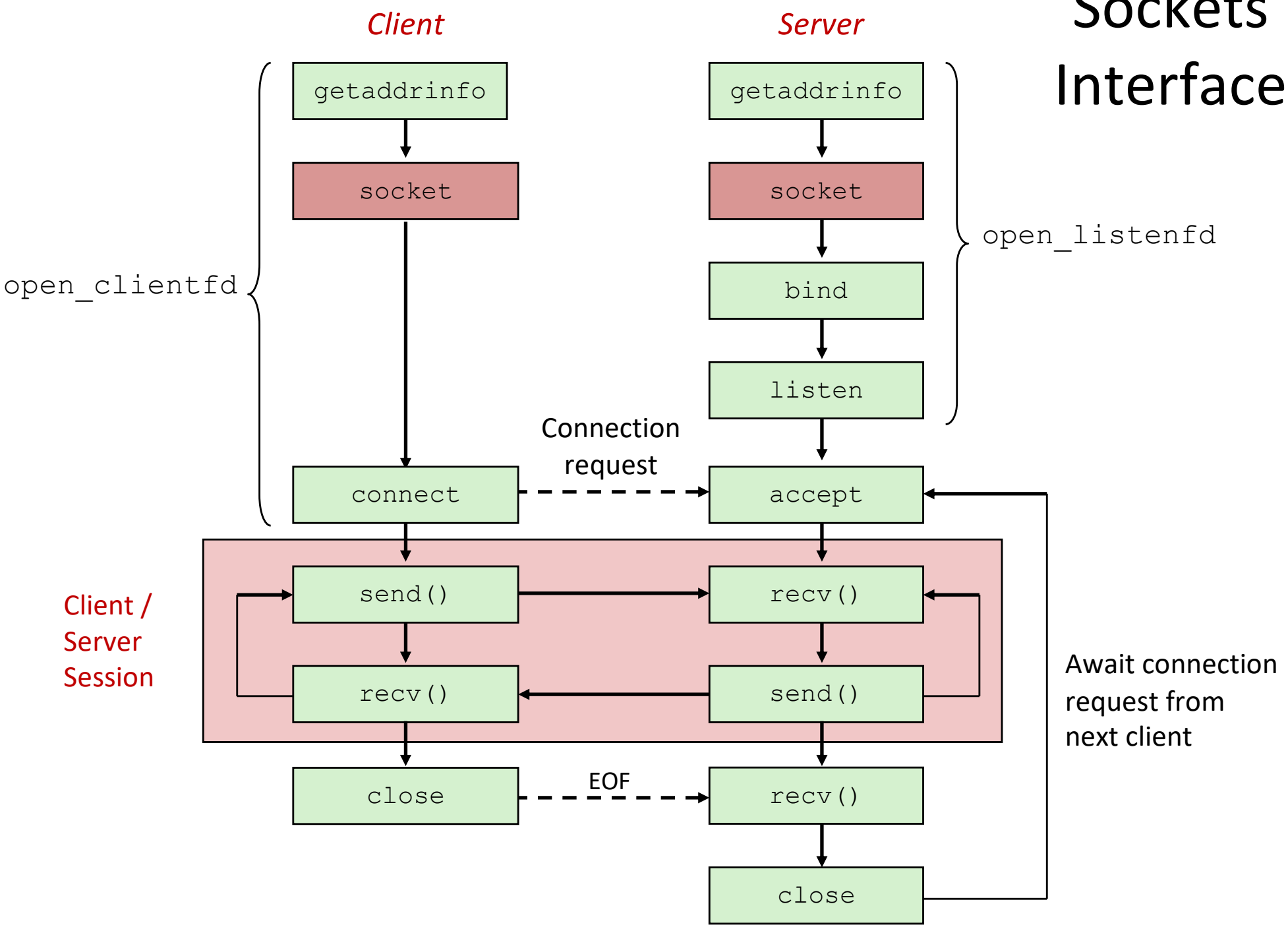
```
int clientfd;                           /* socket descriptor */

/*setup information*/
struct addrinfo hints;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;            /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM;         /* Datagram socket */

/* fill in the server's IP address and port */
struct addrinfo *result = NULL;         /* DNS host entry */
int s = 0;
if ((s = getaddrinfo(hostname, port, &hints, &result)) != 0)
    return -1;
```

Note: This is untested code.

# Sockets Interface



# Client: socket(2)

`socket(2)` creates a socket descriptor on the client

- Allocates and initializes some internal data structures
- `AF_UNSPEC`: indicates that the socket is associated with Internet protocols but either IPv4 or IPv6 is fine
- `SOCK_STREAM`: selects a reliable byte stream connection (TCP)
  - `SOCK_DGRAM` would result in UDP

```
int clientfd;  /* socket descriptor */

if ((clientfd = socket(AF_UNSPEC, SOCK_STREAM, 0) <0)
    return -1; /* check errno for cause of error */

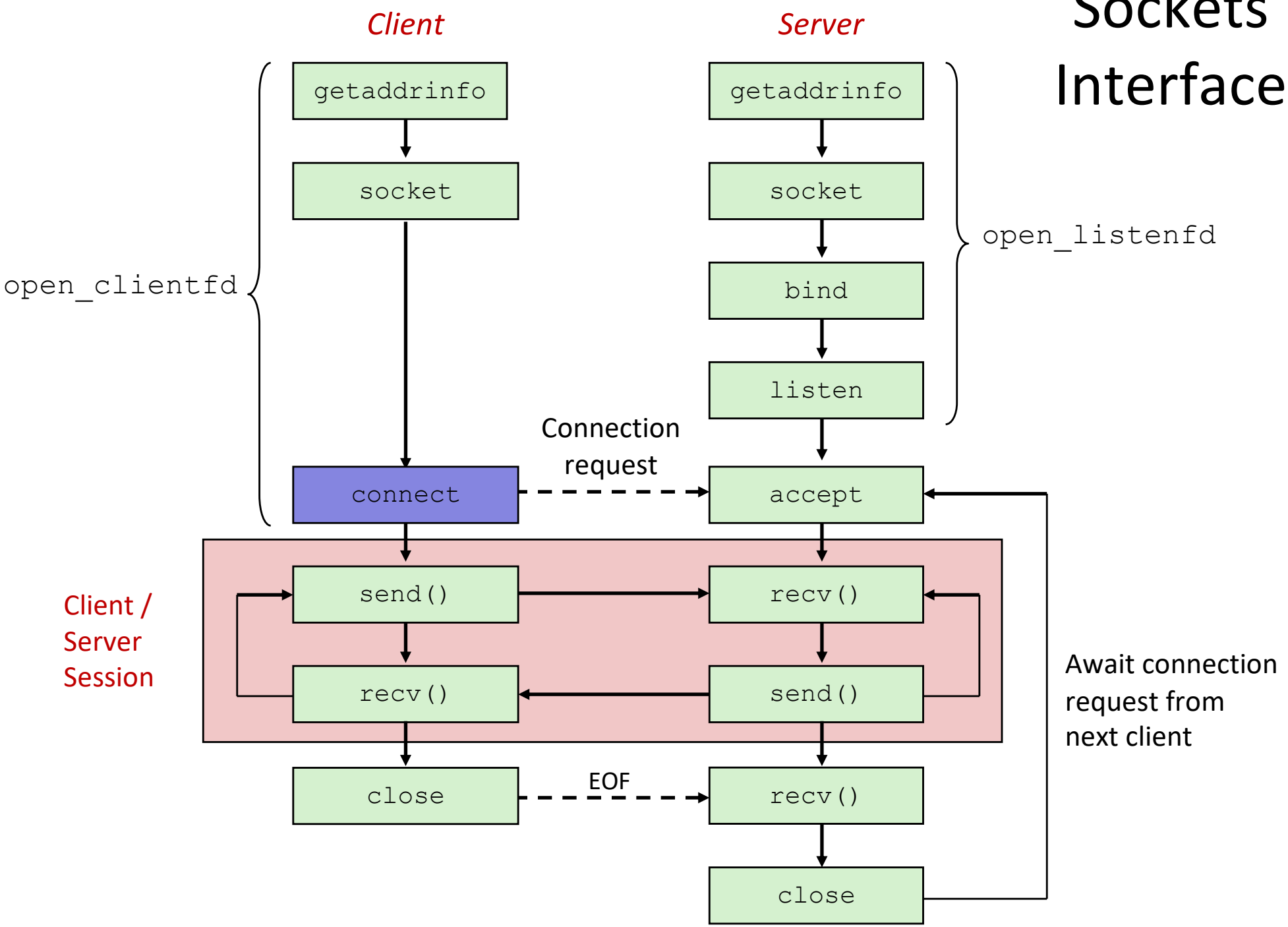
...
```

# Socket Implementation in the OS

- Each socket `fd` has associated socket structure with:
  - Send and receive buffers
  - For listening sockets, queues of incoming connections
  - A *protocol control block* (PCB)

- PCB contains protocol-specific information, such as:
  - Pointer to IP Transmission Control Block with source/destination IP address and port
  - Information about received packets and position in stream
  - Information about unacknowledged sent packets
  - Information about timeouts
  - Information about connection state (setup/teardown)

# Sockets Interface



# Client: `connect(2)`

Finally the client creates a connection with the server

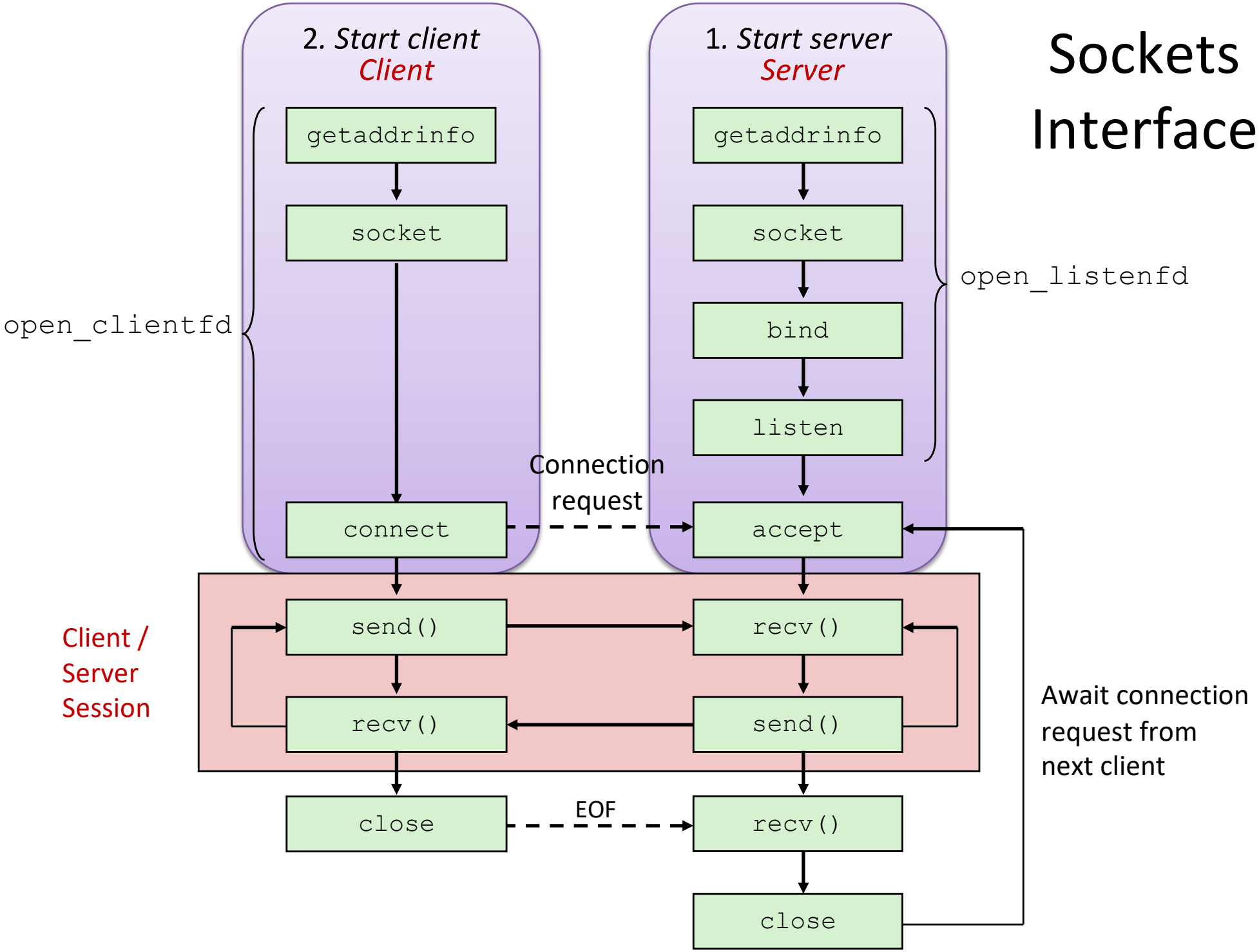
- Client process blocks until the connection is created
- After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls (typically send/recv) on descriptor `clientfd`

```
int clientfd;                    /* socket descriptor */
struct addrinfo *result;         /* DNS host entry initialized in
                                    * call to getaddrbyinfo()*/
typedef struct sockaddr SA;      /* generic sockaddr */

...

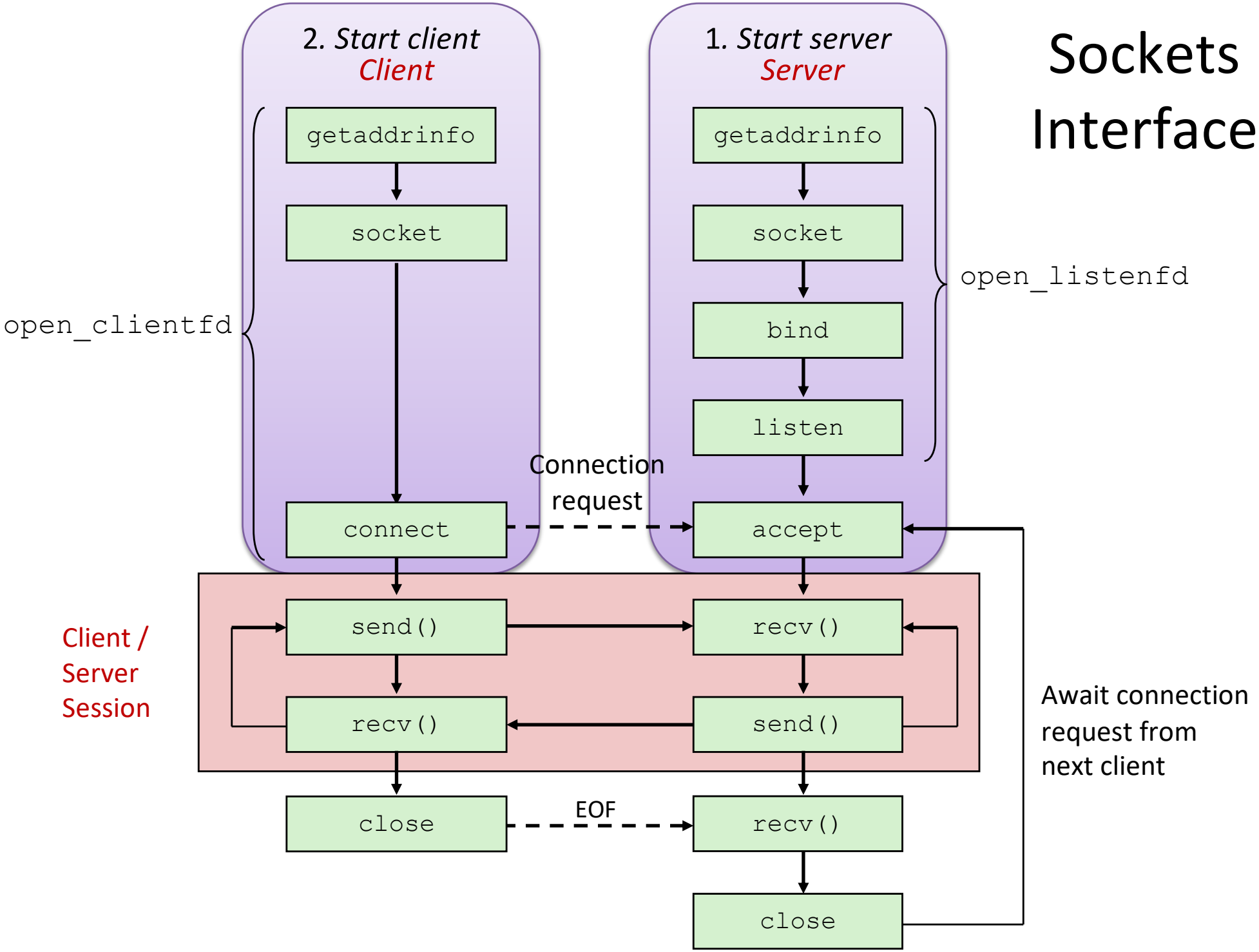
/* Establish a connection with the server */
if (connect(clientfd, result->ai_addr, result->ai_addrlen) < 0)
  return -1;
```

# Sockets Interface

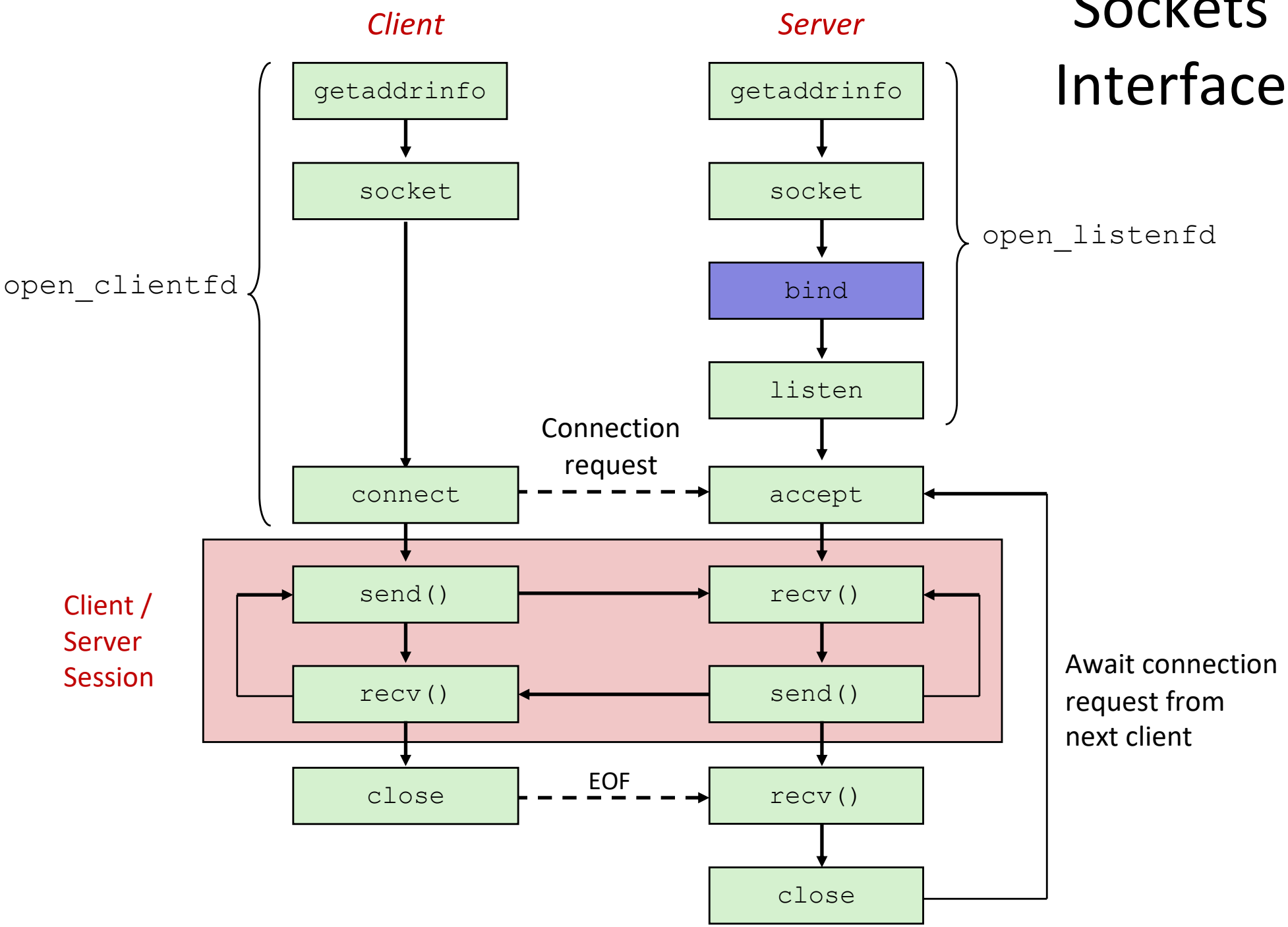


# Sockets Interface: Server

# Sockets Interface



# Sockets Interface

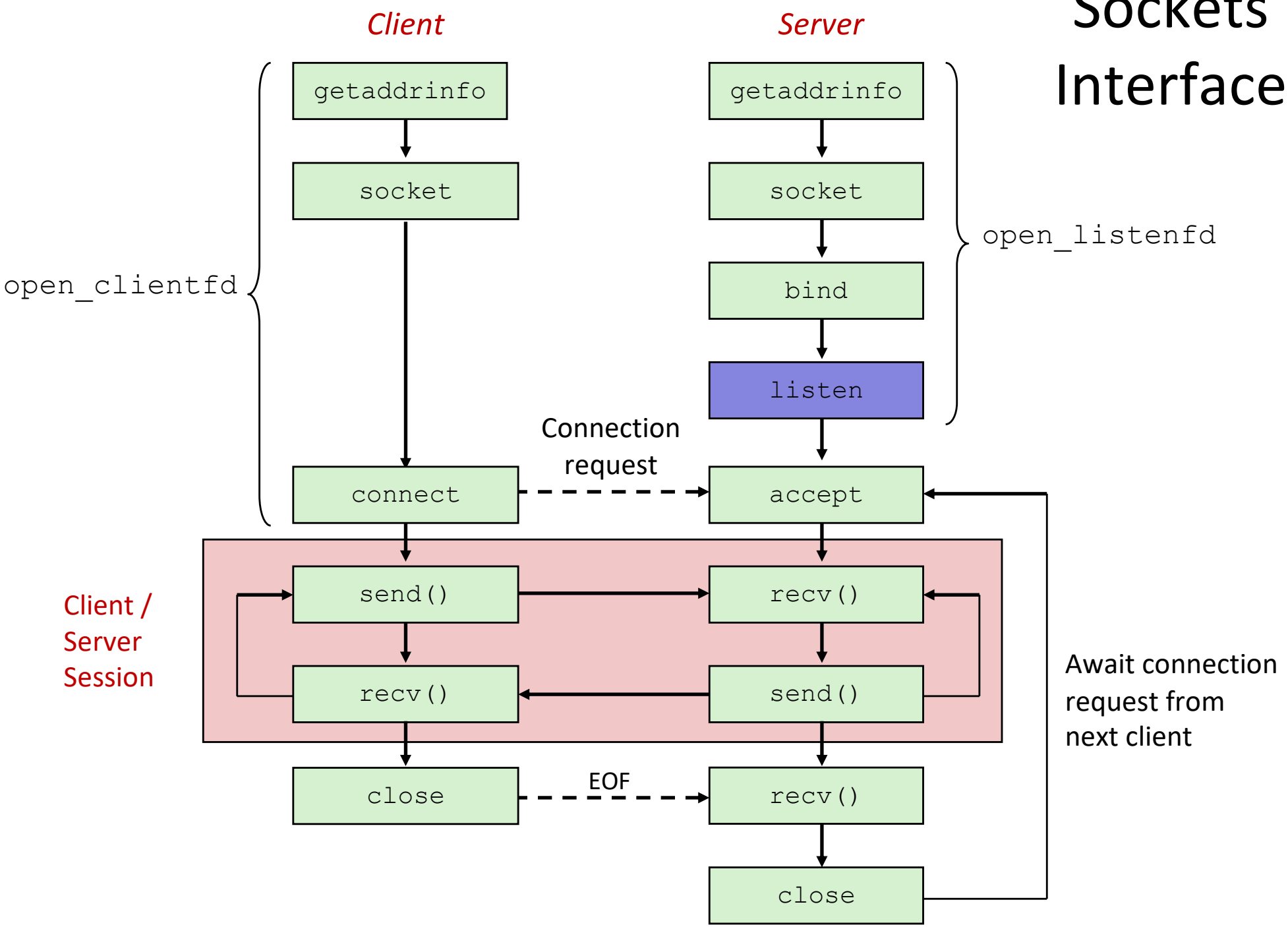


# Server: `bind(2)`

`bind()` associates the socket descriptor with the socket address (created similarly to that of the client)

```
int listenfd;                          /* listening socket */
struct addrinfo *result;               /* DNS host entry initialized in
                                        * call to getaddrbyinfo()*/
...
  /* listenfd will be an endpoint for all requests to port
     on any IP address for this host */
  if (bind(listenfd, rp->ai_addr, rp->ai_addrlen) < 0)
      return -1;
```

# Sockets Interface



# Server: `listen(2)`

`listen()` indicates that this socket will accept connection (`connect`) requests from clients

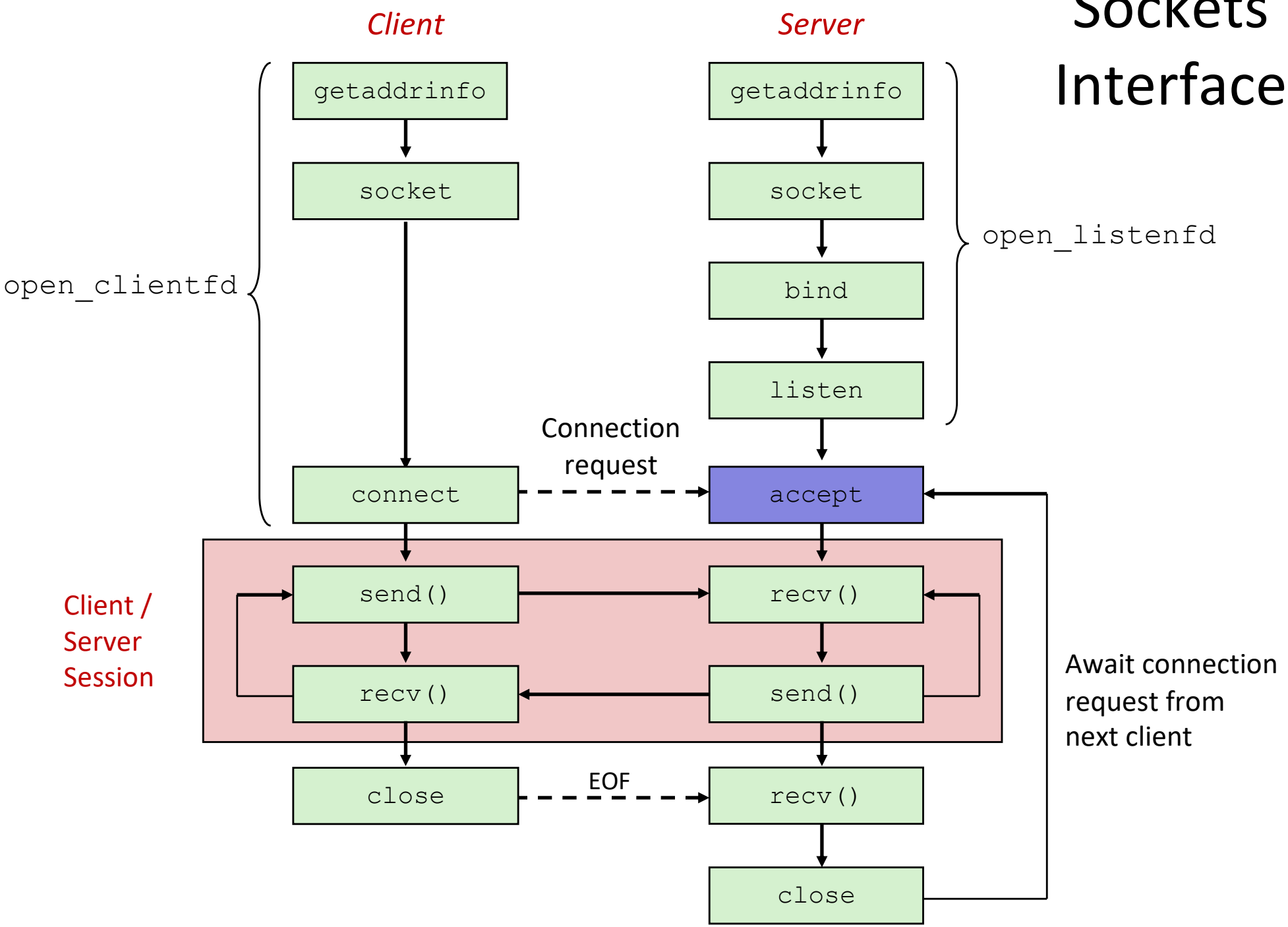
- Kernel assumes an active socket on the client end by default
- `LISTENQ` is a constant indicating how many pending requests are allowed

```
int listenfd; /* listening socket */

...
 /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;
```

We're finally ready to enter the main server loop that accepts and processes client connection requests.

# Sockets Interface



# Server: `accept(2)`

- `accept()` blocks waiting for a connection request

```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
```

- `accept()` returns a *connected descriptor* (`connfd`) with the same properties as the *listening descriptor* (`listenfd`)
  - Returns when the connection between client and server is created and ready for I/O transfers
  - All I/O with the client will be done via the connected socket
- `accept()` also fills in client's IP address

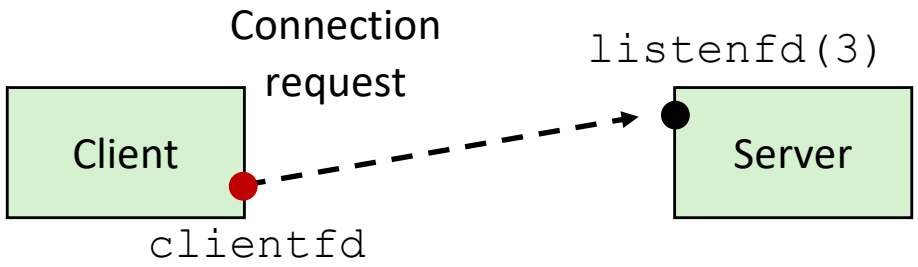
# Connected vs. Listening Descriptors

- 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 create a new thread to handle the request

# `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`*



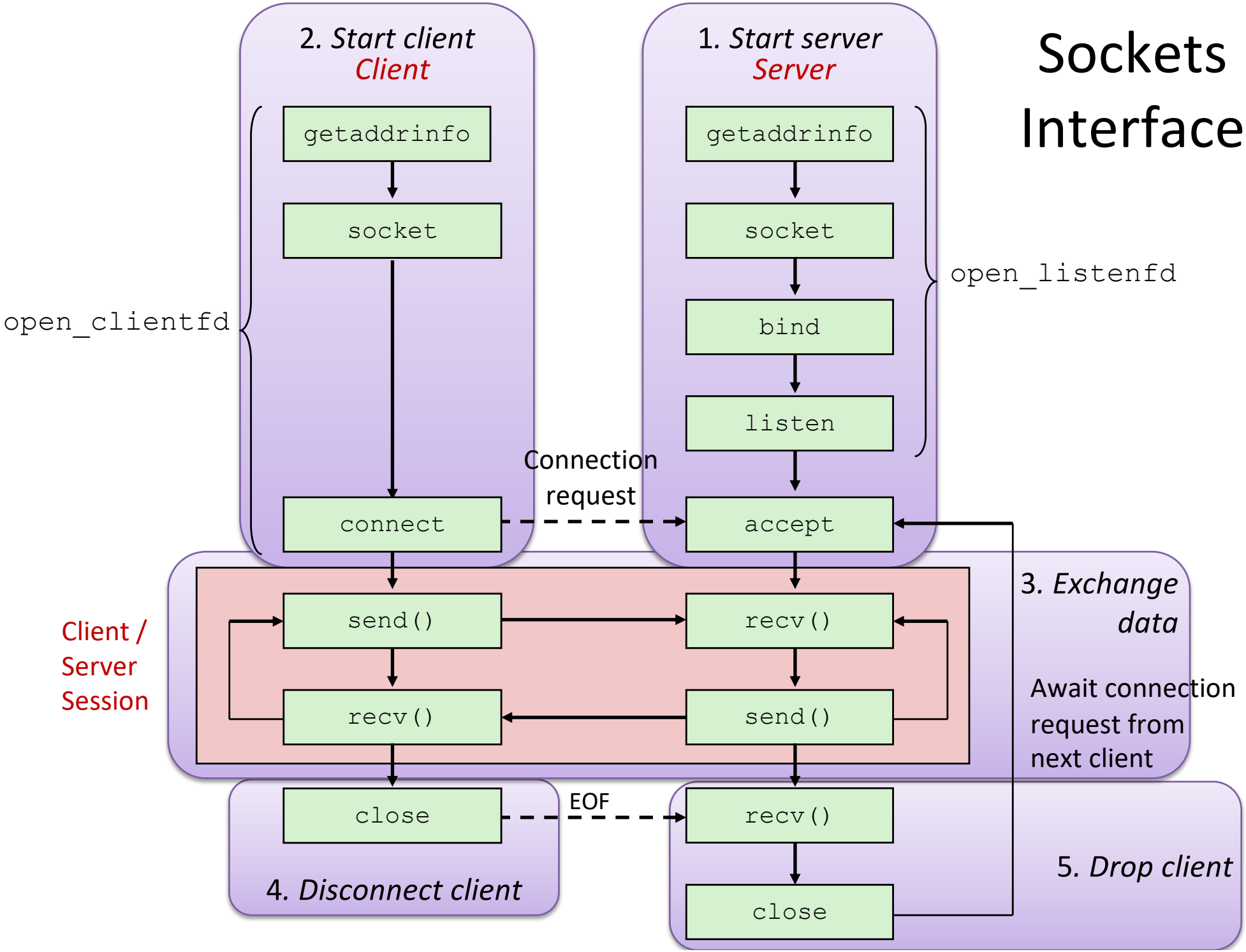

*3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`*

# Question

The listen() call is listening on a different port than the port the server will eventually use to send data to the client.

A. True
B. False

# Sockets Interface



# Parallel Programming

# Parallel Programming

Parallel programming involves:

- Decomposing an algorithm into parts
- Distributing the parts as tasks which are worked on by multiple threads or processes simultaneously
- Coordinating work and communication of those processes (synchronization!)

So we've been doing this…

*How? How have we been parallel programming so far this semester?*

# Two Parallel Programming Models

- Shared memory
  - Processes/threads communicate through shared variables (by using shared memory)
- Message passing
  - Processes communicate by sending and receiving messages over a network
- How do we decide which method to use?
  - Type of parallel architecture being used

# Shared Memory Programming Model

- Processes and threads read and write a set of shared memory locations
  - Communication is very fast and is *implicit*
- OS services used for synchronization
  - Synchronization is *explicit*
- Cooperating processes/threads must all execute on the same computer
  - Difficult to provide when processes are on different machines
- OS services or libraries used for creating processes/threads

# Message Passing Programming Model

- "Shared" data is communicated using send/receive services across an external network
  - Interprocess communication is *explicit*
  - Shared data must be formatted into message chunks for distribution
- Coordination is also via sending/receiving messages
  - Synchronization is *implicit*
- Program components can be run on the same or different computers
  - The computers just need to be connected by a network!
- Standard libraries exist to encapsulate messages:
  - For example: MPI-based libraries