Sockets Programming

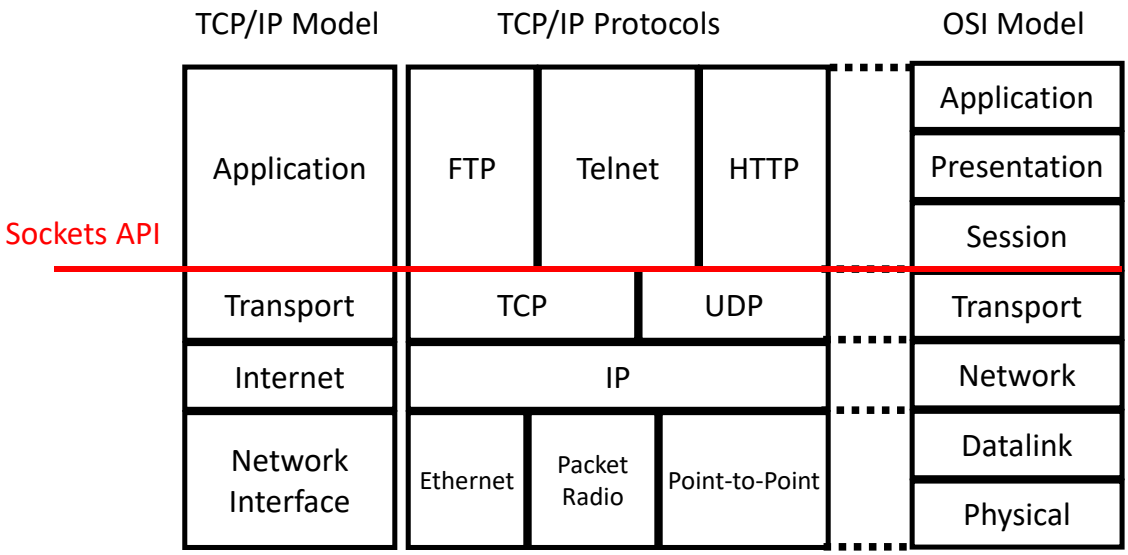
What is a socket?

- A socket is a communication endpoint in the computer network
- Sockets work with UNIX I/O services just like files
- Sockets have special needs:
  - Specifying addresses for communication endpoints
  - Establishing a connection
- For applications to interact with communication protocol suites via sockets, we need sockets API (Application Programming Interface)

Sockets API

- sd = socket(protofamily, type, protocol)
- connect(sd, remoteaddr, addrlen)
- bind(sd, localaddr, addrlen)
- listen(sd, qlen)
- td = accept(sd, &clientaddr, &addrlen)
- close(sd)
- send(td, data, length, flags)
- recv(td, &buffer, length, flags)
- select(maxfd, readset, writeset, exceptset, timeout)
- get/setsocketopt(sd, …)
- Others:
  - sendto, recvfrom, poll, read, write

Where is Sockets API?



What is a client-server paradigm?

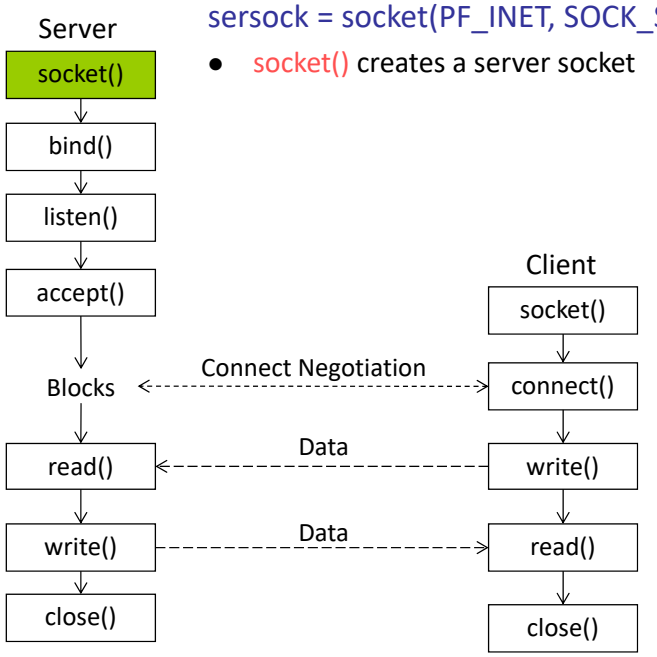
- Server:
  - Server is a process with resources
  - A computer can host multiple servers
  - Server waits for connection requests from clients
  - Server specifies well-known port # when creating socket
- Client:
  - Client is a process that needs resources
  - Client is assigned ephemeral port #
  - Client initiates connection with server
  - Client needs to know server's IP address & port #
  - Server learns client's address & port #

Types of Servers

- Iterative Server
  - Accepts only one connection at any time
- Concurrent Server
  - May accept more than one connection and serve all connections concurrently
  - When the concurrent server receives a request, it forks a child server to serve that request; the server itself waits for a new request
  - When the child server finishes the service, it exits

TCP Sockets Programming

[Server Side] Step 1: Create a TCP Socket



sersock = socket(PF_INET, SOCK_STREAM, 0);

- socket() creates a server socket

Create a TCP Socket

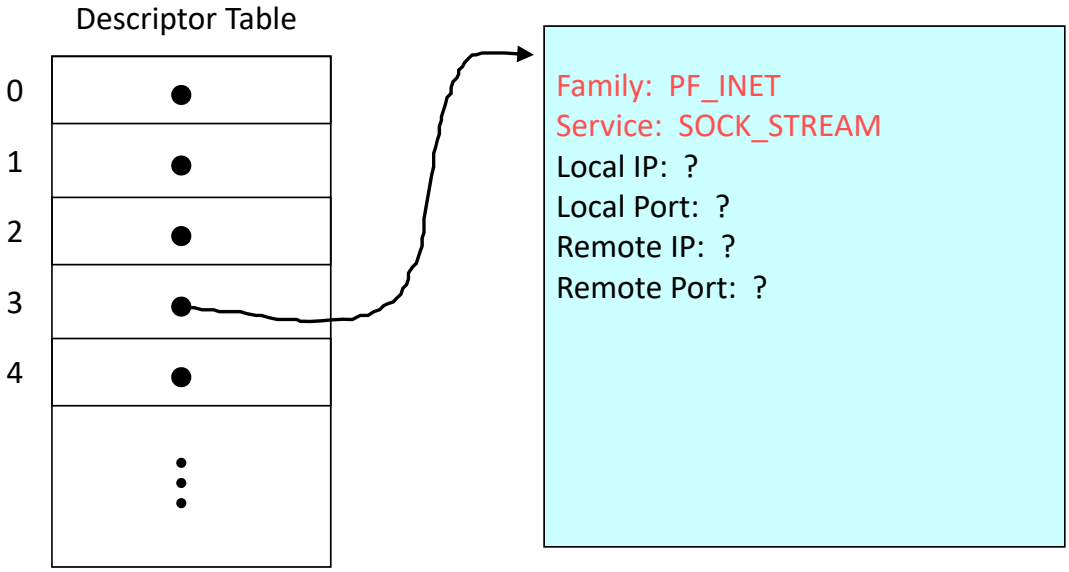
socket()

```
% man –s 2 socket
  cc [flag …] file … -lsocket –lnsl [library …]
  #include <sys/types.h>
  #include <sys/socket.h>
  int socket(int domain, int type, int protocol);
```

- domain (family)
  - PF_UNIX: Unix system internal protocols
  - PF_INET: IPv4 Internet protocols
- type
  - SOCK_STREAM: stream socket, for connection-oriented
  - SOCK_DGRAM: datagram socket, for connectionless
  - SOCK_RAW: raw socket, bypassing the transport layer

- protocol
  - IPPROTO_TCP: used with SOCK_STREAM, TCP Protocol
  - IPPROTO_UDP: used with SOCK_DGRAM, UDP Protocol
  - IPPROTO_IP: used with SOCK_RAW, IP protocol
  - IPPROTO_ICMP: used with SOCK_RAW, ICMP protocol
  - IPPROTO_RAW: used with SOCK_RAW, bypassing the IP layer
  - 0: the default protocol
- socket() system call returns
  - a non-negative integer (socket descriptor/handle)
  - or -1 on error

Socket Descriptor Data Structure



Error Checking and Reporting

- Whenever an error occurs during a system call
  - System call sets the value of a global variable errno
  - To print out the error message, you should use the special "void perror(const char * string)" function right after the error was produced; otherwise, errno may be overwritten in calls to other functions
    - ✓ perror("The following error occurred:");
    - ✗ printf("The error code is %d\n", errno);

Error Checking and Reporting

- Do error checking after every system call
- Example:

```
if ( socket(PF_INET, SOCK_STREAM, 0) < 0 )
{
   perror("The following error occurred:");
   exit(1);
}
```

- Please don't forget to do error checking in your labs!

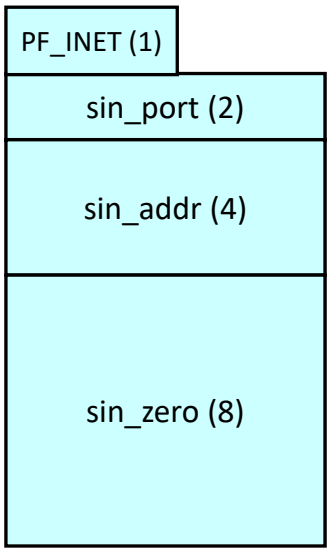
[Server Side] Step 2: Specify an Endpoint Address for Server

- TCP/IP uses an IP address and a port number to specify an endpoint address
  - Other protocol suites (families) may use other addressing schemes
- Ports
  - Each user process associates its connection with the transport layer protocol with a unique port # (16 bits)
  - The same port number can be used for different protocols
    - E.g., you may use port 2000 with TCP and port 2000 with UDP, but you cannot have two TCP services both with port 2000
  - Some well-known ports are reserved for well-known services
    - TCP port 21 for ftp
    - TCP port 80 for http
  - Ports 0 ~ 1023 are reserved, so user processes must use ports $\geq$ 1024

IPv4 Socket Address Structure

```
struct sockaddr_in {
    sa_family_t sin_family; // PF_INET
    in_port_t sin_port;     // 16-bit port #
    struct in_addr sin_addr; // 32-bit IPv4 address
    char sin_zero[8];       // unused
};
```

```
struct in_addr {
  // 32-bit IPv4 address in network byte order
    in_addr_t s_addr;
};
```



Byte Order

- Little-Endian Byte Order
  - Little end of a multi-byte number is stored at the starting address
- Big-Endian Byte Order
  - Big end of a multi-byte number is stored at the starting address

- Example: c = 0x 0001

  c[0] c[1]

  - little-endian:

  - big-endian:

- Different systems have different byte orders (host byte order)
- Internet protocols use big-endian byte ordering!

Network Byte Order

- All values stored in a sockaddr_in must be in network byte order
  - sin_port: a 16-bit TCP or UDP port number
  - sin_addr: a 32-bit IP address

Common Mistake: Ignoring Network Byte Order!

Network Byte Order Functions

'h' : host byte order
'n' : network byte order
's' : short (16 bit)
'l' : long (32 bit)

- uint16_t htons(uint16_t);
- uint16_t ntohs(uint16_t);

- uint32_t htonl(uint32_t);
- uint32_t ntohl(uint32_t);

IPv4 Address Conversion Functions

- in_addr_t inet_addr(const char *)
  - convert an ASCII dotted-decimal IP address to a 32-bit network byte ordered IP address
- char *inet_ntoa(struct in_addr)
  - convert a network byte ordered value to an ASCII dotted-decimal string
- Example:

```
char IP_ADDRESS[16] = "129.186.23.86";
serveraddr.sin_addr.s_addr = inet_addr(IP_ADDRESS);
```

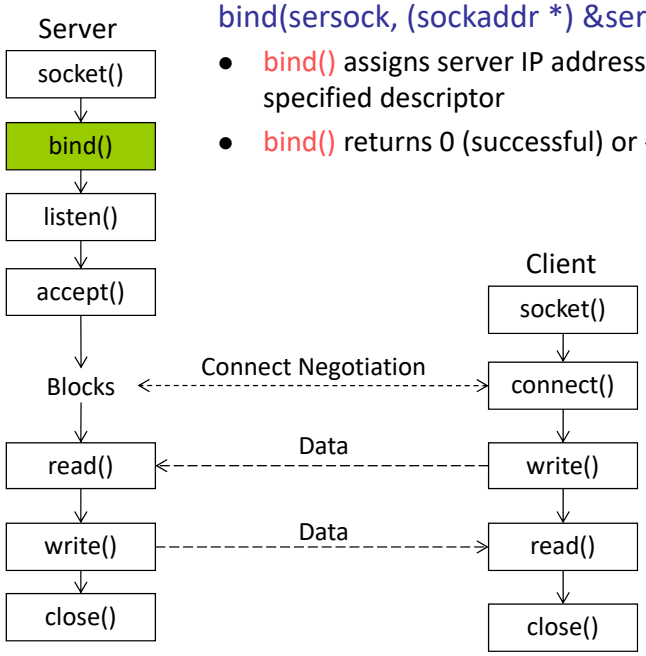

Specify an Endpoint Address for Server

- Specifying an endpoint address using the socket address structure:

```
struct sockaddr_in serveraddr;

serveraddr.sin_family = PF_INET;
serveraddr.sin_port = htons( 80 );
serveraddr.sin_addr.s_addr = inet_addr( IP_ADDRESS );
```

[Server Side] Step 3: Assign Address to Socket



bind(sersock, (sockaddr *) &serveraddr, sizeof(serveraddr));

- bind() assigns server IP address & port # to the socket with the specified descriptor
- bind() returns 0 (successful) or -1 (failure)

Assign Address to Socket

bind()

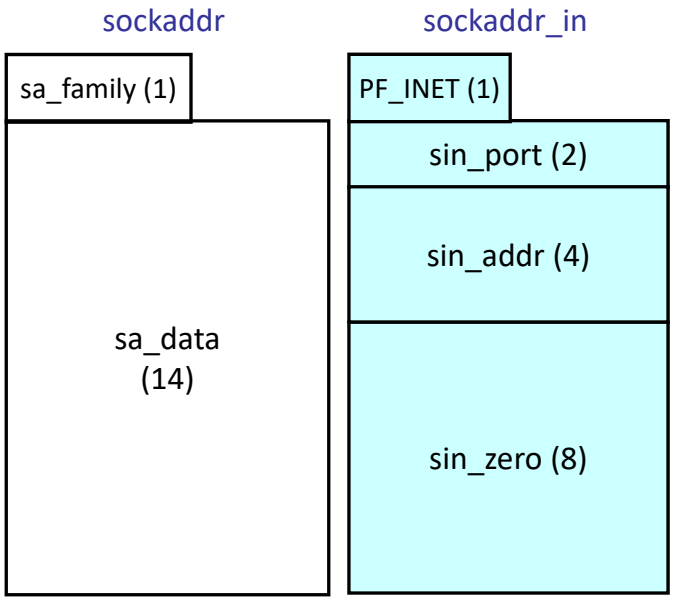
```
% man –s 2 bind
  cc [flag …] file … -lsocket –lnsl [library …]
  #include <sys/types.h>
  #include <sys/socket.h>
  int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

- sockfd
  - Socket descriptor returned by socket()
- myaddr
  - Pointer to a sockaddr structure
- addrlen
  - Length of the sockaddr structure

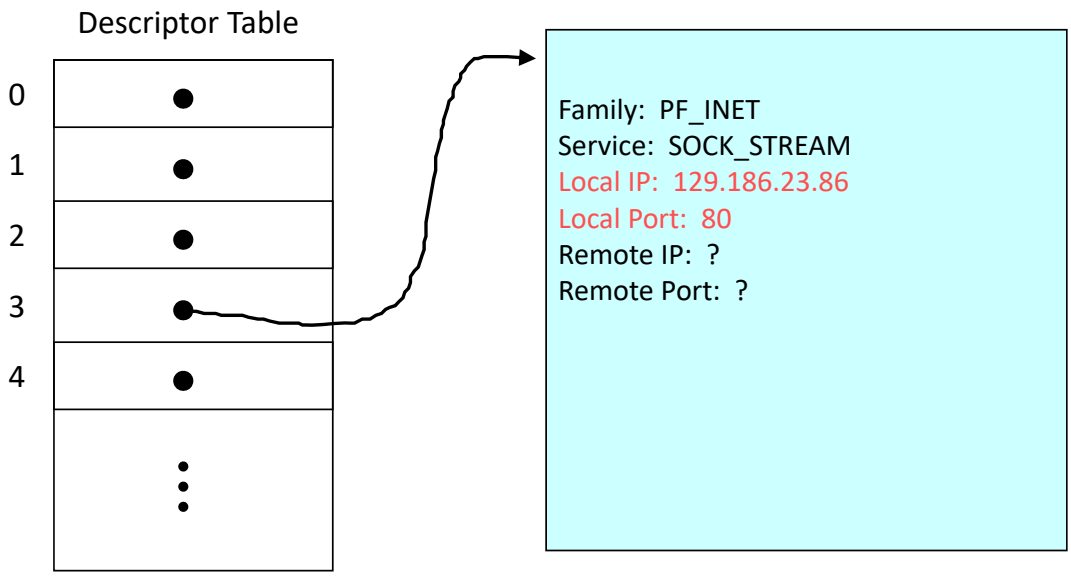
Generic Socket Address Structure

The C functions that make up the sockets API expect structures of generic sockaddr type

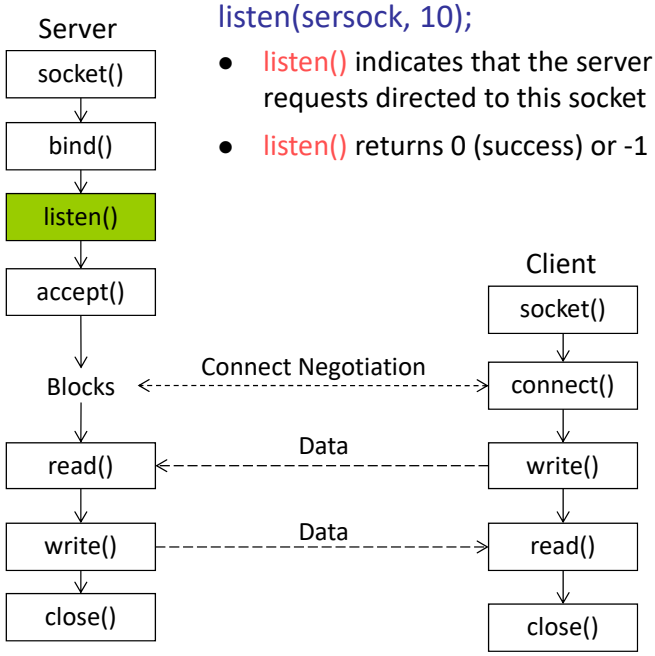
- so if omitting the cast (struct sockaddr *) in front of pointer to a sockaddr_in structure, you will get a warning when compiling



Socket Descriptor Data Structure



[Server Side] Step 4: Make a Passive-Mode Socket



listen(sersock, 10);

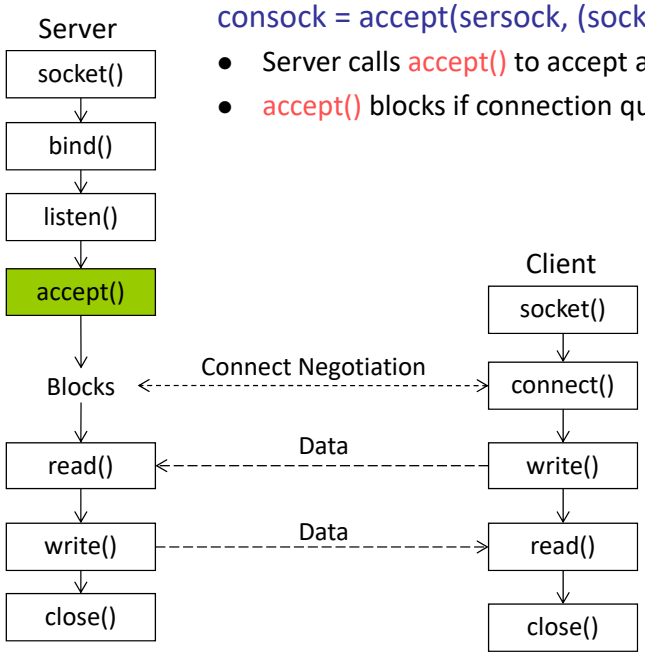
- listen() indicates that the server is willing to accept connection requests directed to this socket
- listen() returns 0 (success) or -1 (failure)

listen()

```
% man –s 2 listen
  cc [flag …] file … -lsocket –lnsl [library …]
  #include <sys/types.h>
  #include <sys/socket.h>
  int listen(int sockfd, int backlog);
```

- sockfd
  - Socket descriptor returned by socket() and already bound to server address via bind()
- backlog
  - Maximum number of connections that may be queued while waiting for the server to accept them

[Server Side] Step 5: Accept a Connection



consock = accept(sersock, (sockaddr *) &clientaddr, &len)

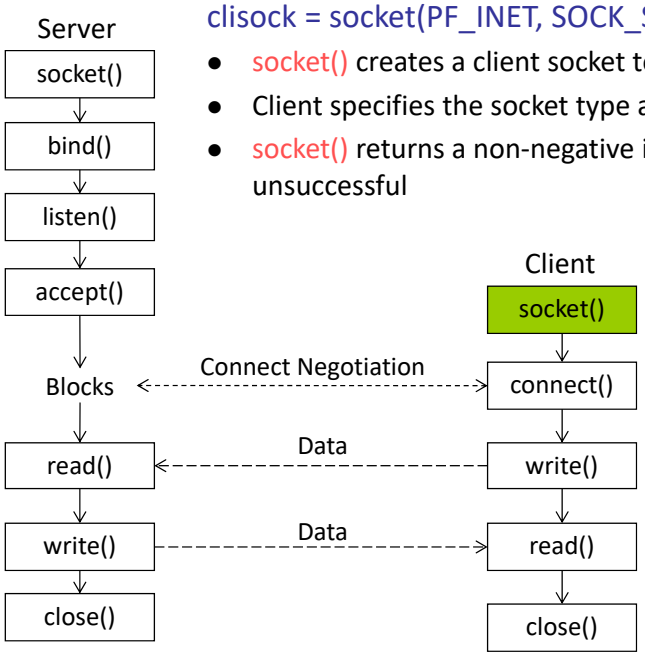
- Server calls accept() to accept an incoming connection request
- accept() blocks if connection queue is empty

accept()

```
%  man –s 2 accept
   cc [flag …] file … -lsocket –lnsl [library …]
   #include <sys/types.h>
   #include <sys/socket.h>
   int accept(int sockfd, struct sockaddr *cliaddr, int *addrlen);
```

- sockfd
  - Socket descriptor of the passive-mode TCP socket
- cliaddr
  - Pointer to sockaddr structure in which the client's address will be stored
- addrlen
  - Pointer to an integer in which the length of the sockaddr structure is returned

[Client Side] Step 1



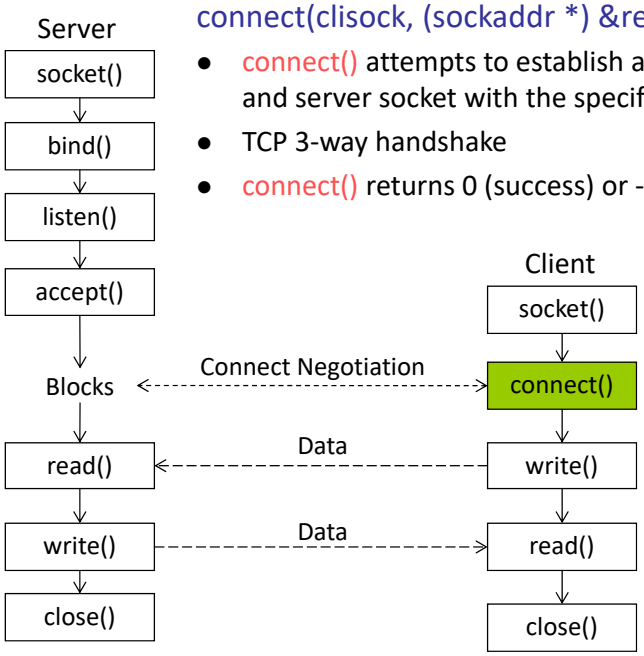
clisock = socket(PF_INET, SOCK_STREAM, 0);

- socket() creates a client socket to connect to the server
- Client specifies the socket type and protocol
- socket() returns a non-negative integer socket descriptor or -1 if unsuccessful

[Client Side] Step 2, 3

- Skipped …
- When using a connection-oriented service, the client does not have to bind its socket:
  - Kernel will take care of assigning the client address (Port #, IP address)

[Client Side] Step 4: Connect to Server



connect(clisock, (sockaddr *) &remoteaddr, sizeof(remoteaddr));

- connect() attempts to establish a connection between client socket and server socket with the specified server IP address and port #
- TCP 3-way handshake
- connect() returns 0 (success) or -1 (failure)

connect()

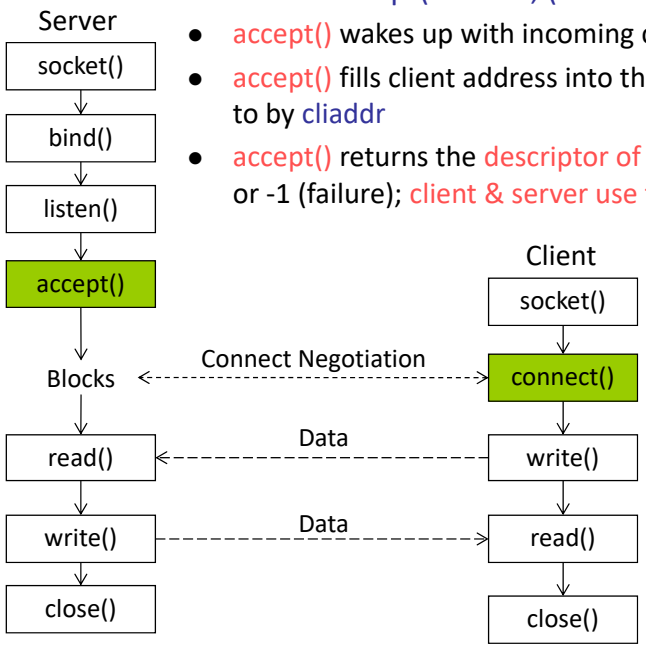
```
% man –s 2 connect
  cc [flag …] file … -lsocket –lnsl [library …]
  #include <sys/types.h>
  #include <sys/socket.h>
  int connect(int sockfd, struct sockaddr *seraddr, socklen_t addrlen);
```

- sockfd
  - Socket descriptor returned by socket() and already implicitly bound
- seraddr
  - Pointer to sockaddr structure that contains the server's endpoint address
- addrlen
  - Size of the sockaddr structure

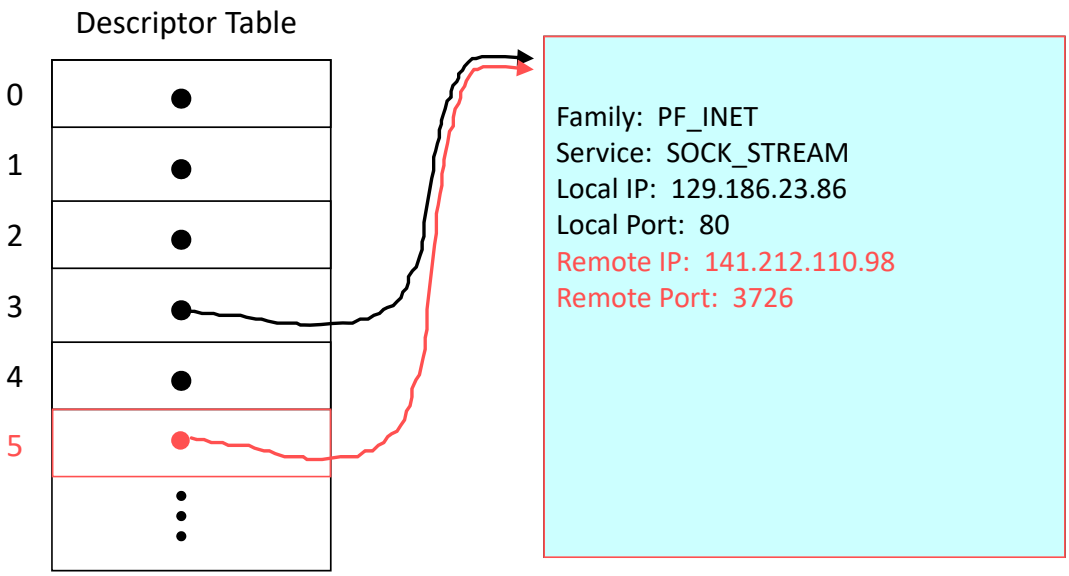
Back to [Server Side] Step 5

consock = accept(sersock, (sockaddr *) &clientaddr, &len)

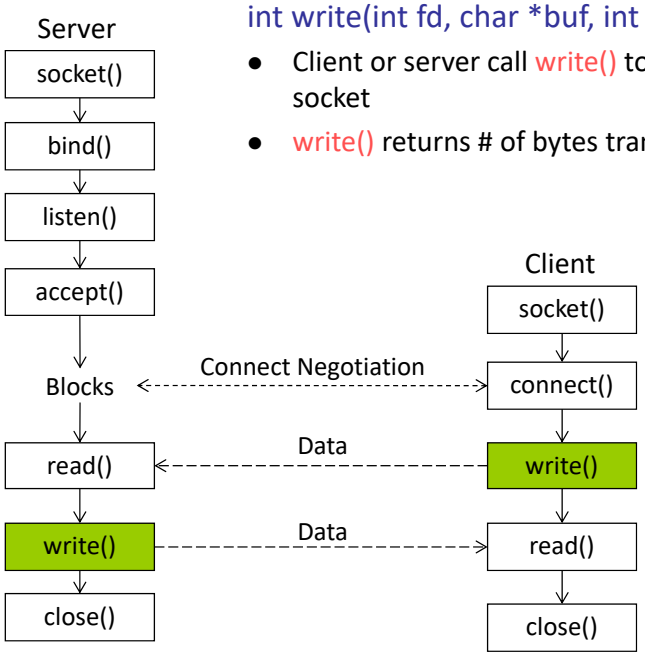
- accept() wakes up with incoming connection request
- accept() fills client address into the socket address structure pointed to by cliaddr
- accept() returns the descriptor of a new connected socket (success) or -1 (failure); client & server use this new socket for data transfer



Socket Descriptor Data Structure



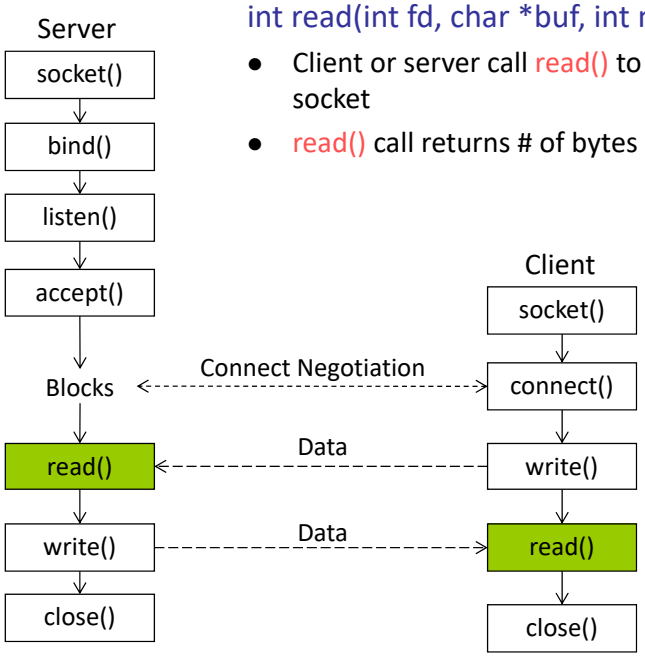
[Server-Client Interaction]



int write(int fd, char *buf, int num);

- Client or server call write() to transmit data into a connected socket
- write() returns # of bytes transferred or -1 (failure)

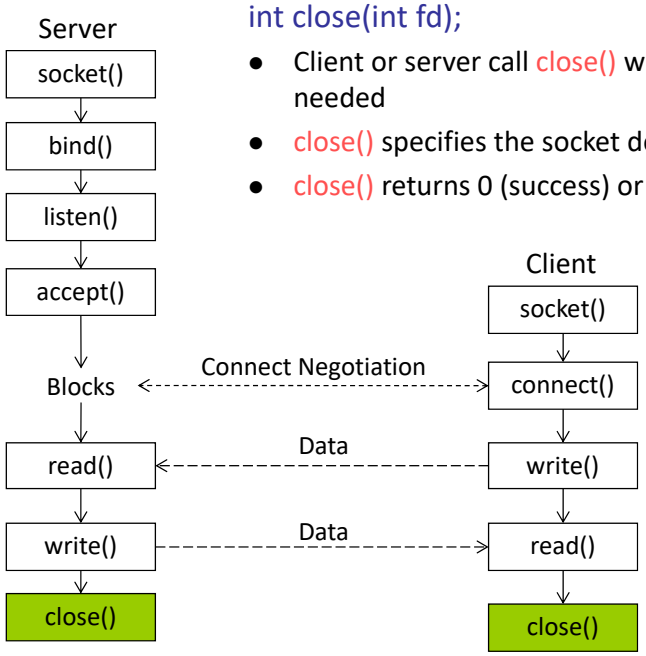
[Server-Client Interaction]



int read(int fd, char *buf, int max);

- Client or server call read() to receive data from a connected socket
- read() call returns # of bytes read or -1 (failure)

[Server or Client Side] Final Step: Connection Termination

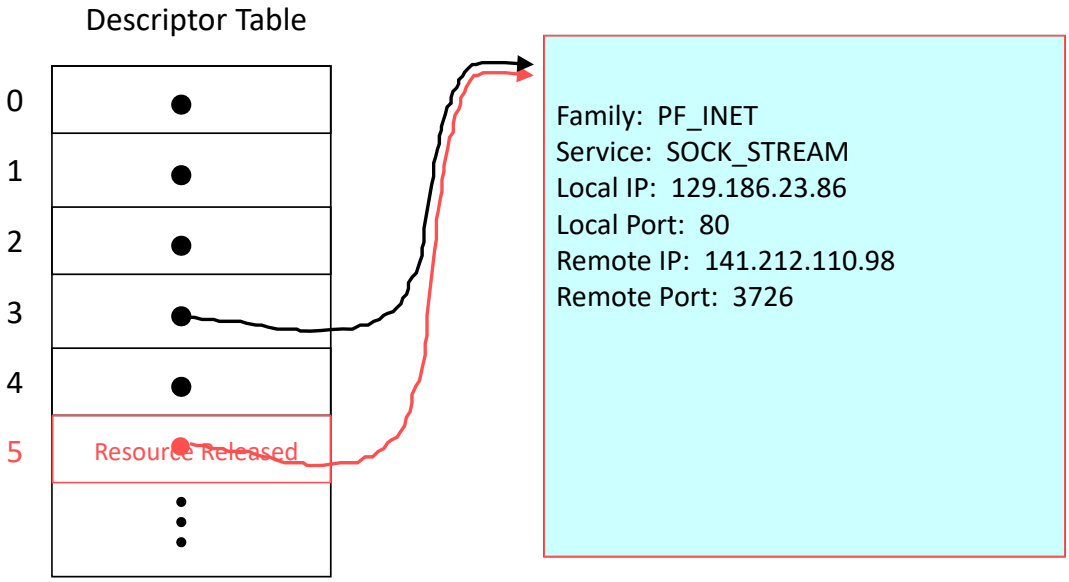


int close(int fd);

- Client or server call close() when the socket is no longer needed
- close() specifies the socket descriptor
- close() returns 0 (success) or -1 (failure)

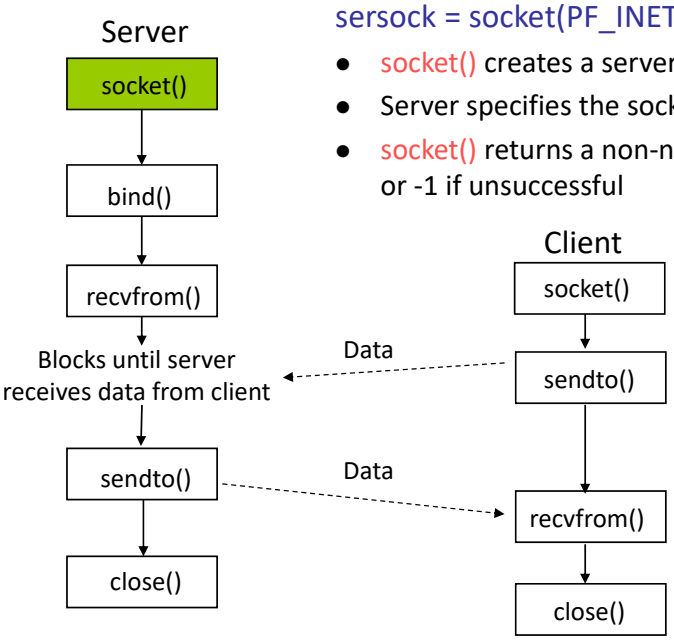
Note: close() initiates the TCP graceful close sequence

Socket Descriptor Data Structure



UDP Sockets Programming

[Server Side] Step 1



sersock = socket(PF_INET, SOCK_DGRAM, 0);

- socket() creates a server socket
- Server specifies the socket type and protocol
- socket() returns a non-negative integer socket descriptor or -1 if unsuccessful

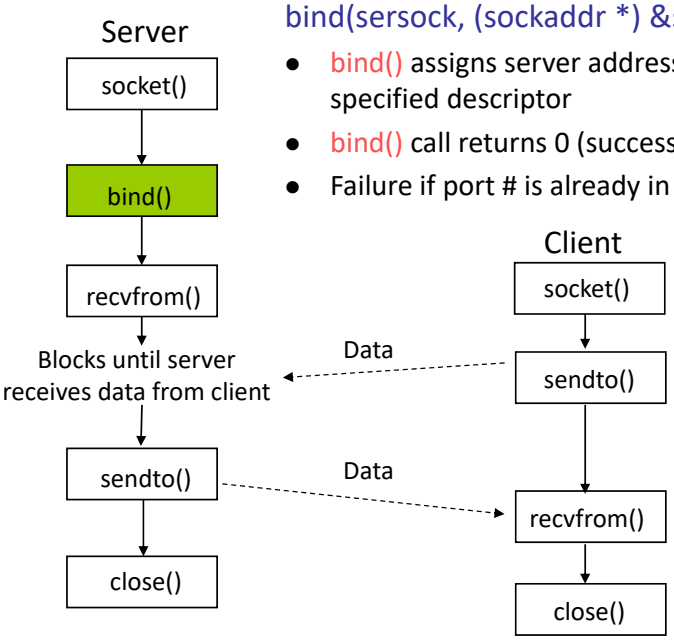
[Server Side] Step 2

- Specifying the server address:

```
struct sockaddr_in serveraddr;

serveraddr.sin_family = PF_INET;
serveraddr.sin_port = htons( 37 );
serveraddr.sin_addr.s_addr = htonl( INADDR_ANY );
```


[Server Side] Step 3



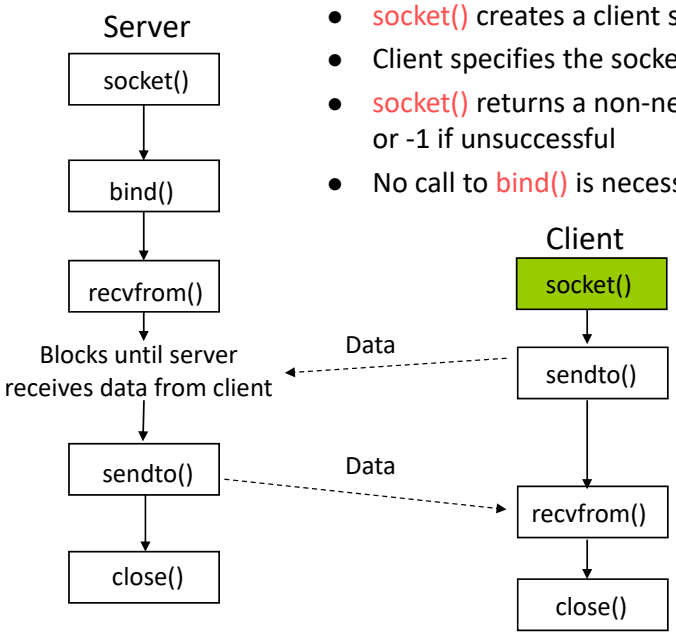
bind(sersock, (sockaddr *) &serveraddr, sizeof(serveraddr));

- bind() assigns server address & port # to the socket with specified descriptor
- bind() call returns 0 (success) or -1 (failure)
- Failure if port # is already in use

[Client Side] Step 1, 2, 3

clisock = socket(PF_INET, SOCK_DGRAM, 0);

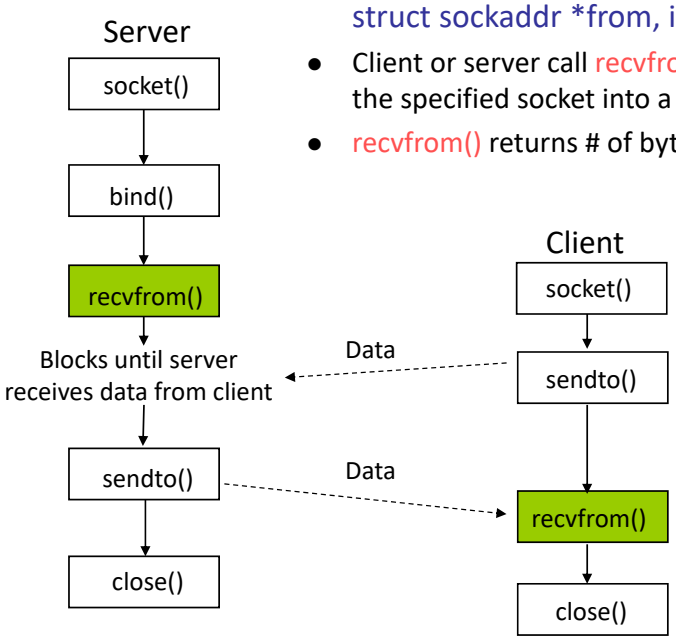
- socket() creates a client socket
- Client specifies the socket type and protocol
- socket() returns a non-negative integer socket descriptor or -1 if unsuccessful
- No call to bind() is necessary!



[Server-Client Interaction]

int recvfrom(int fd, char *buf, int max, int flags,
struct sockaddr *from, int *fromlen);

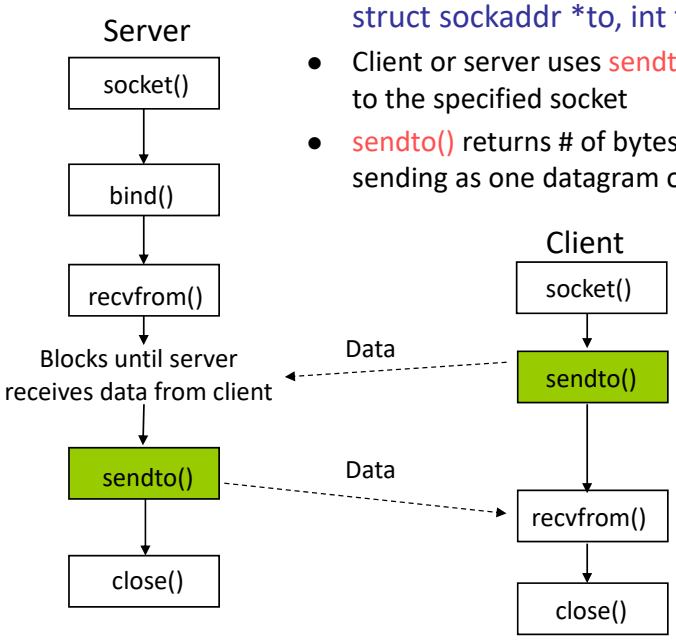
- Client or server call recvfrom() to copy bytes received in the specified socket into a specified location
- recvfrom() returns # of bytes received or -1 (failure)



[Server-Client Interaction]

int sendto(int fd, char *buf, int num, int flags,
struct sockaddr *to, int tolen);

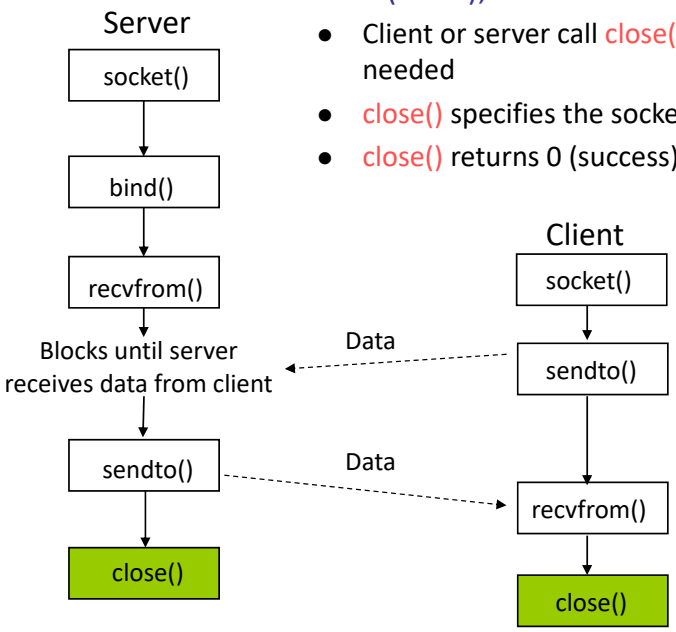
- Client or server uses sendto() to transfer bytes in buffer to the specified socket
- sendto() returns # of bytes accepted by kernel for sending as one datagram or -1 (failure)



You can also connect() to a UDP socket!

- A UDP socket can be used in a call to connect()
- But:
  - This simply tells the kernel the address of the server
  - No handshake is performed
  - No data of any kind is sent on the network as a result of calling connect() on a UDP socket
- connect() is typically used with UDP when communication is with a single peer only
- Once a UDP socket is connected:
  - can use write() and read()

[Server or Client Side] Final Step: Close the Socket



close(int fd);

- Client or server call close() when the socket is no longer needed
- close() specifies the socket descriptor
- close() returns 0 (success) or -1 (failure)

Other Useful Functions

1. int getsockopt(int s, int level, int optname, void *optval, int *optlen);
2. int setsockopt(int s, int level, int optname, const void *optval, int optlen);

- Get and set socket options
- Some useful options
  - SO_REUSEADDR:
    - Allows server to restart
    - Allows multiple servers to bind to the same port with different IP addresses
    - All TCP servers should specify this option to allow the server to be restarted

- Some useful options
  - SO_KEEPALIVE
    - Server sends a probe which client must respond to indicate that it is still alive
  - SO_LINGER
    - Indicates whether close() should return immediately or wait for connection termination
  - SO_RCVBUF and SO_SNDBUF
    - Set receive and send buffer sizes

3. int gethostname(char *name, int namelen);
   - Fills name with the standard host name of the current host

4. int getsockname(int sockfd, struct sockaddr *localaddr, int *addrlen);
   - Fills localaddr with the address of socket sockfd

5. int getpeername(int sockfd, struct sockaddr *peeraddr, int *addrlen);
   - Fills peeraddr with the address of the peer connected to socket sockfd

6. struct hostent *gethostbyname(const char *name);

- name: string that holds the host name
- struct hostent {

```
    char *h_name;       // canonical name of host
    char **h_alias;     // alias list
    int h_addrtype;     // host address type
    int h_length;       // length of address
    char **h_addr_list; // list of addresses
}
```

- Example

```
struct hostent *Address;
Address = gethostbyname("www.ece.iastate.edu");
```


7. struct hostent *gethostbyaddr(const char *addr, int len, int type);

- addr: a pointer to in_addr structure containing the IP address

- Example

```
struct hostent *Address;
unsigned long IP;

IP = inet_addr("129.186.23.86");
Address = gethostbyaddr((const char *) &IP, sizeof(IP), AF_INET);
```