Socket Programming

TCP UDP

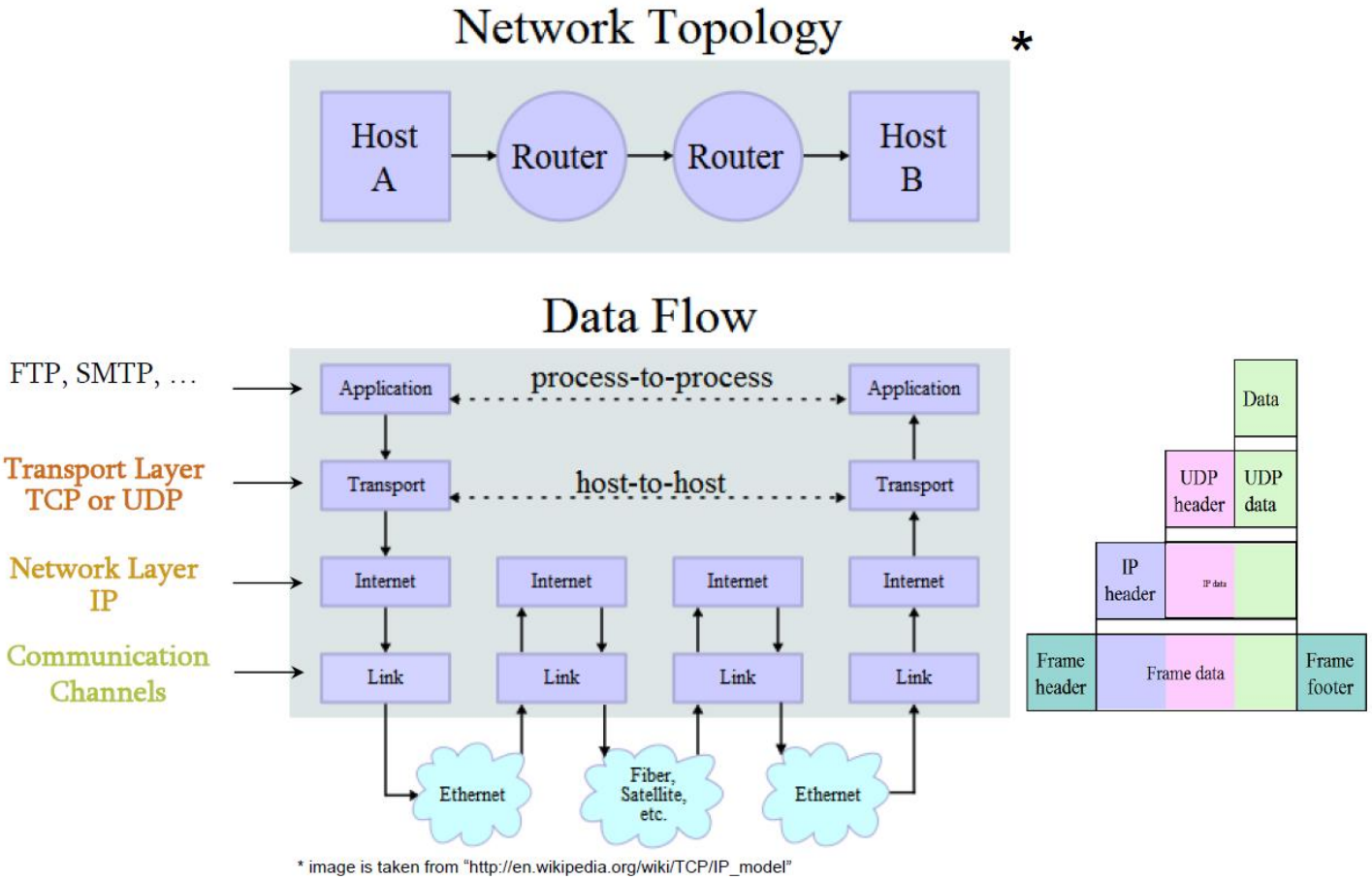
Introduction

- Computer Network
  - hosts, routers, communication channels
- Hosts run applications
- Routers forward information
- Packets: sequence of bytes
  - contain control information
  - e.g. destination host
- Protocol is an agreement
  - meaning of packets
  - structure and size of packets
  - e.g. Hypertext Transfer Protocol (HTTP)

TCP/IP Protocol

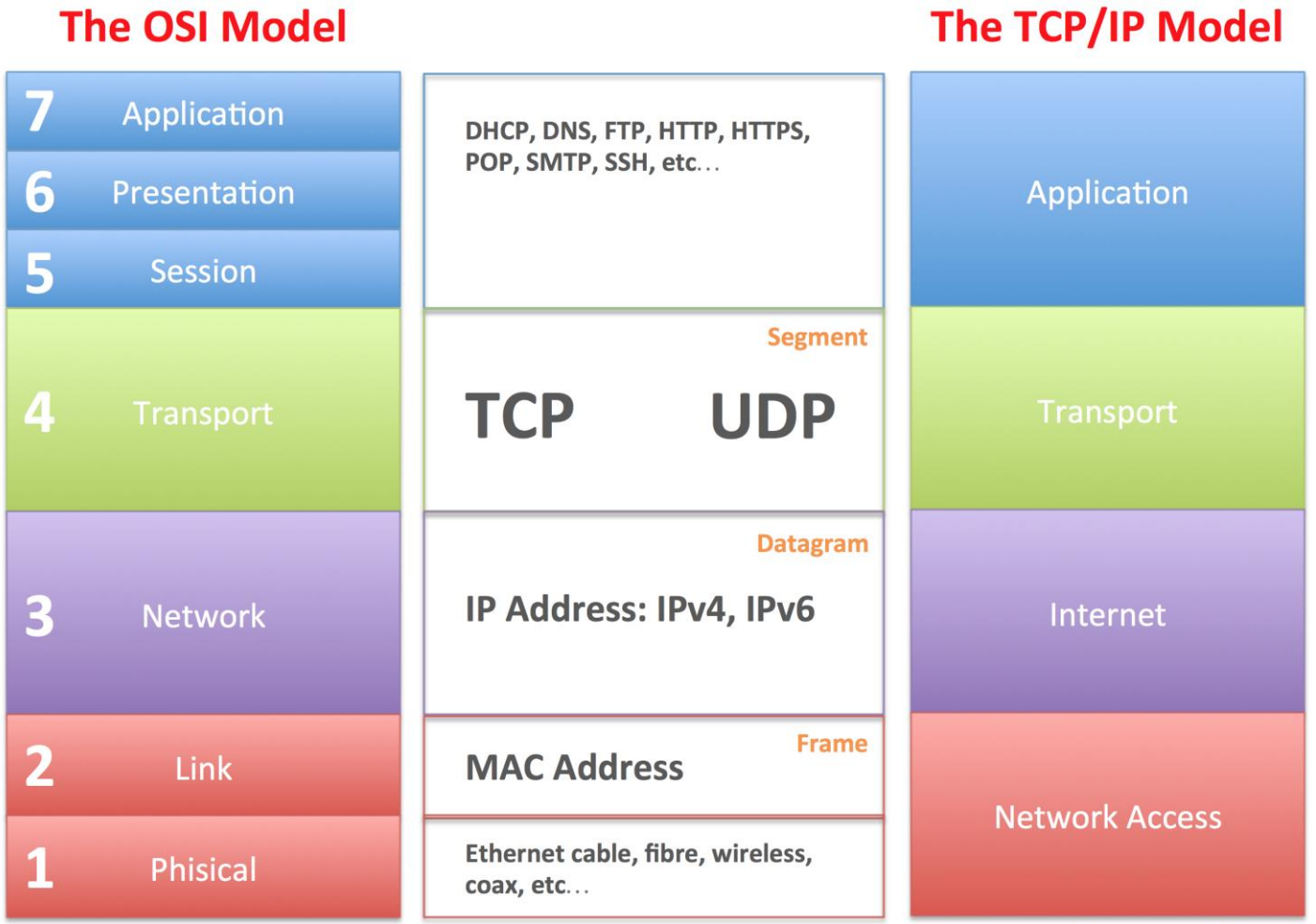
- TCP/IP provides end-to-end connectivity specifying how data should be
  - formatted,
  - addressed,
  - transmitted,
  - routed, and
  - received at the destination
- can be used in the internet and in stand-alone private networks
- it is organized into layers

TCP/IP



* image is taken from "http://en.wikipedia.org/wiki/TCP/IP_model"

OSI Model vs TCP/IP Model



Socket Programming

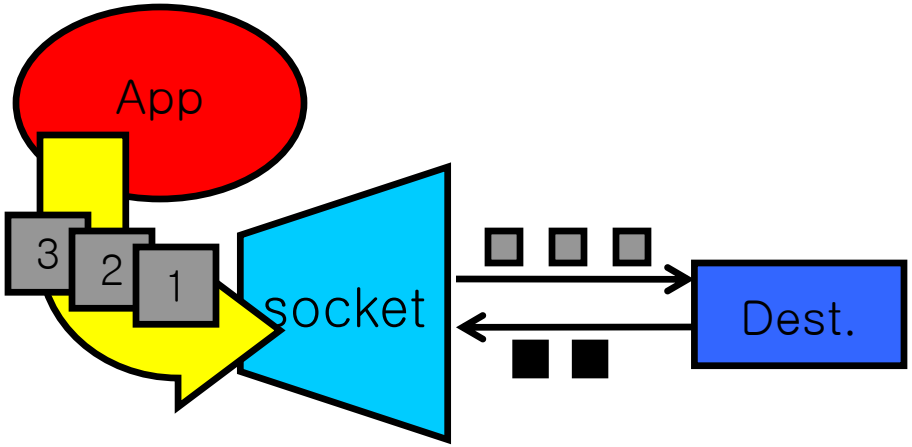
- What is a socket?
- Using sockets
  - Types (Protocols)
  - Associated functions
  - Styles
  - Using sockets in C

What is a socket?

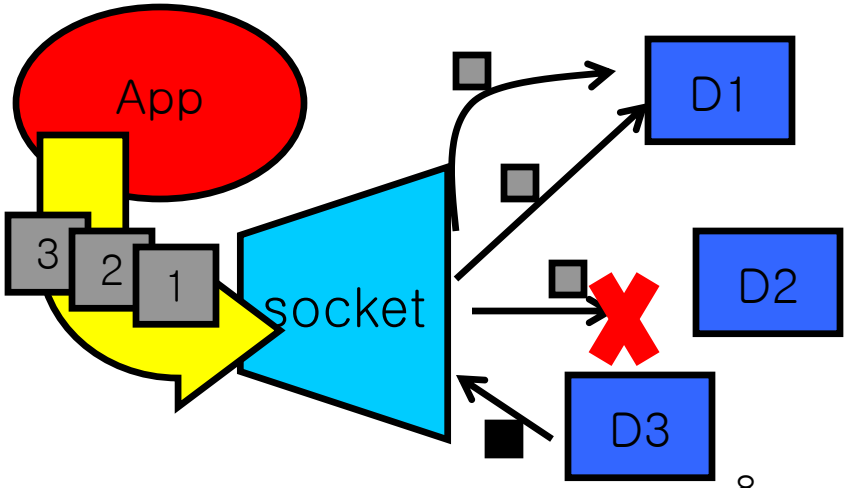
- An interface between application and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional



- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive



Socket Creation in C: socket

- int s = socket(domain, type, protocol);
  - s: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) – typically used
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - protocol: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Ports

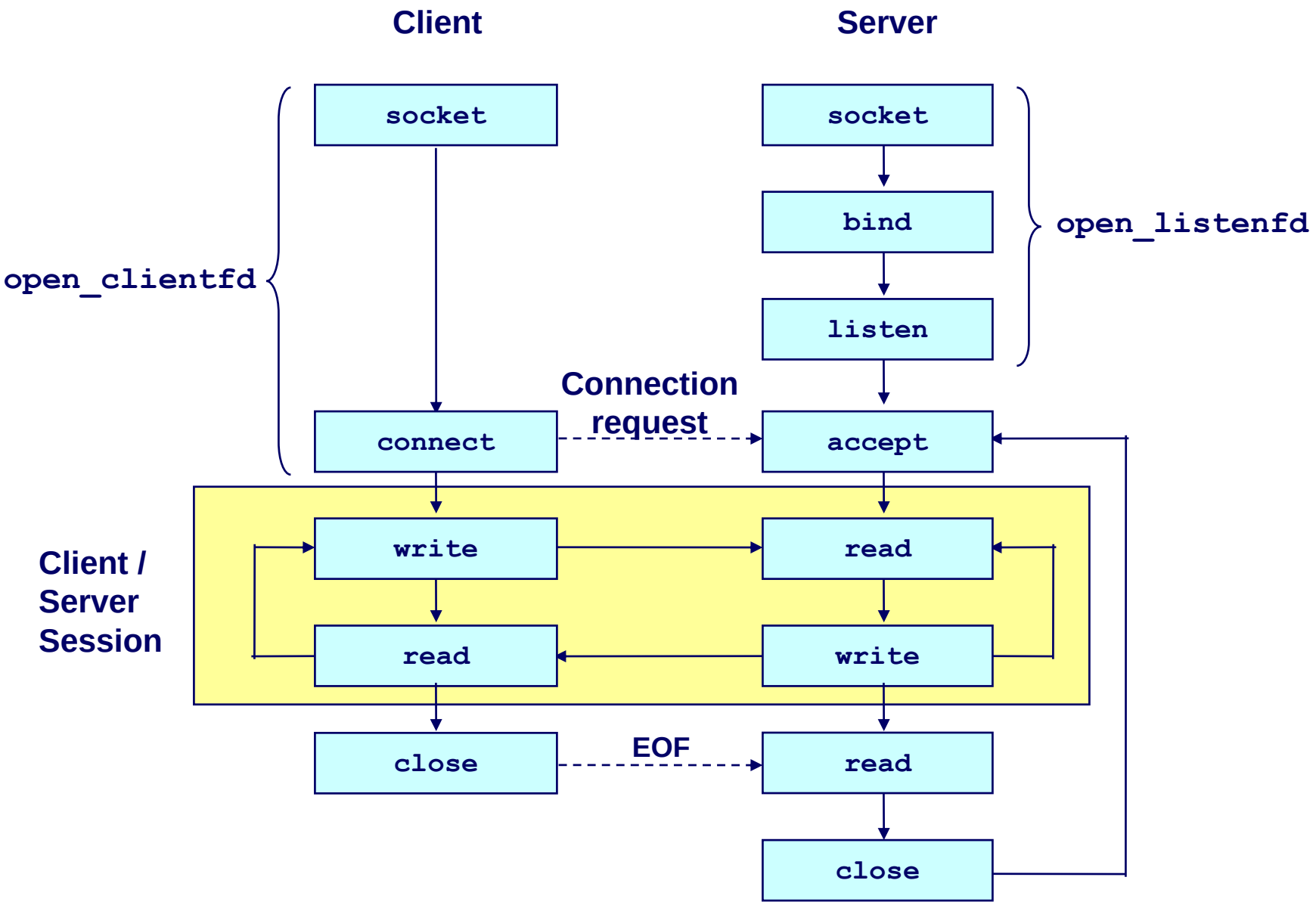
- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 22: SSH
  - 23: Telnet
  - 80: HTTP

- A socket provides an interface to send data to/from the network through a port

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)

Overview


Client
Server
socket
socket
bind
listen
open_clientfd
open_listenfd
Connection
request
connect
accept
Client /
Server
Session
write
read
read
write
close
EOF
read
close

Step 1 – Setup Socket

- **Both client and server need to setup the socket**
  - *int socket(int domain, int type, int protocol);*
- *domain*
  - AF_INET -- IPv4 (AF_INET6 for IPv6)
- *type*
  - SOCK_STREAM -- TCP
  - SOCK_DGRAM -- UDP
- *protocol*
  - 0
- For example,
  - *int sockfd = socket(AF_INET, SOCK_STREAM, 0);*

Step 2 (Server) - Binding

- **Only server need to bind**
  - *int bind(int sockfd, const struct sockaddr *my_addr, socklen_t ad drlen);*
- *sockfd*
  - file descriptor socket() returned
- *my_addr*
  - struct sockaddr_in for IPv4
  - cast (struct sockaddr_in*) to (struct sockaddr*)

```
struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
struct in_addr {
    unsigned long s_addr;  // load with inet_aton()
};
```

What is that Cast?

- bind() takes in protocol-independent (struct sockaddr*)

```
struct sockaddr {
    unsigned short  sa_family;   // address family
    char            sa_data[14]; // protocol address
};
```

  - C's polymorphism
  - There are structs for IPv6, etc.

Step 2 (Server) - Binding contd.

- *addrlen*
  - size of the sockaddr_in

```
struct sockaddr_in saddr;
int sockfd;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) { // from back a couple slides
printf("Error creating socket\n");
...
}

memset(&saddr, '\0', sizeof(saddr));          // zero structure out
saddr.sin_family = AF_INET;                   // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY);    // bind to any local address
saddr.sin_port = htons(port);                 // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
printf("Error binding\n");
...
}
```

What is htonl(), htons()?

- Byte ordering
  - Network order is big-endian
  - Host order can be big- or little-endian
    - x86 is little-endian
    - SPARC is big-endian
- Conversion
  - *htons(), htonl()*: host to network short/long
  - *ntohs(), ntohl()*: network order to host short/long
- What need to be converted?
  - Addresses
  - Port
  - etc.

Step 3 (Server) - Listen

- **Now we can listen**
  - *int listen(int sockfd, int backlog);*
- *sockfd*
  - again, file descriptor socket() returned
- *backlog*
  - number of pending connections to queue
- For example,
  - *listen(sockfd, 5);*

Step 4 (Server) - Accept

- **Server must explicitly accept incoming connections**
  - *int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)*
- *sockfd*
  - again... file descriptor socket() returned
- *addr*
  - pointer to store client address, (struct sockaddr_in *) cast to (stru ct sockaddr *)
- *addrlen*
  - pointer to store the returned size of addr, should be sizeof(*addr)
- For example
  - *int isock=accept(sockfd, (struct sockaddr_in *) &caddr, &clen);*

Put Server Together

```
struct sockaddr_in saddr, caddr;
int sockfd, clen, isock;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) { // from back a couple slides
printf("Error creating socket\n");
...
}

memset(&saddr, '\0', sizeof(saddr));          // zero structure out
saddr.sin_family = AF_INET;                   // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY);    // bind to any local address
saddr.sin_port = htons(port);                 // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
printf("Error binding\n");
...
}

if(listen(sockfd, 5) < 0) {          // listen for incoming connections
printf("Error listening\n");
...
}

clen=sizeof(caddr)
if((isock=accept(sockfd, (struct sockaddr *) &caddr, &clen)) < 0) {   // accept one
printf("Error accepting\n");
...
}
```

What about client?

- Client need not bind, listen, and accept
- **All client need to do is to connect**
  - *int connect(int sockfd, const struct sockaddr *saddr, socklen_t addrlen);*
- For example,
  - *connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr));*

We Are Connected

- Server accepting connections and client connecting to servers
- Send and receive data
  - *ssize_t read(int fd, void *buf, size_t len);*
  - *ssize_t write(int fd, const void *buf, size_t len);*
- For example,
  - *read(sockfd, buffer, sizeof(buffer));*
  - *write(sockfd, “hey\n”, strlen(“hey\n”));*

Close the Socket

- Don’t forget to close the socket descriptor, like a file
  - *int close(int sockfd);*

- Now server can loop around and accept a new connection when the old one finishes

Functions in detail

The bind function

- associates and (can exclusively) reserves a port for use by the socket
- int status = bind(sockid, &addrport, size);
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
  - size: the size (in bytes) of the addrport structure
- bind can be skipped for both types of sockets. When and why?

Skipping the bind

- SOCK_DGRAM:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
  - if receiving, need to bind
- SOCK_STREAM:
  - destination determined during conn. setup
  - don’t need to know port sending from (during connection setup, receiving end is informed of port)

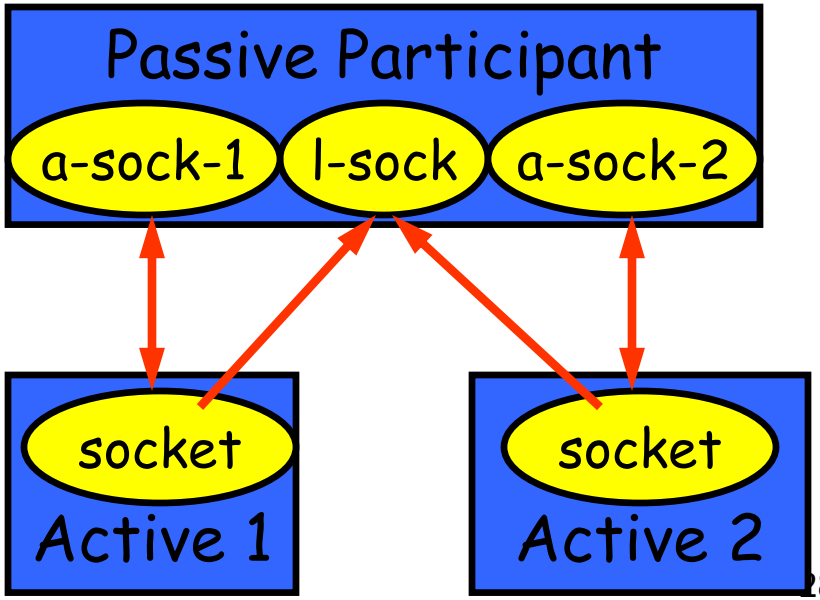
Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are “similar”
  - both can send & receive data
  - either can terminate the connection

Connection setup cont'd

- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants

- Active participant
  - step 2: request & establish connection
  - step 4: data transfer



Connection setup: listen & accept

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can “wait” for a connection
  - listen is **non-blocking**: returns immediately
- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

connect call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- connect is **blocking**

Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

Sending / Receiving Data (cont'd)

- Without a connection (SOCK_DGRAM):
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]

close

- When finished using a socket, the socket should be closed:
- status = close(s);
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

The struct sockaddr

- The generic:

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused