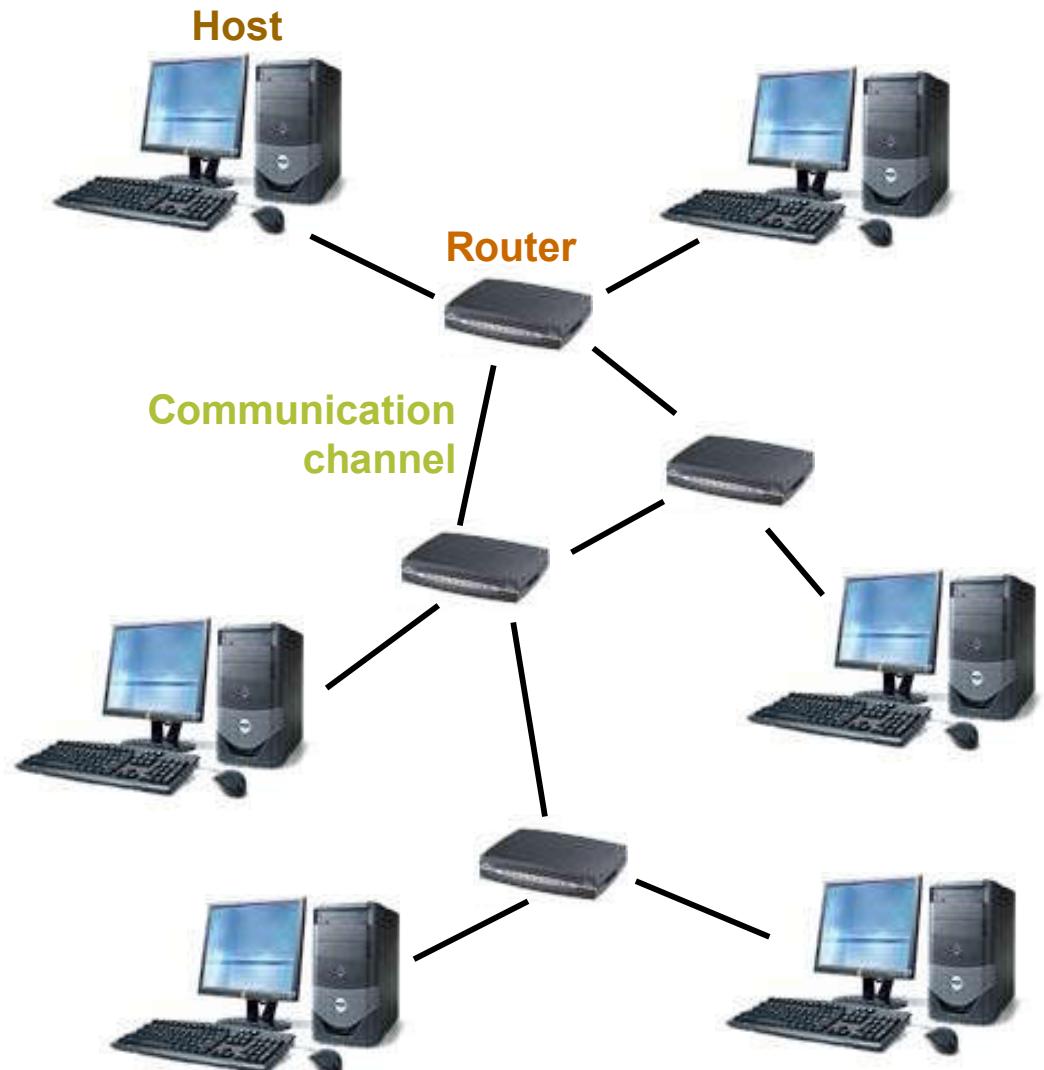


# Introduction to Sockets Programming in C using TCP/IP

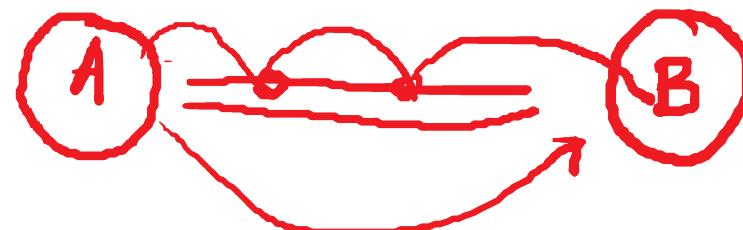
# Introduction

- 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)



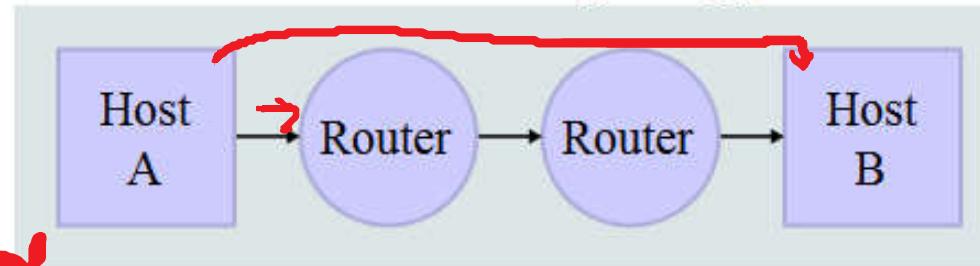
# Protocol Families - TCP/IP

- Several protocols for different problems
  - ☞ Protocol Suites or Protocol Families: TCP/IP
- 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**



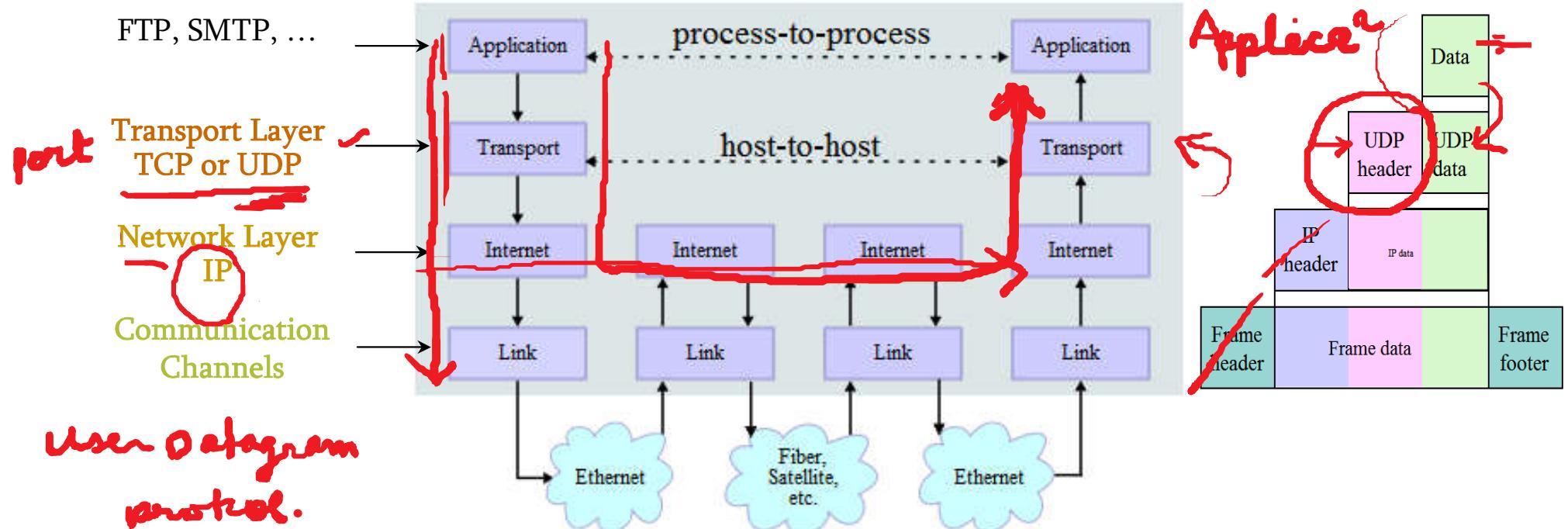
# TCP/IP

## Network Topology \*



*Transmission control protocol.*

Gmail → 0101111  
Data Flow



\* image is taken from "[http://en.wikipedia.org/wiki/TCP/IP\\_model](http://en.wikipedia.org/wiki/TCP/IP_model)"

# Internet Protocol (IP)

- provides a **datagram** service
  - packets are handled and delivered independently
- **best-effort** protocol
  - may loose, reorder or duplicate packets
- each packet must contain an IP address of its destination

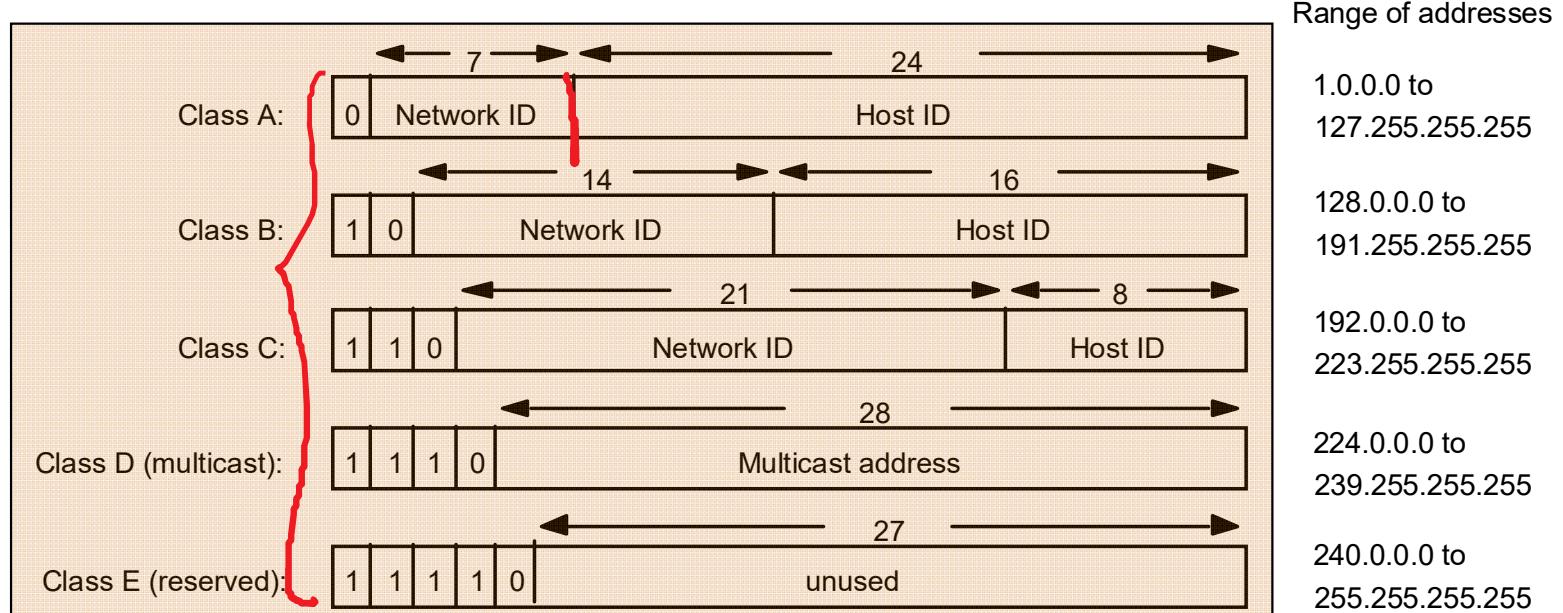
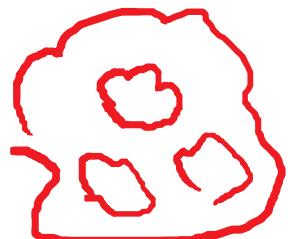


# Addresses - IPv4

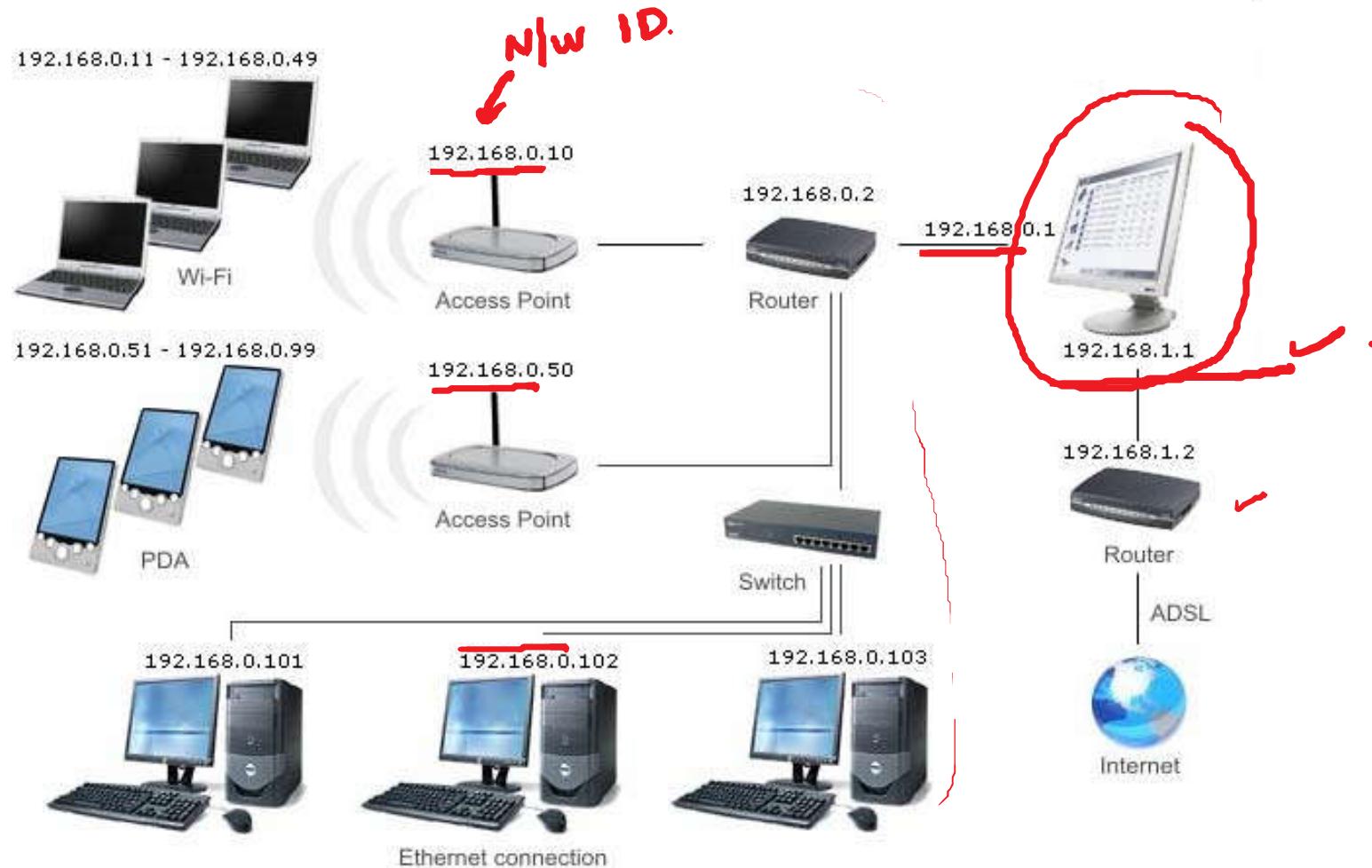
$0 \sim 2^8 - 1$

- The 32 bits of an IPv4 address are broken into **4 octets**, or 8 bit fields (0-255 value in decimal notation).
- For networks of different size,
  - the first one (for large networks) to three (for small networks) octets can be used to identify the **network**, while
  - the rest of the octets can be used to identify the **node** on the network.

192. 168 · 1 · 2 → 3

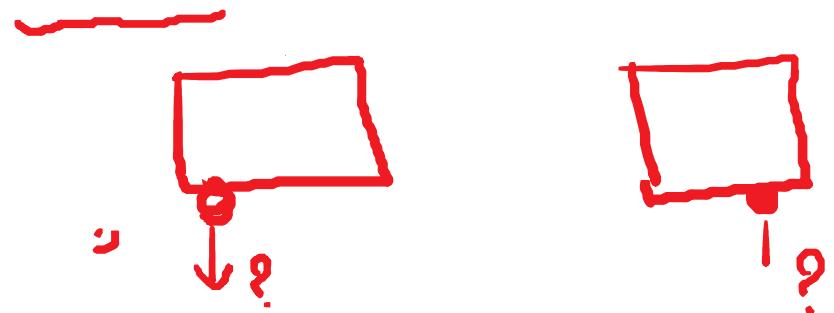
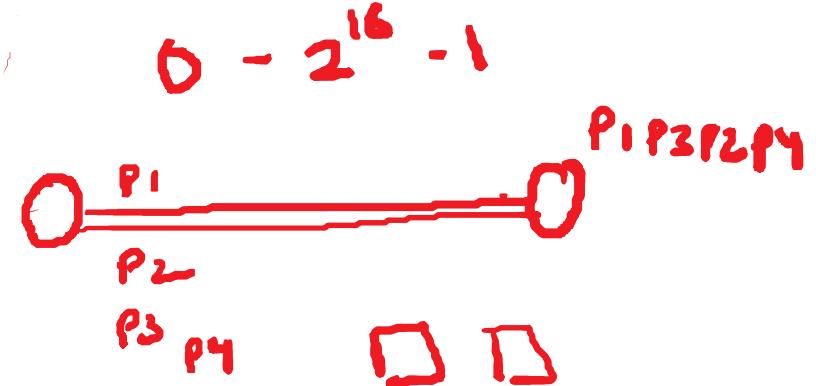


# Local Area Network Addresses - IPv4



# TCP vs UDP - Transport layer

- Both use port numbers
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to  $2^{16} - 1$
  - 👉 to provide end-to-end transport
- UDP: User Datagram Protocol
  - no acknowledgements ✓
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - ✗ □ reliable byte-stream channel (in order, all arrive, no duplicates)
    - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional



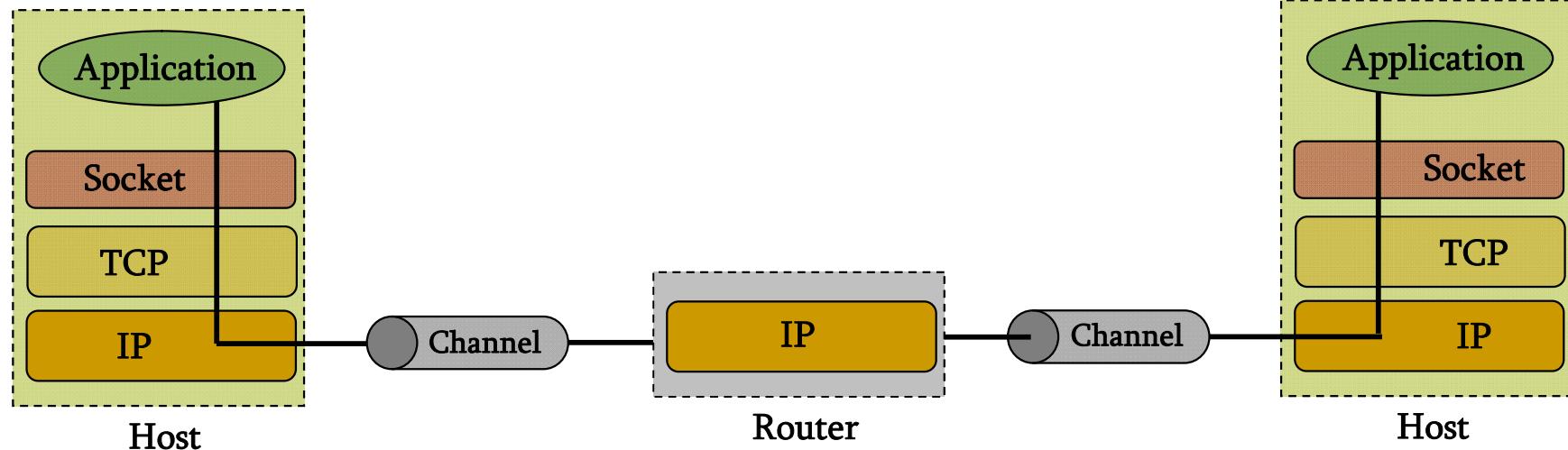
# TCP vs UDP

- ✓TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-reply actions.
- Some common examples of TCP and UDP with their default ports:

	well known ports
DNS lookup	0 - 1023
FTP	0 - 65535
HTTP	
POP3	
Telnet	
UDP	53
TCP	21
TCP	80
TCP	110
TCP	23

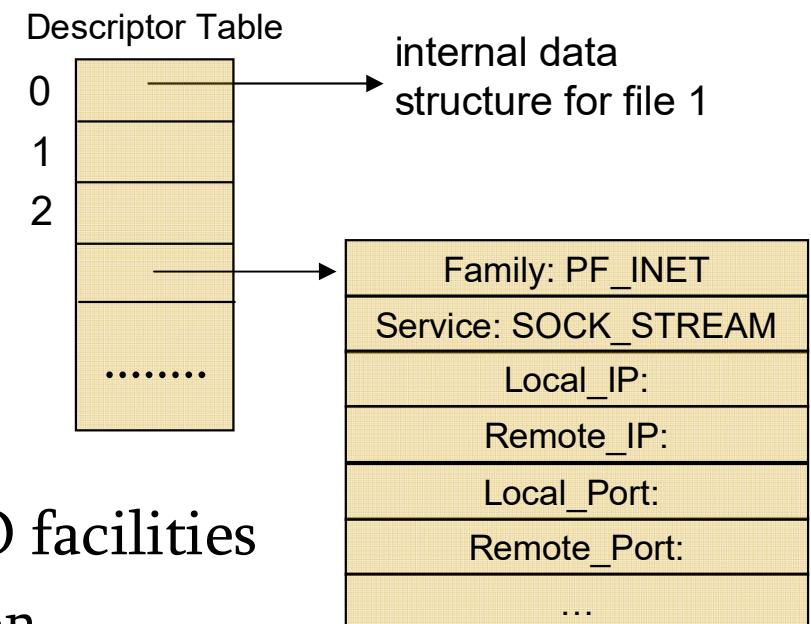
# Berkley Sockets

- Universally known as **Sockets**
- It is an abstraction through which an application may send and receive data
- Provide **generic access** to interprocess communication services
  - e.g. IPX/SPX, Appletalk, TCP/IP
- Standard API for networking

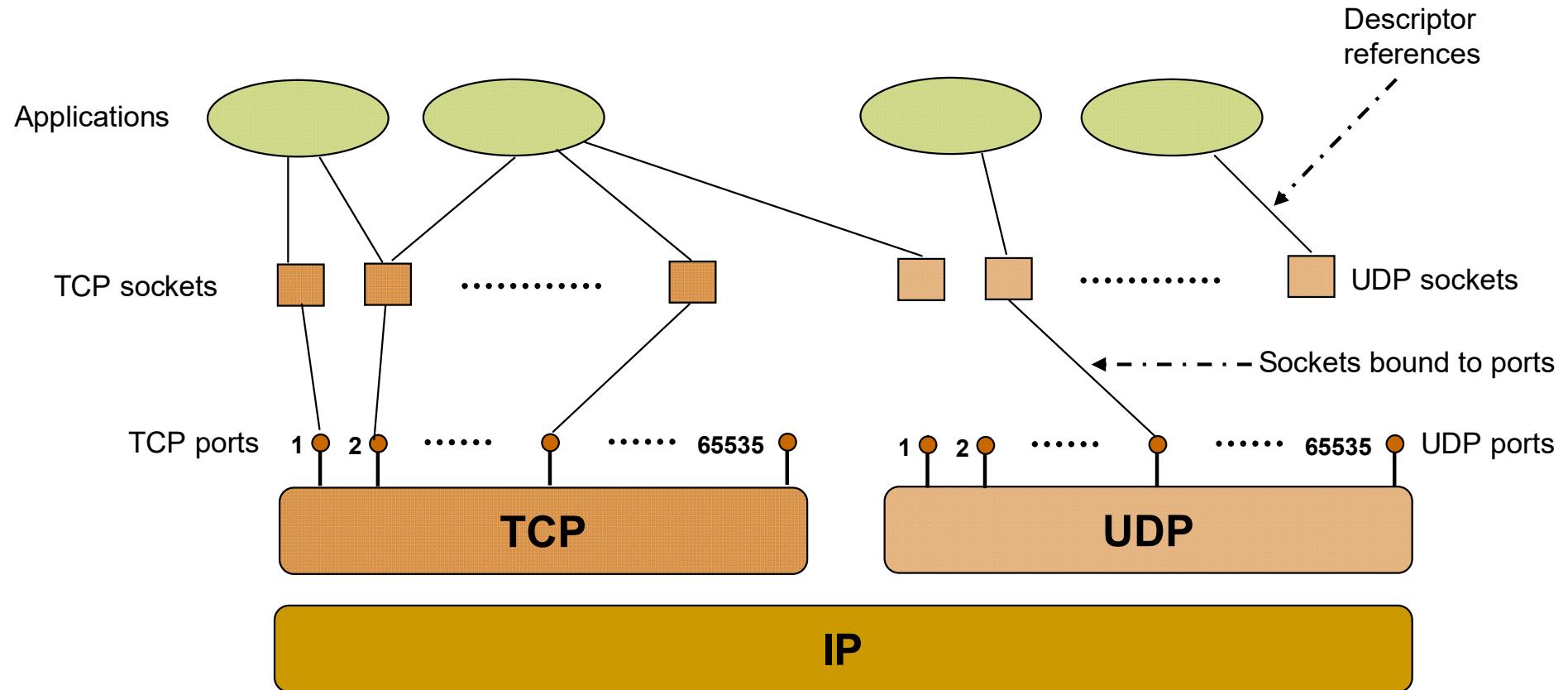


# Sockets

- Uniquely identified by
  - an internet address  
*192.168.1.1 : 80*
  - an end-to-end protocol (e.g. TCP or UDP)
  - a port number
- Two types of (TCP/IP) sockets
  - **Stream** sockets (e.g. uses TCP)
    - provide reliable byte-stream service
  - **Datagram** sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- Sockets extend the conventional UNIX I/O facilities
  - file descriptors for network communication
  - extended the read and write system calls



# Sockets





# Client-Server communication

## ■ Server

- passively waits for and responds to clients
- **passive** socket

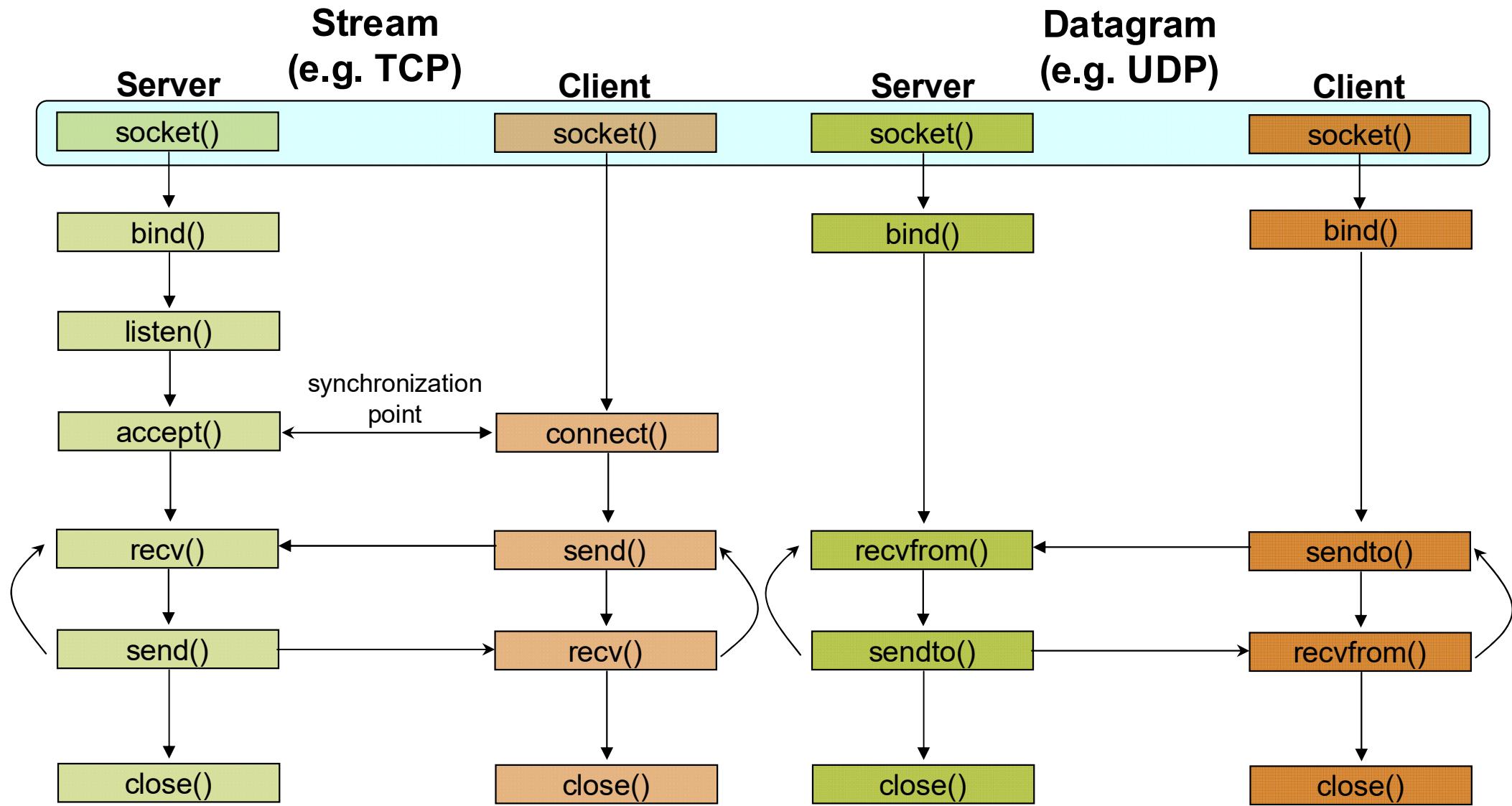
## ■ Client

- initiates the communication
- must know the address and the port of the server
- **active** socket

# Sockets - Procedures

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

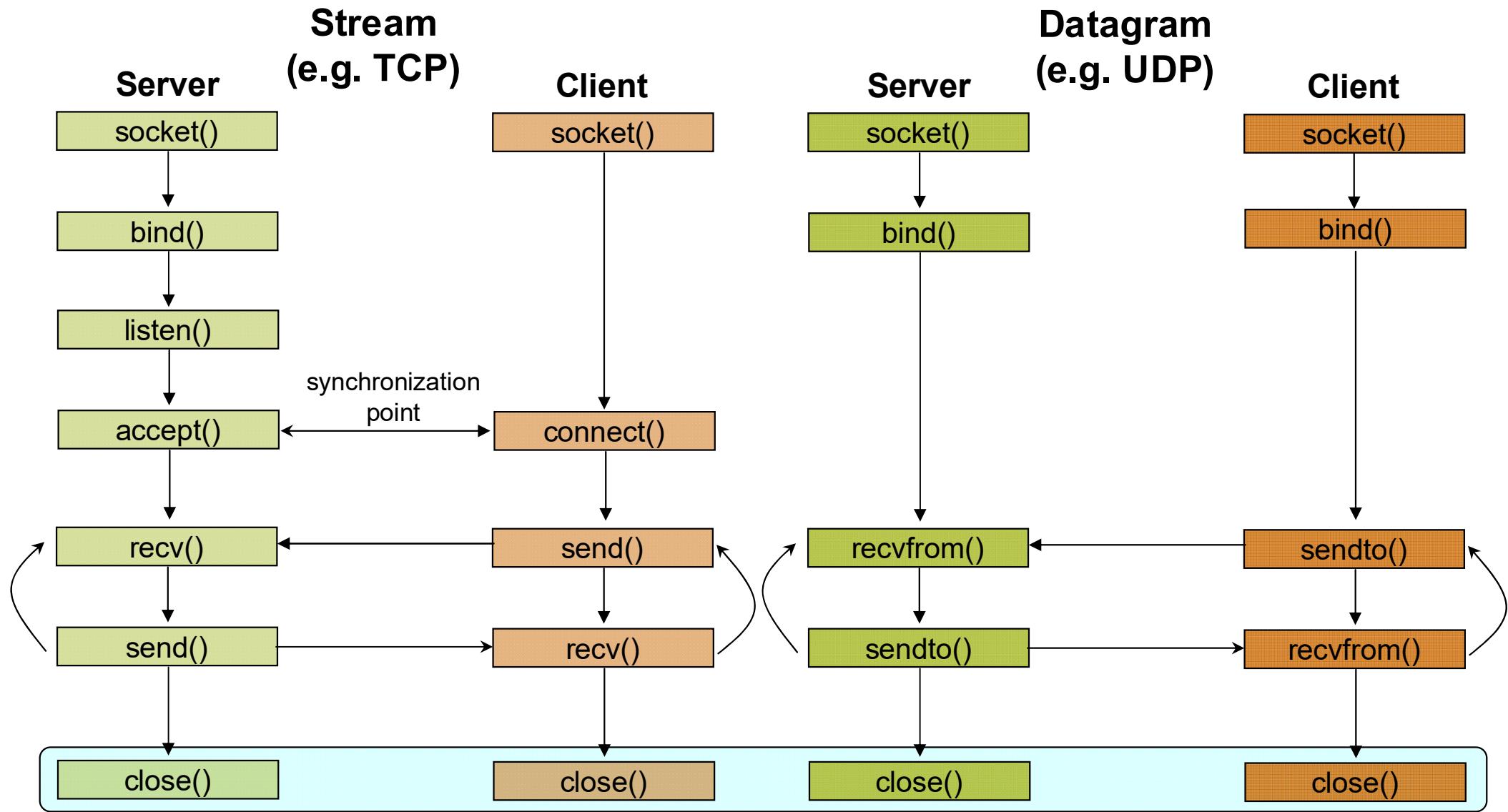
# Client - Server Communication



# Socket creation in C: `socket()`

- `int sockid = socket(family, type, protocol);`
  - `sockid`: socket descriptor, an integer (like a file-handle)
  - `family`: integer, communication domain, e.g.,
    - PF\_INET, IPv4 protocols, Internet addresses (typically used)
    - PF\_UNIX, Local communication, File addresses
  - `type`: communication type
    - SOCK\_STREAM - reliable, 2-way, connection-based service
    - SOCK\_DGRAM - unreliable, connectionless, messages of maximum length
  - `protocol`: specifies protocol
    - IPPROTO\_TCP IPPROTO\_UDP
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1
- ☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

# Client - Server Communication



# Socket close in C: `close()`

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

# Specifying Addresses

- Socket API defines a **generic** data type for addresses:

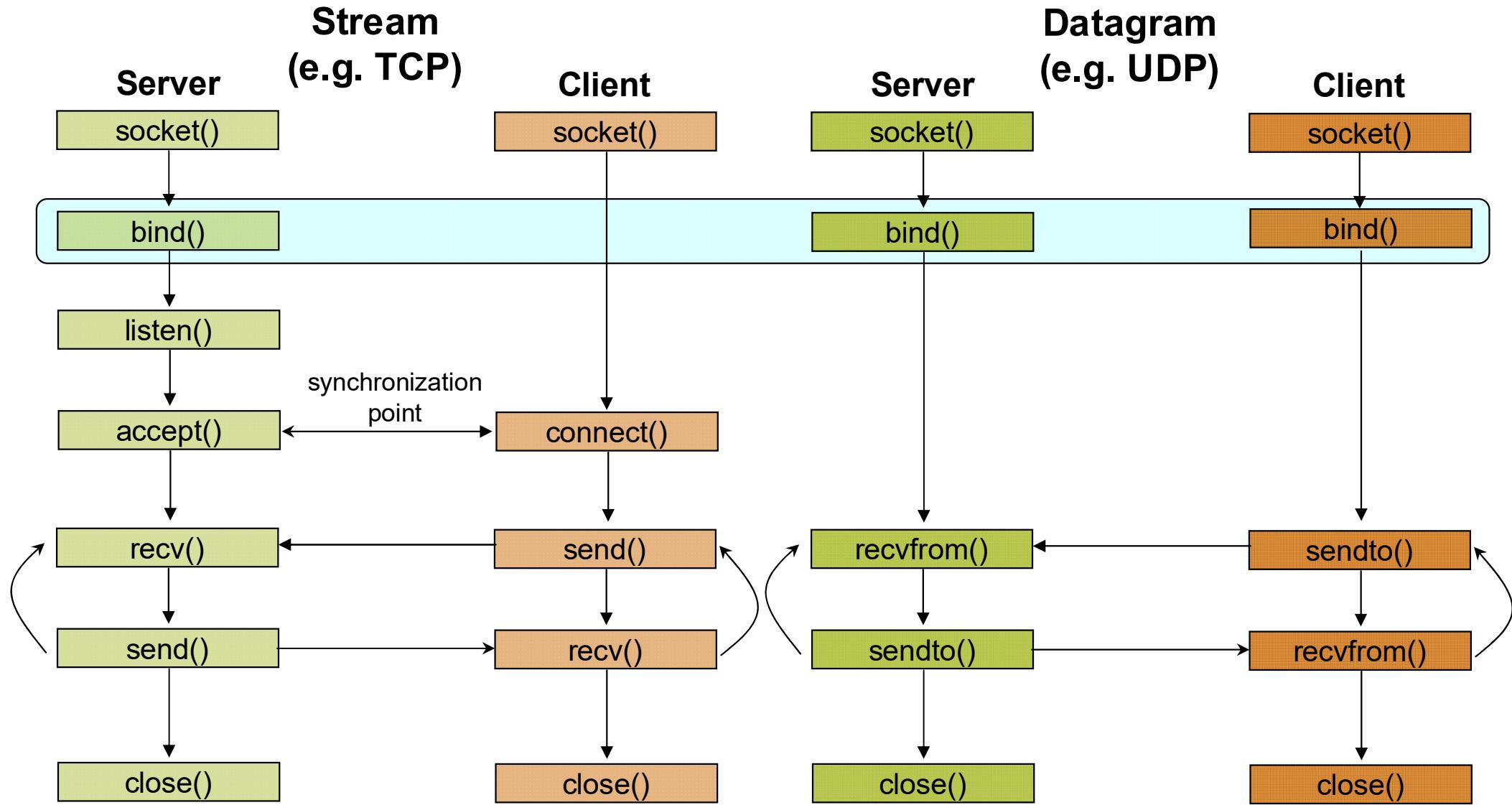
```
struct sockaddr {  
    unsigned short sa_family; /* Address family (e.g. AF_INET) */  
    char sa_data[14];          /* Family-specific address information */  
}
```

- Particular form of the sockaddr used for **TCP/IP** addresses:

```
struct in_addr {  
    unsigned long s_addr;           /* Internet address (32 bits) */  
}  
  
struct sockaddr_in {  
    unsigned short sin_family;      /* Internet protocol (AF_INET) */  
    unsigned short sin_port;         /* Address port (16 bits) */  
    struct in_addr sin_addr;        /* Internet address (32 bits) */  
    char sin_zero[8];               /* Not used */  
}
```

👉 Important: sockaddr\_in can be casted to a sockaddr

# Client - Server Communication



# Assign address to socket: bind()

- associates and reserves a port for use by the socket
- `int status = bind(sockid, &addrport, size);`
  - **sockid**: integer, socket descriptor
  - **addrport**: struct sockaddr, the (IP) address and port of the machine
    - for TCP/IP server, internet address is usually set to INADDR\_ANY, i.e., chooses any incoming interface
  - **size**: the size (in bytes) of the addrport structure
  - **status**: upon failure -1 is returned

# bind () - Example with TCP

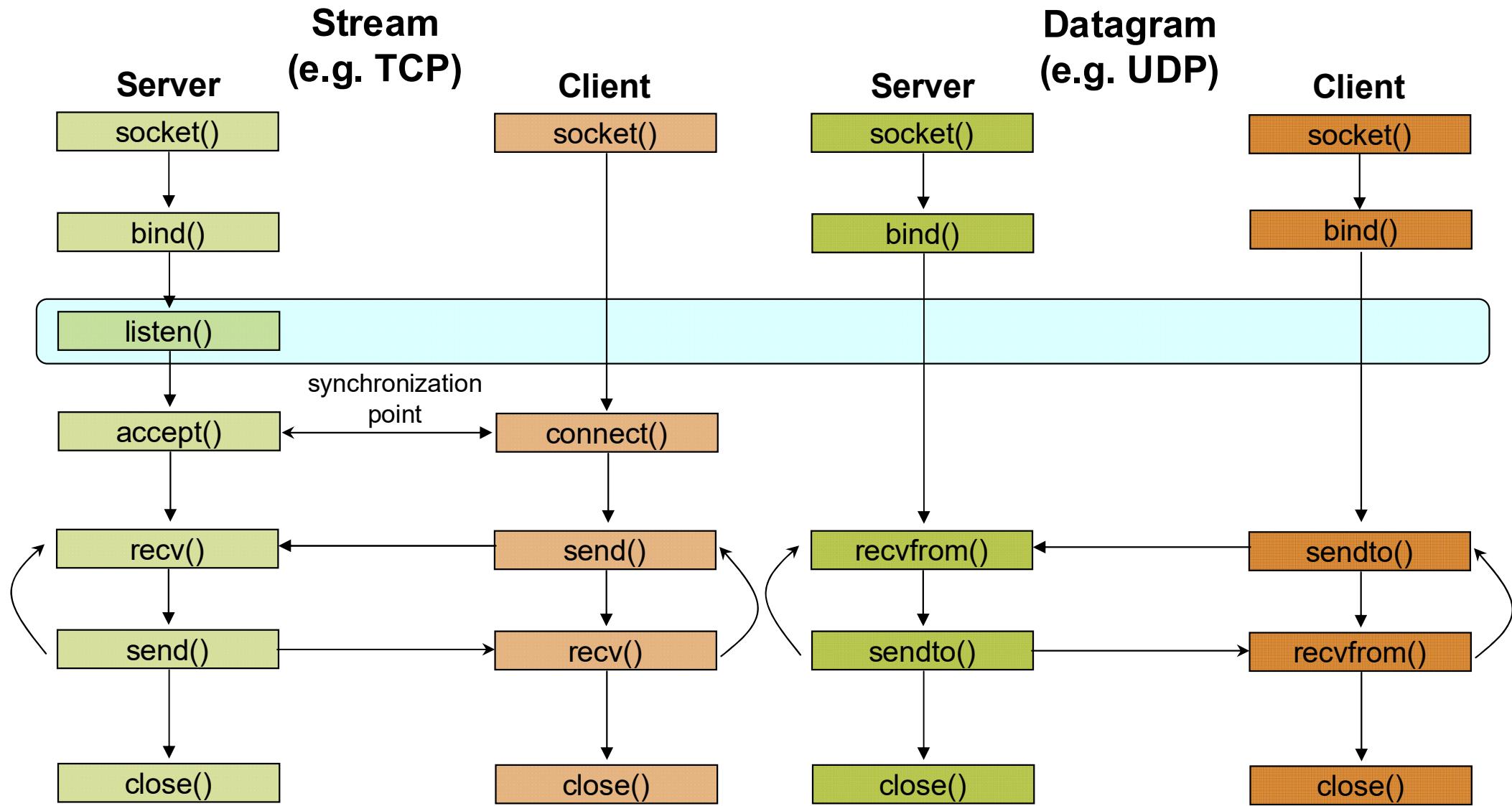
```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *)&addrport, sizeof(addrport)) != -1) {
    ...
}
```

# Skipping the bind ()

- bind can be skipped for both types of sockets
- Datagram socket:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a packet
  - if receiving, need to bind
- Stream socket:
  - destination determined during connection setup
  - don't need to know port sending from (during connection setup, receiving end is informed of port)

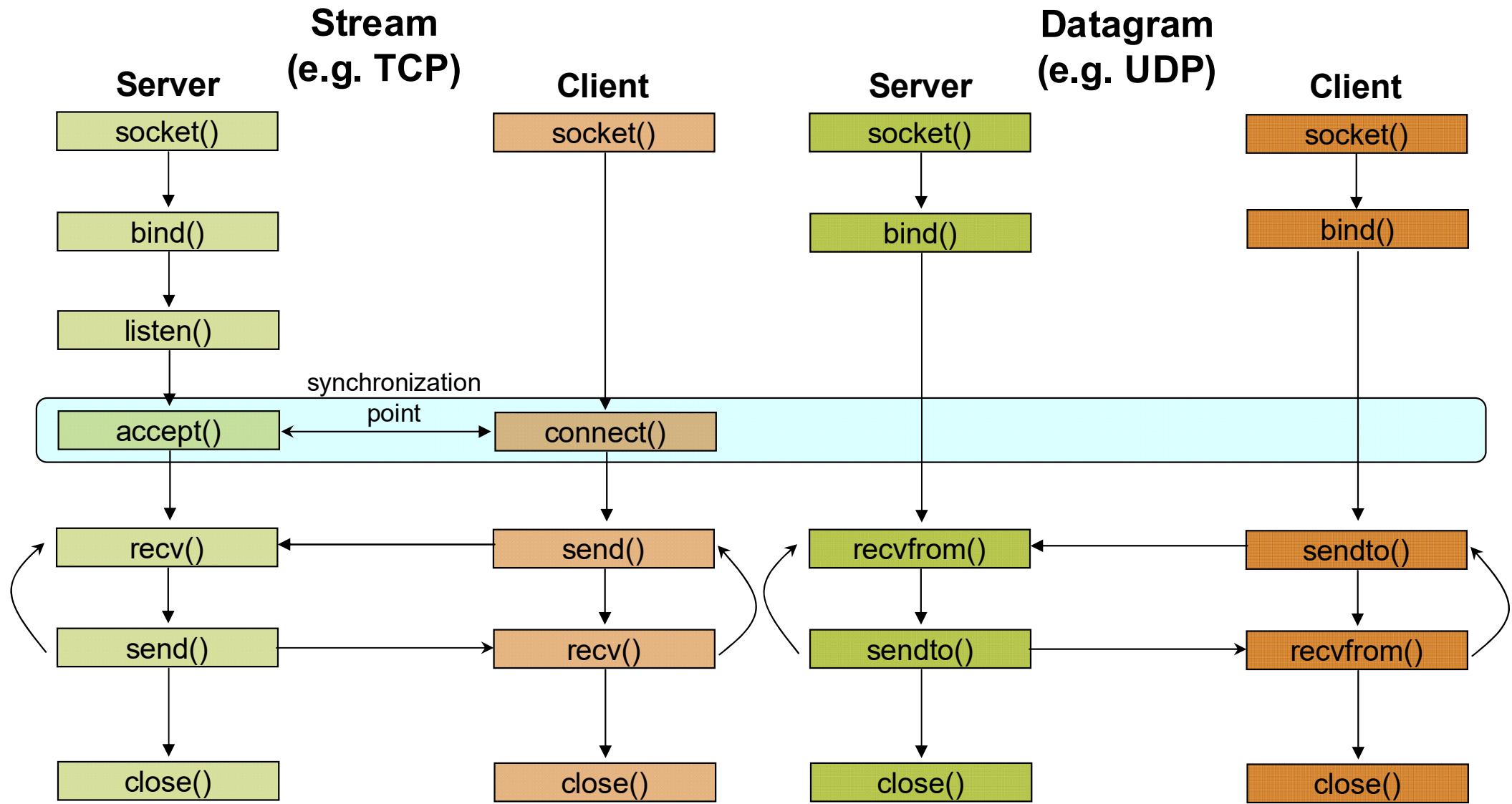
# Client - Server Communication - Unix



# Assign address to socket: bind()

- Instructs TCP protocol implementation to listen for connections
- `int status = listen(sockid, queueLimit);`
  - `sockid`: integer, socket descriptor
  - `queueLen`: integer, # of active participants that can “wait” for a connection
  - `status`: 0 if listening, -1 if error
- `listen()` is **non-blocking**: returns immediately
- The listening socket (`sockid`)
  - is never used for sending and receiving
  - is used by the server only as a way to get new sockets

# Client - Server Communication



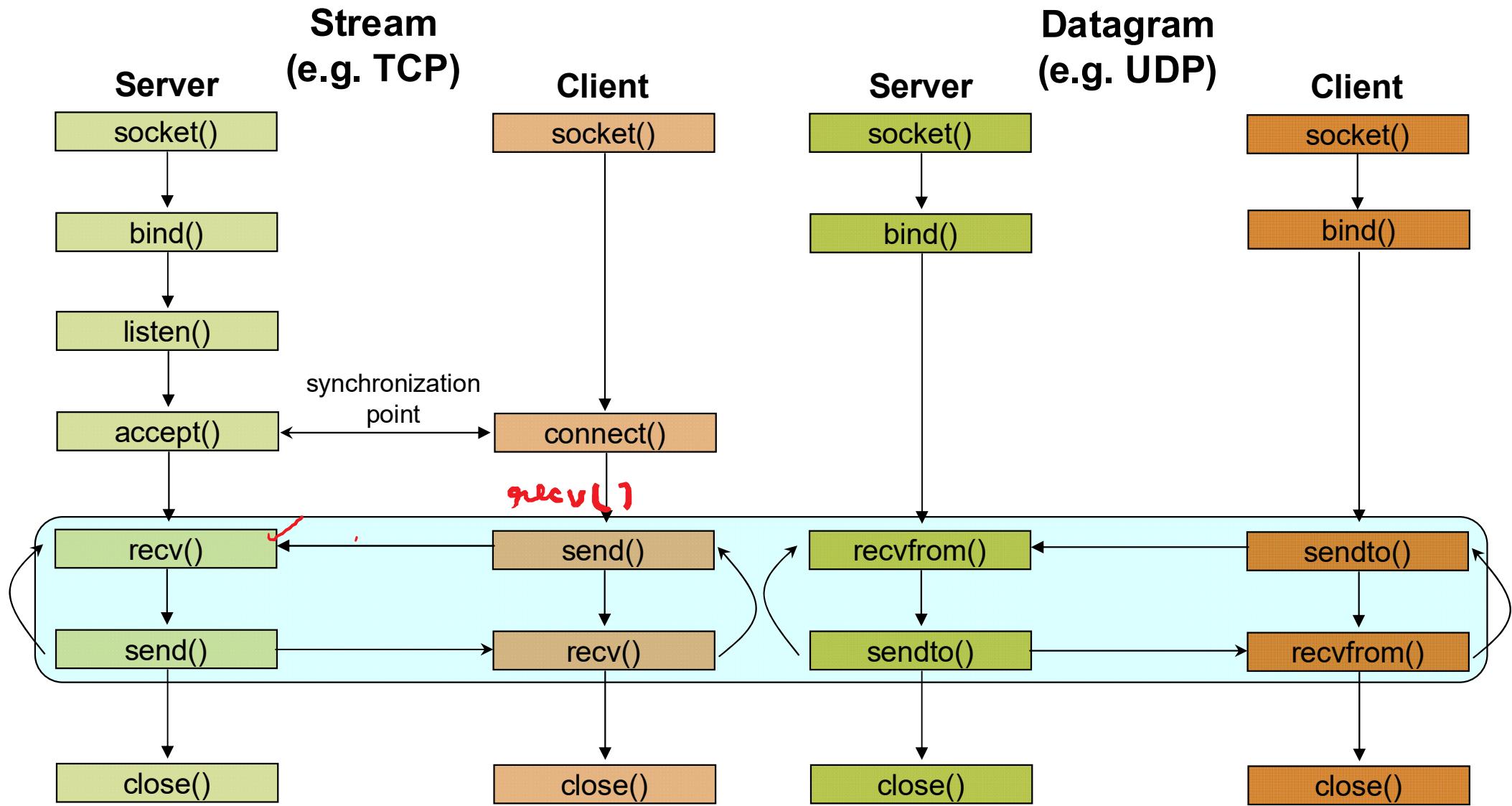
# Establish Connection: connect ()

- The client establishes a connection with the server by calling connect ()
- `int status = connect(sockid, &foreignAddr, addrlen);`
  - `sockid`: integer, socket to be used in connection
  - `foreignAddr`: struct sockaddr: address of the passive participant
  - `addrlen`: integer, sizeof(name)
  - `status`: 0 if successful connect, -1 otherwise
- connect () is **blocking**

# Incoming Connection: accept ()

- The server gets a socket for an incoming client connection by calling accept ()
- `int s = accept(sockid, &clientAddr, &addrLen);`
  - `s`: integer, the new socket (used for data-transfer)
  - `sockid`: integer, the orig. socket (being listened on)
  - `clientAddr`: struct sockaddr, address of the active participant
    - filled in upon return
  - `addrLen`: sizeof(clientAddr): value/result parameter
    - must be set appropriately before call
    - adjusted upon return
- accept ()
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

# Client - Server Communication



# Exchanging data with stream socket

- **int count = send(sockid, msg, msgLen, flags);**
  - **msg**: const void[], message to be transmitted
  - **msgLen**: integer, length of message (in bytes) to transmit
  - **flags**: integer, special options, usually just 0
  - **count**: # bytes transmitted (-1 if error)
- **int count = recv(sockid, recvBuf, bufLen, flags);**
  - **recvBuf**: void[], stores received bytes
  - **bufLen**: # bytes received
  - **flags**: integer, special options, usually just 0
  - **count**: # bytes received (-1 if error)
- Calls are **blocking**
  - returns only after data is sent / received

# Exchanging data with datagram socket

- `int count = sendto(sockid, msg, msgLen, flags, &foreignAddr, addrlen);`
  - `msg`, `msgLen`, `flags`, `count`: same with `send()`
  - `foreignAddr`: struct `sockaddr`, address of the destination
  - `addrLen`: `sizeof(foreignAddr)`
- `int count = recvfrom(sockid, recvBuf, bufLen, flags, &clientAddr, addrlen);`
  - `recvBuf`, `bufLen`, `flags`, `count`: same with `recv()`
  - `clientAddr`: struct `sockaddr`, address of the client
  - `addrLen`: `sizeof(clientAddr)`
- Calls are **blocking**
  - returns only after data is sent / received

# Example - Echo

- A client communicates with an “echo” server
- The server simply echoes whatever it receives back to the client

# Example - Echo using stream socket

The server starts by getting ready to receive client connections...

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
/* Create socket for incoming connections */  
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)  
    DieWithError("socket() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. **Create a TCP socket**
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
echoServAddr.sin_family = AF_INET;           /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);    /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

## Client

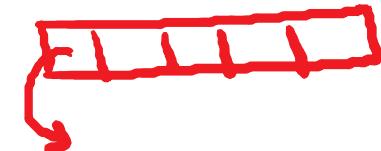
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. **Assign a port to socket**
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
/* Mark the socket so it will listen for incoming connections */  
if (listen(servSock, MAXPENDING) < 0)  
    DieWithError("listen() failed");
```



## Client

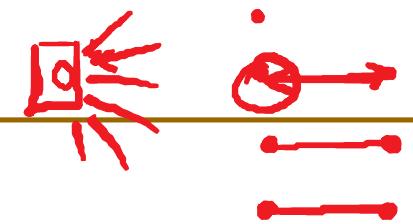
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. **Set socket to listen**
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
for (;;) /* Run forever */  
{  
    clntLen = sizeof(echoClntAddr);  
  
    if ((clientSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen))<0)  
        DieWithError("accept() failed");  
    ...  
}
```



## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

Server is now blocked waiting for connection from a client

...

A client decides to talk to the server

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection



## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. **(Accept new connection)**
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
/* Create a reliable, stream socket using TCP */  
if ((clientSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)  
    DieWithError("socket() failed");
```



## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

inetnet/in.h

```
echoServAddr.sin_family = AF_INET;           /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(echoservIP); /* Server IP address*/
echoServAddr.sin_port = htons(echoServPort);    /* Server port */

if (connect(clientSock, (struct sockaddr *) &echoServAddr,
            sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

## Client ✓

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

Server's accept procedure is now unblocked and returns client's socket

```
for (;;) /* Run forever */  
{  
    clntLen = sizeof(echoClntAddr);  
  
    if ((clientSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen))<0)  
        DieWithError("accept() failed");  
    ...  
}
```

## Client

1. Create a TCP socket
2. **Establish connection**
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
echoStringLen = strlen(echoString); /* Determine input length */  
  
/* Send the string to the server */  
if (send(clientSock, echoString, echoStringLen, 0) != echoStringLen)  
    DieWithError("send() sent a different number of bytes than expected");
```

## Client

1. Create a TCP socket ✓
2. Establish connection
3. **Communicate**
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. Communicate
  - c. Close the connection

# Example - Echo using stream socket

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
/* Send received string and receive again until end of transmission */
while (recvMsgSize > 0) { /* zero indicates end of transmission */
    if (send(clientSocket, echobuffer, recvMsgSize, 0) != recvMsgSize)
        DieWithError("send() failed");
    if ((recvMsgSize = recv(clientSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
        DieWithError("recv() failed");
}
```

## Client

1. Create a TCP socket
2. Establish connection
3. **Communicate**
4. Close the connection

## Server

- 
1. Create a TCP socket
  2. Assign a port to socket
  3. Set socket to listen
  4. Repeatedly:
    - a. ✓ Accept new connection
    - b. **Communicate**
    - c. Close the connection

# Example - Echo using stream socket

Similarly, the client receives the data from the server

## Client

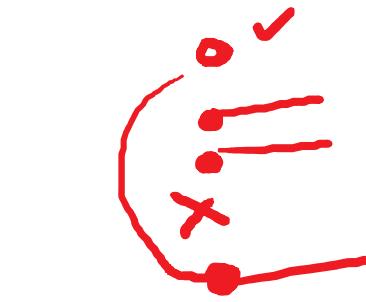
1. Create a TCP socket
2. Establish connection
3. **Communicate**
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. **Communicate**
  - c. Close the connection

# Example - Echo using stream socket

close(clientSock);



close(clientSock);

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. **Close the connection**

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. **Close the connection**

# Example - Echo using stream socket

Server is now blocked waiting for connection from a client

2

Client

1. Create a TCP socket
  2. Establish connection
  3. Communicate
  4. Close the connection

 Server

1. Create a TCP socket
  2. Assign a port to socket
  3. Set socket to listen
  4. Repeatedly:
    - a. **Accept new connection**
    - b. Communicate
    - c. Close the connection

# Example - Echo using datagram socket

```
/* Create socket for sending/receiving datagrams */  
if ((servSock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)  
    DieWithError("socket() failed");
```

```
/* Create a datagram/UDP socket */  
if ((clientSock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)  
    DieWithError("socket() failed");
```

## Client

1. **Create a UDP socket**
2. Assign a port to socket
3. Communicate
4. Close the socket

## Server

1. **Create a UDP socket**
2. Assign a port to socket
3. Repeatedly
  - Communicate

# Example - Echo using datagram socket

```
echoServAddr.sin_family = AF_INET; /* Internet address family */  
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */  
echoServAddr.sin_port = htons(echoServPort); /* Local port */  
  
if (bind(servSock, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr)) < 0)  
    DieWithError("bind() failed");
```

```
echoClientAddr.sin_family = AF_INET; /* Internet address family */  
echoClientAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */  
echoClientAddr.sin_port = htons(echoClientPort); /* Local port */  
  
if(bind(clientSock, (struct sockaddr *)&echoClientAddr, sizeof(echoClientAddr))<0)  
    DieWithError("connect() failed");
```

## Client

1. Create a UDP socket
2. **Assign a port to socket**
3. Communicate
4. Close the socket

## Server

1. Create a UDP socket
2. **Assign a port to socket**
3. Repeatedly
  - Communicate

# Example - Echo using datagram socket

```
echoServAddr.sin_family = AF_INET;           192.168.1.2 /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(echoservIP); /* Server IP address*/
echoServAddr.sin_port = htons(echoServPort);    8080 /* Server port */

echoStringLen = strlen(echoString); /* Determine input length */

/* Send the string to the server */
if (sendto( clientSock, echoString, echoStringLen, 0,
            (struct sockaddr *) &echoServAddr, sizeof(echoServAddr))
    != echoStringLen)
    DieWithError("send() sent a different number of bytes than expected");
```

## Client

1. Create a UDP socket
2. Assign a port to socket
3. **Communicate**
4. Close the socket

## Server

1. Create a UDP socket
2. **Assign a port to socket**
3. Repeatedly
  - Communicate

# Example - Echo using datagram socket

```
for (;;) /* Run forever */  
{  
    clientAddrLen = sizeof(echoClientAddr) /* Set the size of the in-out parameter */  
    /*Block until receive message from client*/  
    if ((recvMsgSize = recvfrom(servSock, echoBuffer, ECHOMAX, 0),  
        (struct sockaddr *)&echoClientAddr, sizeof(echoClientAddr))) < 0)  
        DieWithError("recvfrom() failed");  
  
    if (sendto(servSock, echobuffer, recvMsgSize, 0,  
               (struct sockaddr *)&echoClientAddr, sizeof(echoClientAddr))  
        != recvMsgSize)  
        DieWithError("send() failed");  
}
```



## Client

1. Create a UDP socket
2. Assign a port to socket
3. **Communicate**
4. Close the socket

## Server

1. Create a UDP socket
2. Assign a port to socket
3. Repeatedly
  - **Communicate**

# Example - Echo using datagram socket

Similarly, the client receives the data from the server

## Client

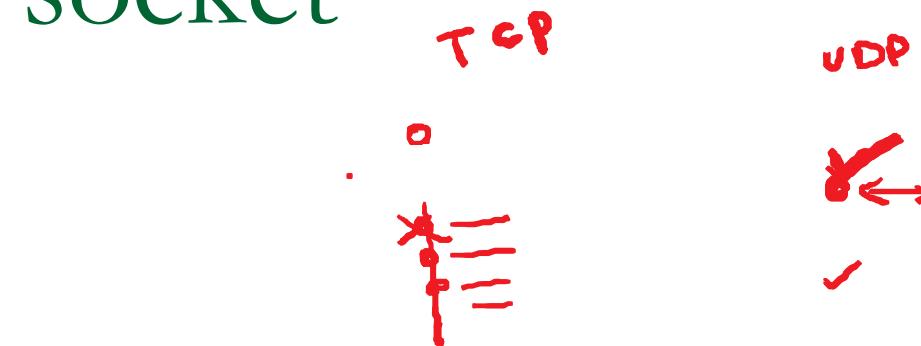
1. Create a UDP socket
2. Assign a port to socket
3. **Communicate**
4. Close the socket

## Server

1. Create a UDP socket
2. Assign a port to socket
3. Repeatedly
  - **Communicate**

# Example - Echo using datagram socket

`close(clientSock);`



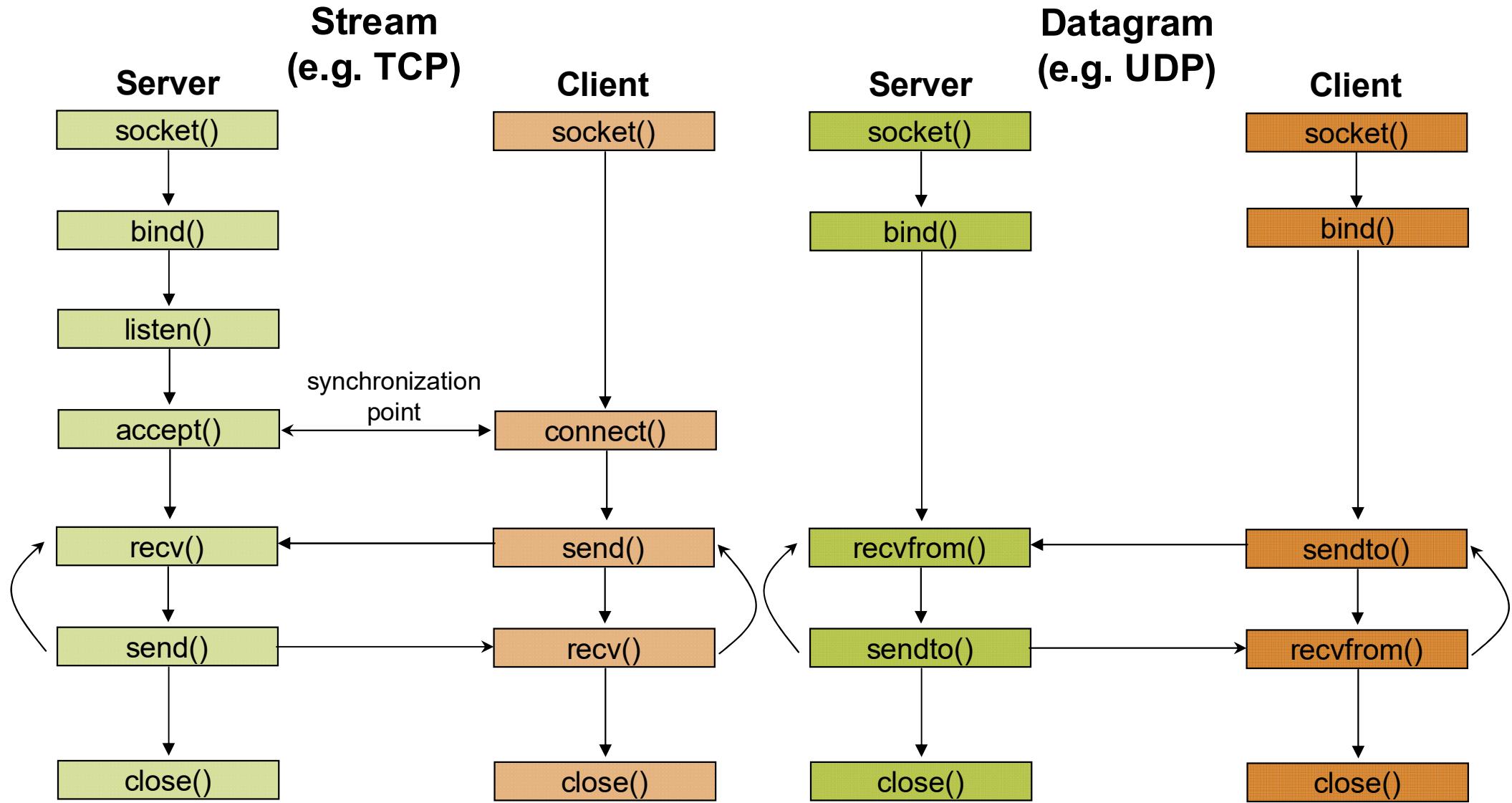
## Client

1. Create a UDP socket
2. Assign a port to socket
3. Communicate
4. **Close the socket**

## Server

1. Create a UDP socket
2. Assign a port to socket
3. **Repeatedly**
  - Communicate

# Client - Server Communication



# Constructing Messages - Encoding Data

- Client wants to send two integers x and y to server
- 1<sup>st</sup> Solution: **Character Encoding**
  - e.g. ASCII
    - 拇指 up the same representation is used to print or display them to screen
    - 拇指 up allows sending arbitrarily large numbers (at least in principle)
- e.g. x = 17,998,720 and y = 47,034,615

49	55	57	57	56	55	50	48	32	52	55	48	51	52	54	49	53	32
1	7	9	9	8	7	2	0	-	4	7	0	3	4	6	1	5	-

```
sprintf(msgBuffer, "%d %d ", x, y);
send(clientSocket, strlen(msgBuffer), 0);
```

# Constructing Messages - Encoding Data

## ■ Pitfalls

- the second delimiter is required
  - otherwise the server will not be able to separate it from whatever it follows
- msgBuffer must be large enough
- strlen counts only the bytes of the message
  - not the null at the end of the string

👎 This solution is not efficient

ஓஓஓ

ஓஓஓ

- 👎 each digit can be represented using 4 bits, instead of one byte
- 👎 it is inconvenient to manipulate numbers

## ■ 2<sup>nd</sup> Solution: **Sending the values** of x and y

# Constructing Messages - Encoding Data

- 2<sup>nd</sup> Solution: **Sending the values** of x and y
  - pitfall: native integer format
  - 👉 a protocol is used
    - how many bits are used for each integer
    - what type of encoding is used (e.g. two's complement, sign/magnitude, unsigned)

## 1<sup>st</sup> Implementation

```
typedef struct {  
    int x,y;  
} msgStruct;  
...  
msgStruct.x = x;  msgStruct.y = y;  
send(clientSock, &msgStruct, sizeof(msgStruct), 0);
```

## 2<sup>nd</sup> Implementation

```
send(clientSock, &x, sizeof(x)), 0);  
send(clientSock, &y, sizeof(y)), 0);
```

**2<sup>nd</sup> implementation  
works in any case?**

# Constructing Messages - Byte Ordering

- Address and port are stored as integers

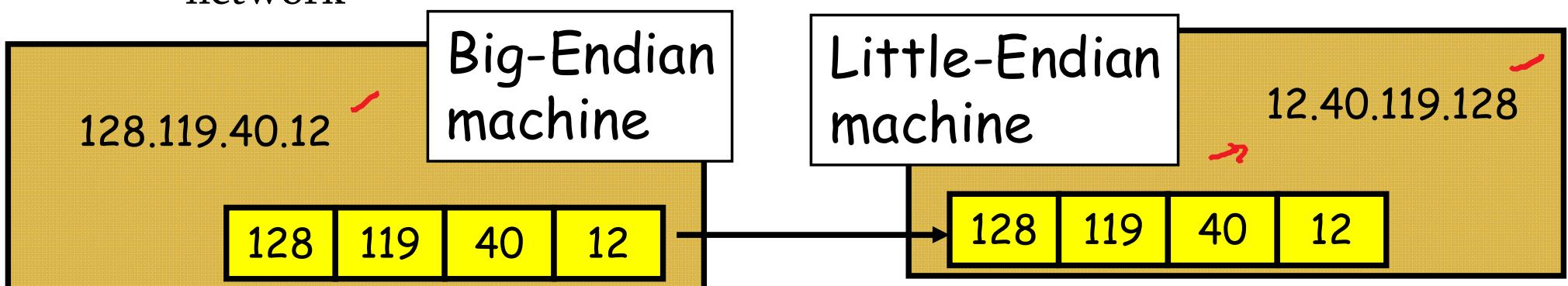
- `u_short sin_port;` (16 bit) ✓
  - `in_addr sin_addr;` (32 bit) ✓



- Problem:

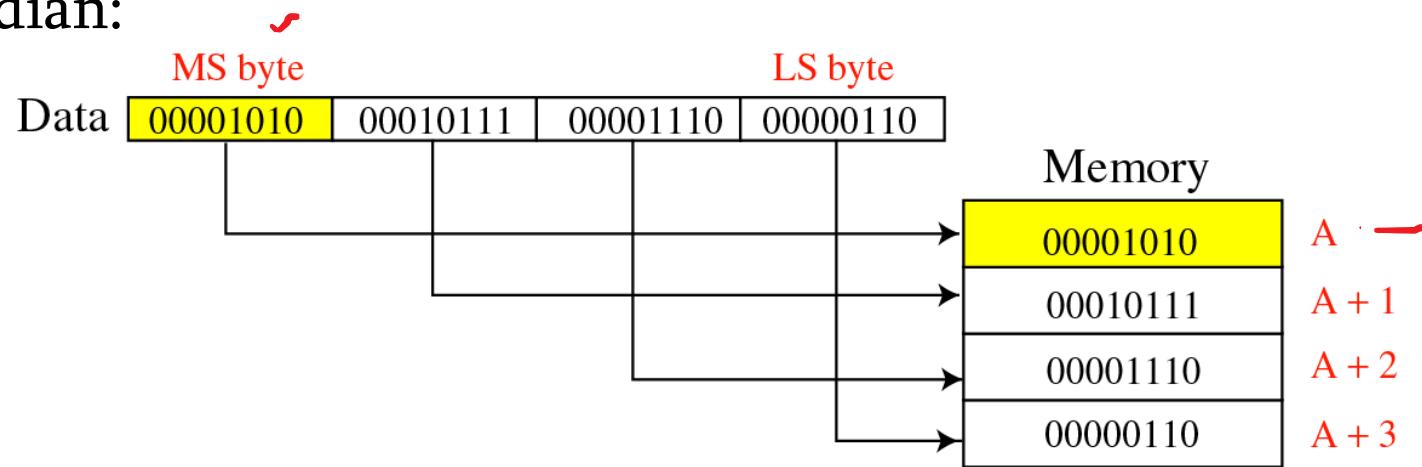
- different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network

128.119.40.12

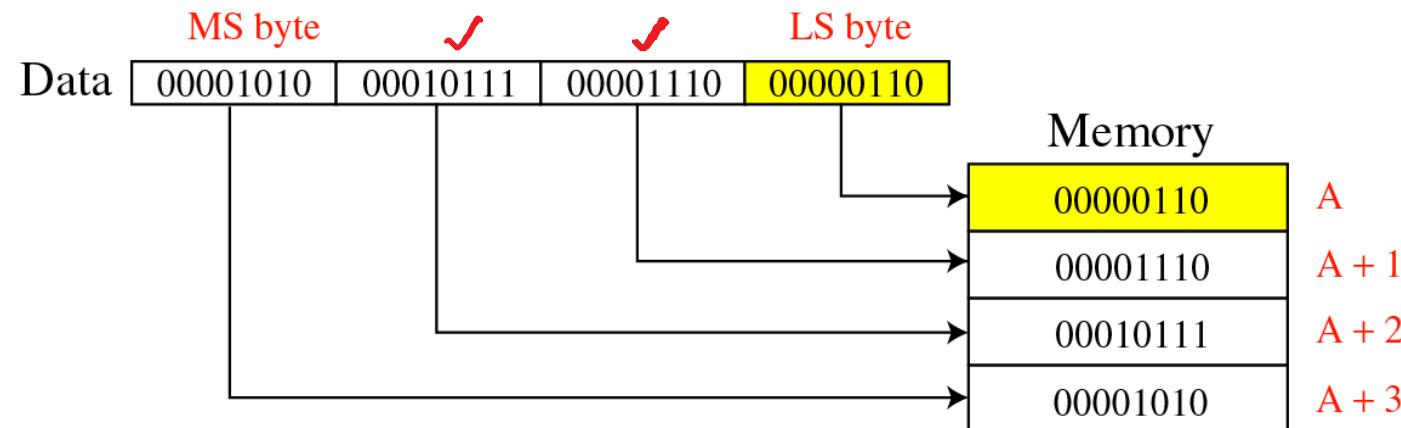


# Constructing Messages - Byte Ordering

## ■ Big-Endian:



## ■ Little-Endian:



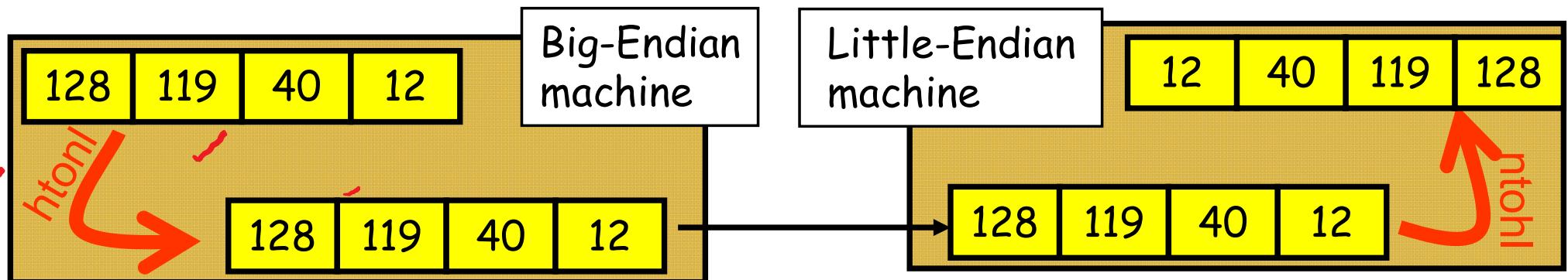
# Constructing Messages - Byte Ordering -

## Solution: Network Byte Ordering

- **Host Byte-Ordering:** the byte ordering used by a host (big or little)
- **Network Byte-Ordering:** the byte ordering used by the network – always big-endian ~~✓~~.

- `u_long htonl(u_long x);`
- `u_short htons(u_short x);`
- `u_long ntohs(u_long x);` → ~~19 addres~~
- `u_short ntohs(u_short x);` ~~port~~

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order



ntoh

# Constructing Messages - Byte Ordering - Example

## Client

```
unsigned short clientPort, message;      unsigned int messageLength;  
  
servPort = 1111;  
message = htons(clientPort);  
messageLength = sizeof(message);  
  
if (sendto( clientSock, message, messageLength, 0,  
           (struct sockaddr *) &echoServAddr, sizeof(echoServAddr))  
    != messageLength)  
    DieWithError("send() sent a different number of bytes than expected");
```

## Server

```
unsigned short clientPort, rcvBuffer;  
unsigned int recvMsgSize ;  
  
if ( recvfrom(servSock, &rcvBuffer, sizeof(unsigned int), 0,  
              (struct sockaddr *) &echoClientAddr, sizeof(echoClientAddr)) < 0)  
    DieWithError("recvfrom() failed");  
  
clientPort = ntohs(rcvBuffer);  
printf ("Client's port: %d", clientPort);
```

# Constructing Messages - Alignment and Padding

- consider the following **12 byte** structure

```
typedef struct {  
    int x;      :;  
    short x2;   :;  
    int y;      :;  
    short y2;   :;  
} msgStruct;
```

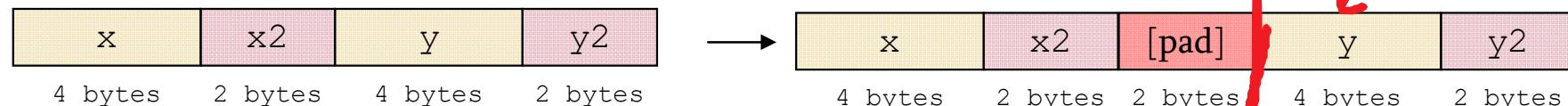
4 , 2

- After compilation it will be a **14 byte** structure!

- Why? → **Alignment!**

- Remember the following rules:

- data structures are maximally aligned, according to the size of the largest native integer
- other multibyte fields are aligned to their size, e.g., a four-byte integer's address will be divisible by four



- This can be avoided

- include padding to data structure
- reorder fields

```
typedef struct {  
    int x;  
    short x2;  
    char pad[2];  
    int y;  
    short y2;  
} msgStruct;
```

```
typedef struct {  
    int x;           4 + 4  
    int y;           4 + 4  
    short x2;        2 * 2 = 4  
    short y2;        2 * 2 = 4  
} msgStruct;
```

# Constructing Messages - Framing and Parsing

- Framing is the problem of formatting the information so that the receiver can parse messages
- Parse means to locate the beginning and the end of message
  - — .
- This is easy if the fields have fixed sizes
  - e.g., msgStruct, Sockaddr Sockaddr.in data
- For text-string representations is harder
  - Solution: use of appropriate delimiters
  - caution is needed since a call of `recv` may return the messages sent by multiple calls of `send`

