NETWORKING LABORATORY

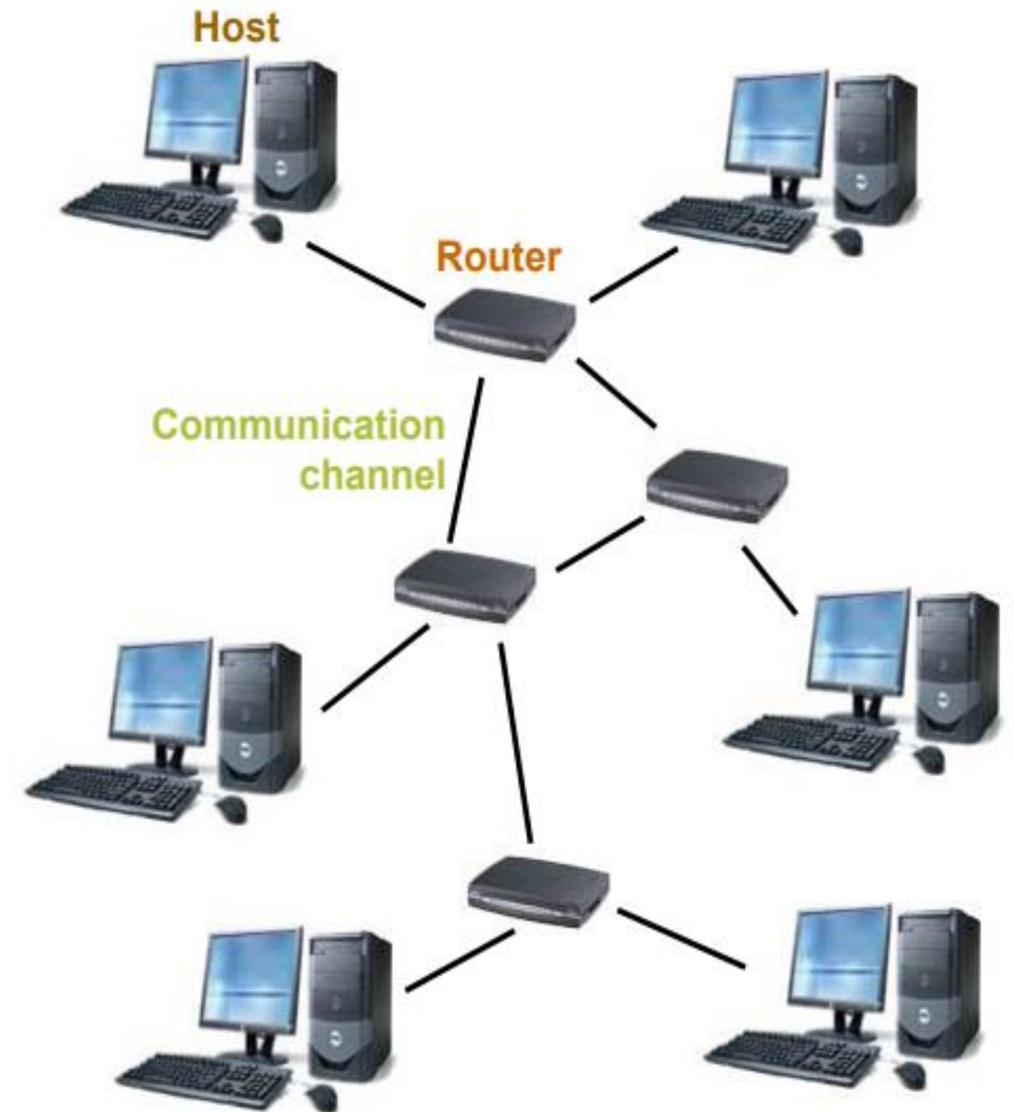
Course Instructor: Ms. Deepika Roselind J

Objectives

- To learn socket programming.
- To learn and use network commands.
- To gain knowledge about the working of routing algorithms.
- To use simulation tools to analyze the performance of protocols in different layers in computer networks

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)

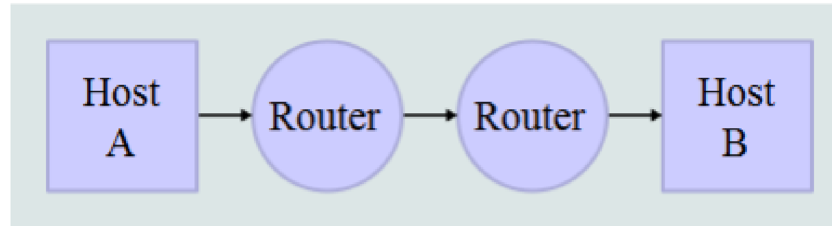


Protocol Families - TCP/IP

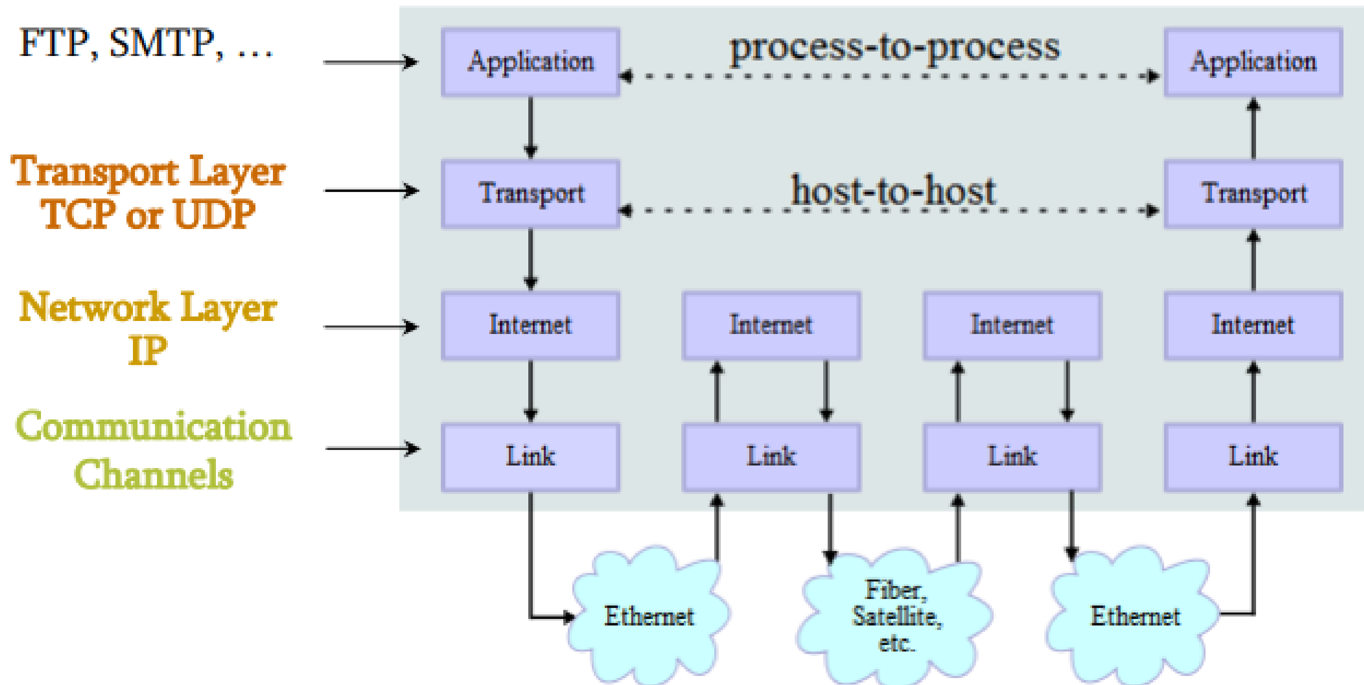
- Several protocols for different problems
  - Protocol Suites or Protocol Families: TCP/IP
- TCP/IP provides end-to-end connectivity specifying how data should be
  - formatted,
  - addressed,
  - transmitted,
  - routed, and
  - received at the destination
- It can be used in the internet and in stand-alone private networks
- It is organized into layers

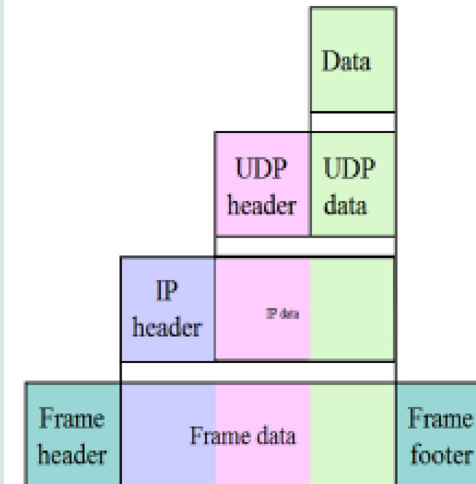
TCP/IP

Network Topology



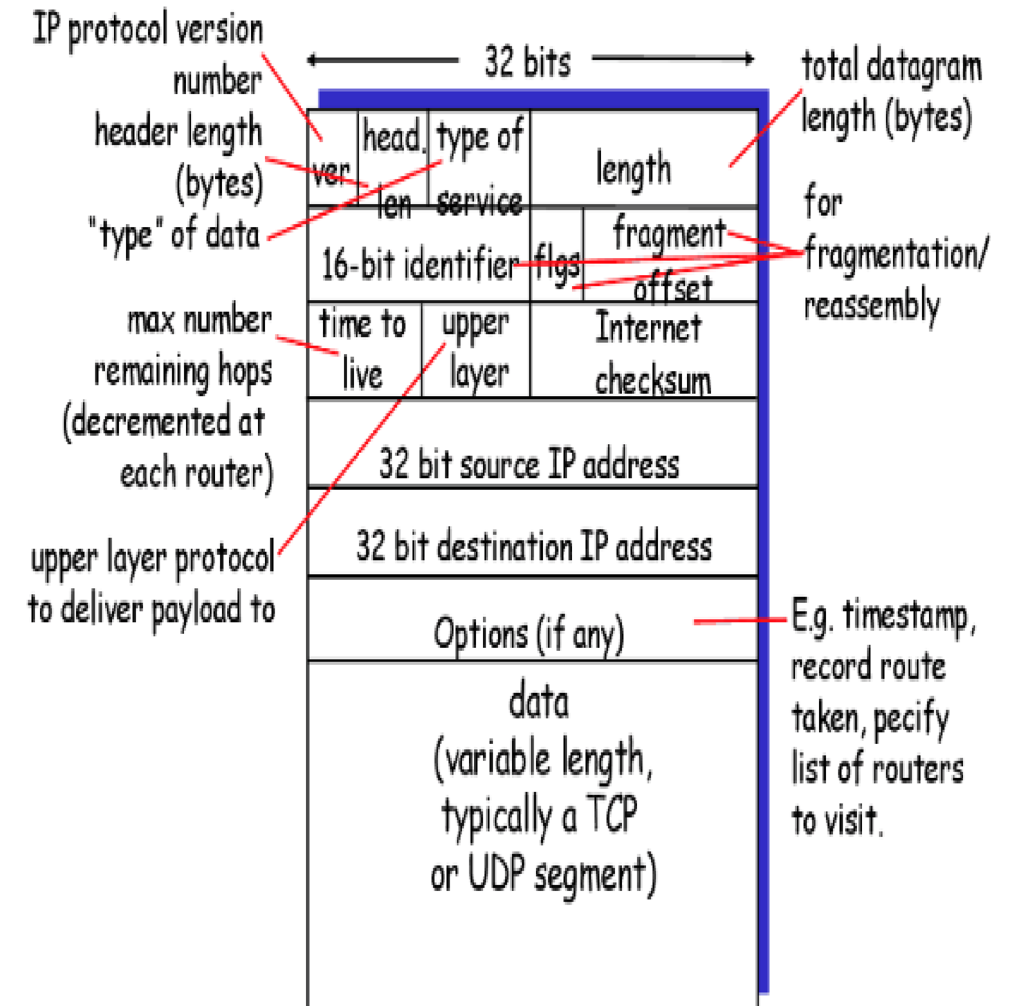
Data Flow





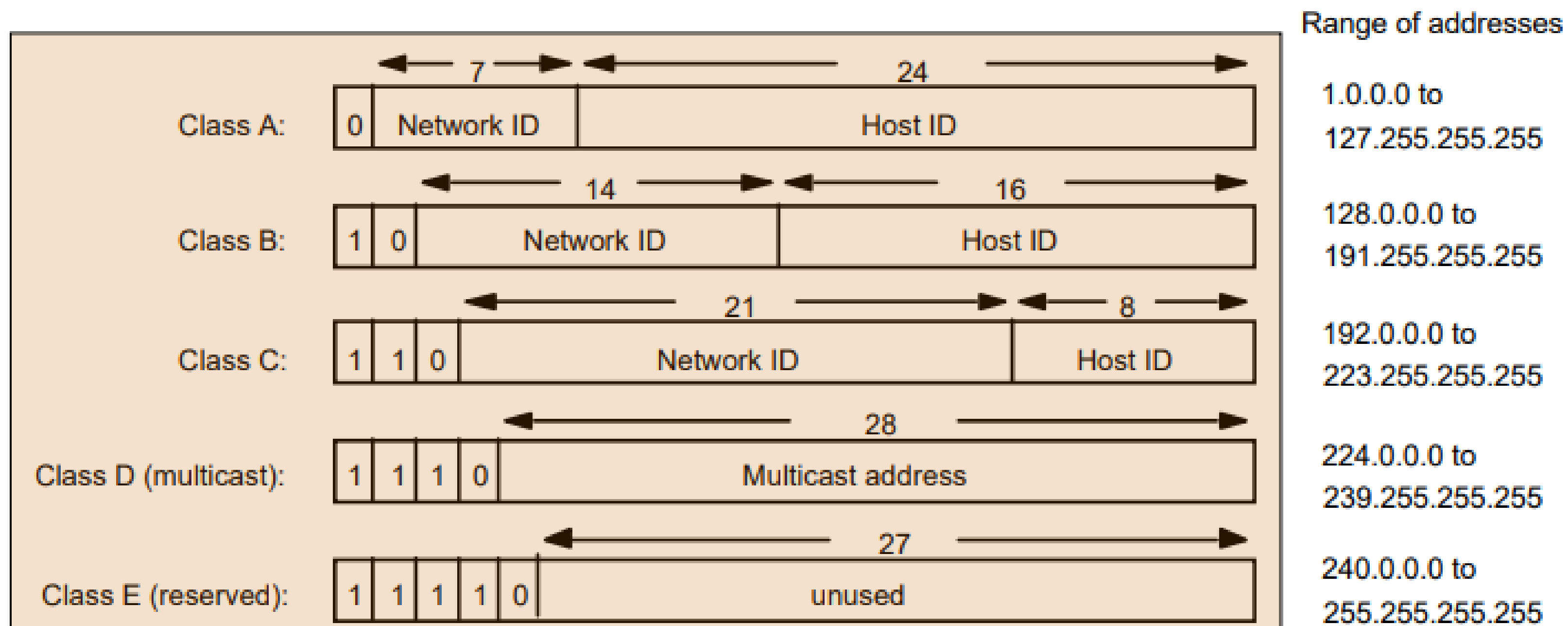
Internet Protocol (IP)

- The Internet Protocol (IP) is the principal communications protocol in the Internet protocol suite
  - Relay datagrams across network boundaries
  - Datagrams are typically structured in header and payload sections
- Its routing function enables internetworking, and essentially establishes the Internet.
- IP has the task of delivering packets from the source host to the destination host
  - based on the IP addresses in the packet headers



Addresses - IPv4

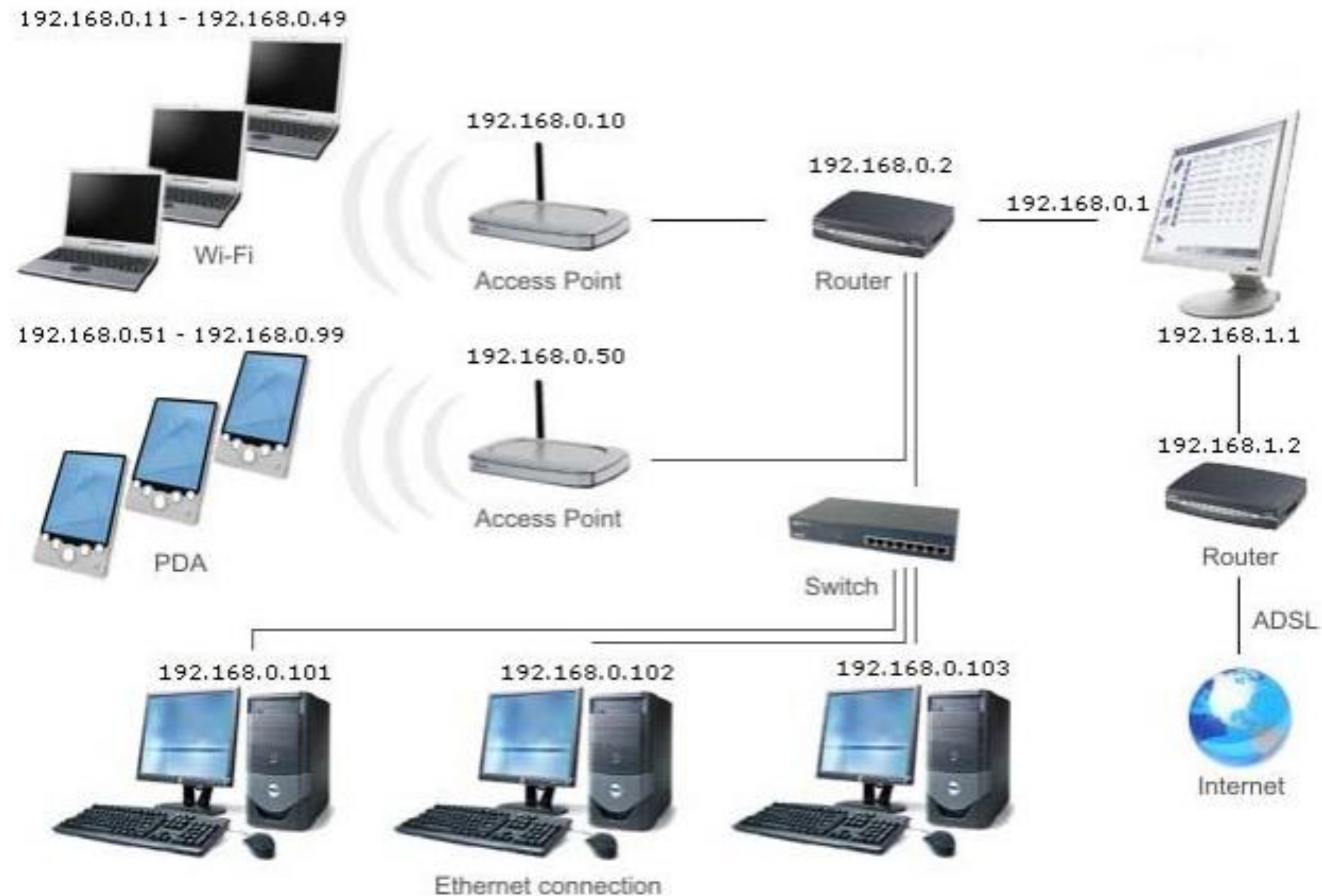
- The 32 bits of an IPv4 address are broken into 4 octets, or 8 bit fields (0-255 value in decimal notation).
- For networks of different size,
  - the first one (for large networks) to three (for small networks) octets can be use d to identify the network, while
  - the rest of the octets can be used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



IP address classes

Class	Address range	Supports
Class A	1.0.0.1 to 126.255.255.254	Supports 16 million hosts on each of 127 networks.
Class B	128.1.0.1 to 191.255.255.254	Supports 65,000 hosts on each of 16,000 networks.
Class C	192.0.1.1 to 223.255.254.254	Supports 254 hosts on each of 2 million networks.
Class D	224.0.0.0 to 239.255.255.255	Reserved for multicast groups.
Class E	240.0.0.0 to 254.255.255.254	Reserved for future use, or research and development purposes.

Local Area Network Addresses - IPv4



TCP vs UDP

TCP	UDP
Reliable, guaranteed	Unreliable. Instead, prompt delivery of packets
Connection-oriented	Connectionless
Used in applications that require safety guarantee. (eg. file applications.)	Used in media applications. (eg. video or voice transmissions.)
Flow control, sequencing of packets, error-control.	No flow or sequence control, user must handle these manually.
Uses byte stream as unit of transfer. (stream sockets)	Uses datagrams as unit of transfer. (datagram sockets)
Allows two-way data exchange, once the connection is established. (full-duplex)	Allows data to be transferred in one direction at once. (half-duplex)
e.g. Telnet uses stream sockets.	e.g. TFTP (trivial file transfer protocol)

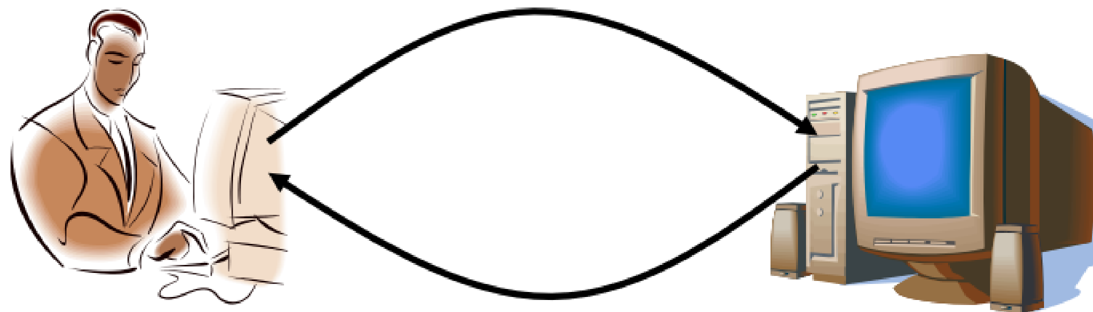
Introduction to Socket Programming

Client

- Initiates a request to the server when interested
- E.g., web browser
- Needs to know the server's address
- Active socket

Server

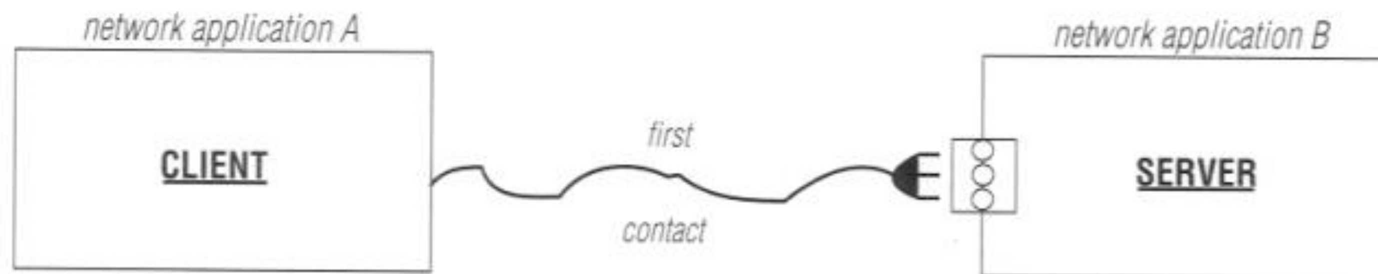
- Serve services to many clients
- E.g., www.google.com
- Not initiate contact with the clients
- Needs a fixed address
- Passive socket



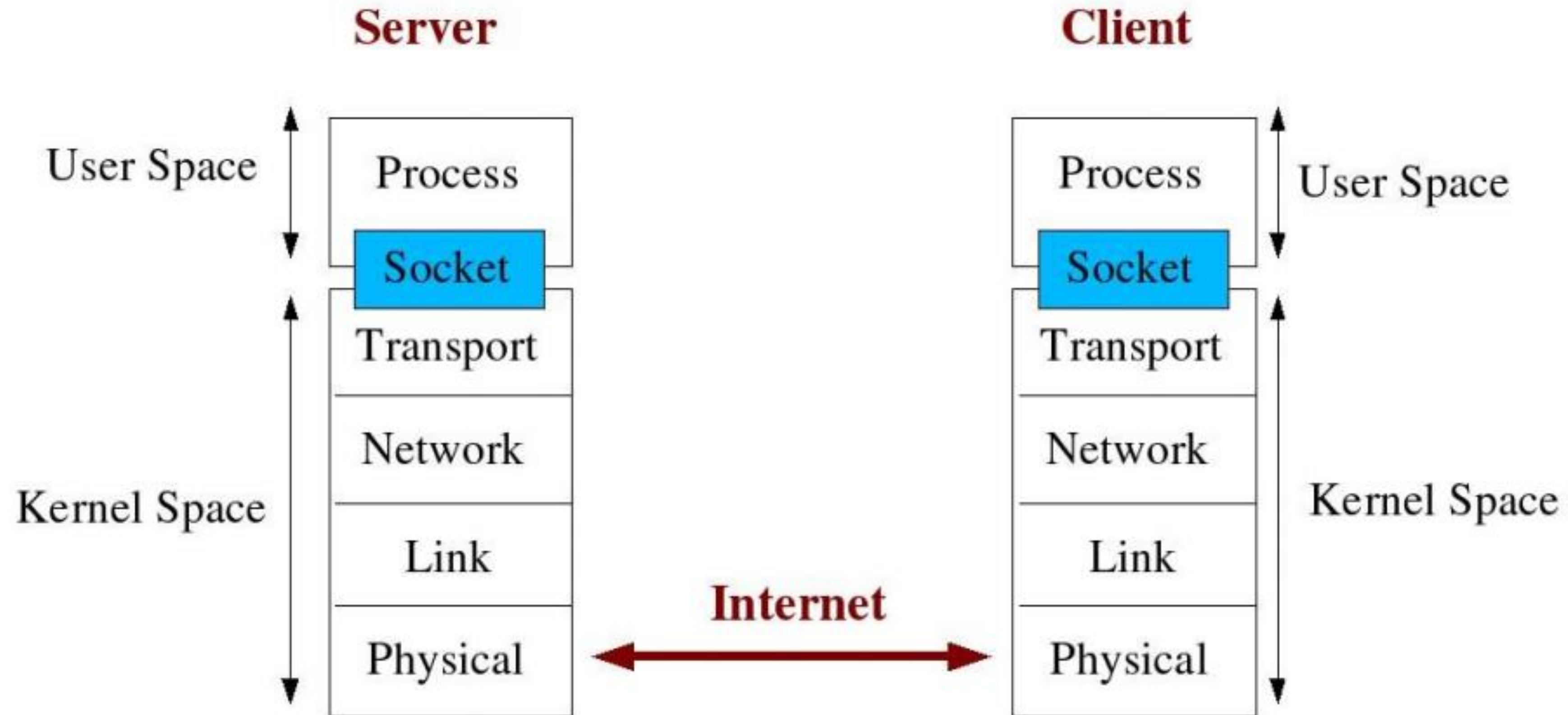
What is a socket?

- Socket: An interface between an application process and transport layer
  - The application process can send/receive messages to/from another application process (local or remote)via a socket

- In Unix jargon, a socket is a file descriptor – an integer associated with an open file



Socket Description



UNIX Socket API

- Socket interface
  - A collection of system calls to write a networking program at user-level.
- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file
  - File is represented by an integer file descriptor
  - Data written into socket on one host can be read out of socket on other host
- System calls for sockets
  - Client: create, connect, write, read, close
  - Server: create, bind, listen, accept, read, write, close

Sockets versus File I/O

File I/O	Network I/O
Open a file	Open a socket
	Name the socket
	Associate with another socket
Read and Write	Send & Receive between sockets
Close the file	Close the socket

Byte Ordering

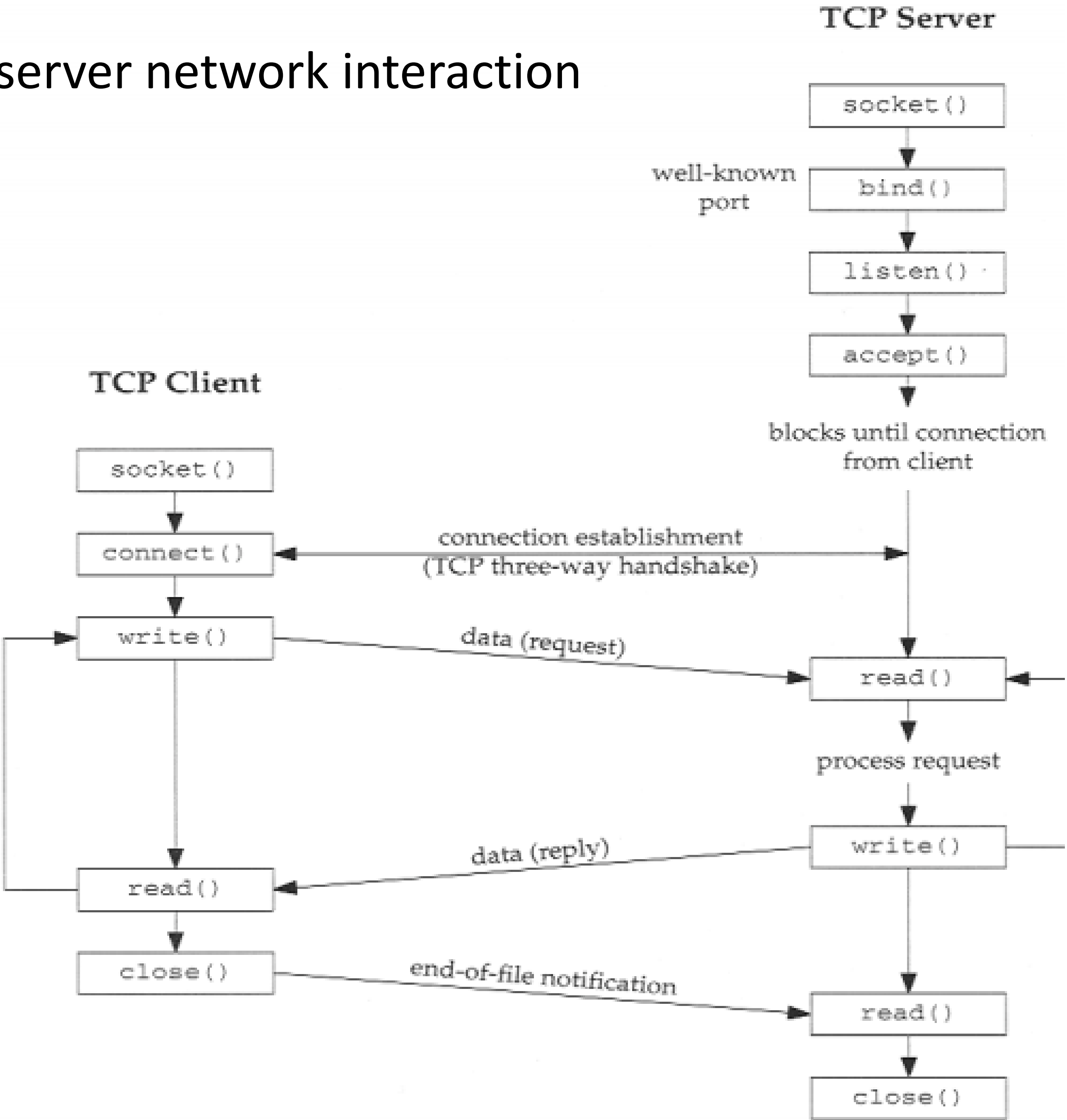
- Two types of “Byte ordering”
  - Network Byte Order: High-order byte of the number is stored in memory at the lowest address
  - Host Byte Order: Low-order byte of the number is stored in memory at the lowest address
  - Network stack (TCP/IP) expects Network Byte Order

- Conversions
  - htons() - Host to Network Short
  - htonl() - Host to Network Long
  - ntohs() - Network to Host Short
  - ntohl() - Network to Host Long

convert port numbers (16 bits)

convert port numbers (32 bits)

Outline of a client-server network interaction



Ports

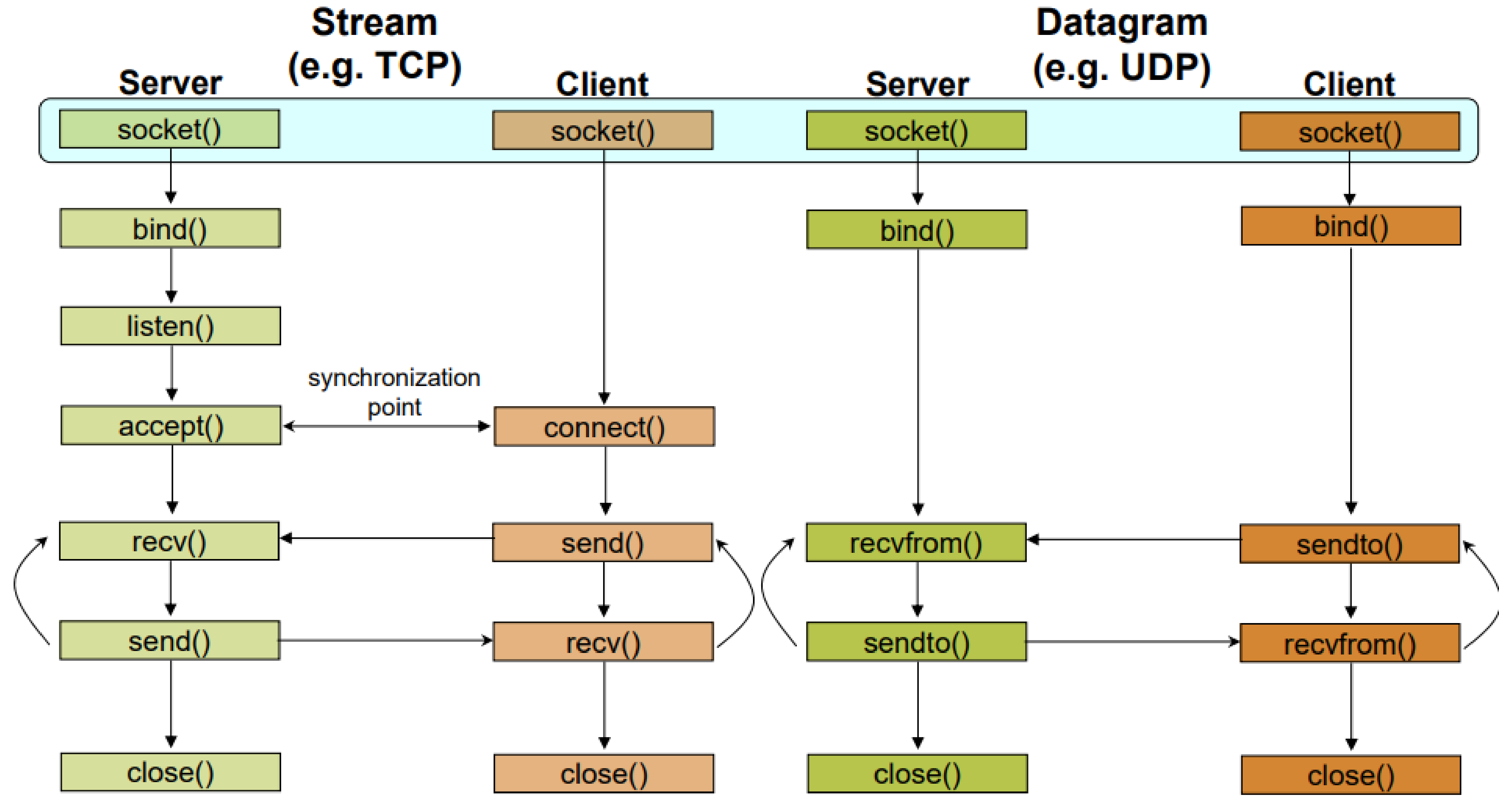
- Sockets are uniquely identified by Internet address, end-to-end protocol, and port number.
- When a socket is first created it is vital to match it with a valid IP address and a port number.
- Ports are software objects to multiplex data between different applications.
- When a host receives a packet, it travels up the protocol stack and finally reaches the application layer.
- Consider a user running an ftp client, a telnet client, and a web browser concurrently. To which application should the packet be delivered?
- Well part of the packet contains a value holding a port number, and it is this number which determines to which application the packet should be delivered.
- So when a client first tries to contact a server, which port number should the client specify? For many common services, standard port numbers are defined.

- Ports 0 – 1023 → reserved and servers or clients that you create will not be able to bind to these ports unless you have root privilege.
- Ports 1024 – 65535 → available for use by your programs, but beware other network applications maybe running and using these port numbers.

Sockets - Procedures

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

Client - Server Communication - Unix



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

Specifying Addresses

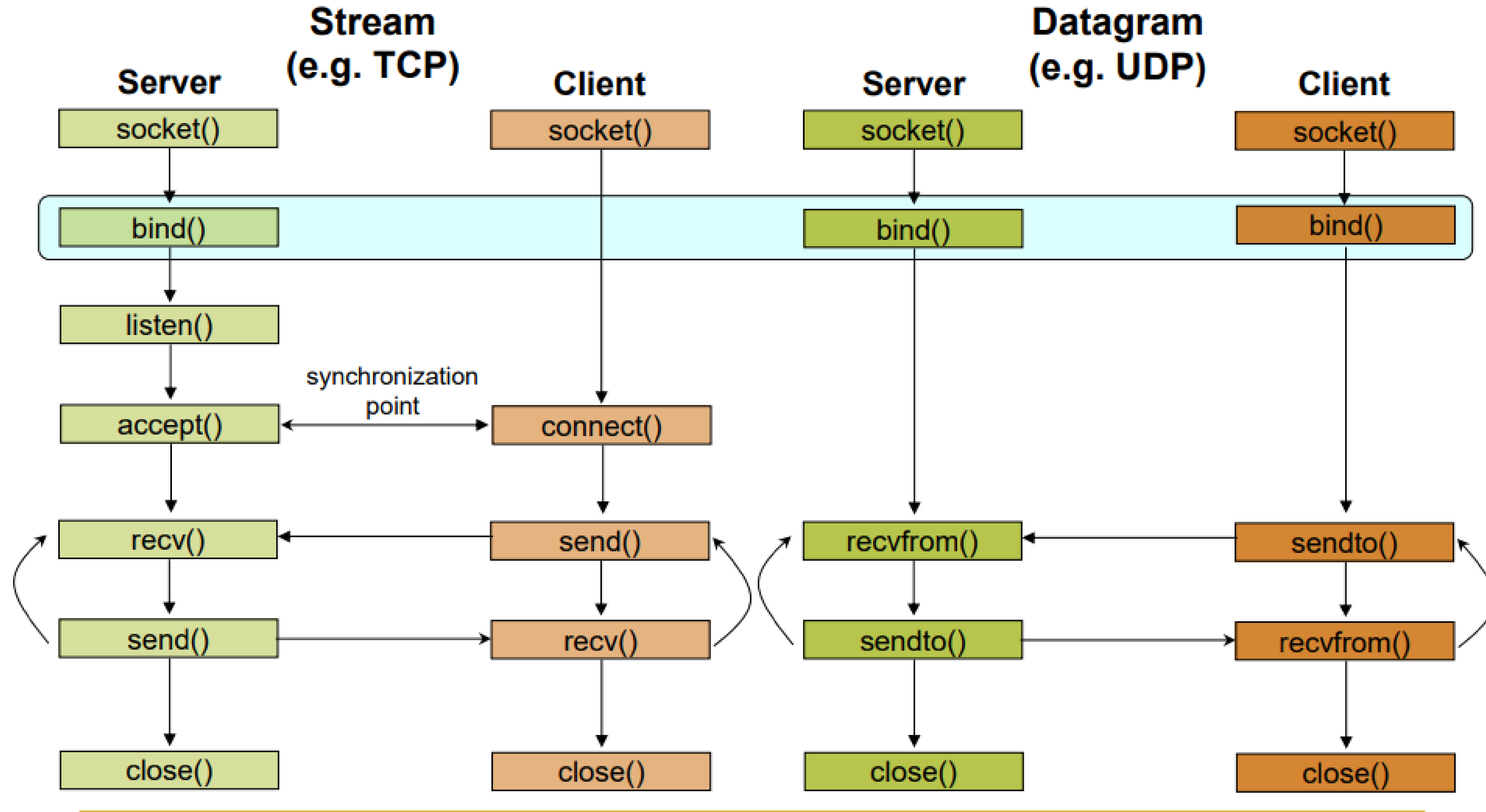
- Socket API defines a generic data type for addresses:

```
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 */
}
```

bind()



Assign address to socket: bind()

- associates and reserves a port for use by the socket

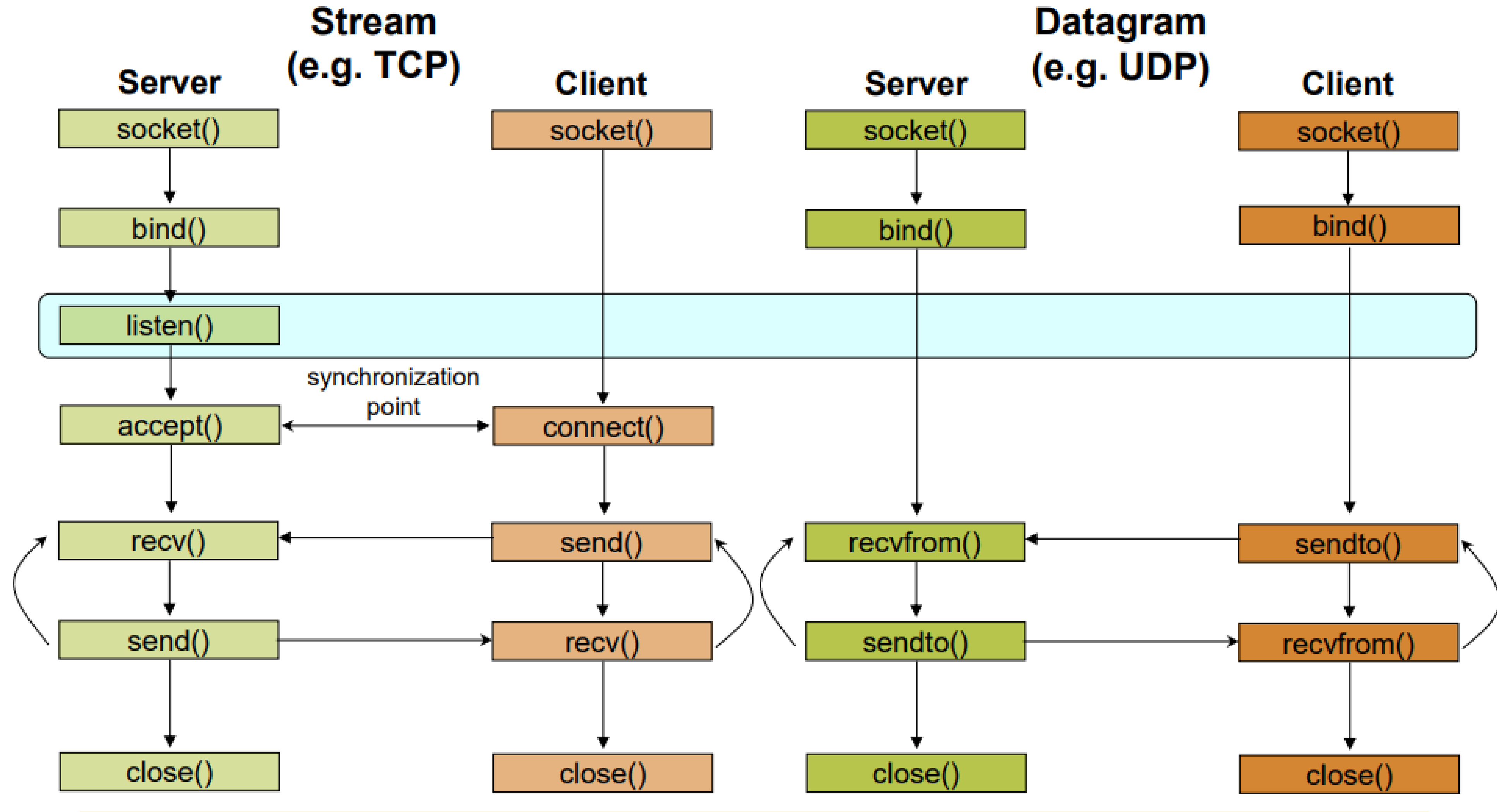
int status = bind(sockid, &addrport, size);

- **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) {                    …}
```

Listen()



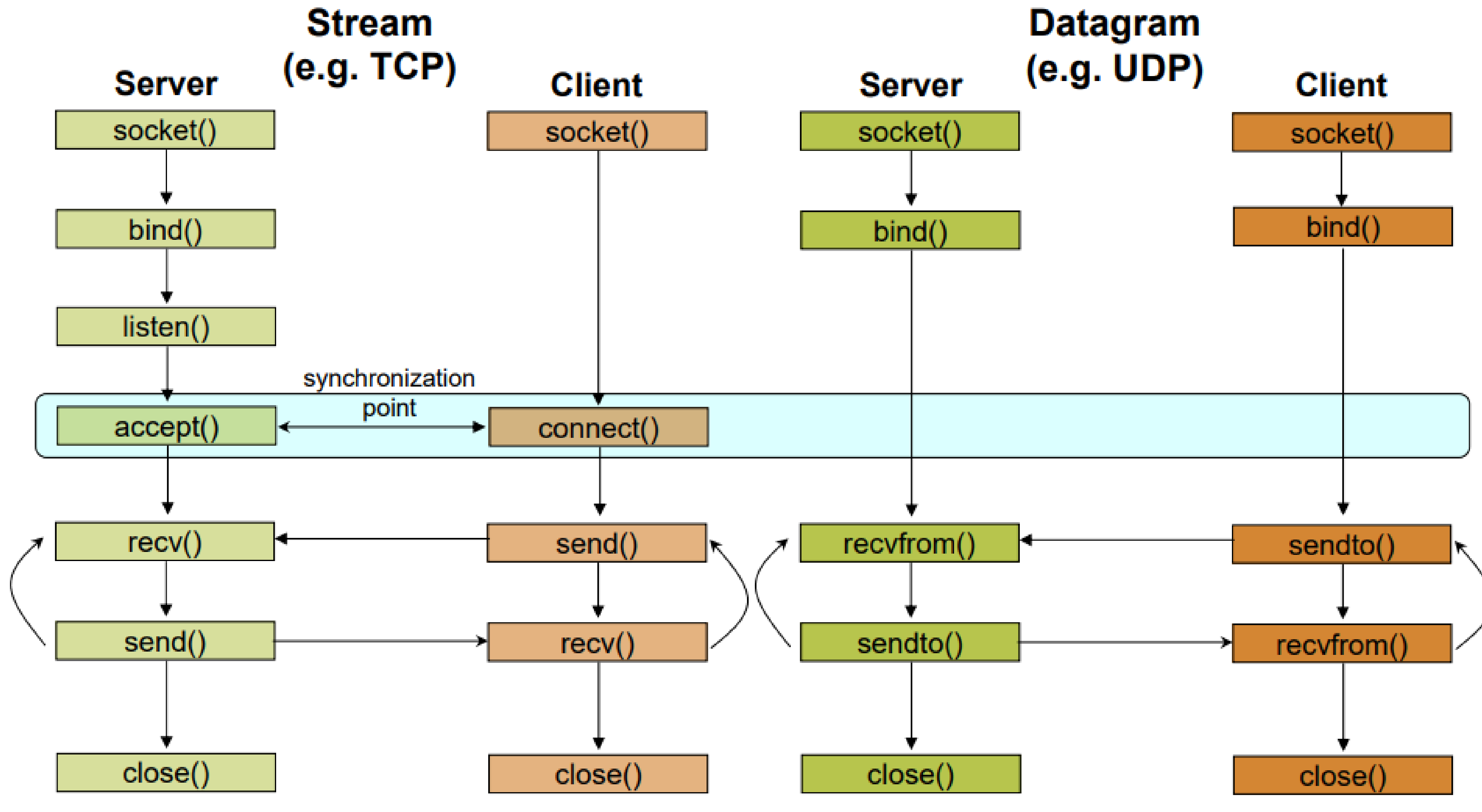
Listen()

- 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

Accept()



Establish Connection: connect()

- The client establishes a connection with the server by calling connect()

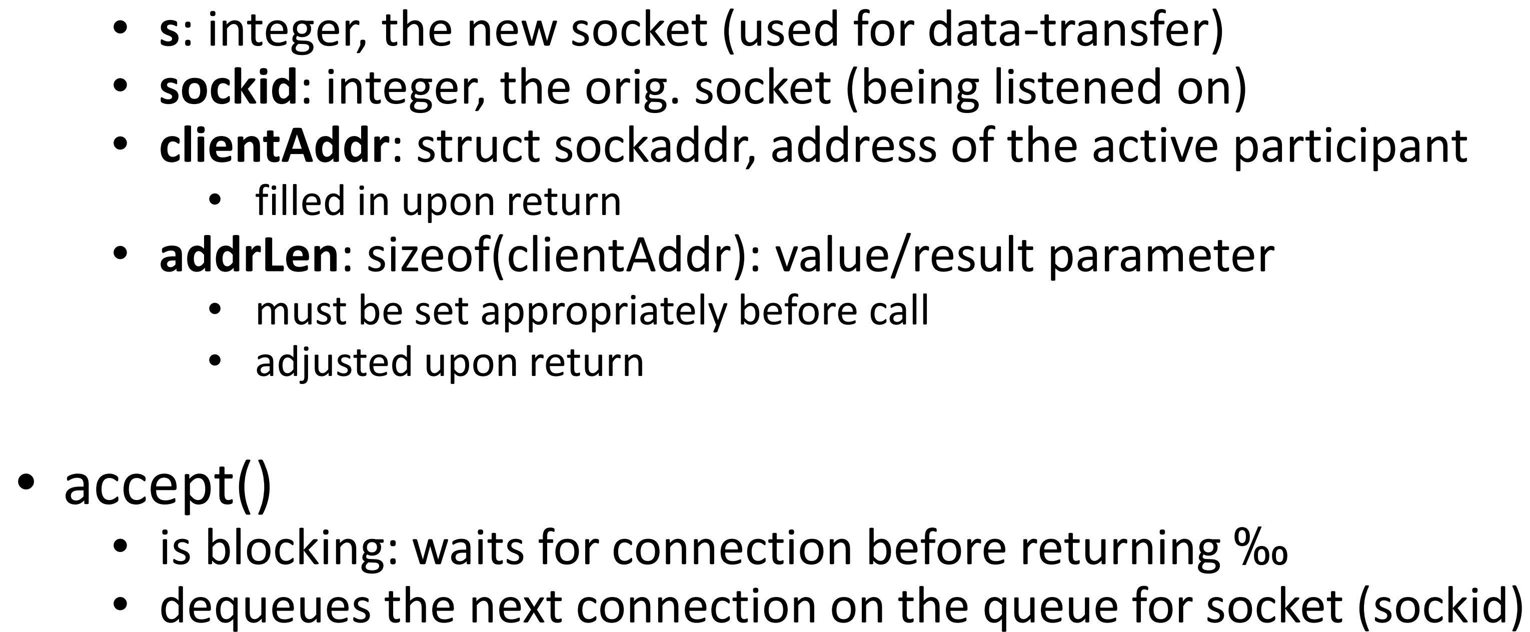
int status = connect(sockid, &foreignAddr, addrlen);

  - **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()

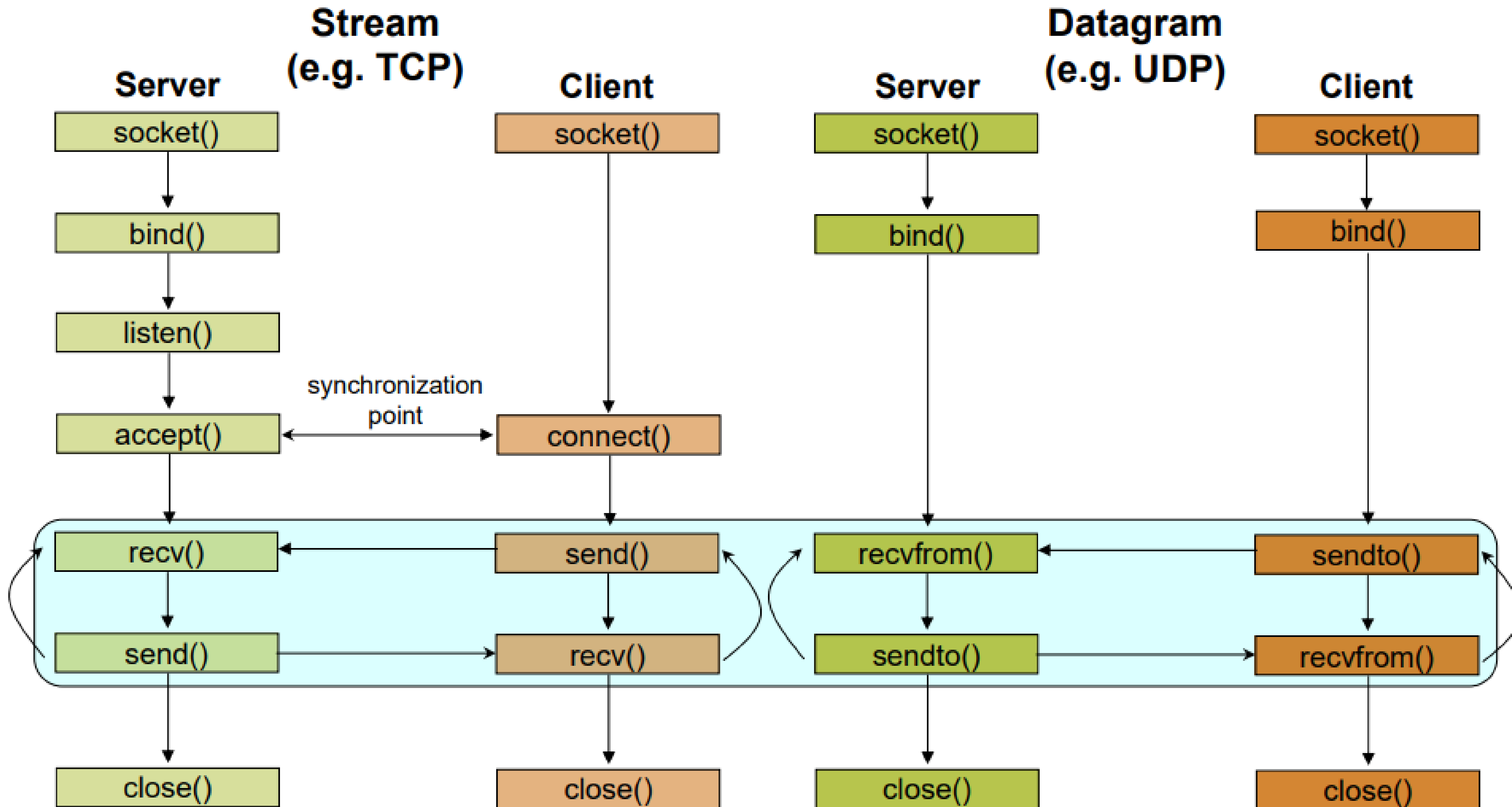


int s = accept(sockid, &clientAddr, &addrLen);

  - **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)

Send() & Recv()



Send() & Recv()

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