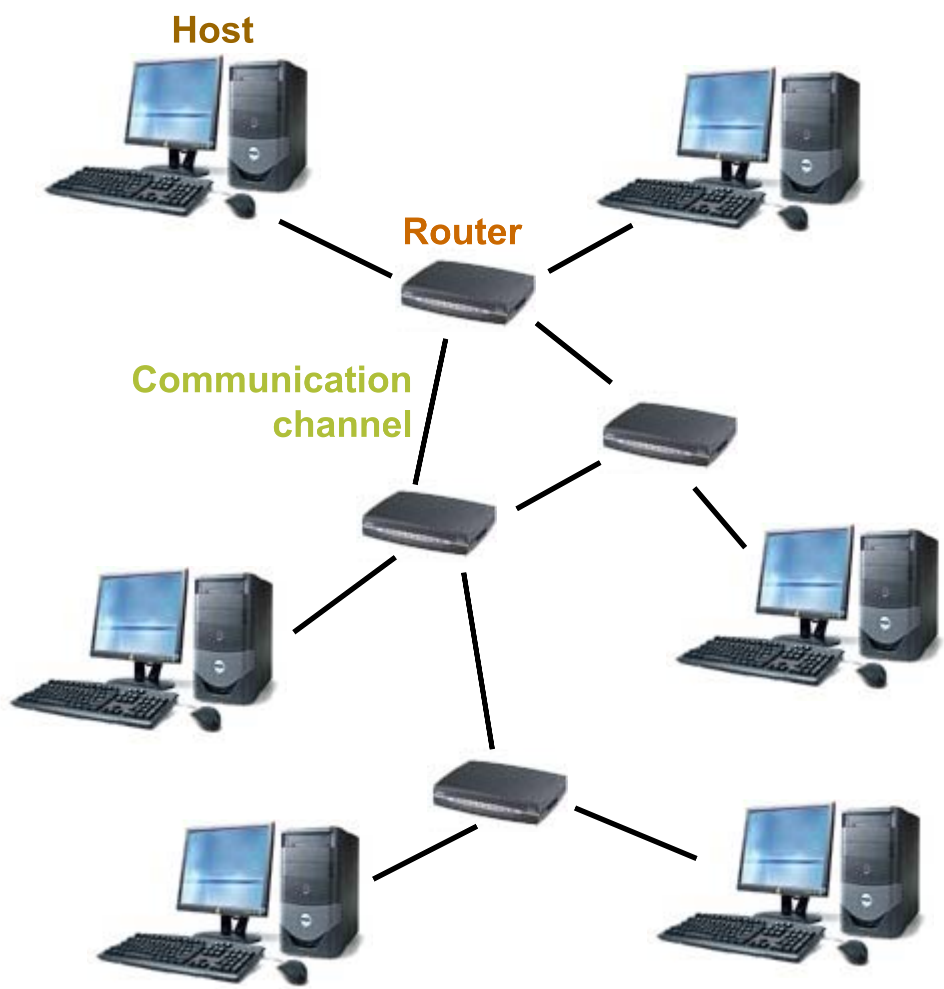
Introduction to Sockets Programming in C using TCP/IP

Professor: Panagiota Fatourou
TA: Eleftherios Kosmas
CSD - May 2012

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

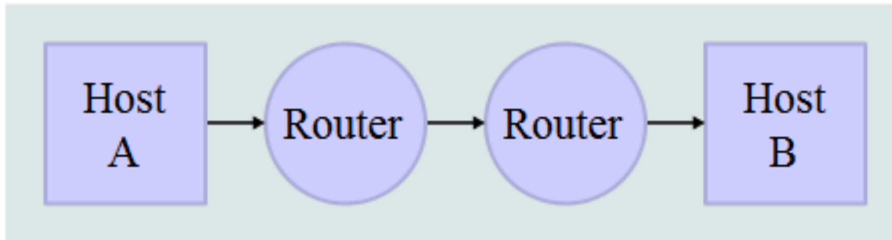
- Several protocols for different problems

☞ **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 *



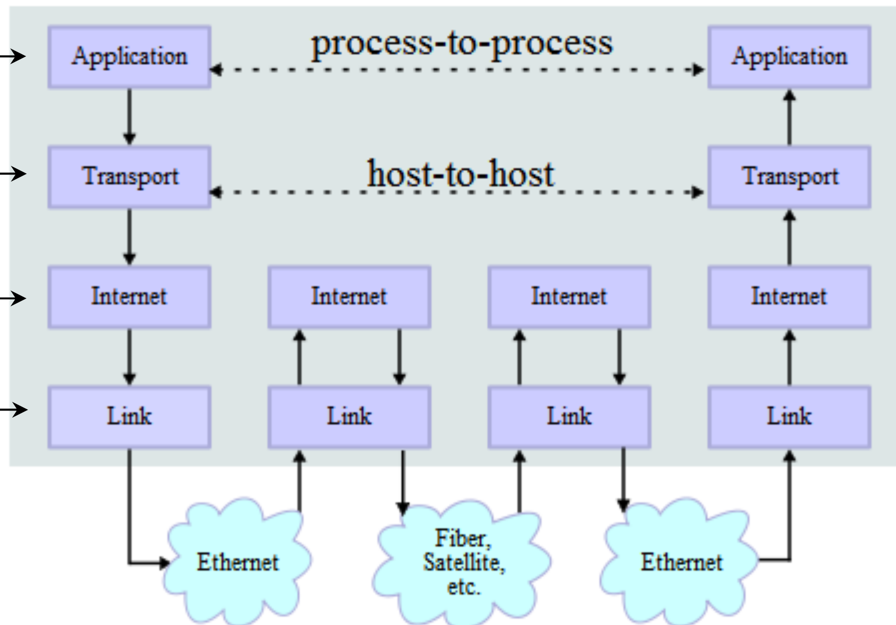
Data Flow

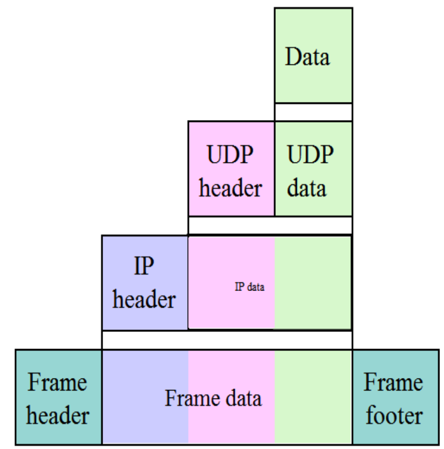
FTP, SMTP, …

Transport Layer
TCP or UDP

Network Layer
IP

Communication
Channels



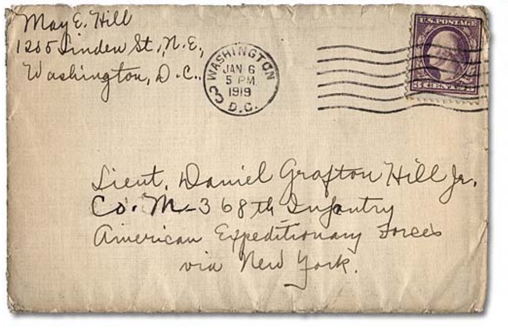


* image is taken from “http://en.wikipedia.org/wiki/TCP/IP_model”

Internet Protocol (IP)

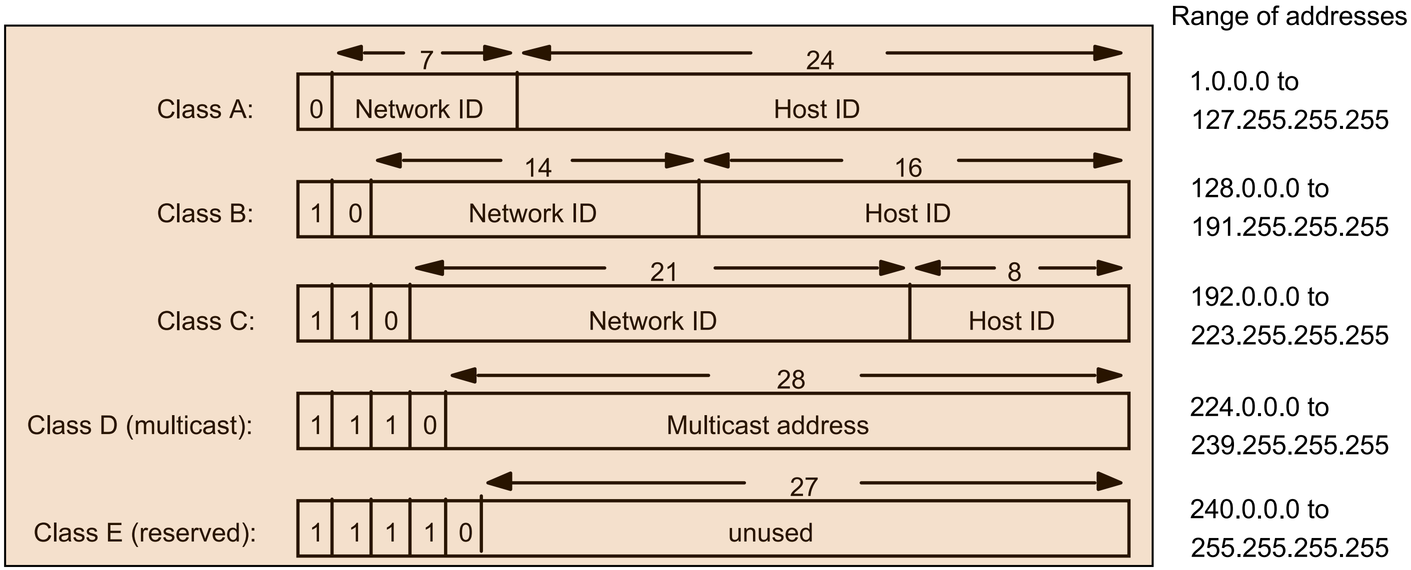
- provides a datagram service
  - packets are handled and delivered independently
- best-effort protocol
  - may loose, reorder or duplicate packets

- each packet must contain an IP address of its destination

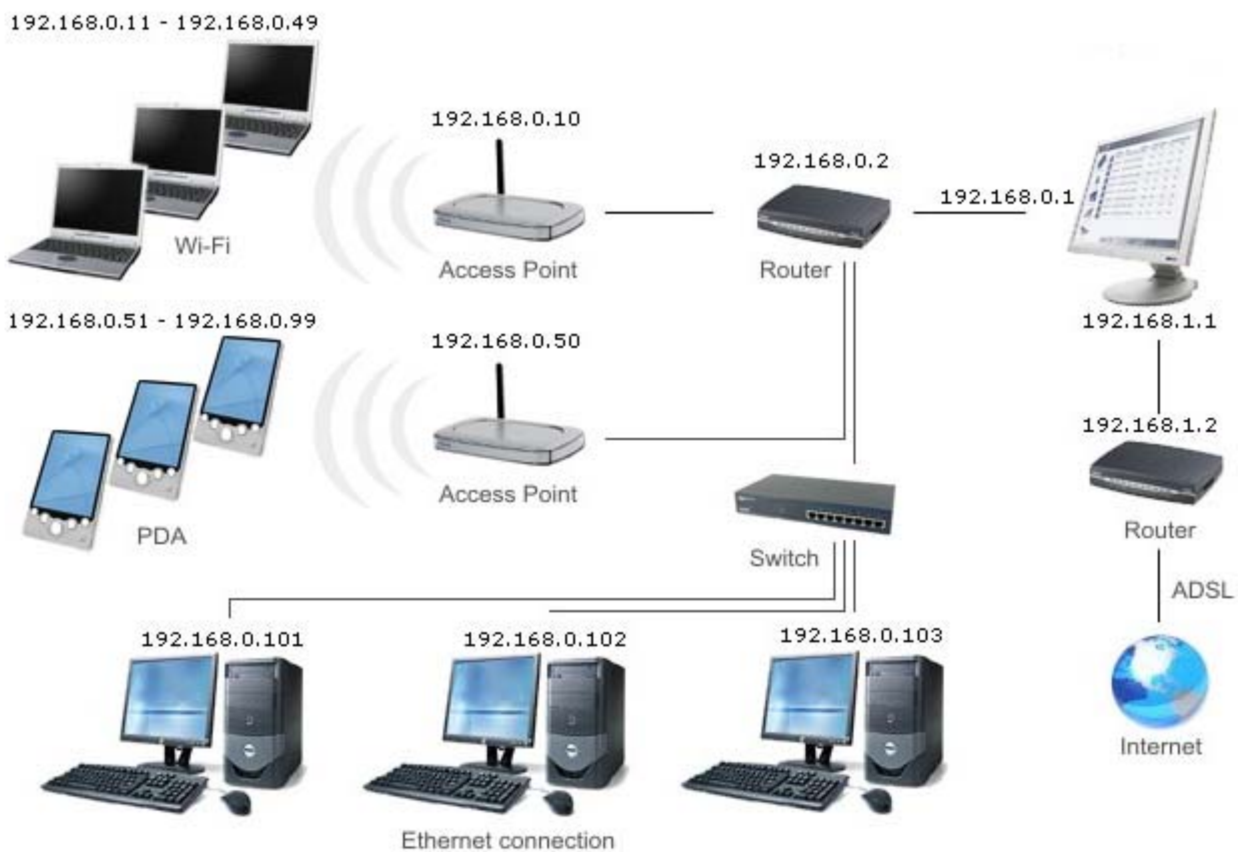


Addresses - IPv4

- 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.



Local Area Network Addresses - IPv4



TCP vs UDP

- Both use **port numbers**
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - ☞ 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

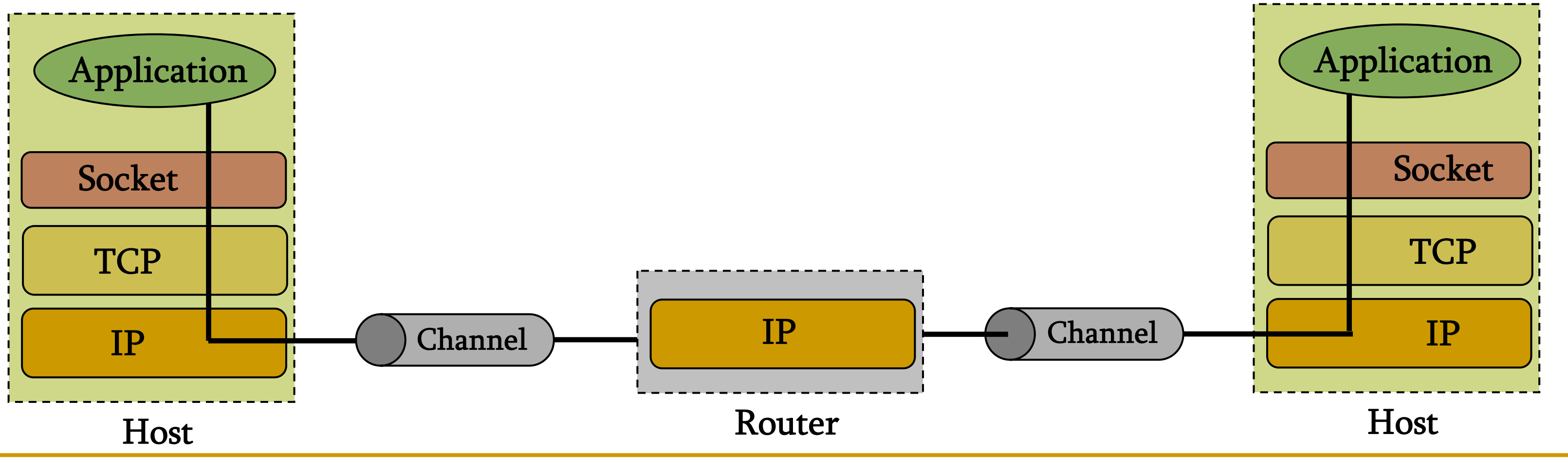
- TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-reply actions.

- Some common examples of TCP and UDP with their default ports:

DNS lookup	UDP	53
FTP	TCP	21
HTTP	TCP	80
POP3	TCP	110
Telnet	TCP	23

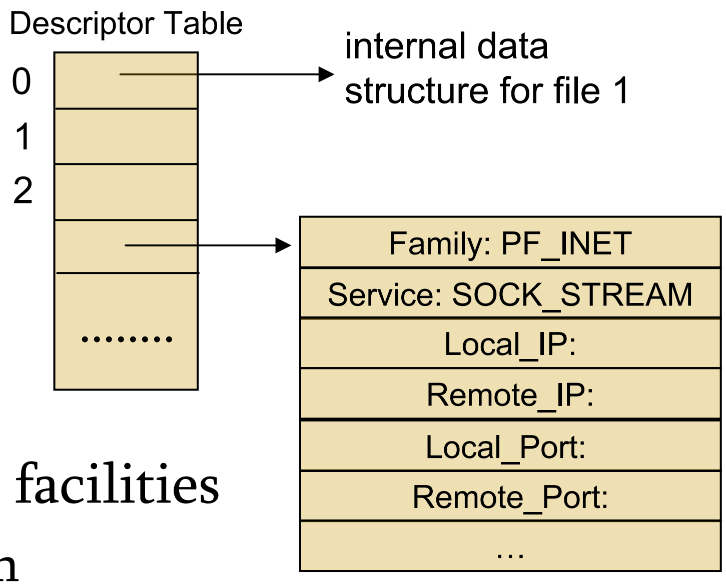
Berkley Sockets

- 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

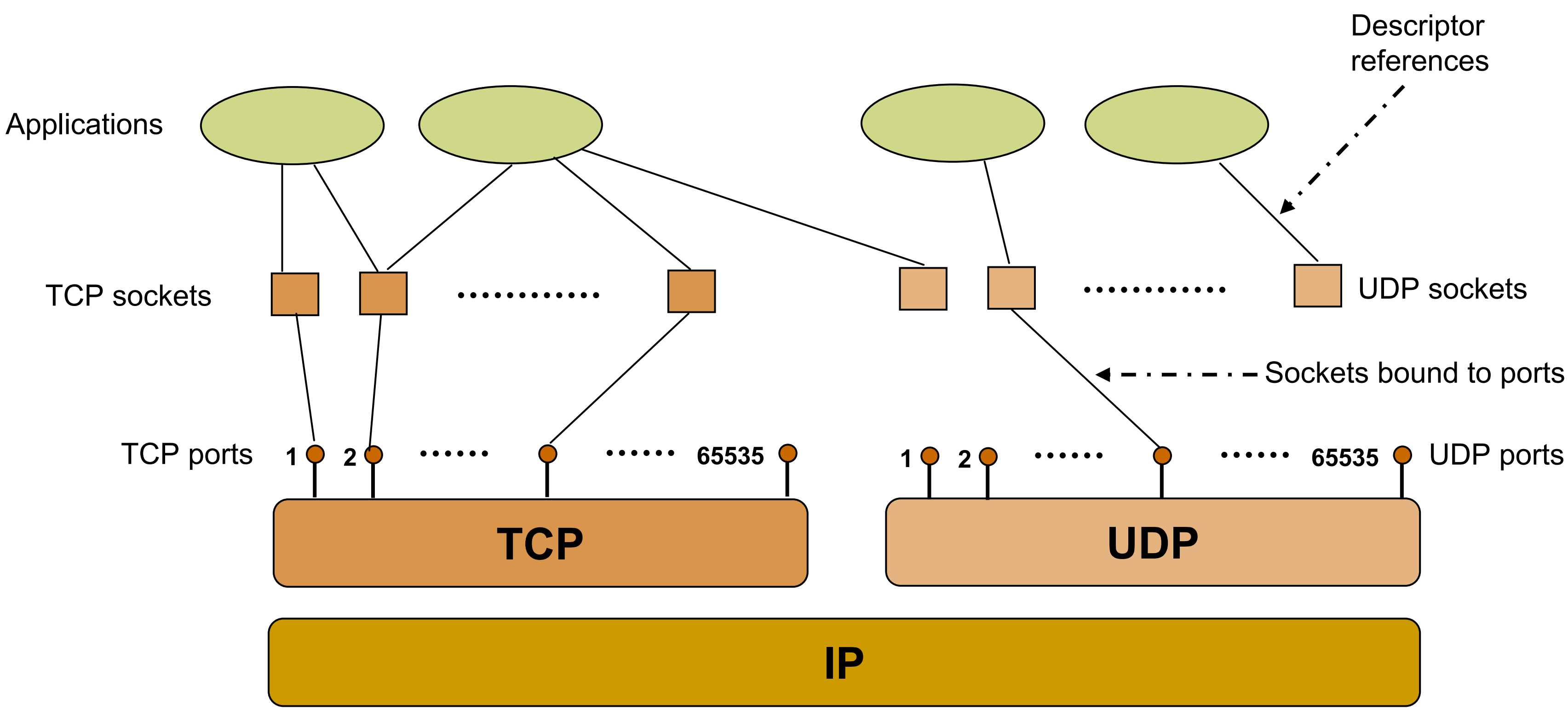


Sockets

- Uniquely identified by
  - an internet address
  - 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
- Socket extend the convectional UNIX I/O facilities
  - file descriptors for network communication
  - extended the read and write system calls

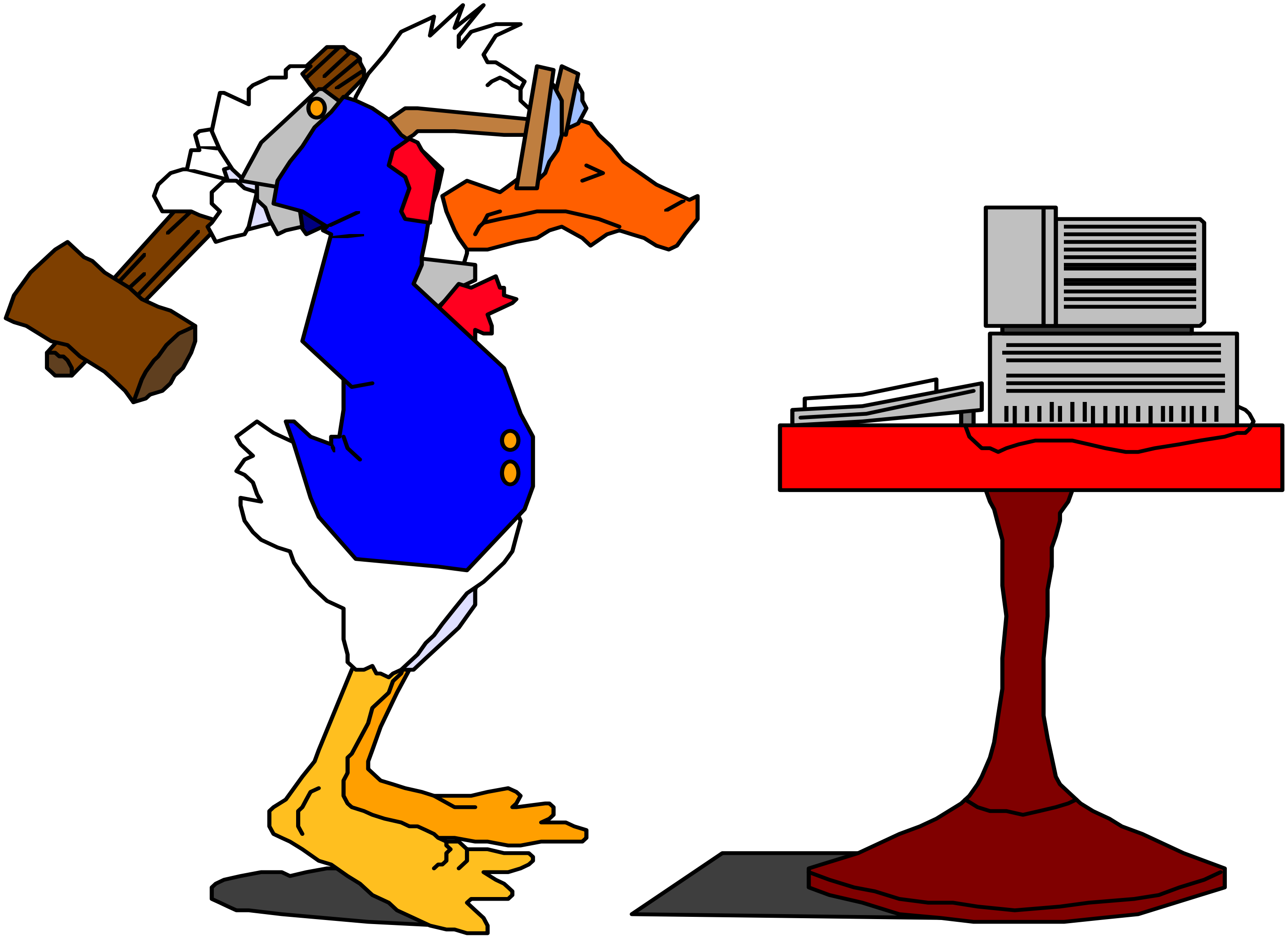


Sockets



Descriptor references
Applications
TCP sockets
UDP sockets
Sockets bound to ports
TCP ports
1
2
65535
1
2
65535
UDP ports
TCP
UDP
IP

Socket Programming



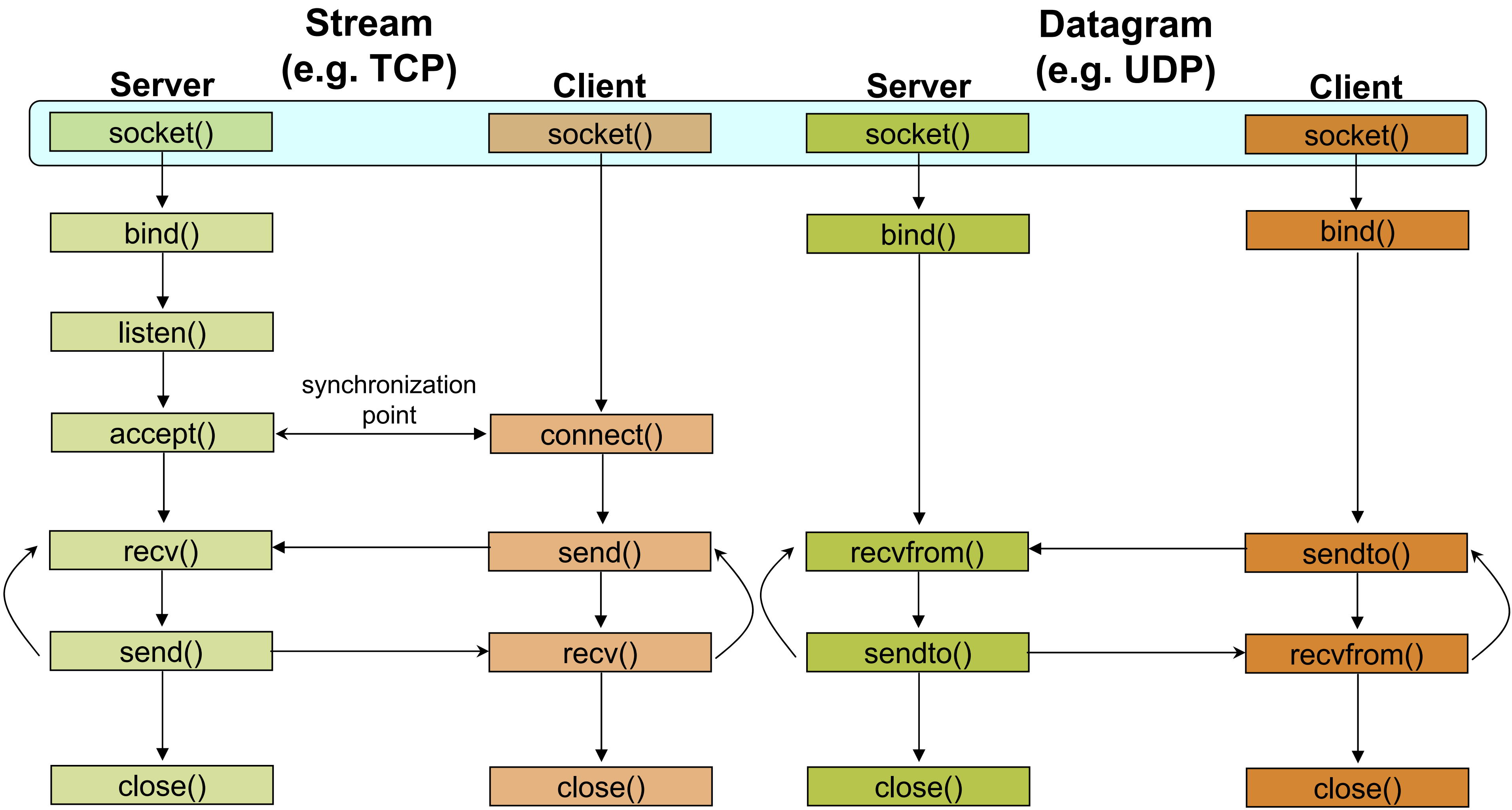
Client-Server communication

- 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 - Unix

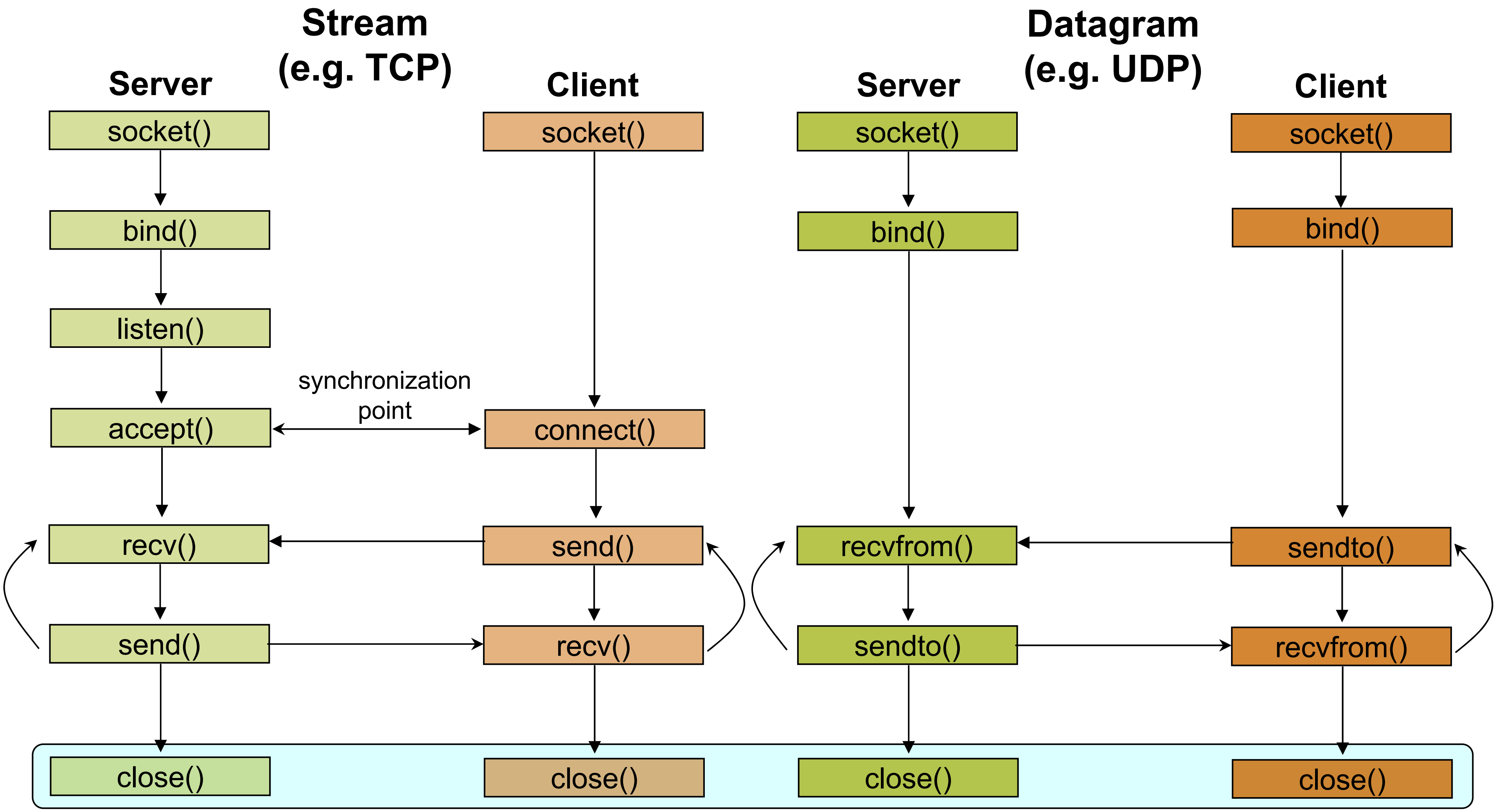


Socket creation in C: socket()

- ```
  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

☞ 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 - Unix



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:

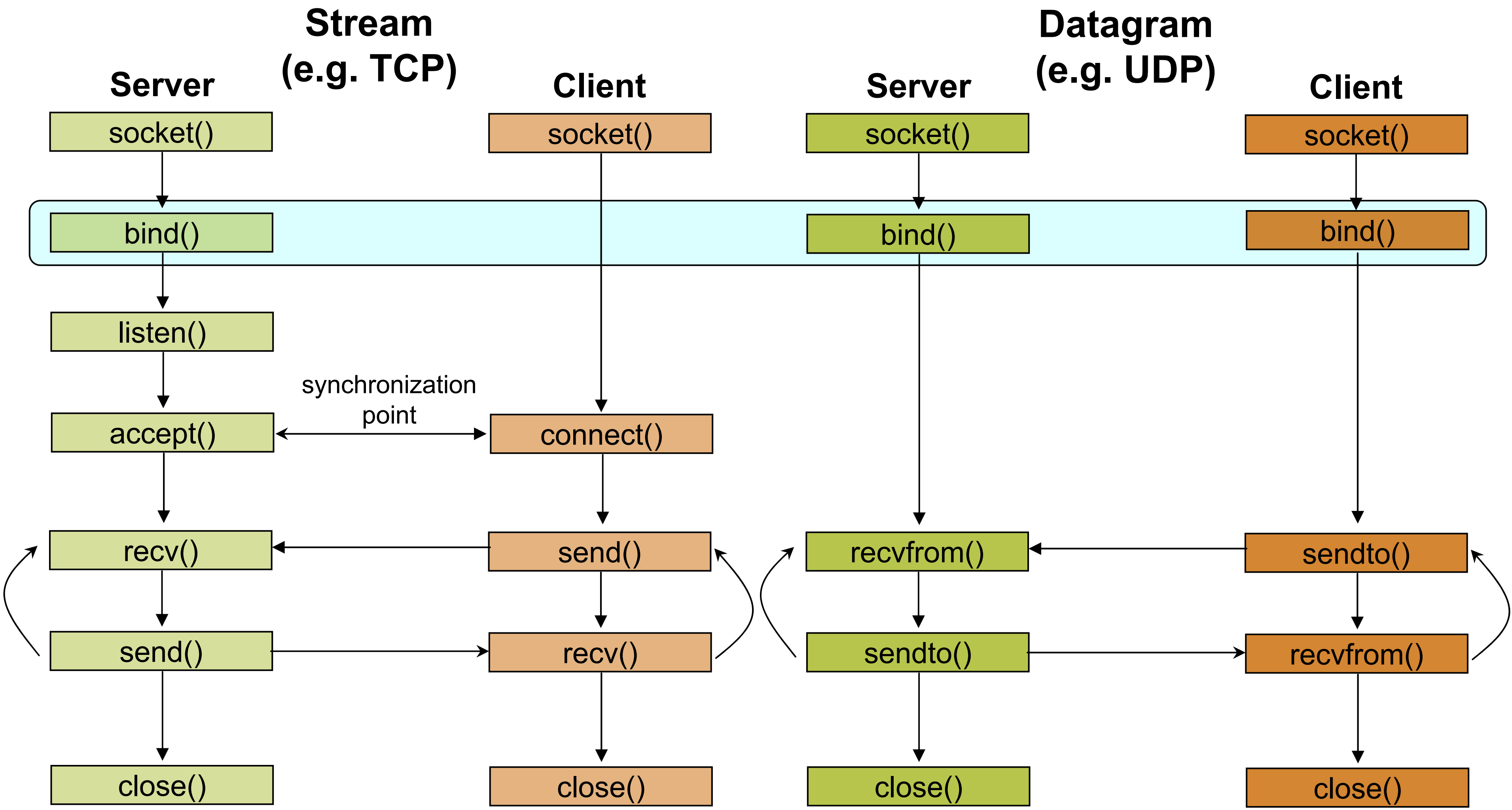
```
struct sockaddr {
    unsigned short sa_family;   /* Address family (e.g. AF_INET) */
    char sa_data[14];           /* Family-specific address information */
}
```

- 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 - Unix



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);

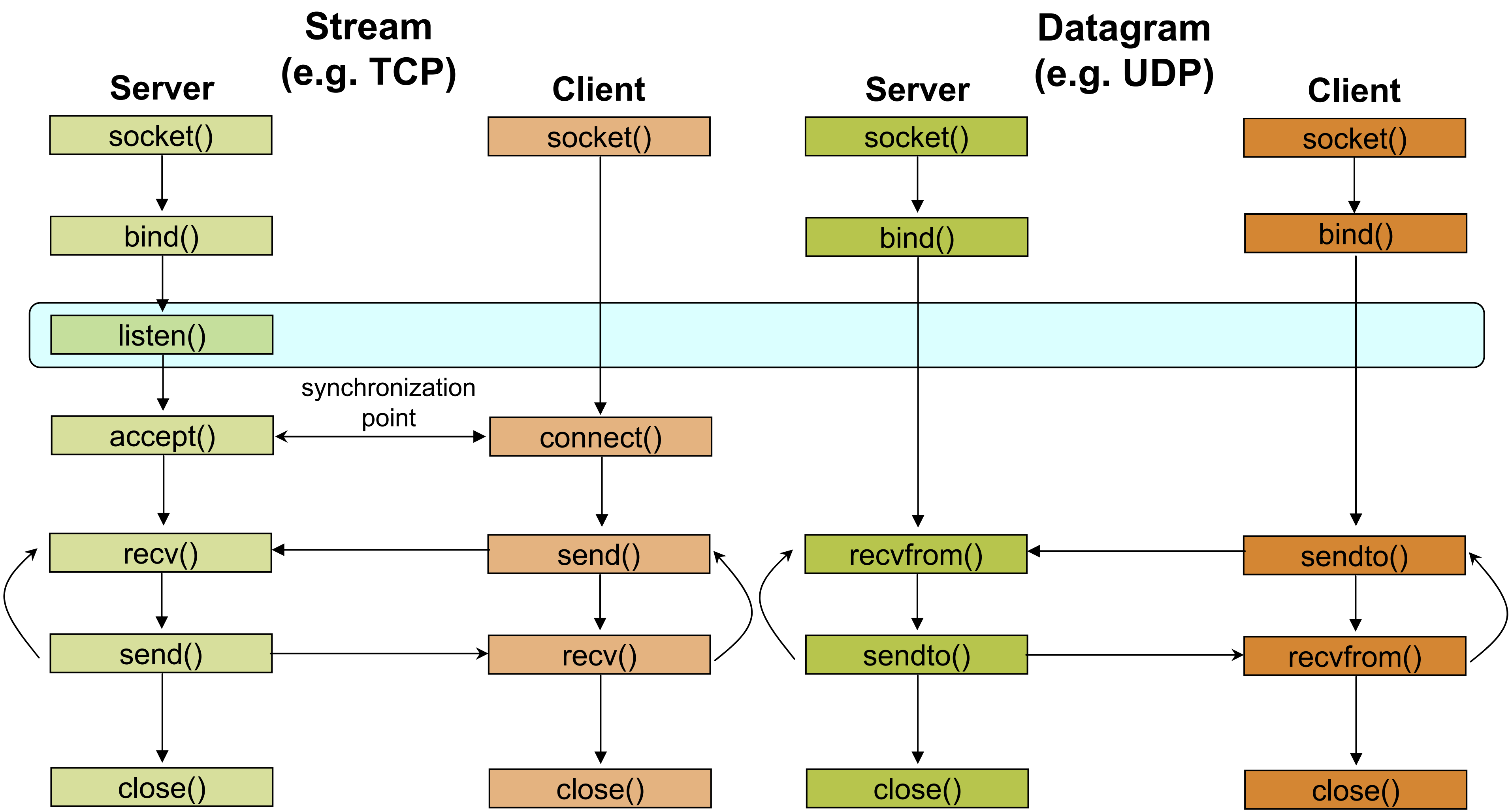
addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
    …}
```

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



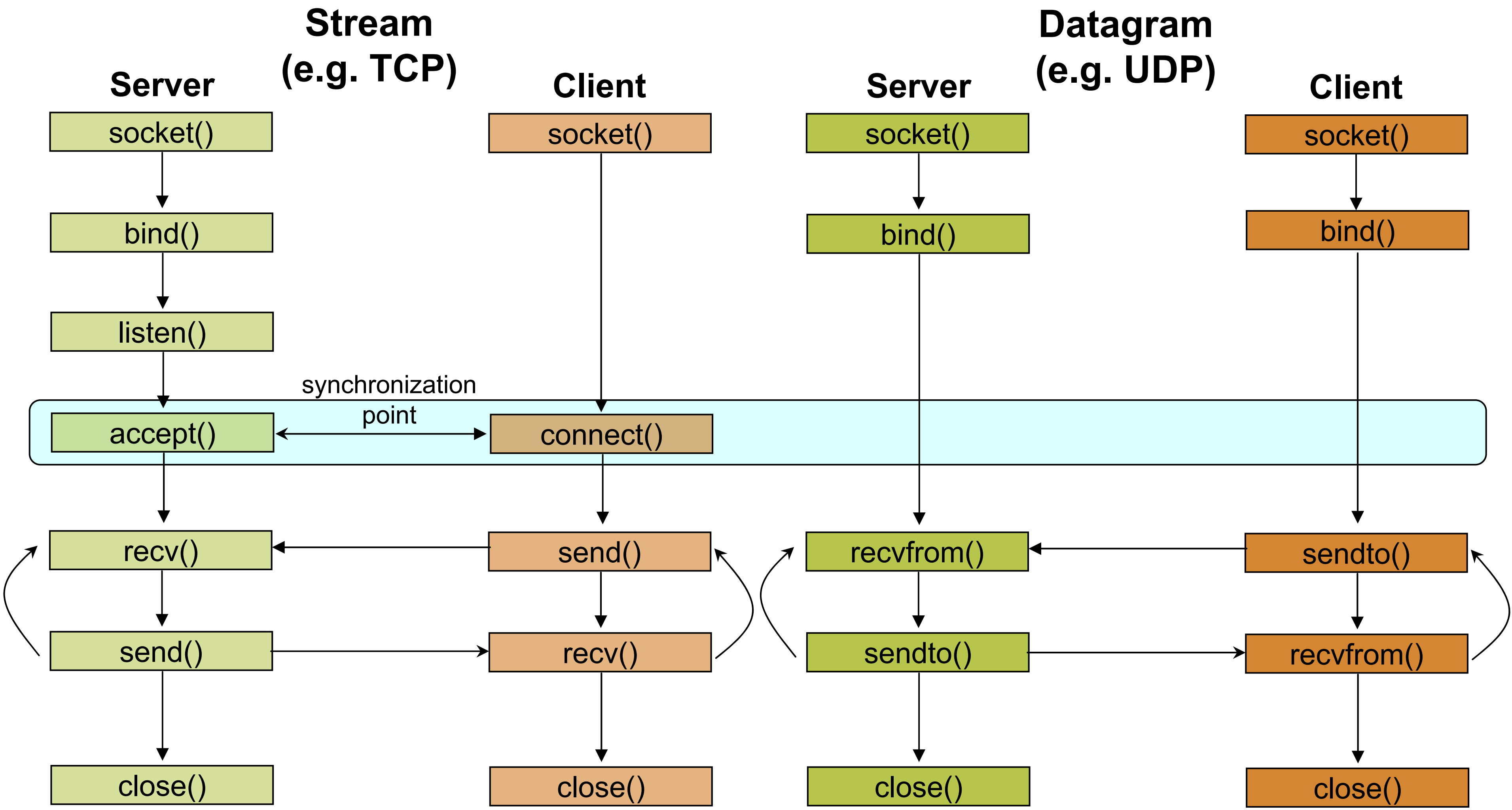
Assign address to socket: `bind()`

- 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 - Unix



Establish Connection: `connect()`

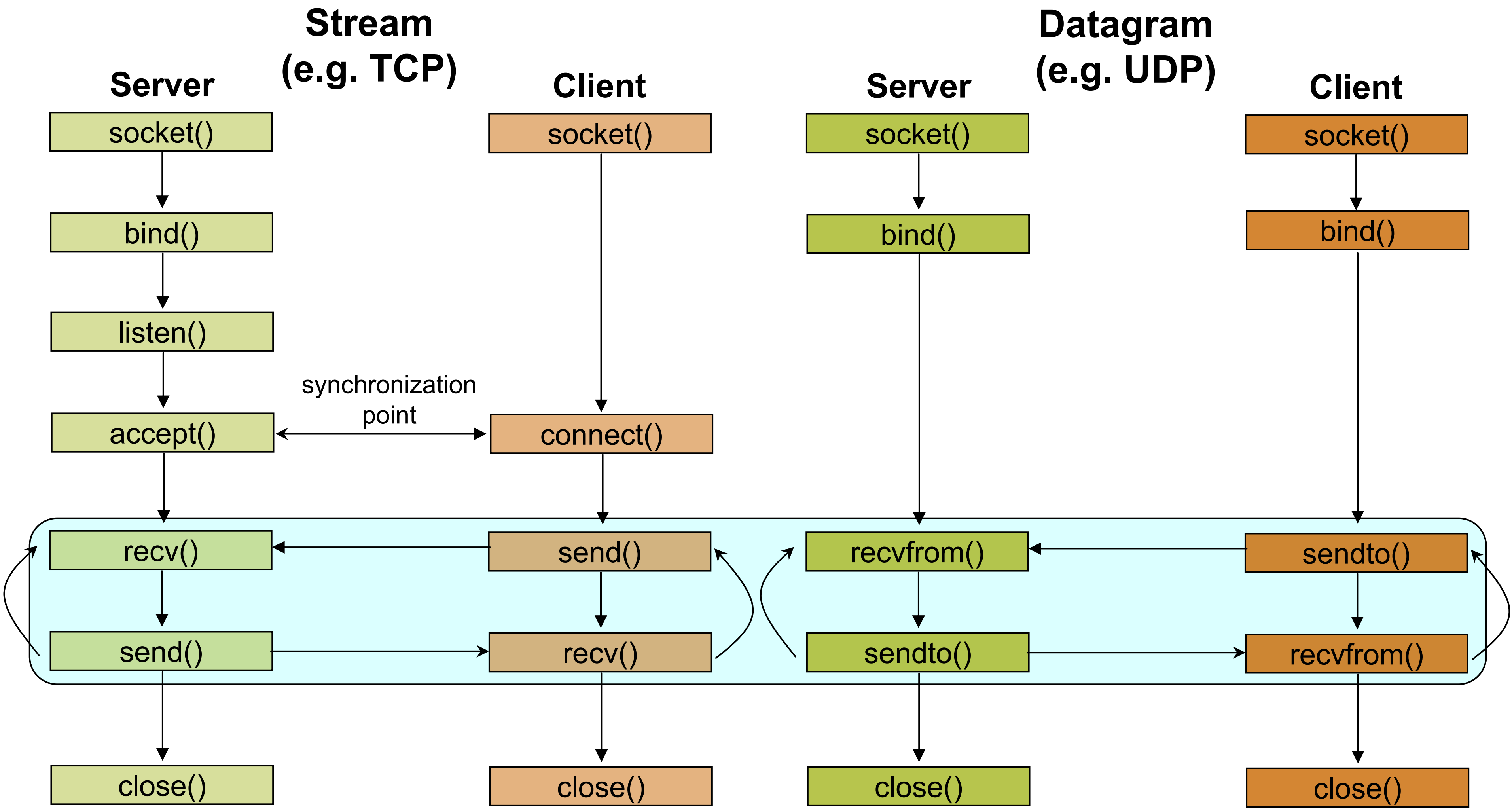
- The client establishes a connection with the server by calling `connect()`

- ```
  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 - Unix



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

```
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 in 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, RECVBUFSIZE, 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 …

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

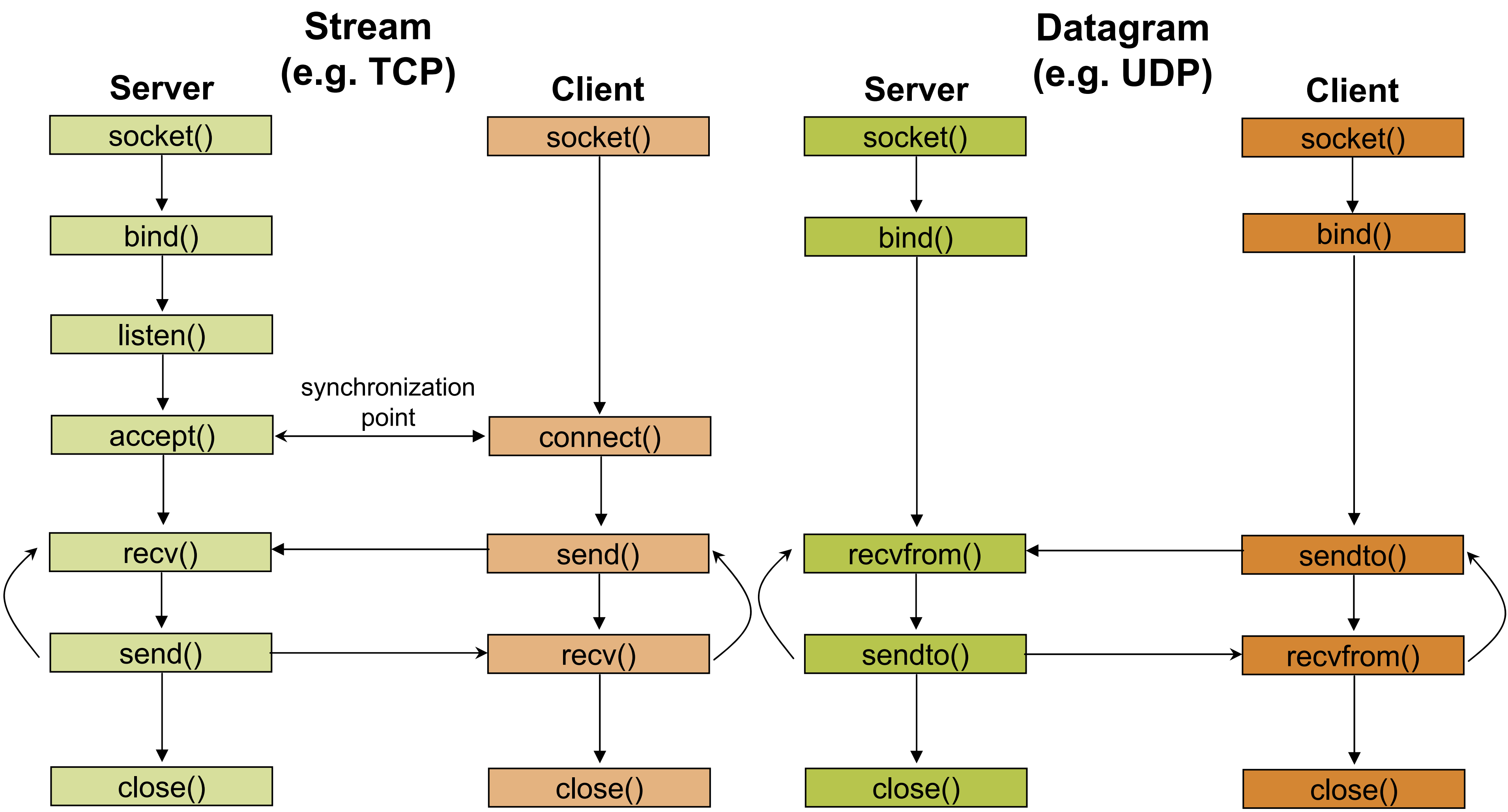
Client

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 - Unix



Constructing Messages - Encoding Data

- Client wants to send two integers `x` and `y` to server
- 1st Solution: **Character Encoding**
  - e.g. ASCII
  - the same representation is used to print or display them to screen
  - 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
  - each digit can be represented using 4 bits, instead of one byte
  - it is inconvenient to manipulate numbers
- 2nd Solution: Sending the values of `x` and `y`

Constructing Messages - Encoding Data

- 2nd 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)

1st Implementation

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

2nd Implementation

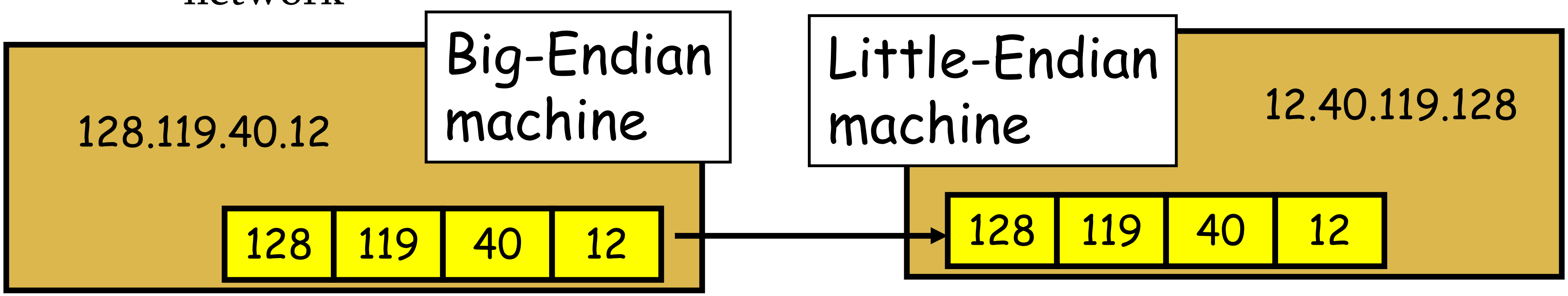
```
send(clientSock, &x, sizeof(x)), 0);
send(clientSock, &y, sizeof(y)), 0);
```

2nd implementation works in any case?

Constructing Messages - Byte Ordering

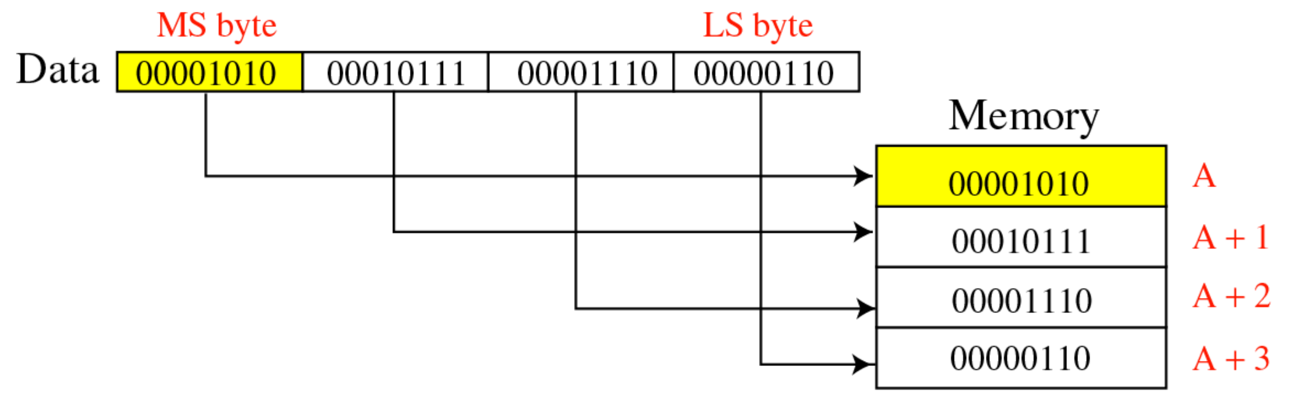
- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

- 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

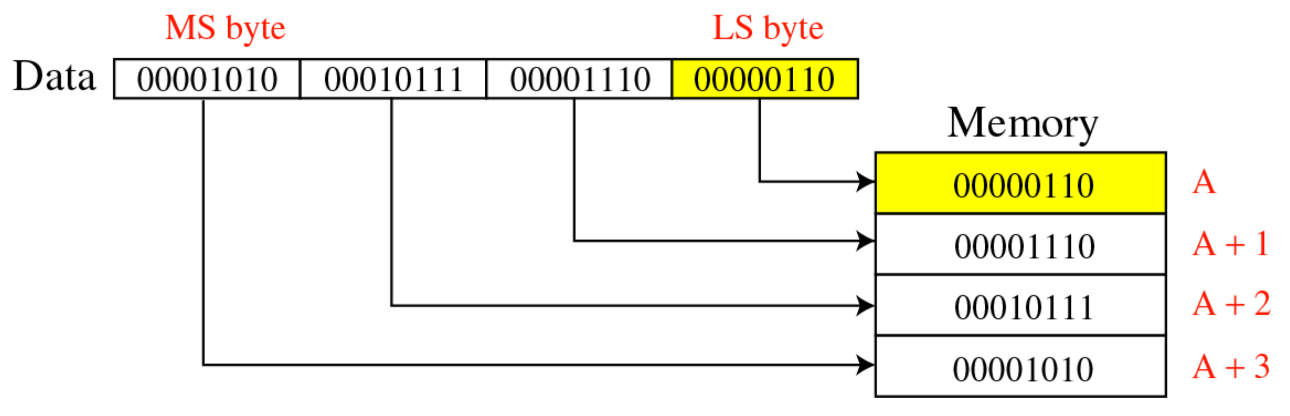


Constructing Messages - Byte Ordering

- Big-Endian:



- Little-Endian:

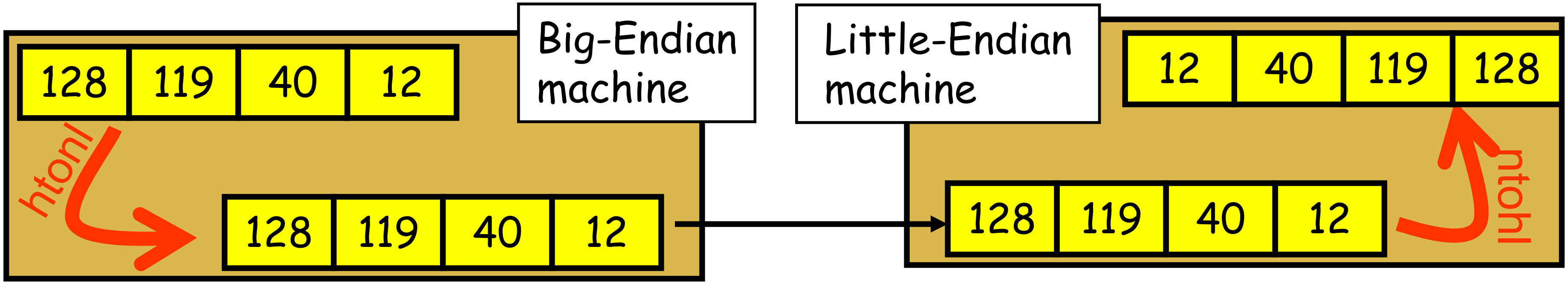


Constructing Messages - Byte Ordering -
Solution: Network Byte Ordering

- 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 ntohl(u_long x);
u_short ntohs(u_short x);
```

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



Constructing Messages - Byte Ordering - Example

Client

```
unsigned short clientPort, message;   unsigned int messageLenth;

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;
```

- 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

x	x2	y	y2
4 bytes	2 bytes	4 bytes	2 bytes

→

x	x2	[pad]	y	y2
4 bytes	2 bytes	2 bytes	4 bytes	2 bytes

- 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;
  int y;
  short x2;
  short y2;
} 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`
- 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`

Socket Options

- `getsockopt` and `setsockopt` allow socket options values to be queried and set, respectively
- **`int getsockopt (sockid, level, optName, optVal, optLen);`**
  - sockid: integer, socket descriptor
  - level: integer, the layers of the protocol stack (socket, TCP, IP)
  - optName: integer, option
  - optVal: pointer to a buffer; upon return it contains the value of the specified option
  - optLen: integer, in-out parameter

  it returns -1 if an error occured
- **`int setsockopt (sockid, level, optName, optVal, optLen);`**
  - optLen is now only an input parameter

Socket Options
- Table

optName	Type	Values	Description
SOL_SOCKET Level			
SO_BROADCAST	**int**	0,1	Broadcast allowed
SO_KEEPALIVE	**int**	0,1	Keepalive messages enabled (if implemented by the protocol)
SO_LINGER	**linger**{}	time	Time to delay `close()` return waiting for confirmation (see Section 6.4.2)
SO_RCVBUF	**int**	bytes	Bytes in the socket receive buffer (see code on page 44 and Section 6.1)
SO_RCVLOWAT	**int**	bytes	Minimum number of available bytes that will cause `recv()` to return
SO_REUSEADDR	**int**	0,1	Binding allowed (under certain conditions) to an address or port already in use (see Section 6.4 and 6.5)
SO_SNDLOWAT	**int**	bytes	Minimum bytes to send a packet
SO_SNDBUF	**int**	bytes	Bytes in the socket send buffer (see Section 6.1)
IPPROTO_TCP Level			
TCP_MAX	**int**	seconds	Seconds between keepalive messages.
TCP_NODELAY	**int**	0,1	Disallow delay for data merging (Nagle's algorithm)
IPPROTO_IP Level			
IP_TTL	**int**	0-255	Time-to-live for unicast IP packets
IP_MULTICAST_TTL	**unsigned char**	0-255	Time-to-live for multicast IP packets (see `MulticastSender.c` on page 81)
IP_MULTICAST_LOOP	**int**	0,1	Enables multicast socket to receive packets it sent
IP_ADD_MEMBERSHIP	**ip_mreq**{}	group address	Enables reception of packets addressed to the specified multicast group (see `MulticastReceiver.c` on page 83)—set only
IP_DROP_MEMBERSHIP	**ip_mreq**{}	group address	Disables reception of packets addressed to the specified multicast group—set only

Socket Options - Example

- Fetch and then double the current number of bytes in the socket's receive buffer

```
int rcvBufferSize;
int sockOptSize;
…
/* Retrieve and print the default buffer size */
sockOptSize = sizeof(recvBuffSize);
if (getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize, &sockOptSize) < 0)
   DieWithError("getsockopt() failed");
printf("Initial Receive Buffer Size: %d\n", rcvBufferSize);

/* Double the buffer size */
recvBufferSize *= 2;

/* Set the buffer size to new value */
if (setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize,
               sizeof(rcvBufferSize)) < 0)
 DieWithError("getsockopt() failed");
```

Dealing with blocking calls

- Many of the functions we saw block (by default) until a certain event
  - **accept**: until a connection comes in
  - **connect**: until the connection is established
  - **recv**, **recvfrom**: until a packet (of data) is received
    - what if a packet is lost (in datagram socket)?
  - **send**: until data are pushed into socket's buffer
  - **sendto**: until data are given to the network subsystem
- For **simple programs**, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Dealing with blocking calls

- Non-blocking Sockets
- Asynchronous I/O
- Timeouts

Non-blocking Sockets

- If an operation can be completed immediately, success is returned; otherwise, a failure is returned (usually -1)
  - `errno` is properly set, to distinguish this (blocking) failure from other - (`EINPROGRESS` for `connect`, `EWOULDBLOCK` for the other)
- 1st Solution: **`int fcntl (sockid, command, argument);`**
  - sockid: integer, socket descriptor
  - command: integer, the operation to be performed (**`F_GETFL`**, **`F_SETFL`**)
  - argument: long, e.g. **`O_NONBLOCK`**
  - ☞ **`fcntl (sockid, F_SETFL, O_NONBLOCK);`**
- 2nd Solution: flags parameter of `send`, `recv`, `sendto`, `recvfrom`
  - **`MSG_DONTWAIT`**
  - not supported by all implementations

Signals

- Provide a mechanism for operating system to notify processes that certain events occur
  - e.g., the user typed the “interrupt” character, or a timer expired
- signals are delivered **asynchronously**
- upon signal delivery to program
  - it may be ignored, the process is never aware of it
  - the program is forcefully terminated by the OS
  - a signal-handling routine, specified by the program, is executed
    - this happens in a different thread
  - the signal is blocked, until the program takes action to allow its delivery
    - each process (or thread) has a corresponding mask
- Each signal has a default behavior
  - e.g. SIGINT (i.e., Ctrl+C) causes termination
  - it can be changed using `sigaction()`
- Signals can be **nested** (i.e., while one is being handled another is delivered)

Signals

- `int sigaction(whichSignal, &newAction, &oldAction);`
  - whichSignal: integer
  - newAction: `struct sigaction`, defines the new behavior
  - oldAction: `struct sigaction`, if not NULL, then previous behavior is copied
  - it returns 0 on success, -1 otherwise

```
struct sigaction {
    void (*sa_handler)(int);  /* Signal handler */
    sigset_t sa_mask;         /* Signals to be blocked during handler execution */
    int sa_flags;             /* Flags to modify default behavior */
};
```

- `sa_handler` determines which of the first three possibilities occurs when signal is delivered, i.e., it is not masked
  - `SIG_IGN`, `SIG_DFL`, address of a function
- `sa_mask` specifies the signals to be blocked while handling whichSignal
  - whichSignal is always blocked
  - it is implemented as a set of boolean flags

```
int sigemptyset (sigset_t *set);  /* unset all the flags */
int sigfullset (sigset_t *set);   /* set all the flags */
int sigaddset(sigset_t *set, int whichSignal);  /* set individual flag */
int sigdelset(sigset_t *set, int whichSignal);  /* unset individual flag */
```

Signals - Example

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void DieWithError(char *errorMessage);
void InterruptSignalHandler(int signalType);

int main (int argc, char *argv[]) {
   struct sigaction handler;                      /* Signal handler specification structure */
   handler.sa_handler = InterruptSignalHandler;   /* Set handler function */
   if (sigfillset(&handler.sa_mask) < 0)          /* Create mask that masks all signals */
      DieWithError ("sigfillset() failed");
   handler.sa_flags = 0;
   if (sigaction(SIGINT, &handler, 0) < 0)        /* Set signal handling for interrupt signals */
      DieWithError ("sigaction() failed");
   for(;;) pause();                               /* Suspend program until signal received */
   exit(0);
}

void InterruptHandler (int signalType) {
   printf ("Interrupt received. Exiting program.\n);
   exit(1);
}
```

Asynchronous I/O

- Non-blocking sockets require "**polling**"
- With asynchronous I/O the **operating system informs** the program when a socket call is completed
  - the `SIGIO` signal is delivered to the process, when some I/O-related event occurs on the socket
- Three steps:

/* i. inform the system of the desired disposition of the signal */

```
struct sigaction handler;
handler.sa_handler = SIGIOHandler;
if (sigfillset(&handler.sa_mask) < 0) DiewithError("…");
handler.sa_flags = 0;
if (sigaction(SIGIO, &handler, 0) < 0) DieWithError("…");
```

/* ii. ensure that signals related to the socket will be delivered to this process */

```
if (fcntl(sock, F_SETOWN, getpid()) < 0) DieWithError();
```

/* iii. mark the socket as being primed for asynchronous I/O */

```
if (fcntl(sock, F_SETFL, O_NONBLOCK | FASYNC) < 0) DieWithError();
```

Timeouts

- Using asynchronous I/O the operating system informs the program for the occurrence of an I/O related event
  - what happens if a UPD packet is lost?
- We may need to know if something doesn't happen after some time
- **`unsigned int alarm (unsigned int secs);`**
  - starts a timer that expires after the specified number of seconds (**`secs`**)
  - returns
    - the number of seconds remaining until any previously scheduled alarm was due to be delivered,
    - or zero if there was no previously scheduled alarm
  - process receives **`SIGALARM`** signal when timer expires and `errno` is set to **`EINTR`**

Asynchronous I/O - Example

/* Inform the system of the desired disposition of the signal */

```
struct sigaction myAction;
myAction.sa_handler = CatchAlarm;
if (sigfillset(&myAction.sa_mask) < 0) DiewithError("…");
myAction.sa_flags = 0;
if (sigaction(SIGALARM, &handler, 0) < 0) DieWithError("…");
```

/* Set alarm */

```
alarm(TIMEOUT_SECS);
```

/* Call blocking receive */

```
if (recvfrom(sock, echoBuffer, ECHOMAX, 0, … ) < 0) {
    if (errno = EINTR) … /*Alarm went off */
    else DieWithError("recvfrom() failed");
}
```

Iterative Stream Socket Server

- Handles one client at a time
- Additional clients can connect while one is being served
  - connections are established
  - they are able to send requests

  but, the server will respond after it finishes with the first client
- Works well if each client required a small, bounded amount of work by the server
- otherwise, the clients experience long delays

Iterative Server - Example: echo using stream socket

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), connect(), recv() and send() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5    /* Maximum outstanding connection requests */

void DieWithError(char *errorMessage);  /* Error handling function */
void HandleTCPClient(int clntSocket);   /* TCP client handling function */

int main(int argc, char *argv[]) {
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;     /* Server port */
    unsigned int clntLen;            /* Length of client address data structure */

    if (argc != 2) {     /* Test for correct number of arguments */
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */

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

    ...
```

Iterative Server - Example: echo using stream socket

```
    ...

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    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 */

    /* Bind to the local address */
    if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

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

    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);

        /* Wait for a client to connect */
        if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
                               &clntLen)) < 0)
            DieWithError("accept() failed");

        /* clntSock is connected to a client! */

        printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

        HandleTCPClient(clntSock);
    }
    /* NOT REACHED */
}
```

Iterative Server - Example: echo using stream socket

```
#define RCVBUFSIZE 32   /* Size of receive buffer */

void HandleTCPClient(int clntSocket)
{
    char echoBuffer[RCVBUFSIZE];        /* Buffer for echo string */
    int recvMsgSize;                    /* Size of received message */

    /* 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 */
    {
        /* Echo message back to client */
        if (send(clntSocket, echoBuffer, recvMsgSize, 0) != recvMsgSize)
            DieWithError("send() failed");

        /* See if there is more data to receive */
        if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
            DieWithError("recv() failed");
    }

    close(clntSocket);    /* Close client socket */
}
```

Multitasking - Per-Client Process

- For each client connection request, a new process is created to handle the communication
- **`int fork();`**
  - a new process is created, identical to the calling process, except for its process ID and the return value it receives from `fork()`
  - returns 0 to child process, and the process ID of the new child to parent

Caution:

  - when a child process terminates, it does not automatically disappears
  - use `waitpid()` to parent in order to "harvest" zombies

Multitasking - Per-Client Process

- Example: echo using stream socket

```
#include <sys/wait.h>                    /* for waitpid() */

int main(int argc, char *argv[]) {
   int servSock;                         /* Socket descriptor for server */
   int clntSock;                         /* Socket descriptor for client */
   unsigned short echoServPort;          /* Server port */
   pid_t processID;                      /* Process ID from fork()*/
   unsigned int childProcCount = 0;      /* Number of child processes */

   if (argc != 2) {      /* Test for correct number of arguments */
      fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
      exit(1);
   }
   echoServPort = atoi(argv[1]);         /* First arg:  local port */

   servSock = CreateTCPServerSocket(echoServPort);

   for (;;) { /* Run forever */
      clntSock = AcceptTCPConnection(servSock);

      if ((processID = fork()) < 0) DieWithError (“fork() failed”);  /* Fork child process */
      else if (processID = 0) {          /* This is the child process */
         close(servSock);                /* child closes listening socket */
         HandleTCPClient(clntSock);
         exit(0);                        /* child process terminates */
      }

      close(clntSock);                   /* parent closes child socket */
      childProcCount++;                  /* Increment number of outstanding child processes */

      ...
```

Multitasking - Per-Client Process

- Example: echo using stream socket

```
        ...
        while (childProcCount) {                                    /* Clean up all zombies */
           processID = waitpid((pid_t) -1, NULL, WHOANG);           /* Non-blocking wait */
           if (processID < 0) DieWithError ("...");
           else if (processID == 0) break;                          /* No zombie to wait */
           else childProcCount--;                                   /* Cleaned up after a child */
        }
    }
    /* NOT REACHED */
}
```

Multitasking - Per-Client Thread

- Forking a new process is expensive
  - duplicate the entire state (memory, stack, file/socket descriptors, …)
- Threads decrease this cost by allowing multitasking within the same process
  - threads share the same address space (code and data)

An example is provided using POSIX Threads

Multitasking - Per-Client Thread

- Example: echo using stream socket

```
#include <pthread.h>                  /* for POSIX threads */

void *ThreadMain(void *arg)           /* Main program of a thread */

struct ThreadArgs {                   /* Structure of arguments to pass to client thread */
   int clntSock;                      /* socket descriptor for client */
};

int main(int argc, char *argv[]) {
   int servSock;                      /* Socket descriptor for server */
   int clntSock;                      /* Socket descriptor for client */
   unsigned short echoServPort;       /* Server port */
   pthread_t threadID;                /* Thread ID from pthread_create()*/
   struct ThreadArgs *threadArgs;     /* Pointer to argument structure for thread */

   if (argc != 2) {       /* Test for correct number of arguments */
      fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
      exit(1);
   }
   echoServPort = atoi(argv[1]);      /* First arg:  local port */

   servSock = CreateTCPServerSocket(echoServPort);

   for (;;) { /* Run forever */
      clntSock = AcceptTCPConnection(servSock);

      /* Create separate memory for client argument */
      if ((threadArgs = (struct ThreadArgs *) malloc(sizeof(struct ThreadArgs)))) == NULL) DieWithError("…");
      threadArgs -> clntSock = clntSock;

      /* Create client thread */
      if (pthread_create (&threadID, NULL, ThreadMain, (void *) threadArgs) != 0) DieWithError("…");
   }
   /* NOT REACHED */
}
```

Multitasking - Per-Client Thread

- Example: echo using stream socket

```
void *ThreadMain(void *threadArgs)
{
    int clntSock;                     /* Socket descriptor for client connection */

    pthread_detach(pthread_self()); /* Guarantees that thread resources are deallocated upon return */

    /* Extract socket file descriptor from argument */
    clntSock = ((struct ThreadArgs *) threadArgs) -> clntSock;
    free(threadArgs);                /* Deallocate memory for argument */

    HandleTCPClient(clntSock);

    return (NULL);
}
```

Multitasking - Constrained

- Both process and thread incurs **overhead**
  - creation, scheduling and context switching
- As their numbers increases
  - this overhead increases
  - after some point it would be better if a client was blocked
- Solution: **Constrained multitasking**. The server:
  - begins, creating, binding and listening to a socket
  - creates a number of processes, each loops forever and accept connections from the same socket
  - when a connection is established
    - the client socket descriptor is returned to only one process
    - the other remain blocked

Multitasking - Constrained

- Example: echo using stream socket

```
void ProcessMain(int servSock);      /* Main program of process */

int main(int argc, char *argv[]) {
    int servSock;                    /* Socket descriptor for server*/
    unsigned short echoServPort;     /* Server port */
    pid_t processID;                 /* Process ID */
    unsigned int processLimit;       /* Number of child processes to create */
    unsigned int processCt;          /* Process counter */

    if (argc != 3) {      /* Test for correct number of arguments */
        fprintf(stderr,"Usage:  %s <SERVER PORT> <FORK LIMIT>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg: local port */
    processLimit = atoi(argv[2]);  /* Second arg:  number of child processes */

    servSock = CreateTCPServerSocket(echoServPort);

    for (processCt=0; processCt < processLimit; processCt++)
        if ((processID = fork()) < 0) DieWithError("fork() failed");    /* Fork child process */
        else if (processID == 0) ProcessMain(servSock);                 /* If this is the child process */

    exit(0);  /* The children will carry on */
}

void ProcessMain(int servSock) {
    int clntSock;                  /* Socket descriptor for client connection */

    for (;;) { /* Run forever */
        clntSock = AcceptTCPConnection(servSock);
        printf("with child process: %d\n", (unsigned int) getpid());
        HandleTCPClient(clntSock);
    }
}
```

Multiplexing

- So far, we have dealt with a **single** I/O channel
- We may need to cope with **multiple** I/O channels
  - e.g., supporting the echo service over multiple ports
- **Problem**: from which socket the server should accept connections or receive messages?
  - it can be solved using non-blocking sockets
    - but it requires polling
- **Solution**: `select()`
  - specifies a list of descriptors to check for pending I/O operations
  - blocks until one of the descriptors is ready
  - returns which descriptors are ready

Multiplexing

- `int select (maxDescPlus1, &readDescs, &writeDescs, &exceptionDescs, &timeout);`
  - maxDescsPlus1: integer, hint of the maximum number of descriptors
  - readDescs: `fd_set`, checked for immediate input availability
  - writeDescs: `fd_set`, checked for the ability to immediately write data
  - exceptionDescs: `fd_set`, checked for pending exceptions
  - timeout: `struct timeval`, how long it blocks (NULL → forever)

  \- **returns** the total number of ready descriptors, -1 in case of error

  \- **changes** the descriptor lists so that only the corresponding positions are set

```
int FD_ZERO  (fd_set *descriptorVector);                   /* removes all descriptors from vector */
int FD_CLR   (int descriptor, fd_set *descriptorVector);   /* remove descriptor from vector */
int FD_SET   (int descriptor, fd_set *descriptorVector);   /* add descriptor to vector */
int FD_ISSET (int descriptor, fd_set *descriptorVector);   /* vector membership check */
```

```
struct timeval {
   time_t tv_sec;    /* seconds */
   time_t tv_usec;   /* microseconds */
};
```

Multiplexing - Example: echo using stream socket

```
#include <sys/time.h>       /* for struct timeval {} */

int main(int argc, char *argv[])
{
    int *servSock;                    /* Socket descriptors for server */
    int maxDescriptor;                /* Maximum socket descriptor value */
    fd_set sockSet;                   /* Set of socket descriptors for select() */
    long timeout;                     /* Timeout value given on command-line */
    struct timeval selTimeout;        /* Timeout for select() */
    int running = 1;                  /* 1 if server should be running; 0 otherwise */
    int noPorts;                      /* Number of port specified on command-line */
    int port;                         /* Looping variable for ports */
    unsigned short portNo;            /* Actual port number */

    if (argc < 3) {     /* Test for correct number of arguments */
        fprintf(stderr, "Usage:  %s <Timeout (secs.)> <Port 1> ...\n", argv[0]);
        exit(1);
    }

    timeout = atol(argv[1]);         /* First arg: Timeout */
    noPorts = argc - 2;              /* Number of ports is argument count minus 2 */

    servSock = (int *) malloc(noPorts * sizeof(int)); /* Allocate list of sockets for incoming connections */
    maxDescriptor = -1;                               /* Initialize maxDescriptor for use by select() */

    for (port = 0; port < noPorts; port++) {          /* Create list of ports and sockets to handle ports */
        portNo = atoi(argv[port + 2]);                /* Add port to port list. Skip first two arguments */
        servSock[port] = CreateTCPServerSocket(portNo); /* Create port socket */

        if (servSock[port] > maxDescriptor)           /* Determine if new descriptor is the largest */
            maxDescriptor = servSock[port];
    }
    ...
```

Multiplexing - Example: echo using stream socket

```
    printf("Starting server:  Hit return to shutdown\n");
    while (running) {
        /* Zero socket descriptor vector and set for server sockets */
        /* This must be reset every time select() is called */
        FD_ZERO(&sockSet);
        FD_SET(STDIN_FILENO, &sockSet); /* Add keyboard to descriptor vector */
        for (port = 0; port < noPorts; port++) FD_SET(servSock[port], &sockSet);

        /* Timeout specification */
        /* This must be reset every time select() is called */
        selTimeout.tv_sec = timeout;        /* timeout (secs.) */
        selTimeout.tv_usec = 0;             /* 0 microseconds */

        /* Suspend program until descriptor is ready or timeout */
        if (select(maxDescriptor + 1, &sockSet, NULL, NULL, &selTimeout) == 0)
            printf("No echo requests for %ld secs...Server still alive\n", timeout);
        else  {
            if (FD_ISSET(0, &sockSet)) { /* Check keyboard */
                printf("Shutting down server\n");
                getchar();
                running = 0;
            }
            for (port = 0; port < noPorts; port++)
                if (FD_ISSET(servSock[port], &sockSet)) {
                    printf("Request on port %d:  ", port);
                    HandleTCPClient(AcceptTCPConnection(servSock[port]));
                }
        }
    }
    for (port = 0; port < noPorts; port++) close(servSock[port]);  /* Close sockets */
    free(servSock);                                                 /* Free list of sockets */
    exit(0);
}
```

Multiple Recipients

- So far, all sockets have dealt with **unicast** communication
  - i.e., an one-to-one communication, where one copy ("**uni**") of the data is sent ("**cast**")
- what if we want to send data to multiple recipients?
- 1^{st} **Solution**: unicast a copy of the data to each recipient
  - inefficient, e.g.,
    - consider we are connected to the internet through a 3Mbps line
    - a video server sends 1-Mbps streams
    - then, server can support only three clients simultaneously
- 2^{nd} **Solution**: using network support
  - **broadcast**, all the hosts of the network receive the message
  - **multicast**, a message is sent to some subset of the host
  - for IP: only **UDP sockets** are allowed to broadcast and multicast

Multiple Recipients - Broadcast

- Only the IP address changes
- Local broadcast: to address `255.255.255.255`
  - send the message to every host on the same broadcast network
  - not forwarded by the routers
- Directed broadcast:
  - for network identifier `169.125` (i.e., with subnet mask `255.255.0.0`)
  - the directed broadcast address is `169.125.255.255`
- No network-wide broadcast address is available
  - why?

✍ In order to use broadcast the options of socket must change:

```
int broadcastPermission = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (void*)
   &broadcastPermission, sizeof(broadcastPermission));
```

Multiple Recipients - Multicast

- Using class D addresses
  - range from `224.0.0.0` to `239.255.255.255`
- hosts send multicast requests for specific addresses
- a multicast group is formed

✍ we need to set TTL (time-to-live), to limit the number of hops
- using `sockopt()`

✍ no need to change the options of socket

Useful Functions

- `int atoi(const char *nptr);`
  - converts the initial portion of the string pointed to by `nptr` to int
- `int inet_aton(const char *cp, struct in_addr *inp);`
  - converts the Internet host address `cp` from the IPv4 numbers-and-dots notation into binary form (in network byte order)
  - stores it in the structure that `inp` points to.
  - it returns nonzero if the address is valid, and 0 if not
- `char *inet_ntoa(struct in_addr in);`
  - converts the Internet host address `in`, given in network byte order, to a string in IPv4 dotted-decimal notation

```
typedef uint32_t in_addr_t;

struct in_addr {
   in_addr_t s_addr;
};
```

Useful Functions

- `int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
  - returns the address (IP and port) of the peer connected to the socket `sockfd`, in the buffer pointed to by `addr`
  - 0 is returned on success; -1 otherwise

- `int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
  - returns the current address to which the socket `sockfd` is bound, in the buffer pointed to by `addr`
  - 0 is returned on success; -1 otherwise

Domain Name Service

- **struct hostent *gethostbyname(const char *name);**
  - returns a structure of type `hostent` for the given host name
  - `name` is a hostname, or an IPv4 address in standard dot notation

  e.g. **gethostbyname("www.csd.uoc.gr");**

- **struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);**
  - returns a structure of type hostent for the given host address addr of length len and address type `type`

```
struct hostent {
   char   *h_name;        /* official name of host */
   char **h_aliases;      /* alias list (strings) */
   int     h_addrtype;    /* host address type (AF_INET) */
   int     h_length;      /* length of address */
   char **h_addr_list;    /* list of addresses (binary in network byte order) */
}

#define h_addr h_addr_list[0]  /* for backward compatibility */
```

Domain Name Service

- **`struct servent *getservbyname(const char *name, const char *proto);`**
  - returns a `servent` structure for the entry from the database that matches the service name using protocol `proto`.
  - if `proto` is NULL, any protocol will be matched.

  e.g. **`getservbyname("echo", "tcp");`**

- **`struct servent *getservbyport(int port, const char *proto);`**
  - returns a `servent` structure for the entry from the database that matches the service name using port `port`

```
struct servent {
   char   *s_name;        /* official service name */
   char  **s_aliases;     /* list of alternate names (strings)*/
   int     s_port;        /* service port number */
   char   *s_proto;       /* protocol to use ("tcp" or "udp")*/
}
```

Compiling and Executing

- include the required header files
- Example:

```
milo:~/CS556/sockets> gcc -o TCPEchoServer TCPEchoServer.c DieWithError.c HandleTCPClient.c
milo:~/CS556/sockets> gcc -o TCPEchoClient TCPEchoClient.c DieWithError.c
milo:~/CS556/sockets> TCPEchoServer 3451 &
[1] 6273
milo:~/CS556/sockets> TCPEchoClient 0.0.0.0 hello! 3451
Handling client 127.0.0.1
Received: hello!
milo:~/CS556/sockets> ps
  PID TTY          TIME CMD
 5128 pts/9    00:00:00 tcsh
 6273 pts/9    00:00:00 TCPEchoServer
 6279 pts/9    00:00:00 ps
milo:~/CS556/sockets> kill 6273
milo:~/CS556/sockets>
[1]    Terminated                   TCPEchoServer 3451
milo:~/CS556/sockets>
```

The End - Questions