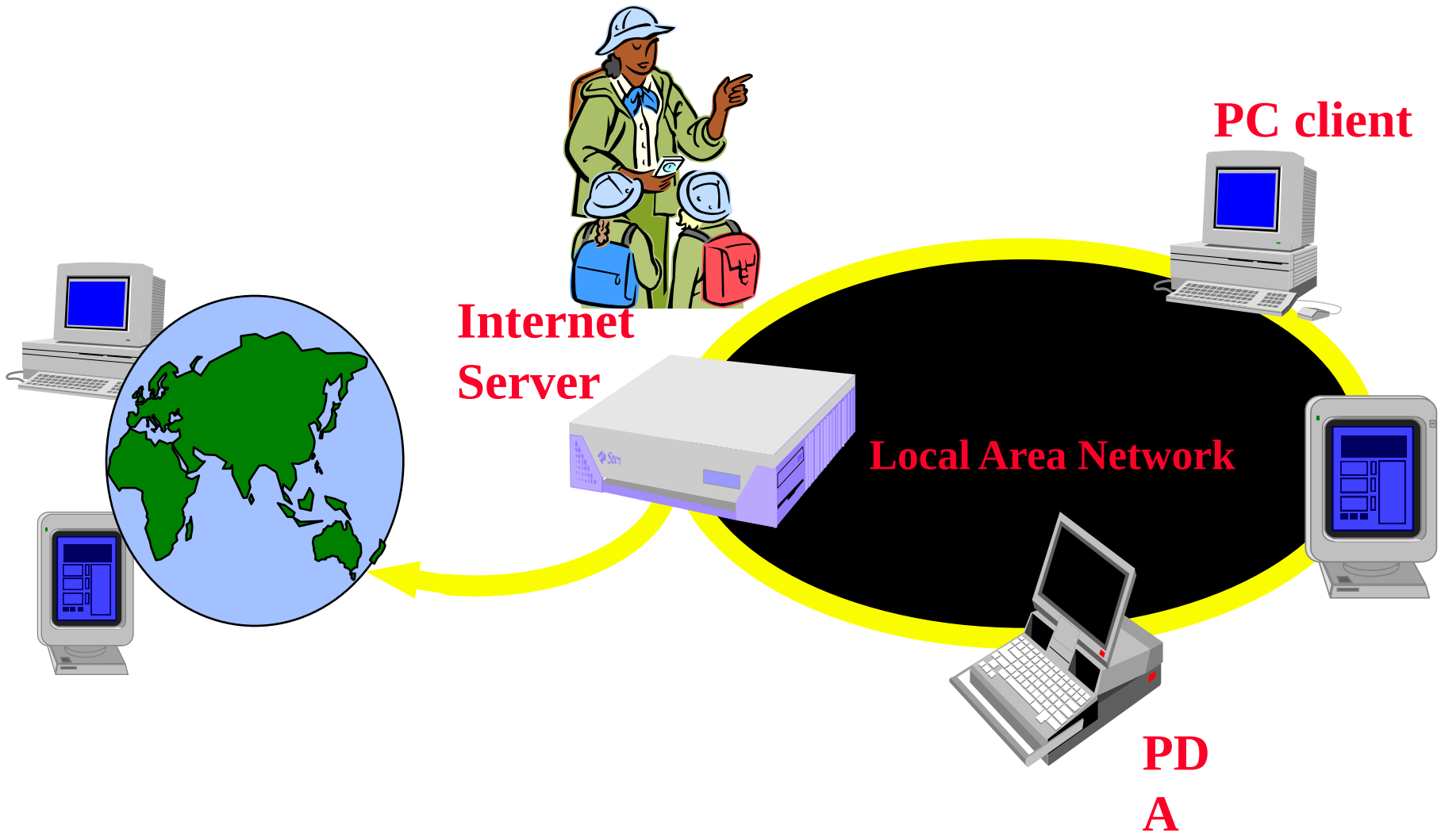
Agenda

- Introduction
- Elements of Client Server Computing
- Networking Basics
- Understanding Ports and Sockets
- Java Sockets
  - Implementing a Server
  - Implementing a Client
- Sample Examples
- Conclusions

Introduction

- Internet and WWW have emerged as global ubiquitous media for communication and changing the way we conduct science, engineering, and commerce.
- They also changing the way we learn, live, enjoy, communicate, interact, engage, etc. It appears like the modern life activities are getting completely centered around the Internet.

Internet Applications Serving Local and Remote Users

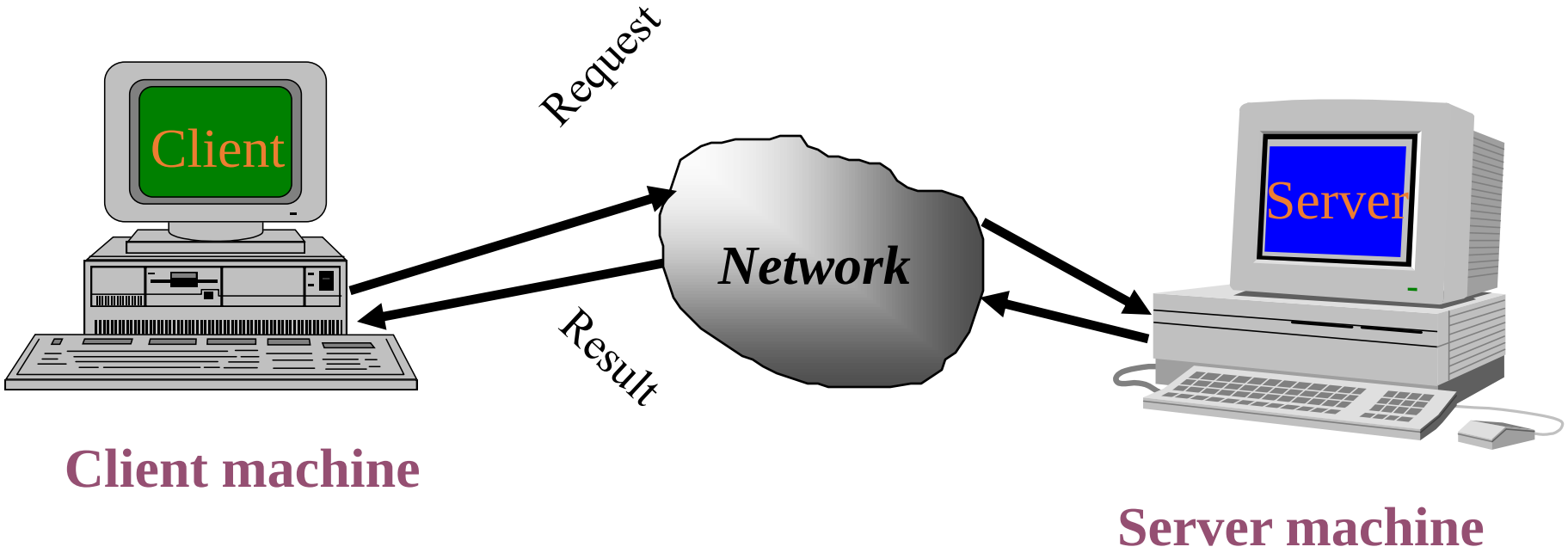


Increased demand for Internet applications

- To take advantage of opportunities presented by the Internet, businesses are continuously seeking new and innovative ways and means for offering their services via the Internet.
- This created a huge demand for software designers with skills to create new Internet-enabled applications or migrate existing/legacy applications on the Internet platform.
- Object-oriented Java technologies—Sockets, threads, RMI, clustering, Web services-- have emerged as leading solutions for creating portable, efficient, and maintainable large and complex Internet applications.

Elements of C-S Computing

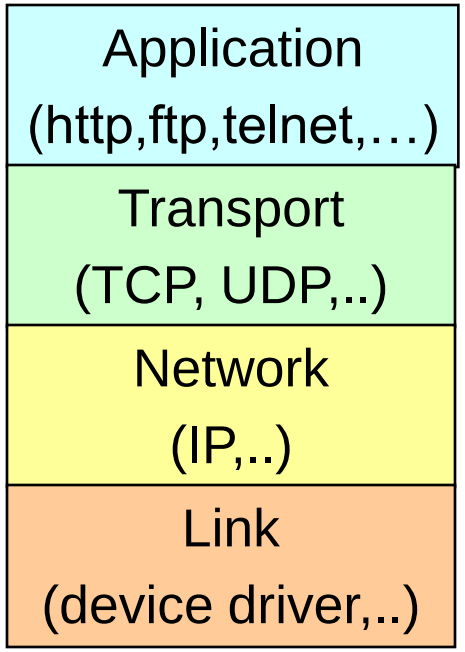
a client, a server, and network



Networking Basics

- Applications Layer
  - Standard apps
    - HTTP
    - FTP
    - Telnet
  - User apps
- Transport Layer
  - TCP
  - UDP
  - Programming Interface:
    - Sockets
- Network Layer
  - IP
- Link Layer
  - Device drivers

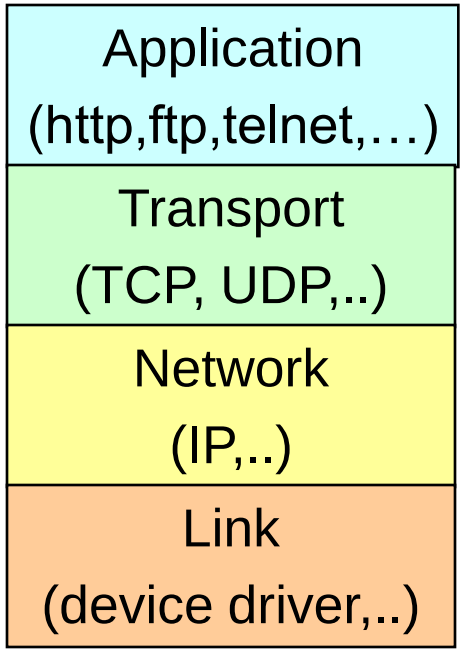
- TCP/IP Stack



Networking Basics

- TCP (Transport Control Protocol) is a connection-oriented protocol that provides a reliable flow of data between two computers.
- Example applications:
  - HTTP
  - FTP
  - Telnet

- TCP/IP Stack

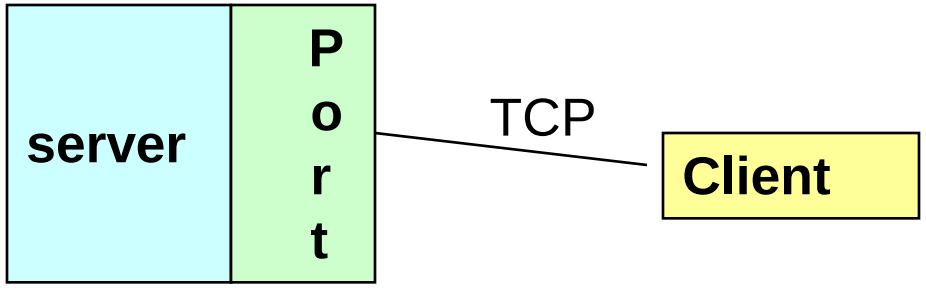


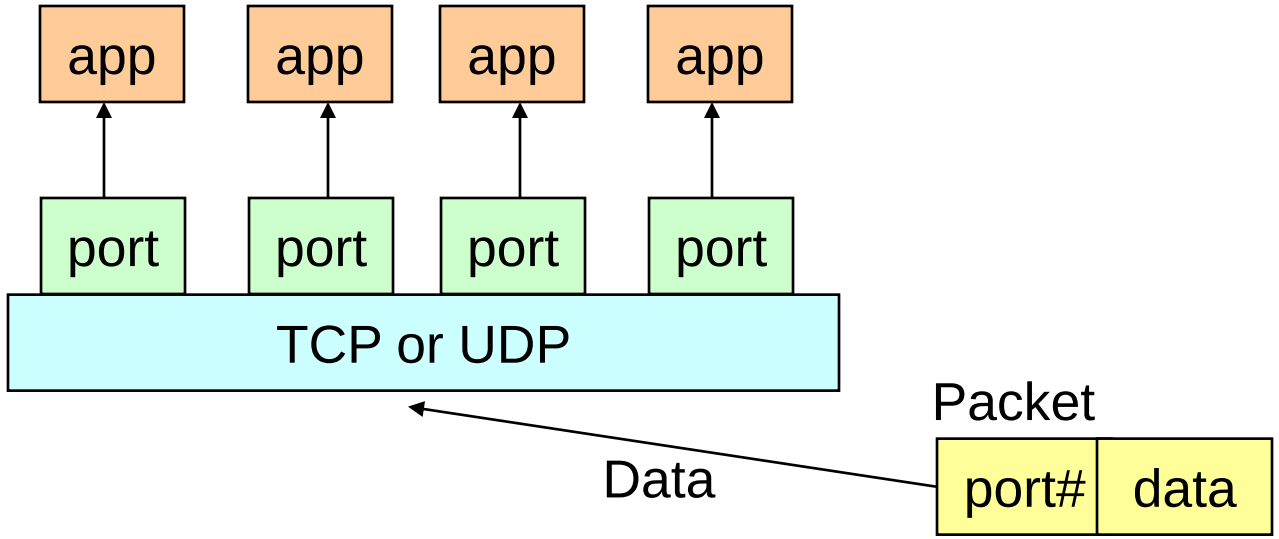
Networking Basics

- UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called *datagrams*, from one computer to another with no guarantees about arrival.
- Example applications:
  - Clock server
  - Ping

Understanding Ports

- The TCP and UDP protocols use *ports* to map incoming data to a particular *process* running on a computer.



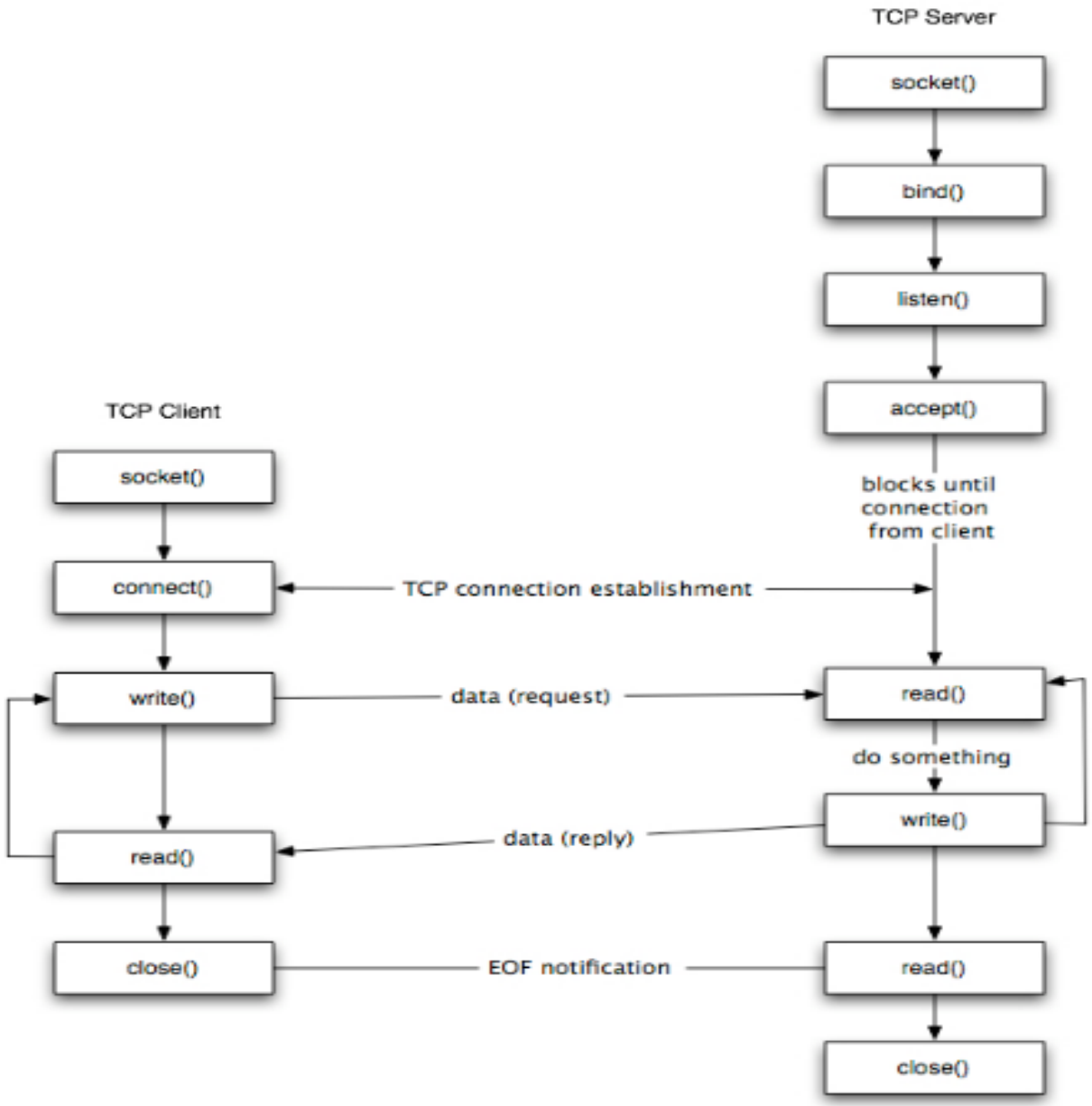


Understanding Ports

- Port is represented by a **positive (16-bit) integer value**
- Some ports have been reserved to support common/well known services:
  - ftp 21/tcp
  - telnet 23/tcp
  - smtp 25/tcp
  - login 513/tcp
- User level process/services generally use port number value >= 1024

Sockets

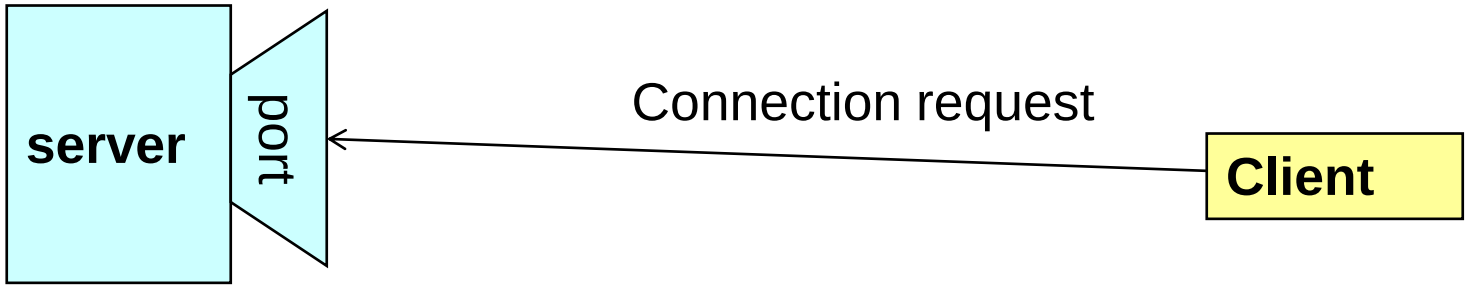
- Sockets provide an **interface** for programming networks at the transport layer.
- Network communication using Sockets is very much **similar to performing file I/O**
  - In fact, socket handle is treated like file handle.
  - The streams used in file I/O operation are also applicable to socket-based I/O
- Socket-based communication is **programming language independent**.
  - That means, a socket program written in Java language can also communicate to a program written in Java or non-Java socket program.



Communication Diagram

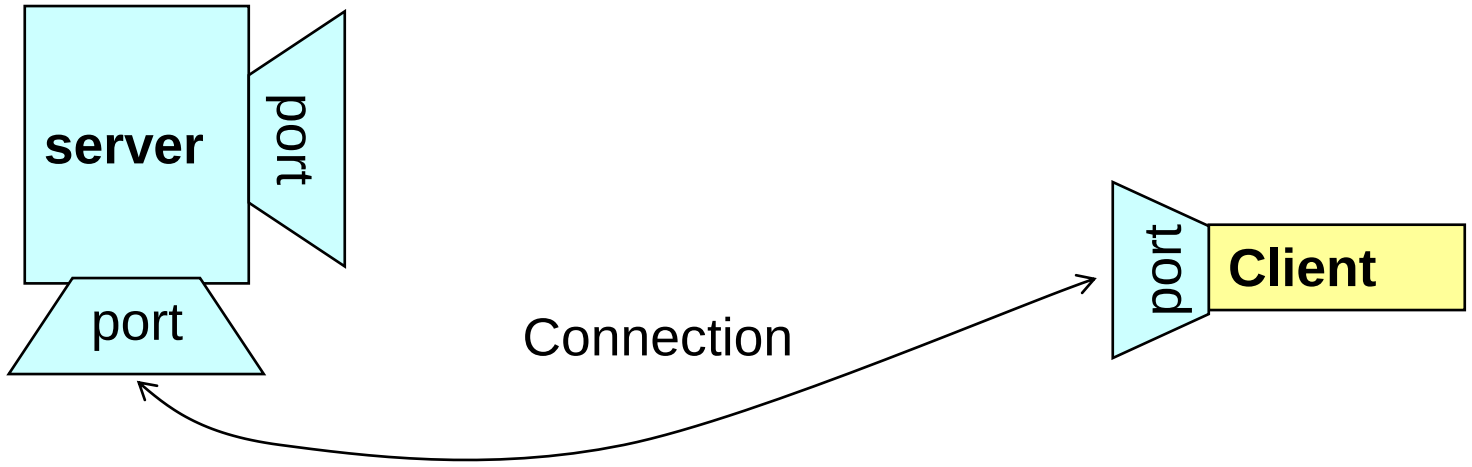
Socket Communication

- A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server waits and listens to the socket for a client to make a connection request.



Socket Communication

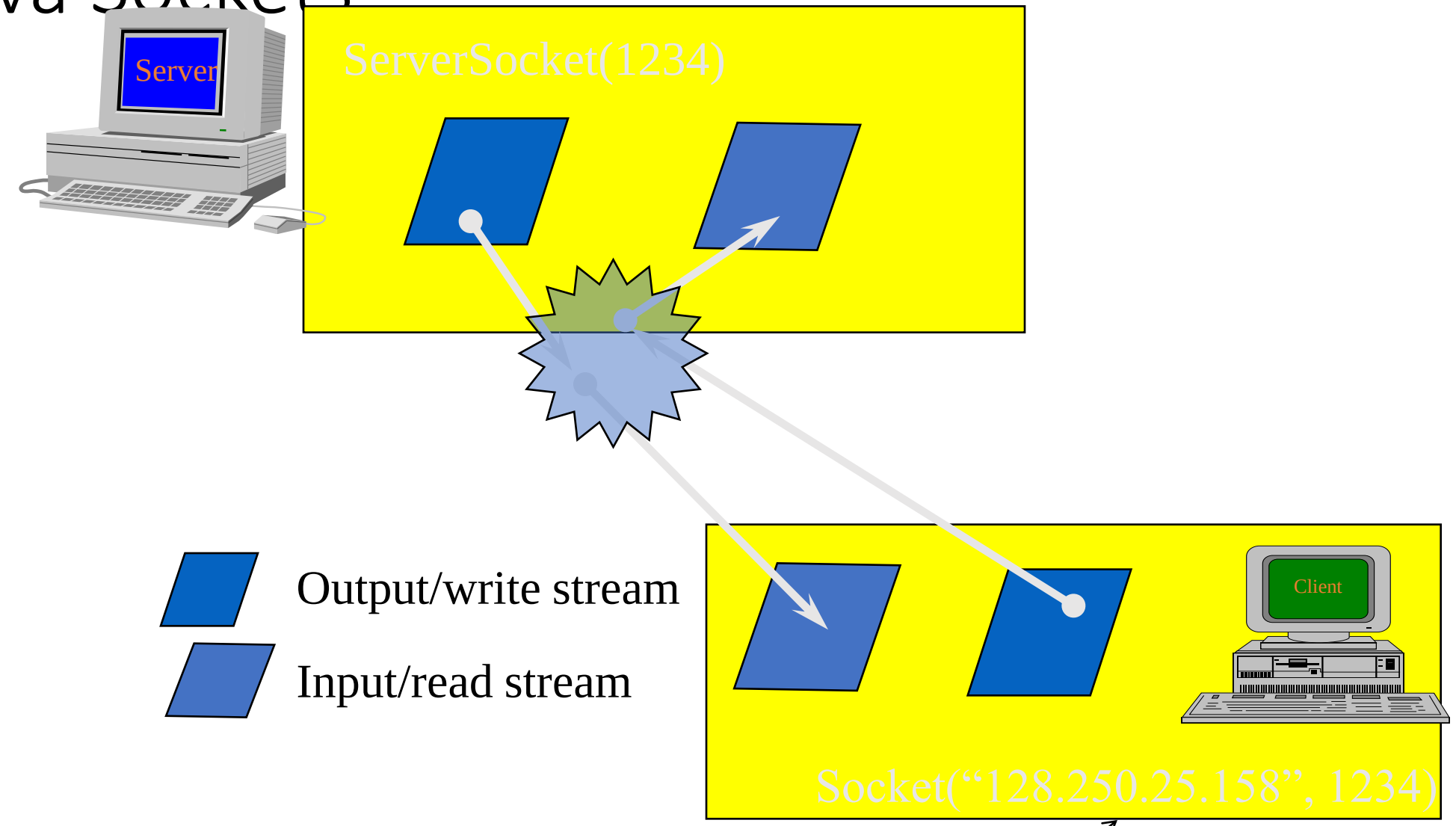
- If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bounds to a different port. It needs a new socket (consequently a different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client.



Sockets and Java Socket Classes

- A socket is an endpoint of a two-way communication link between two programs running on the network.
- A socket is bound to a port number so that the TCP layer can identify the application that data destined to be sent.
- Java's .net package provides two classes:
  - Socket – for implementing a client
  - ServerSocket – for implementing a server

Java Sockets



It can be host_name like "mandroo.cs.mu.oz.au"

Implementing a Server

1. Open the Server Socket:

```
ServerSocket server;
DataOutputStream os;
DataInputStream is;
server = new ServerSocket( PORT );
```

2. Wait for the Client Request:

```
Socket client = server.accept();
```

3. Create I/O streams for communicating to the client

```
is = new DataInputStream( client.getInputStream() );
os = new DataOutputStream( client.getOutputStream() );
```

4. Perform communication with client

```
Receive from client: String line = is.readLine();
Send to client: os.writeBytes("Hello\n");
```

5. Close sockets: `client.close();`

For multithreaded server:

```
while(true) {
```

i. wait for client requests (step 2 above)

ii. create a thread with "client" socket as parameter (the thread creates streams (as in step (3) and does communication as stated in (4). Remove thread once service is provided.

```
}
```

Implementing a Client

1. Create a Socket Object:

```
client = new Socket( server, port_id );
```

2. Create I/O streams for communicating with the server.

```
is = new DataInputStream(client.getInputStream() );
os = new DataOutputStream( client.getOutputStream() );
```

3. Perform I/O or communication with the server:
   - Receive data from the server:

```
String line = is.readLine();
```

   - Send data to the server:

```
os.writeBytes("Hello\n");
```

4. Close the socket when done:

```
client.close();
```

A simple server (simplified code)

```
// SimpleServer.java: a simple server program
import java.net.*;
import java.io.*;
public class SimpleServer {
  public static void main(String args[]) throws IOException {
    // Register service on port 1234
    ServerSocket s = new ServerSocket(1234);
    Socket s1=s.accept(); // Wait and accept a connection
    // Get a communication stream associated with the socket
    OutputStream s1out = s1.getOutputStream();
    DataOutputStream dos = new DataOutputStream (s1out);
    // Send a string!
    dos.writeUTF("Hi there");
    // Close the connection, but not the server socket
    dos.close();
    s1out.close();
    s1.close();
  }
}
```

A simple client (simplified code)

```
// SimpleClient.java: a simple client program
import java.net.*;
import java.io.*;
public class SimpleClient {
  public static void main(String args[]) throws IOException {
    // Open your connection to a server, at port 1234
    Socket s1 = new Socket("mundroo.cs.mu.oz.au",1234);
    // Get an input file handle from the socket and read the input
    InputStream s1In = s1.getInputStream();
    DataInputStream dis = new DataInputStream(s1In);
    String st = new String (dis.readUTF());
    System.out.println(st);
    // When done, just close the connection and exit
    dis.close();
    s1In.close();
    s1.close();
  }
}
```

Run

- Run Server on mundroo.cs.mu.oz.au
  - [raj@mundroo] java SimpleServer &

- Run Client on any machine (including mundroo):
  - [raj@mundroo] java SimpleClient
    Hi there

- If you run client when server is not up:
  - [raj@mundroo] sockets [1:147] java SimpleClient

```
Exception in thread "main" java.net.ConnectException: Connection refused
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:320)
        at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:133)
        at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:120)
        at java.net.Socket.<init>(Socket.java:273)
        at java.net.Socket.<init>(Socket.java:100)
        at SimpleClient.main(SimpleClient.java:6)
```

Socket Exceptions

```
try {
   Socket client = new Socket(host, port); handleConnection(client);
}
catch(UnknownHostException uhe) { System.out.println("Unknown host: " + host);
   uhe.printStackTrace();
}
catch(IOException ioe) {
System.out.println("IOException: " + ioe); ioe.printStackTrace();
}
```

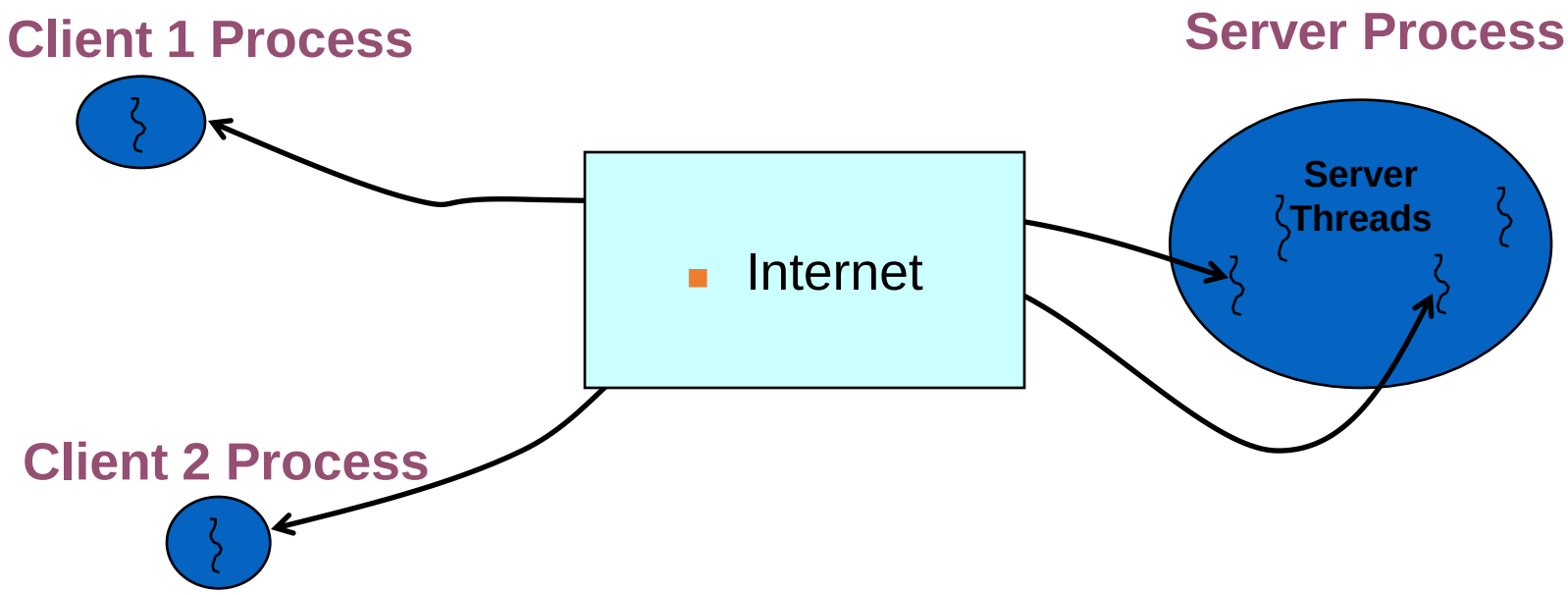
ServerSocket & Exceptions

- public **ServerSocket**(int port) throws IOException
  - Creates a server socket on a specified port.
  - A port of 0 creates a socket on any free port. You can use **getLocalPort**() to identify the (assigned) port on which this socket is listening.
  - The maximum queue length for incoming connection indications (a request to connect) is set to 50. If a connection indication arrives when the queue is full, the connection is refused.
- Throws:
  - IOException - if an I/O error occurs when opening the socket.
  - SecurityException - if a security manager exists and its checkListen method doesn't allow the operation.

Server in Loop: Always up

```
// SimpleServerLoop.java: a simple server program that runs forever in a single thead
import java.net.*;
import java.io.*;
public class SimpleServerLoop {
  public static void main(String args[]) throws IOException {
    // Register service on port 1234
    ServerSocket s = new ServerSocket(1234);
    while(true)
    {
          Socket s1=s.accept(); // Wait and accept a connection
          // Get a communication stream associated with the socket
          OutputStream s1out = s1.getOutputStream();
          DataOutputStream dos = new DataOutputStream (s1out);
          // Send a string!
          dos.writeUTF("Hi there");
          // Close the connection, but not the server socket
          dos.close();
          s1out.close();
          s1.close();
    }
  }
}
```

Multithreaded Server: For Serving Multiple Clients Concurrently



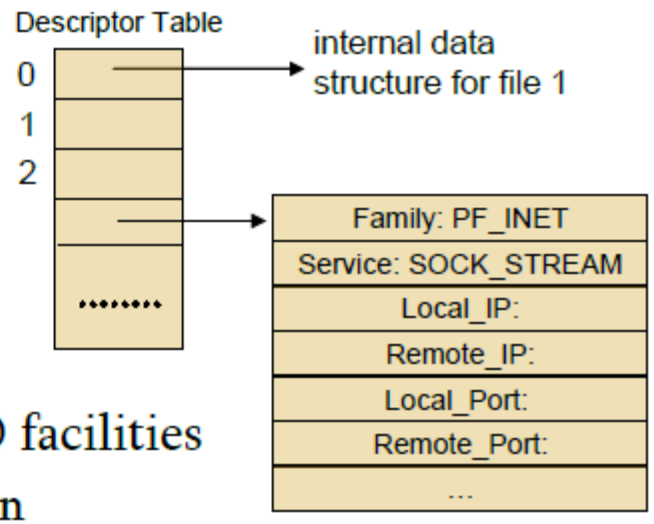
Conclusion

- Programming client/server applications in Java is fun and challenging.
- Programming socket programming in Java is much easier than doing it in other languages such as C.
- Keywords:
  - Clients, servers, TCP/IP, port number, sockets, Java sockets

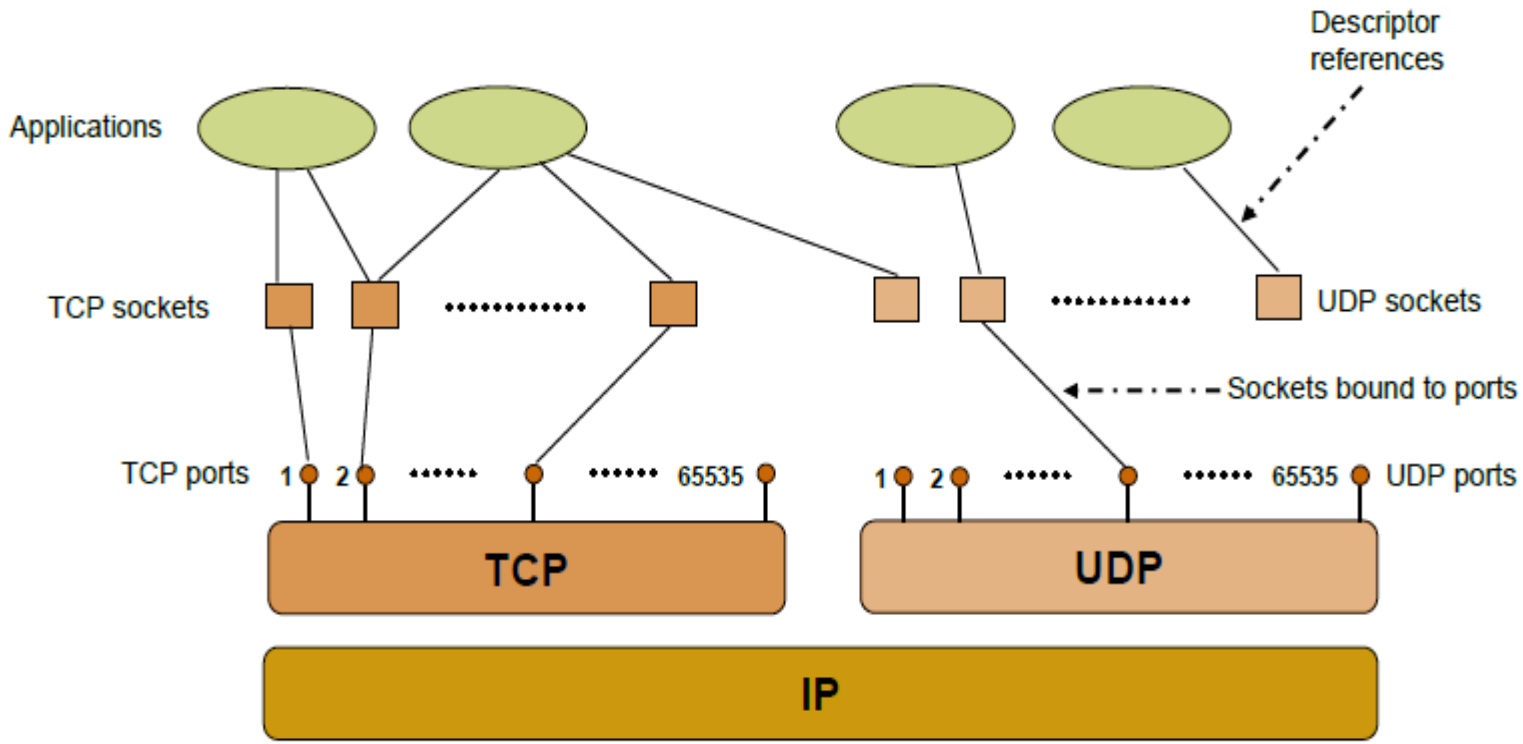
Socket Programming in C

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

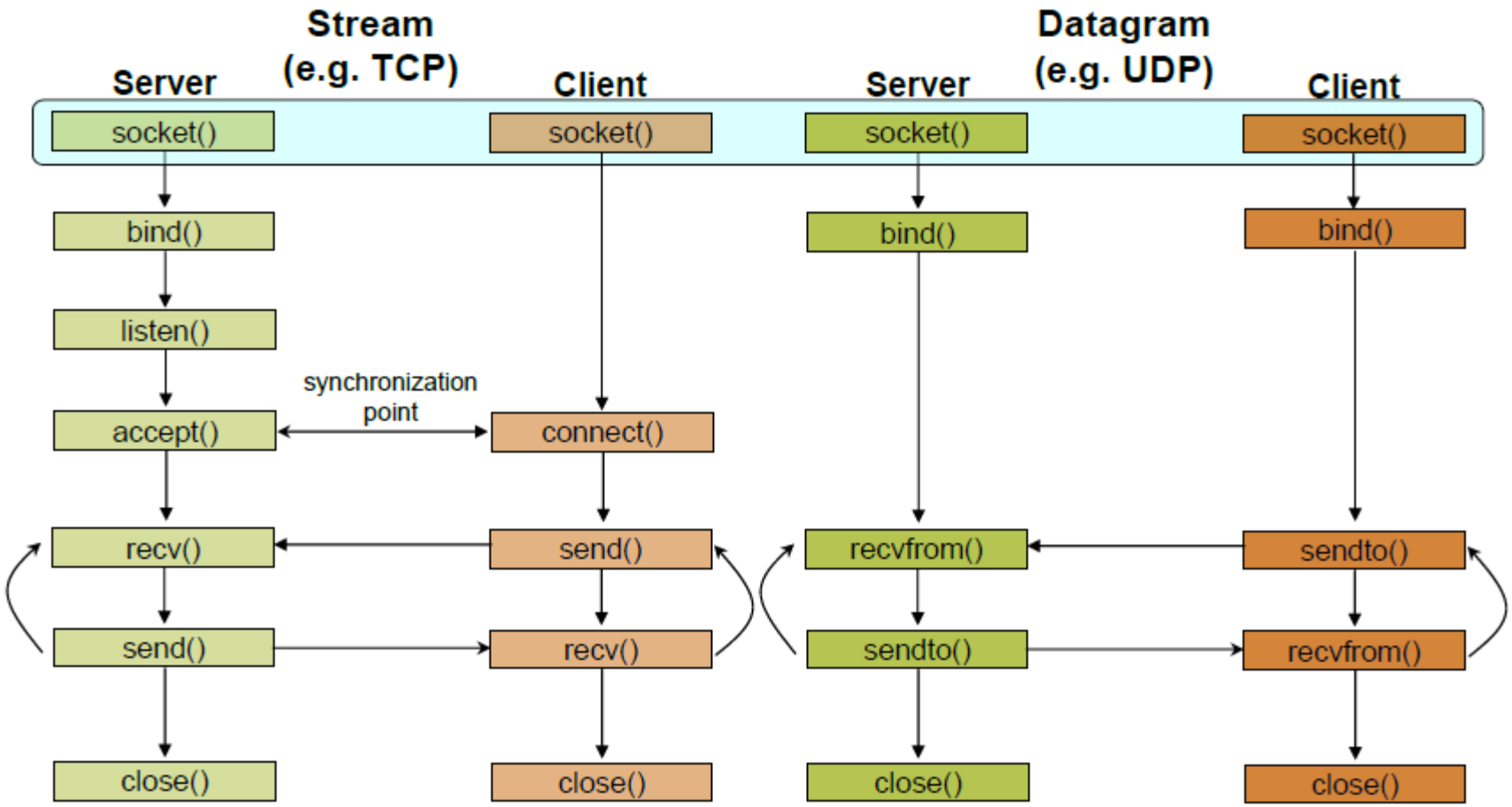
Client-Server communication

- **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

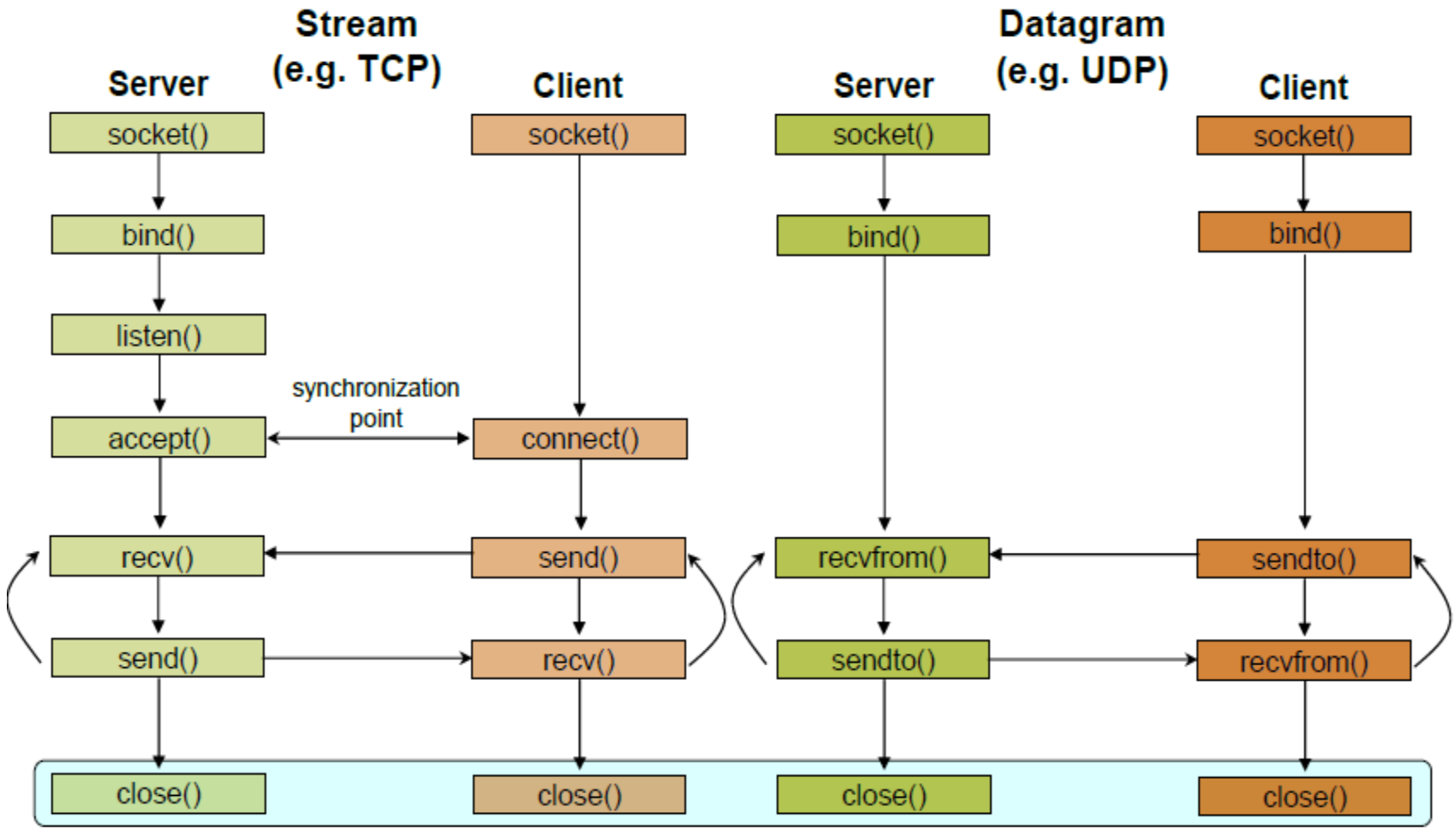


Socket creation in C: `socket()`

- `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:

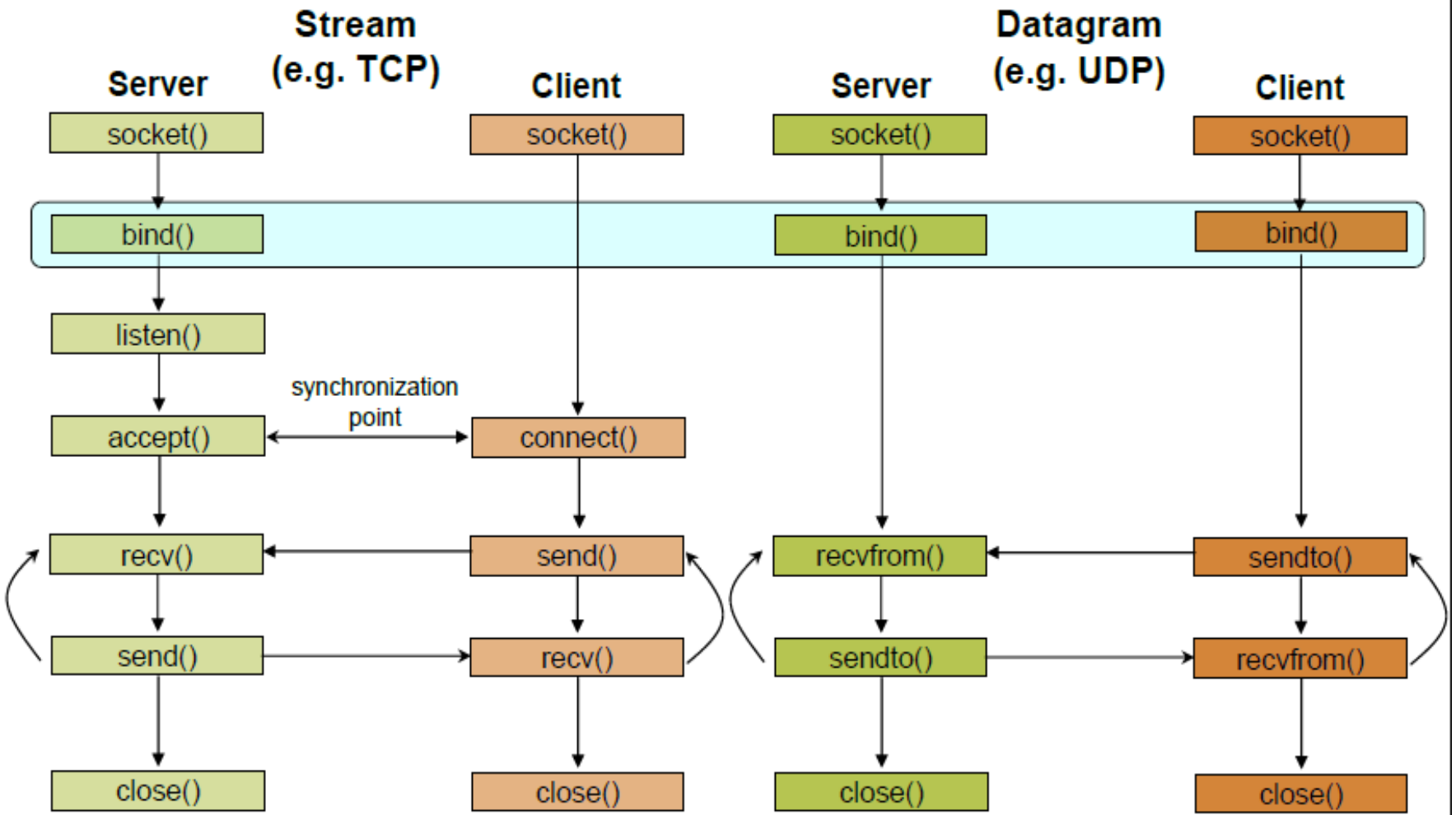
```
struct sockaddr {
   unsigned short sa_family;  /* Address family (e.g. AF_INET) */
   char sa_data[14];          /* Family-specific address information */
}
```

- 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

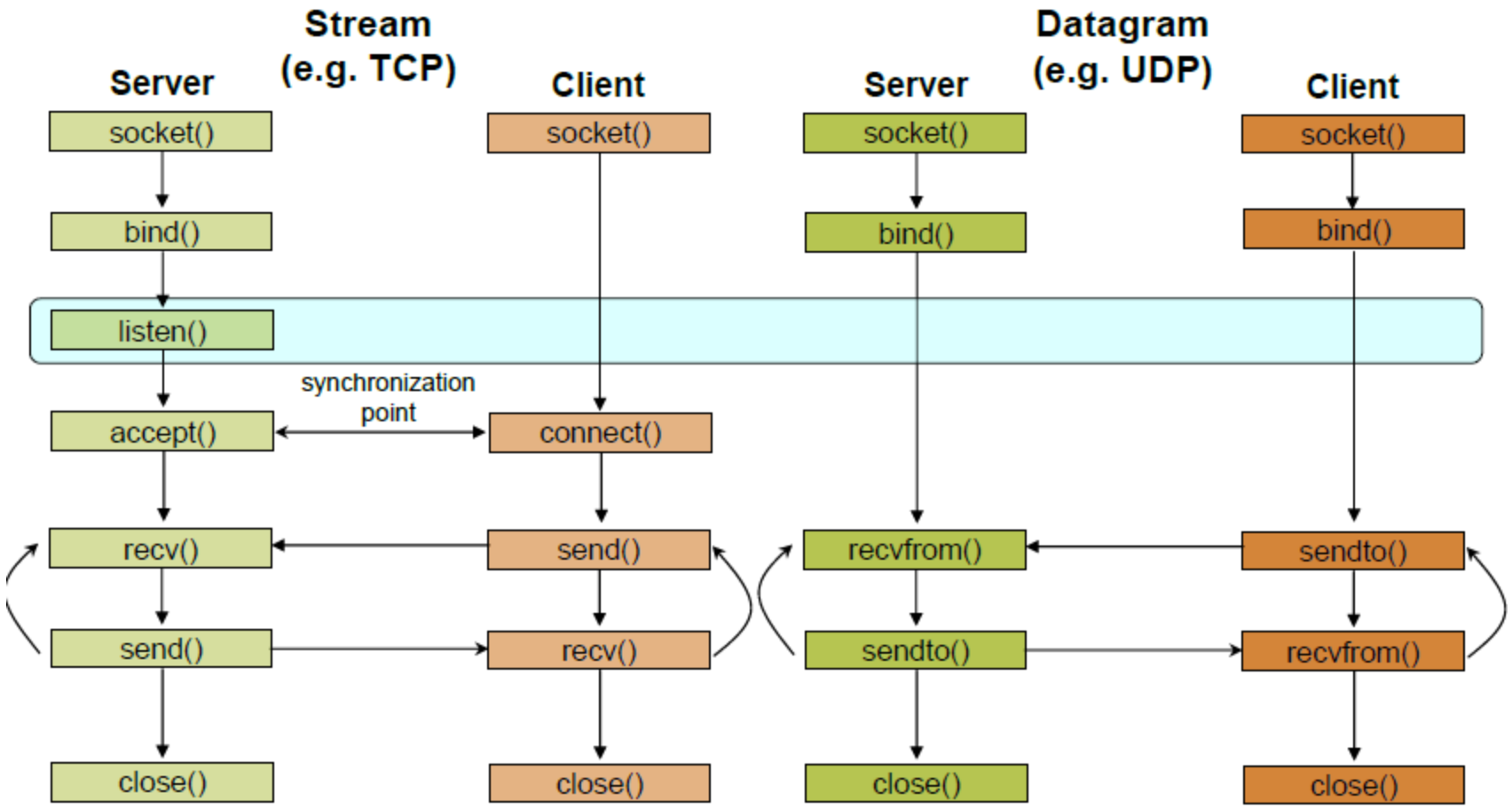
```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

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



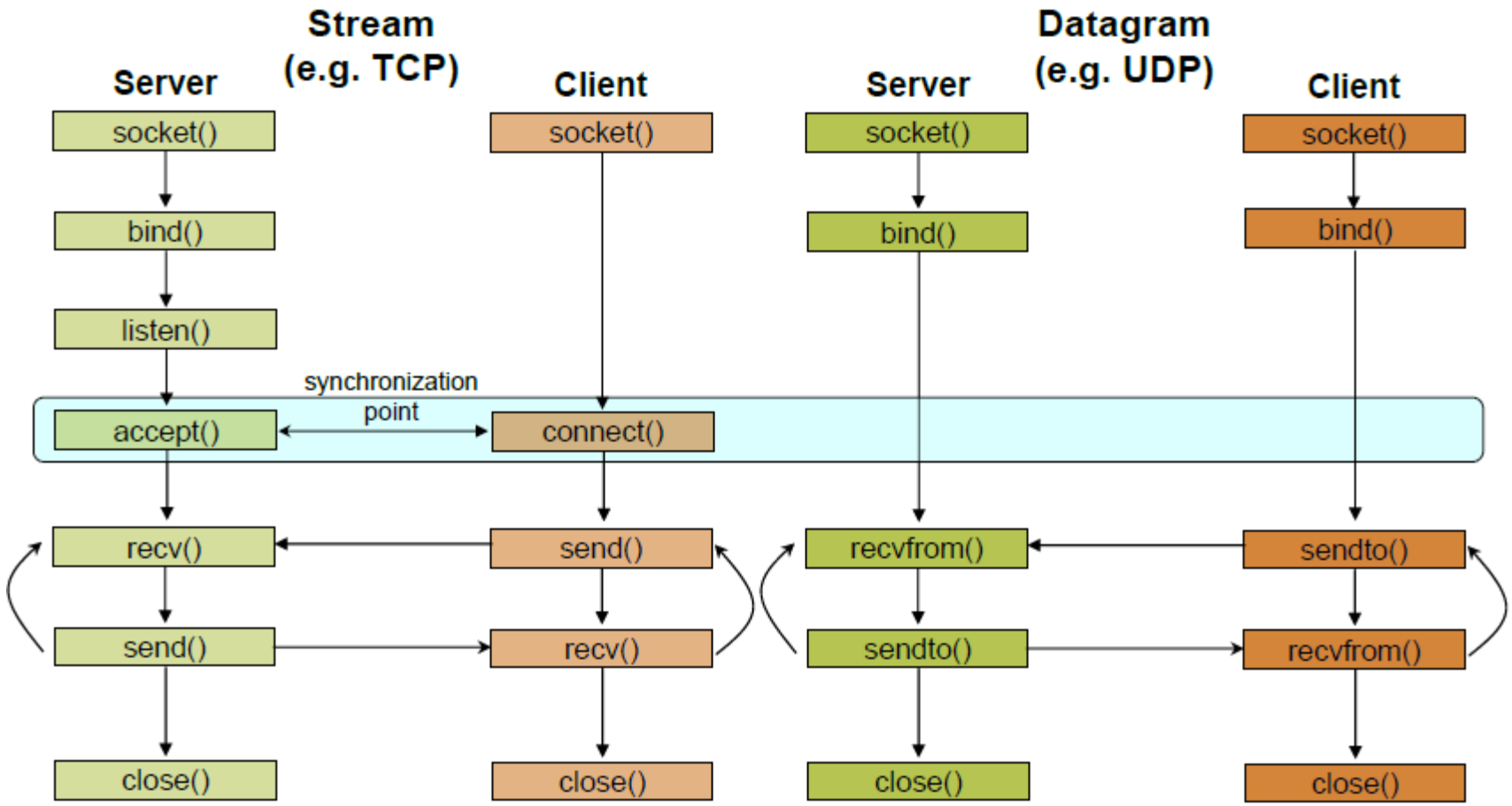
Assign address to socket: `bind()`

- 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()`

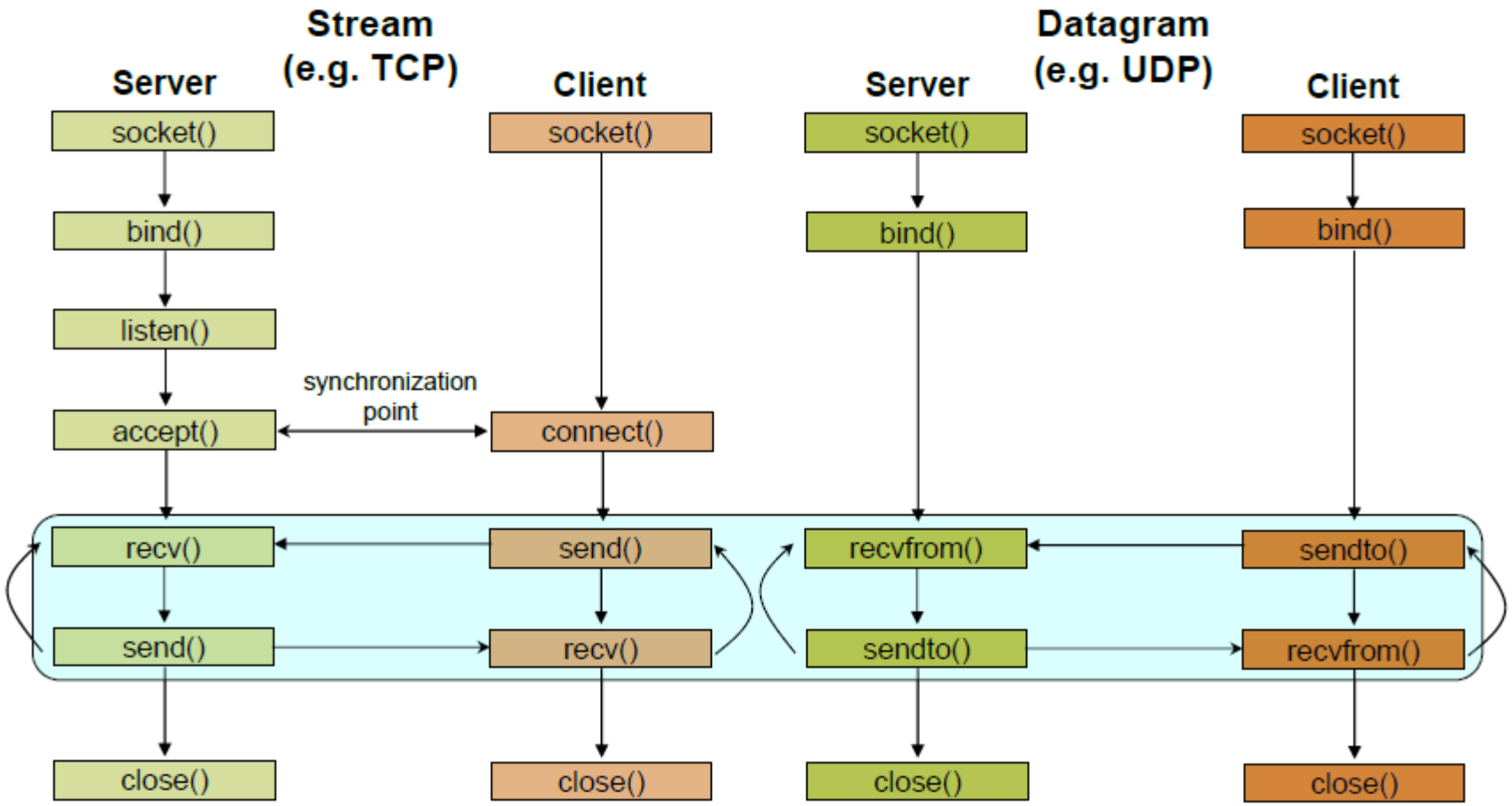
- The client establishes a connection with the server by calling `connect()`

- `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

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