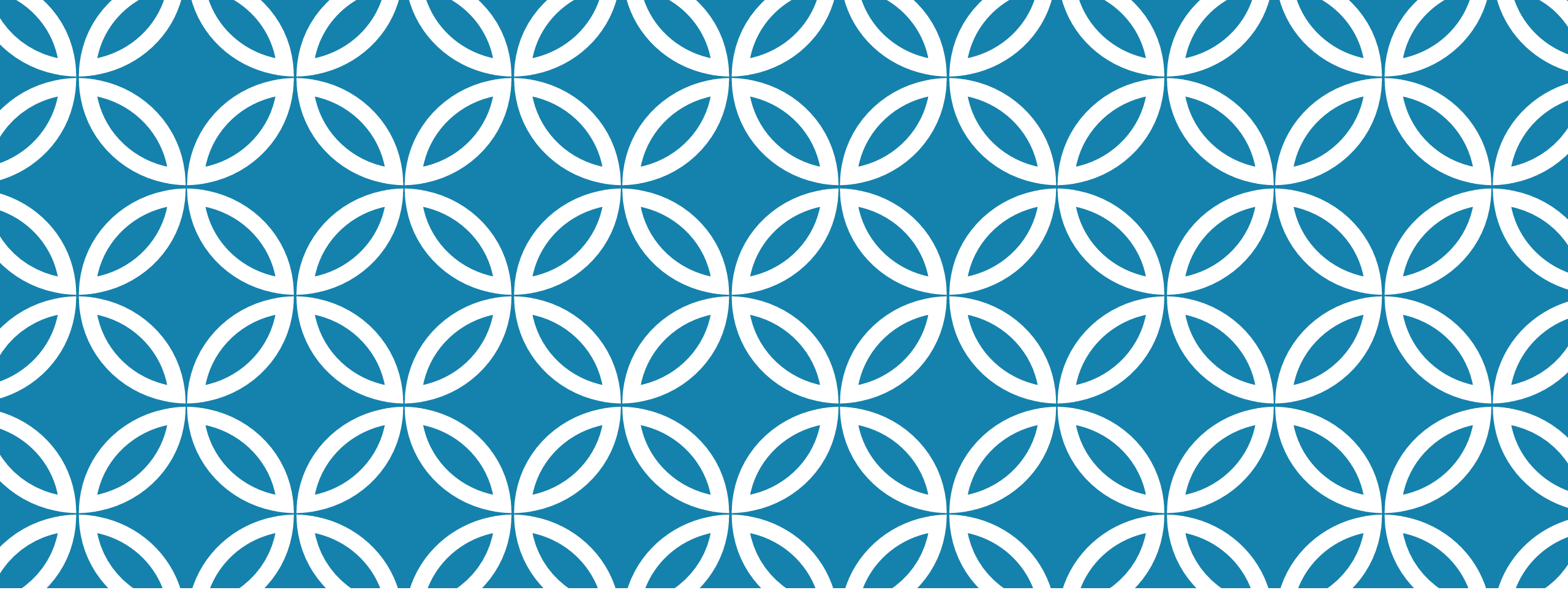
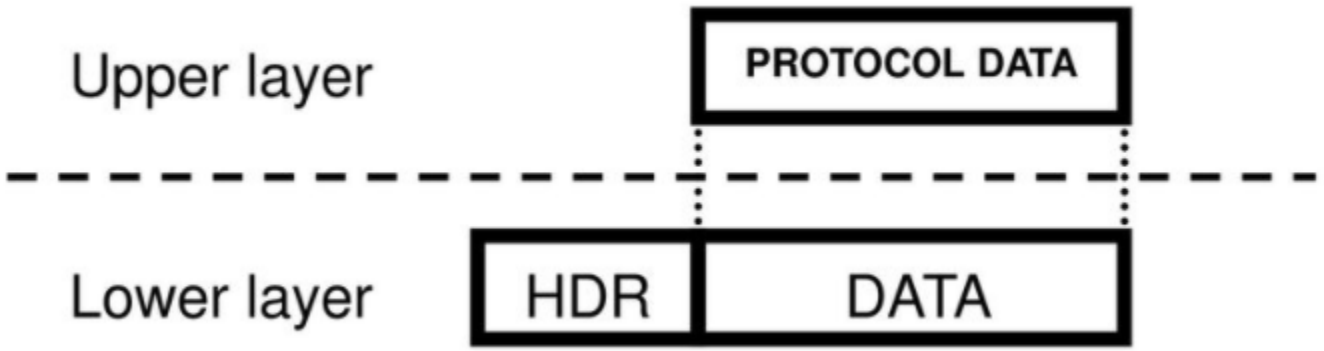


TCP-UDP

PROTOCOL LAYERS

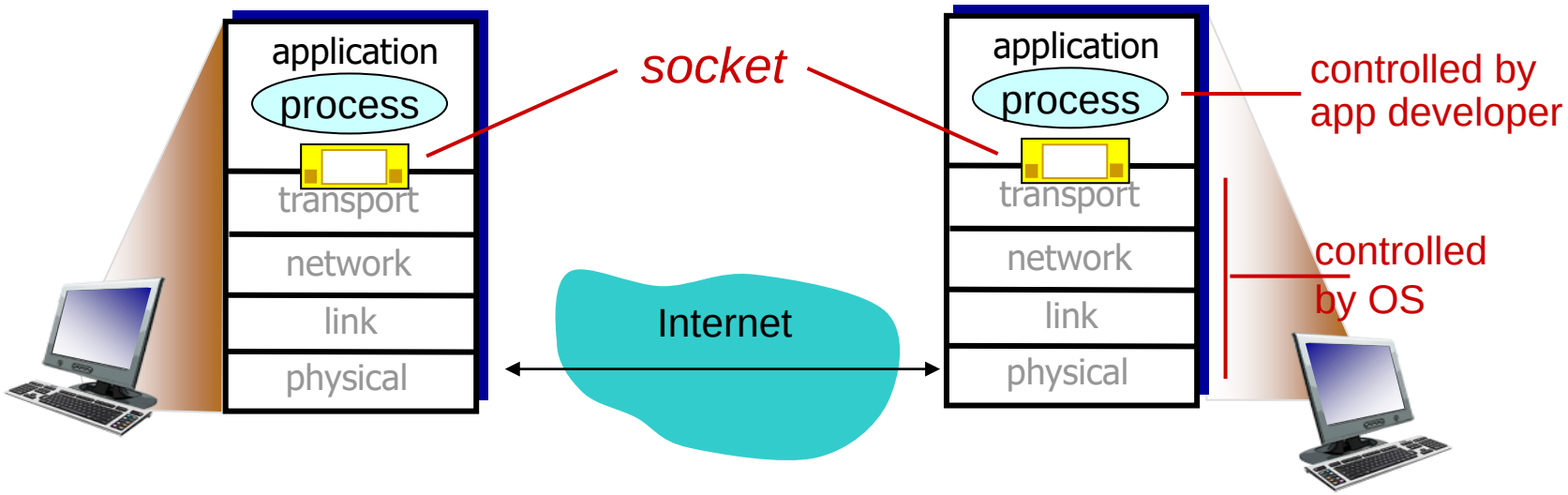
Each layer uses the layer below



SOCKET PROGRAMMING

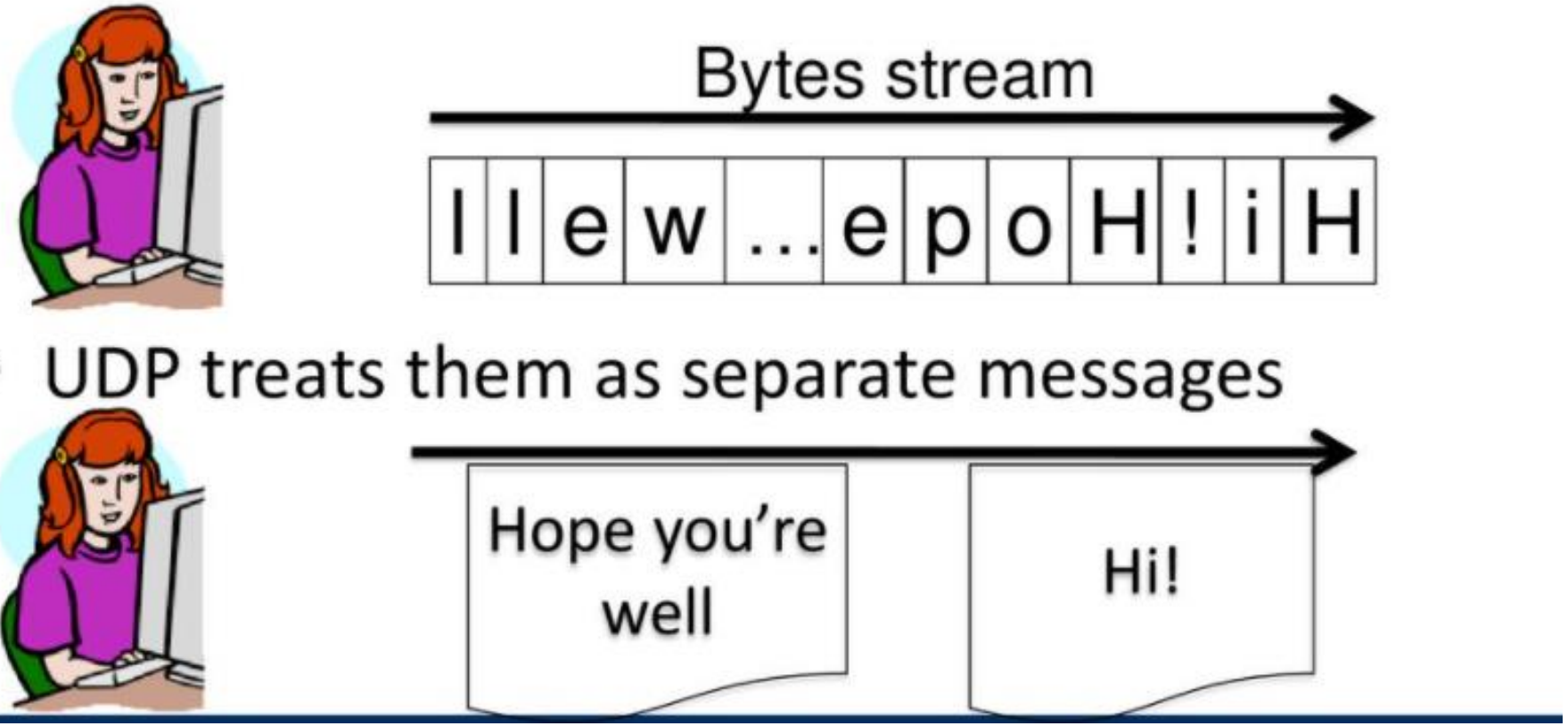
goal: learn how to build client/server applications that communicate using sockets

socket: dropbox between application process and end-end-transport protocol

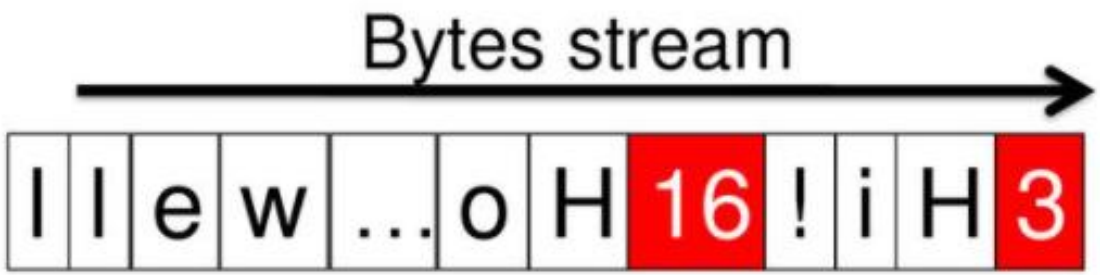


SENDING “HI” AND “HOPE YOU ARE DOING WELL”

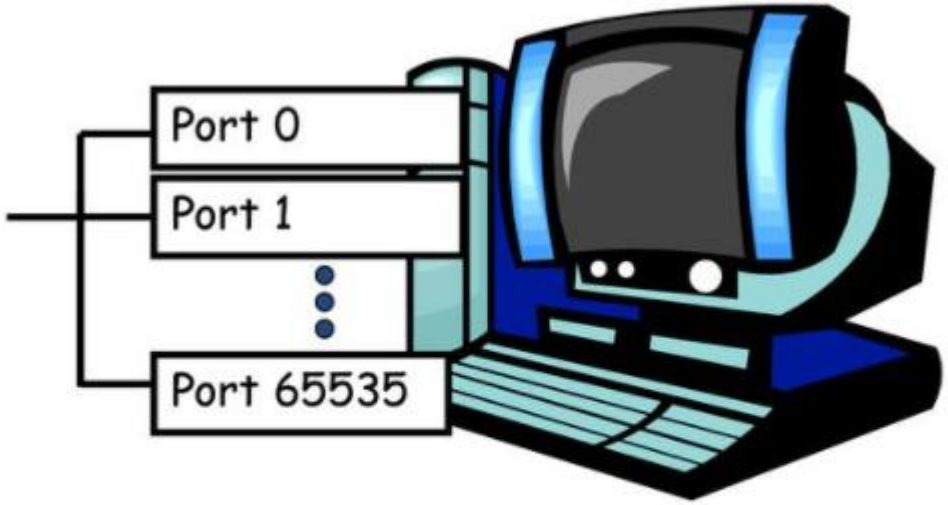
TCP treats as a single byte stream



- Thus, TCP needs application-level message boundary.
  - By carrying length in application-level header



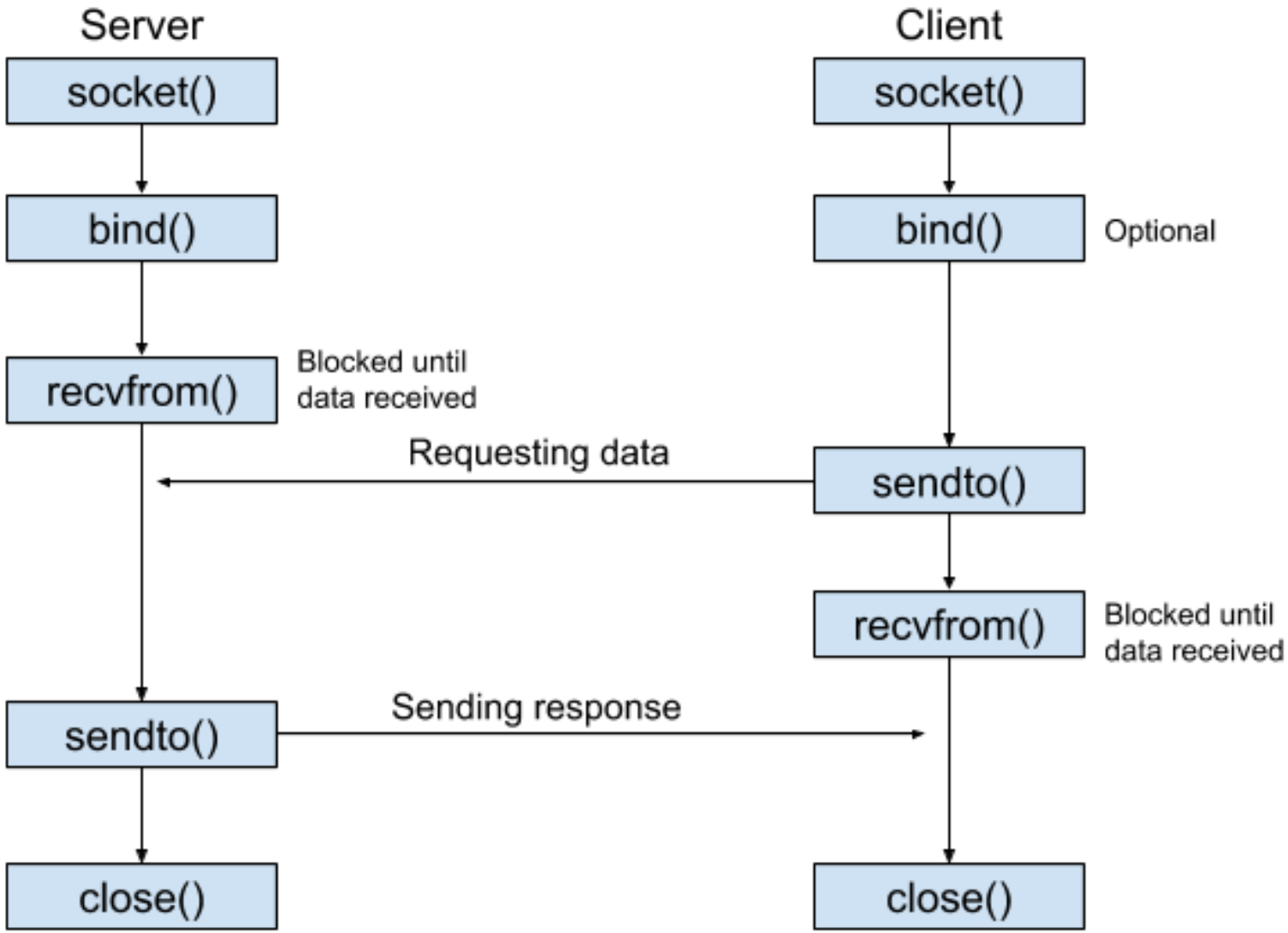
SOCKETS CONNECTED



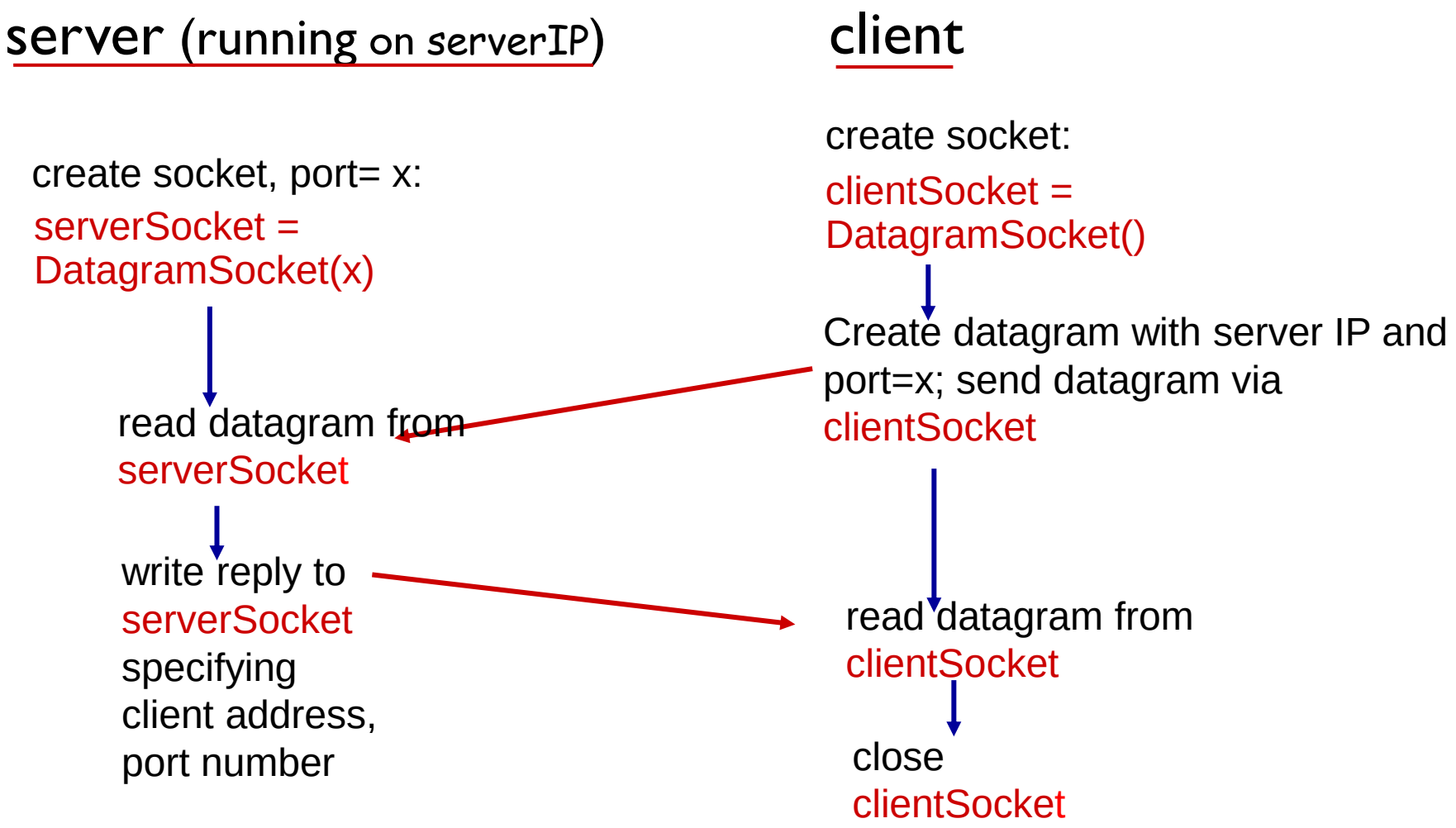
A socket provides an interface to send data to/from the network through a port

- Server opens a specific port
  - The one associated with its s
  - Then just waits for requests
  - Server is the passive opener
- Clients get ephemeral ports
  - Guaranteed unique, 1024 or greater
  - Uses them to communicate with server
  - Client is the active opener

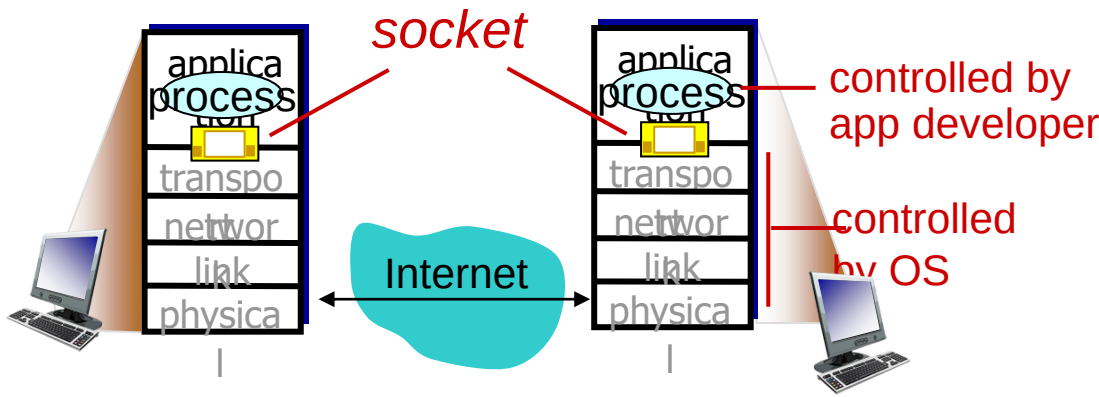
Assigned by the OS; no significance after the connection closes


Server
Client
socket()
socket()
bind()
bind()
Optional
recvfrom()
Blocked until
data received
Requesting data
sendto()
recvfrom()
Blocked until
data received
sendto()
Sending response
close()
close()

CLIENT/SERVER SOCKET INTERACTION: UDP



Example: Java client (UDP)




```
import java.io.*;
import java.net.*;

class UDPClient {
   public static void main(String args[]) throws Exception
   {
```

create input stream →

```
      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));
```

create client socket →

```
      DatagramSocket clientSocket = new DatagramSocket();
```

translate hostname to IP addr using DNS →

```
      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Example: Java client (UDP)

create datagram with data-to-send, length, IP addr, port

send datagram to server

read datagram from server

```
DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length,
                                IPAddress, 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket =
   new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}

}
```

Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
   {
      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
       {
         DatagramPacket receivePacket =
           new DatagramPacket(receiveData, receiveData.length);
         serverSocket.receive(receivePacket);
```

create datagram socket at port 9876 →

create space for received datagram →

receive datagram →

Example: Java server (UDP)

```
                String sentence = new String(receivePacket.getData());

get IP addr
port #, of
sender      →   InetAddress IPAddress = receivePacket.getAddress();

            →   int port = receivePacket.getPort();

                        String capitalizedSentence = sentence.toUpperCase();

                sendData = capitalizedSentence.getBytes();

create datagram
to send to client →  DatagramPacket sendPacket =
                       new DatagramPacket(sendData, sendData.length, IPAddress,
                                port);

write out
datagram
to socket   →   serverSocket.send(sendPacket);
              }
            }
          }
```

end of while loop,
loop back and wait for
another datagram

Example app: UDP client

Python UDPClient

include Python's socket library

```
import socket
serverName = 'hostname'
serverPort = 12000
clientSocket = socket.socket(socket.AF_INET,
                             socket.SOCK_DGRAM)
```

create UDP socket for server

get user keyboard input

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message,(serverName, serverPort))
```

read reply characters from socket into string

```
modifiedMessage, serverAddress =
                             clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print modifiedMessage

clientSocket.close()
```

bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
message = bytesAddressPair[0]
address = bytesAddressPair[1]

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((‘<>', serverPort))
print “The server is ready to receive”
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)

modifiedMessage = message.upper()

serverSocket.sendto(modifiedMessage, clientAddress)
```

create UDP socket

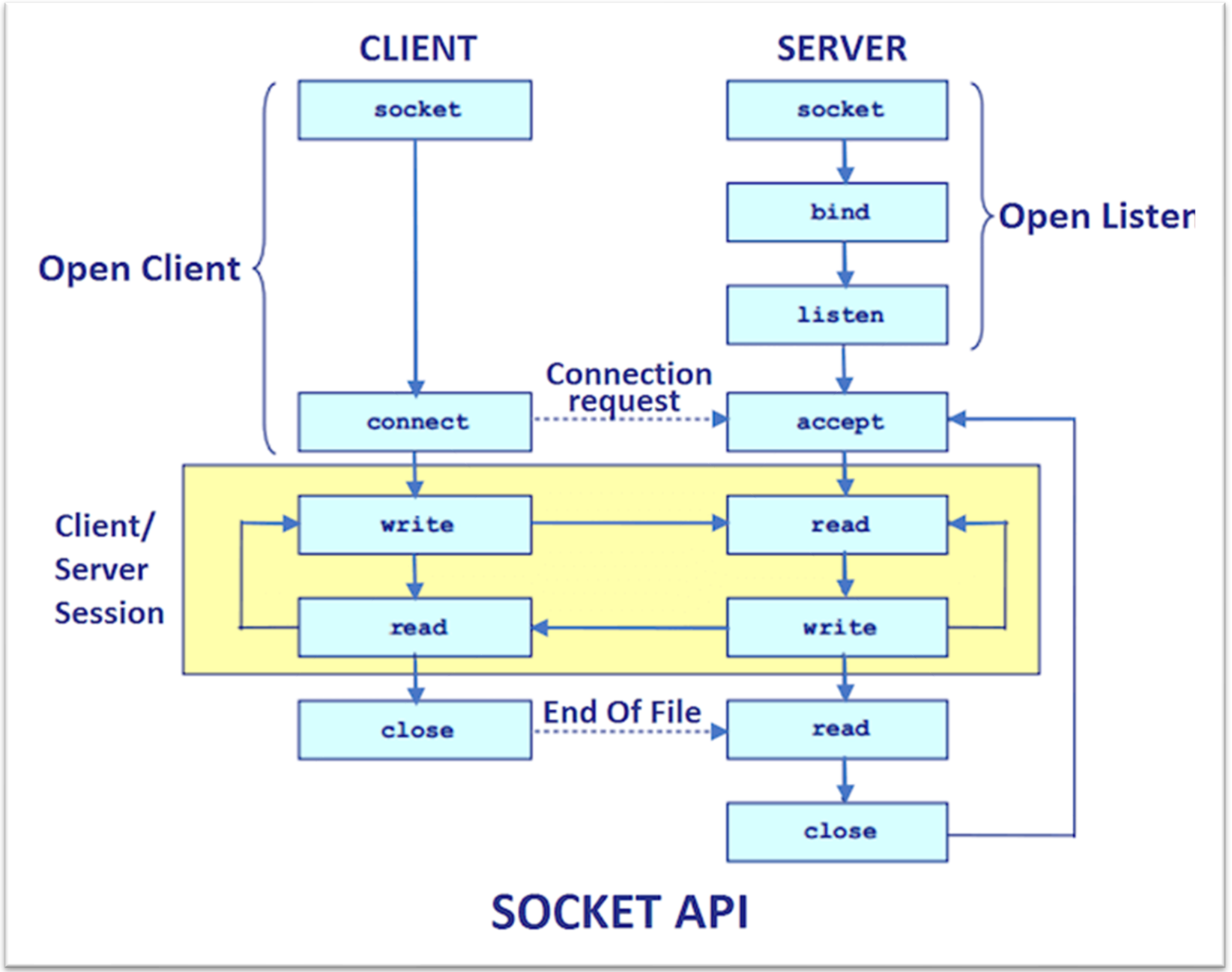
bind socket to local port number 12000

loop forever

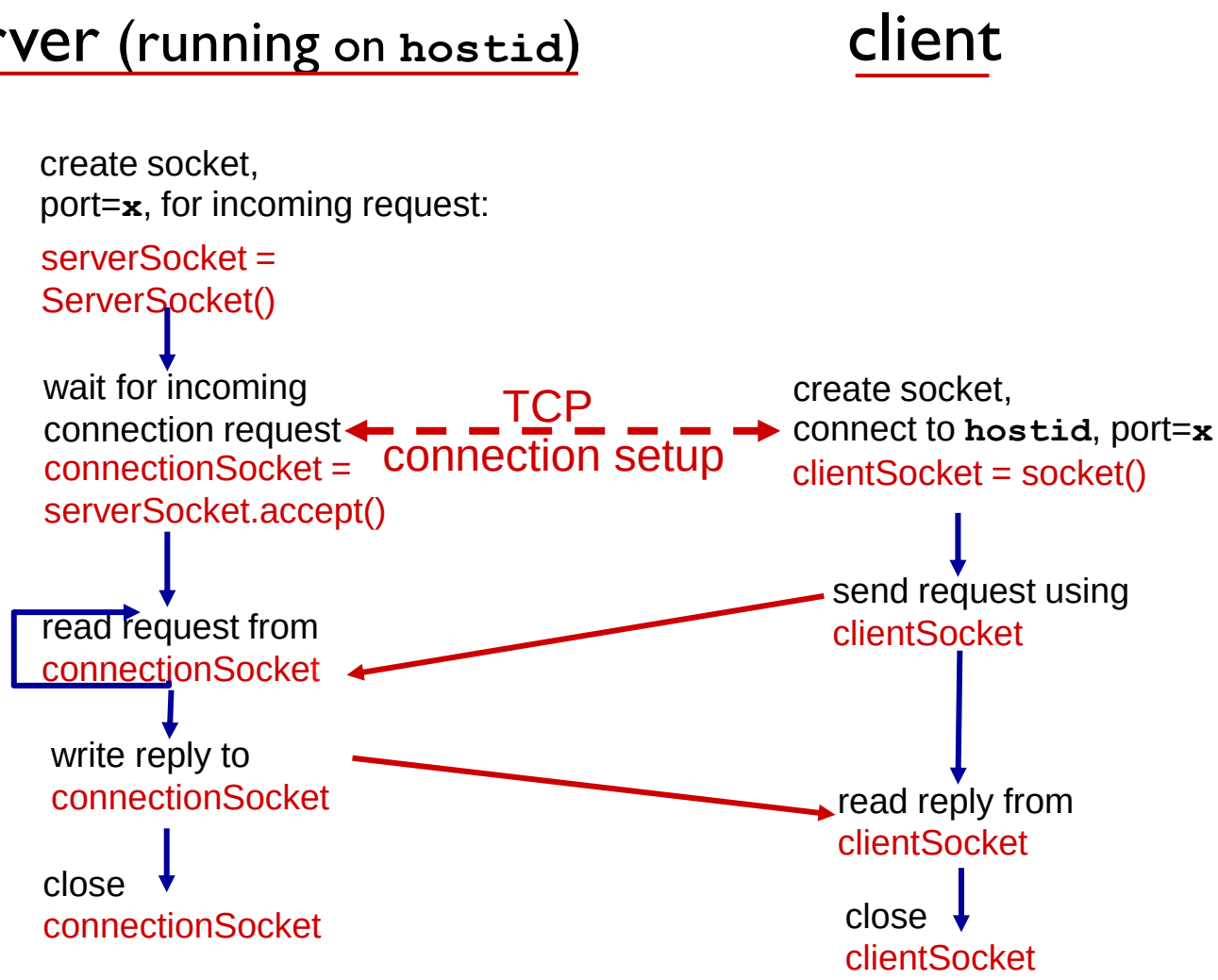
Read from UDP socket into message, getting client’s address (client IP and port)

send upper case string back to this client

TCP SOCKET

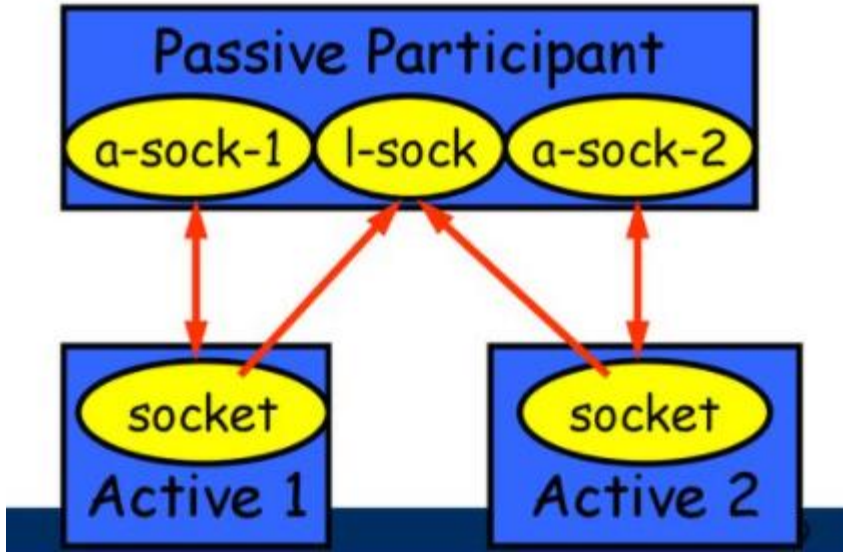

CLIENT
SERVER
socket
socket
bind
listen
Open Client
Open Listen
Connection request
connect
accept
Client/ Server Session
write
read
read
write
close
End Of File
read
close
SOCKET API

CLIENT/SERVER SOCKET INTERACTION: TCP

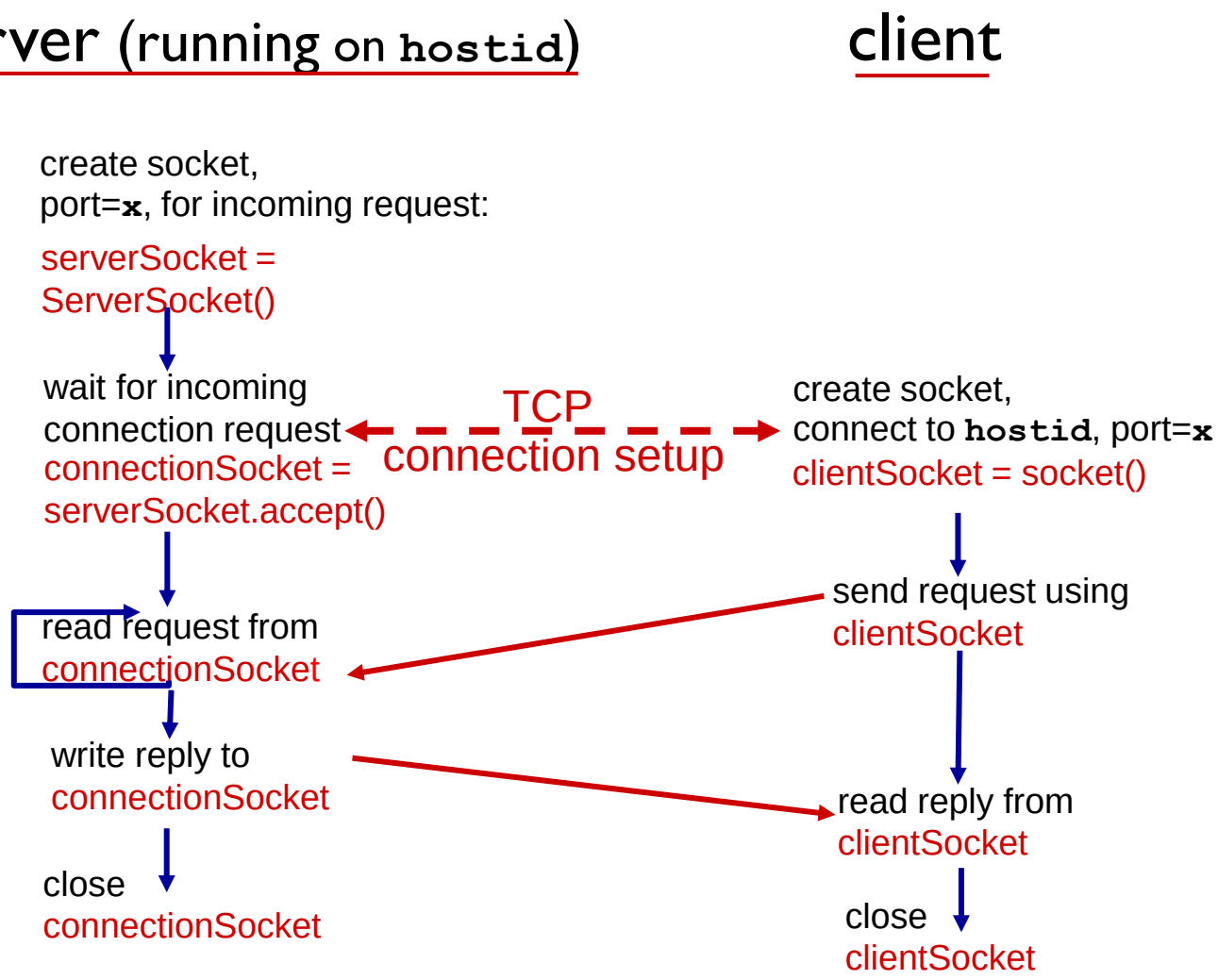


- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- Active participant
  - step 2: request & establish connection
  - step 4: data transfer

Sockets come in two primary flavors: active and passive. An active socket is connected to a remote active socket via an open data connection. Closing the connection destroys the active sockets at each end point. A passive socket is not connected, but rather awaits an incoming connection, which will spawn a new active socket.



CLIENT/SERVER SOCKET INTERACTION: TCP



EXAMPLE: JAVA CLIENT (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

this package defines Socket() and ServerSocket() classes

server name, e.g., www.umass.edu

server port #

create input stream

create clientSocket object of type Socket, connect to server

create output stream attached to socket

Example: Java client (TCP)

create input stream attached to socket →

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
```

send line to server →

```
outToServer.writeBytes(sentence + '\n');
```

read line from server →

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);
```

close socket (clean up behind yourself!) →

```
clientSocket.close();

    }
}
```

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

       Socket connectionSocket = welcomeSocket.accept();

        BufferedReader inFromClient =
          new BufferedReader(new
          InputStreamReader(connectionSocket.getInputStream()));
```

create welcoming socket at port 6789

wait, on welcoming socket accept() method for client contact create, *new* socket on return

create input stream, attached to socket

Example: Java server (TCP)

create output stream, attached to socket →

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

read in line from socket →

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

write out line to socket →

```
outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

end of while loop, loop back and wait for another client connection

Example app: TCP client

Python TCPClient

```
import socket
serverName = 'servername'
serverPort = 12000
clientSocket = socket.socket(socket.AF_INET,
                             socket.SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'

while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

NODE.JS

```
1 var http = require("http");
2
3 http.createServer(function(request, response) {
4 response.writeHead(200, {"Content-Type":
"text/plain"});
5 response.write("Hello World");
6 response.end();
7 }).listen(8888);
```

The first line *require*s the *http* module that ships with Node.js and makes it accessible through the variable *http*.

```
var server = http.createServer();
server.listen(8888);
```

We then call one of the functions the http module offers: *createServer*.

This function returns an object, and this object has a method named *listen*, and takes a numeric value which indicates the port number our HTTP server is going to listen on.