Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Processes communicating

process: program running within a host.

- within same host, two processes communicate using inter-process communication (defined by OS).
- processes in different hosts communicate by exchanging messages

client process: process that initiates communication

server process: process that waits to be contacted

- aside: applications with P2P architectures have client processes & server processes

Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

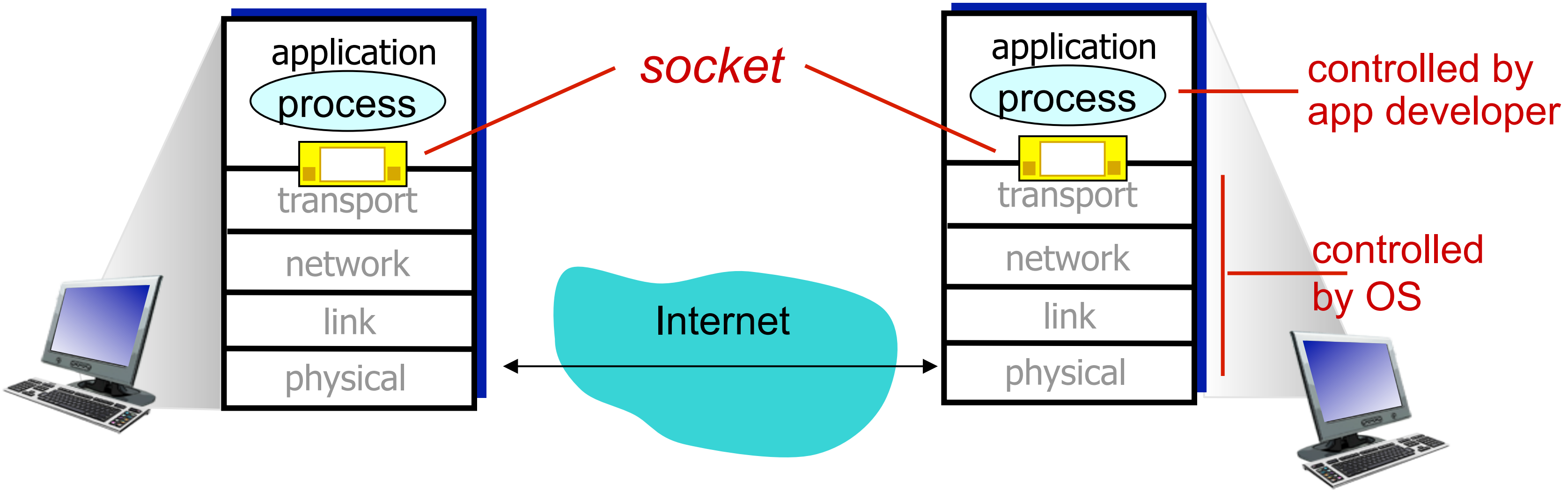
security

- encryption, data integrity, …

Socket programming

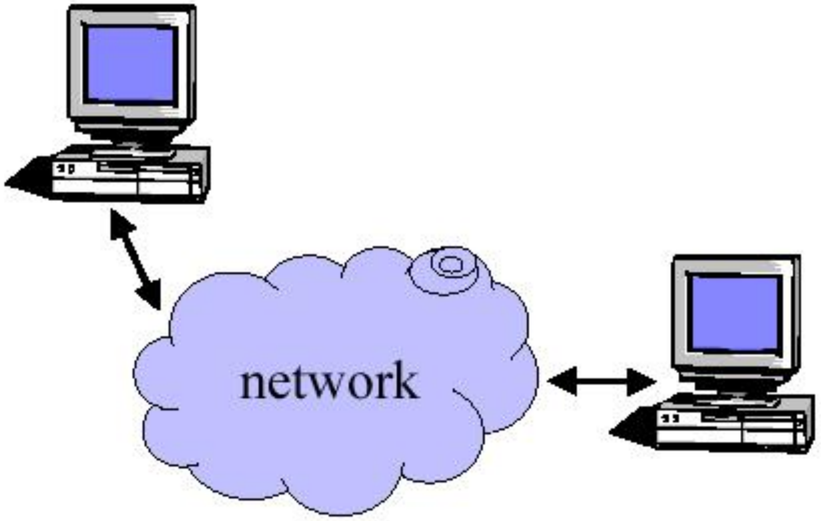
goal: learn how to build client/server applications that communicate using sockets

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



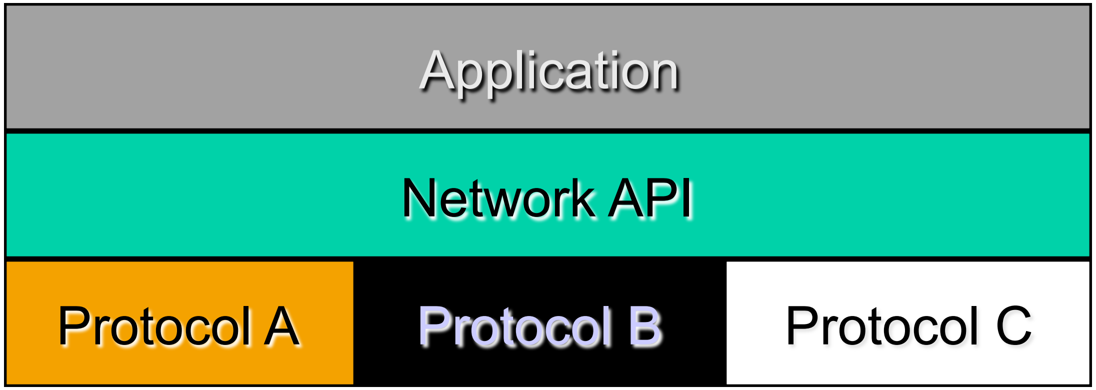
Why do we need sockets?

Provides an abstraction for interprocess communication



Definition

- The services provided (often by the operating system) that provide the interface between application and protocol software.



Functions

- Define an “end- point” for communication
- Initiate and accept a connection
- Send and receive data
- Terminate a connection gracefully

Examples

- File transfer apps (FTP), Web browsers
- (HTTP), Email (SMTP/ POP3), etc…

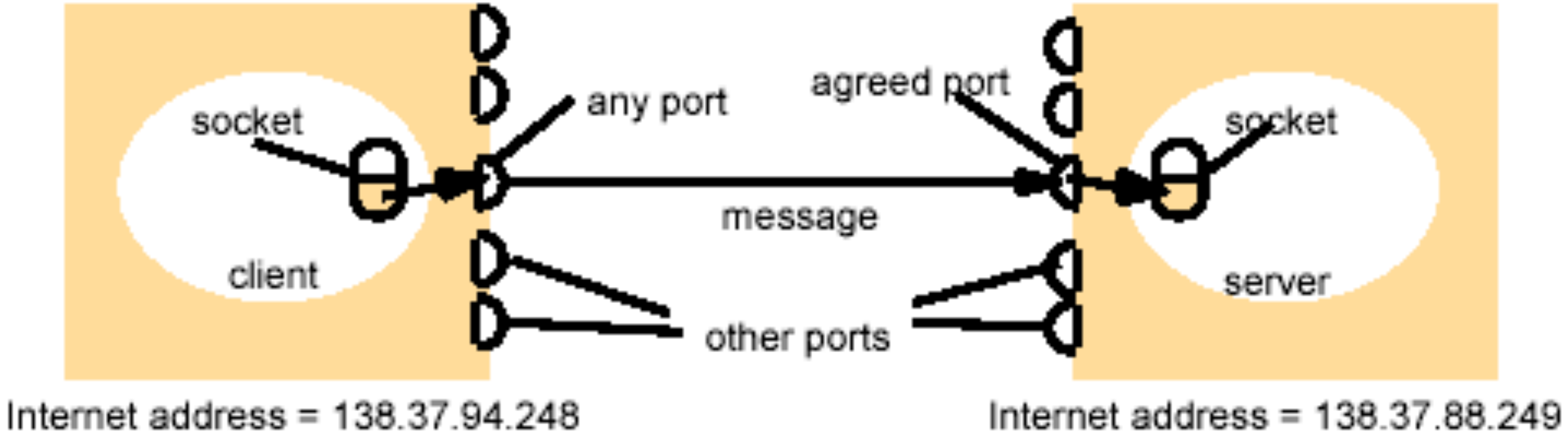
Types of Sockets

- Two different types of sockets :
  - stream vs. datagram

- Stream socket :(*a. k.* a. connection- oriented socket)
  - It provides reliable, connected networking service
  - Error free; no out- of- order packets (uses TCP)
  - applications: telnet/ ssh, http, …

- Datagram socket :(*a. k.* a. connectionless socket)
  - It provides unreliable, best- effort networking service
  - Packets may be lost; may arrive out of order (uses UDP)
  - applications: streaming audio/ video (realplayer), …

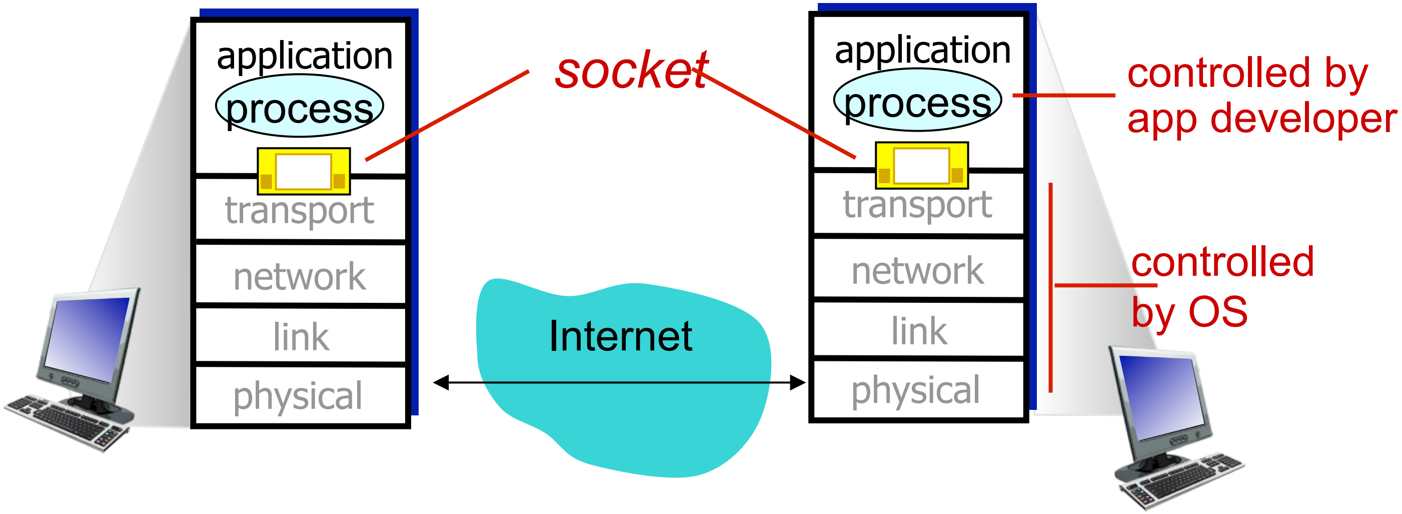
Addressing


socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

Client
Server

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket programming

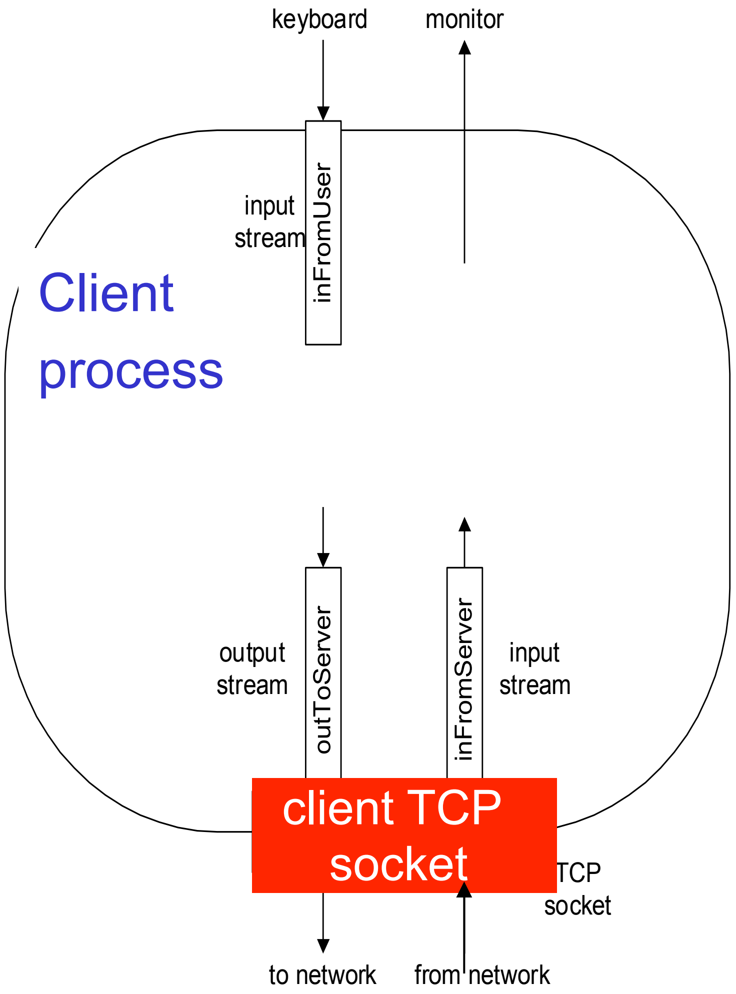
Example client-server app:

1) client reads line from standard input (**`inFromUser`** stream) , sends to server via socket (**`outToServer`** stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (**`inFromServer`** stream)



Socket programming *with UDP*

UDP: no "connection" between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

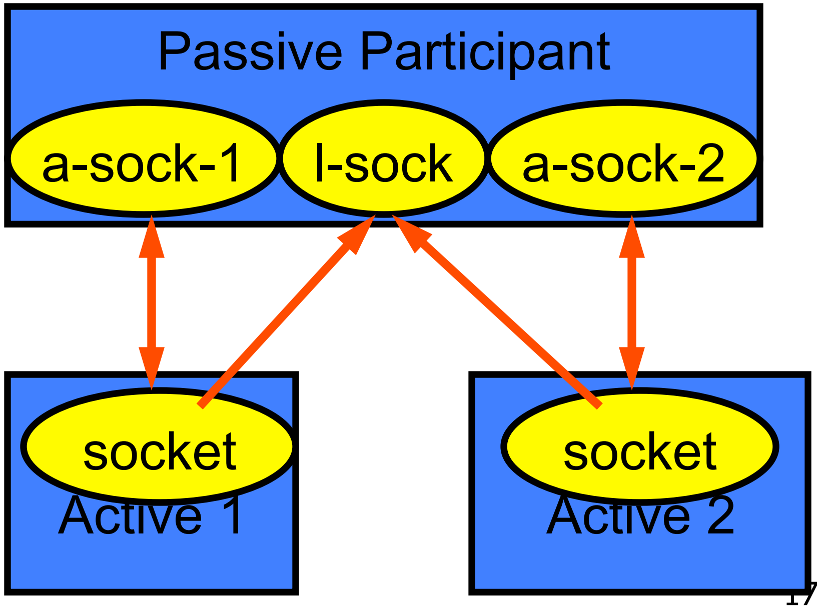
Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are "similar"
  - both can send & receive data
  - either can terminate the connection

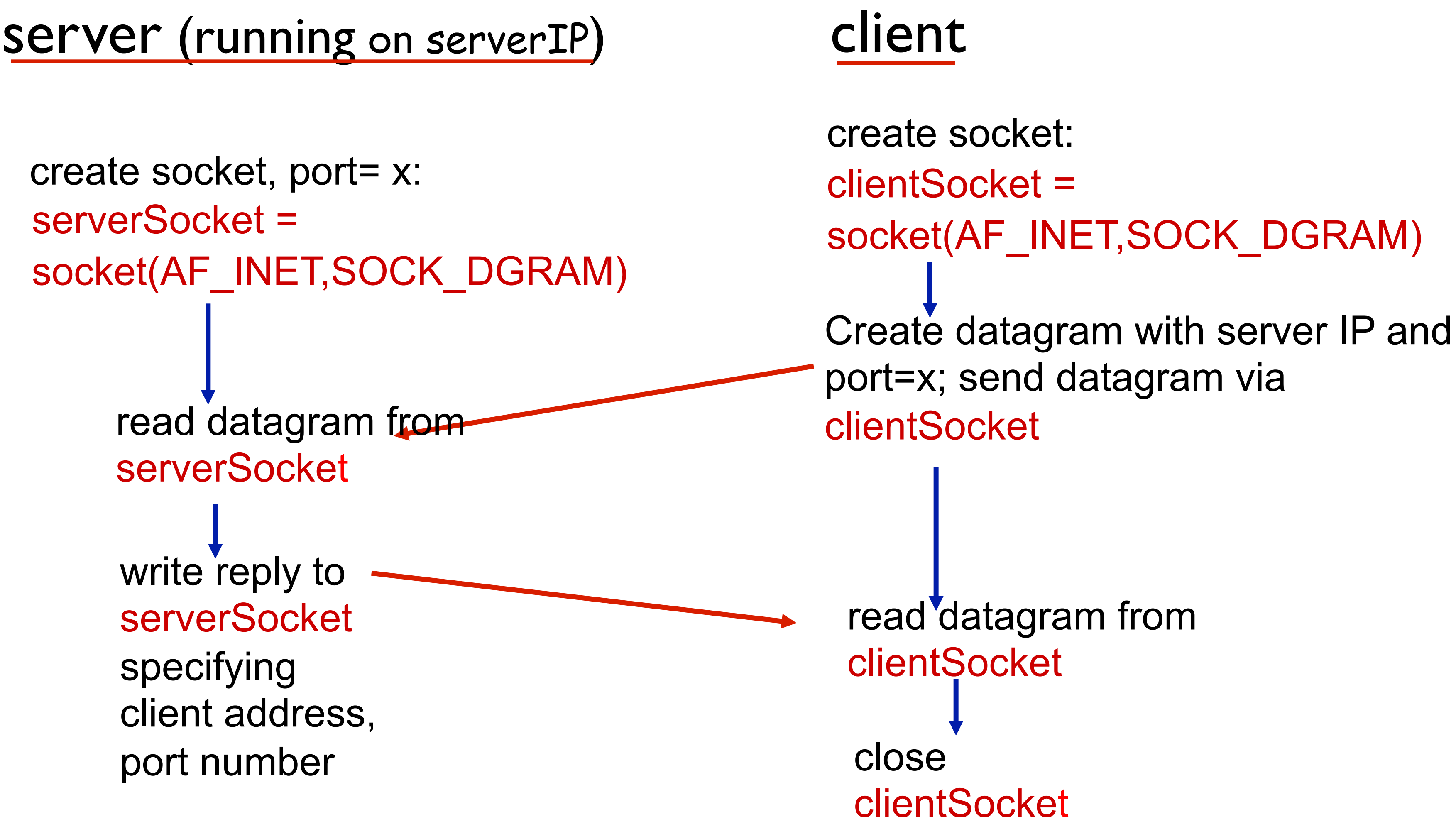
Connection setup cont’d

- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants
- Why?

- Active participant
  - step 2: request & establish connection
  - step 4: data transfer



Client/server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

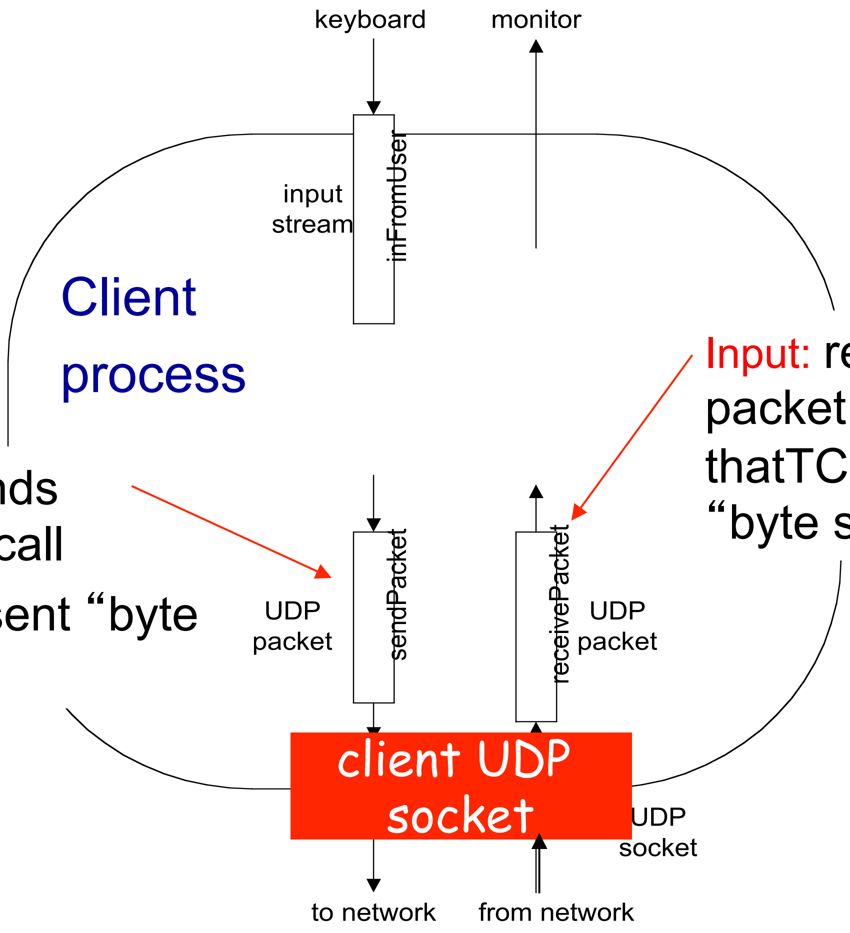
create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example: Java client (UDP)



Output: sends packet (recall that TCP sent "byte stream")

Input: receives packet (recall thatTCP received "byte stream")

Example app: UDP client

Python UDPClient

include Python's socket library

```
from socket import *
serverName = ‘hostname’
serverPort = 12000
```

create UDP socket for server

```
clientSocket = socket(socket.AF_INET,
                          socket.SOCK_DGRAM)
```

get user keyboard input

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

Attach server name, port to message; send into socket

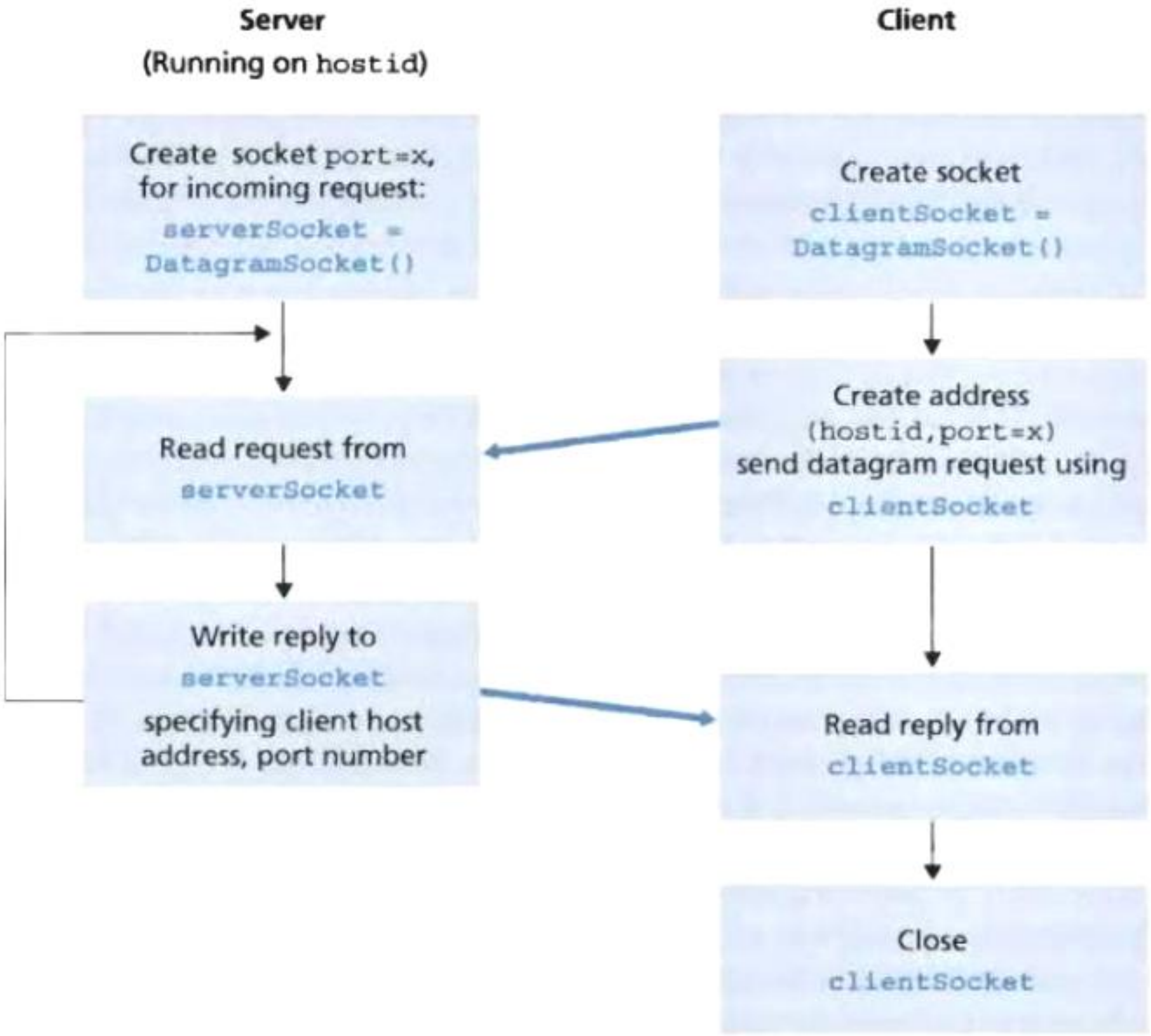
```
clientSocket.sendto(message,(serverName, serverPort))
```

read reply characters from socket into string

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

print out received string and close socket

```
print modifiedMessage
clientSocket.close()
```



Server

(Running on hostid)

Create socket port=x, for incoming request:

serverSocket = DatagramSocket()

Read request from serverSocket

Write reply to serverSocket specifying client host address, port number

Client

Create socket clientSocket = DatagramSocket()

Create address (hostid,port=x) send datagram request using clientSocket

Read reply from clientSocket

Close clientSocket

Figure 1. The client-server application, using connectionless transport services

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

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 address 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), cont.

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

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

send datagram
to server

```
clientSocket.send(sendPacket);

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

read datagram
from server

```
      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
    {
```

create datagram socket at port 9876

```
      DatagramSocket serverSocket = new DatagramSocket(9876);

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

      while(true)
        {
```

create space for received datagram

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

receive datagram

```
           serverSocket.receive(receivePacket);
```

Example: Java server (UDP), cont

get IP addr port #, of sender

create datagram to send to client

write out datagram to socket

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

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

          InetAddress IPAddress = receivePacket.getAddress();

          int port = receivePacket.getPort();

                      String capitalizedSentence = sentence.toUpperCase();

          sendData = capitalizedSentence.getBytes();

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

          serverSocket.send(sendPacket);
        }
      }
    }
```

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

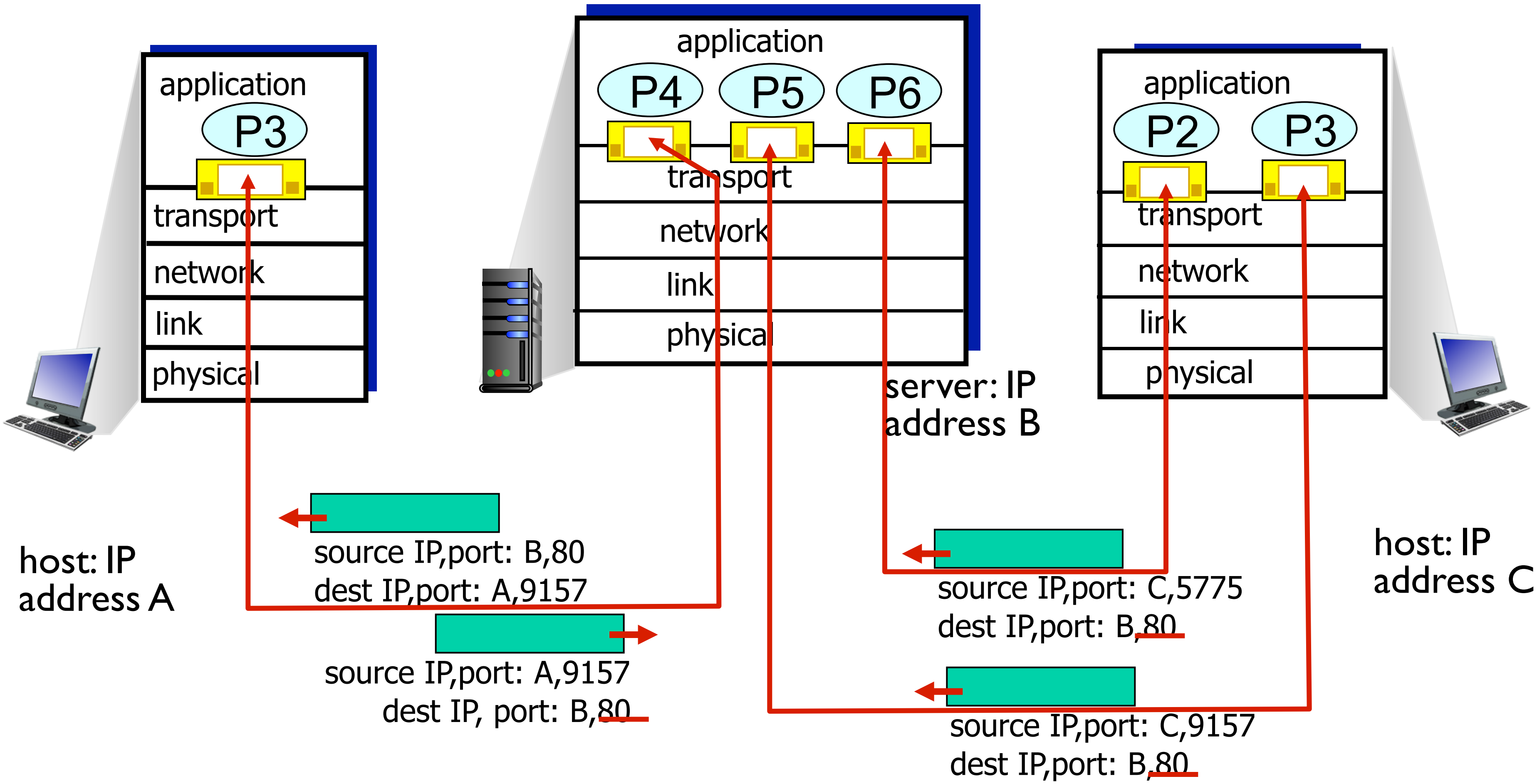
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Connection-oriented demux: example



host: IP
address A

server: IP
address B

host: IP
address C

source IP,port: B,80
dest IP,port: A,9157

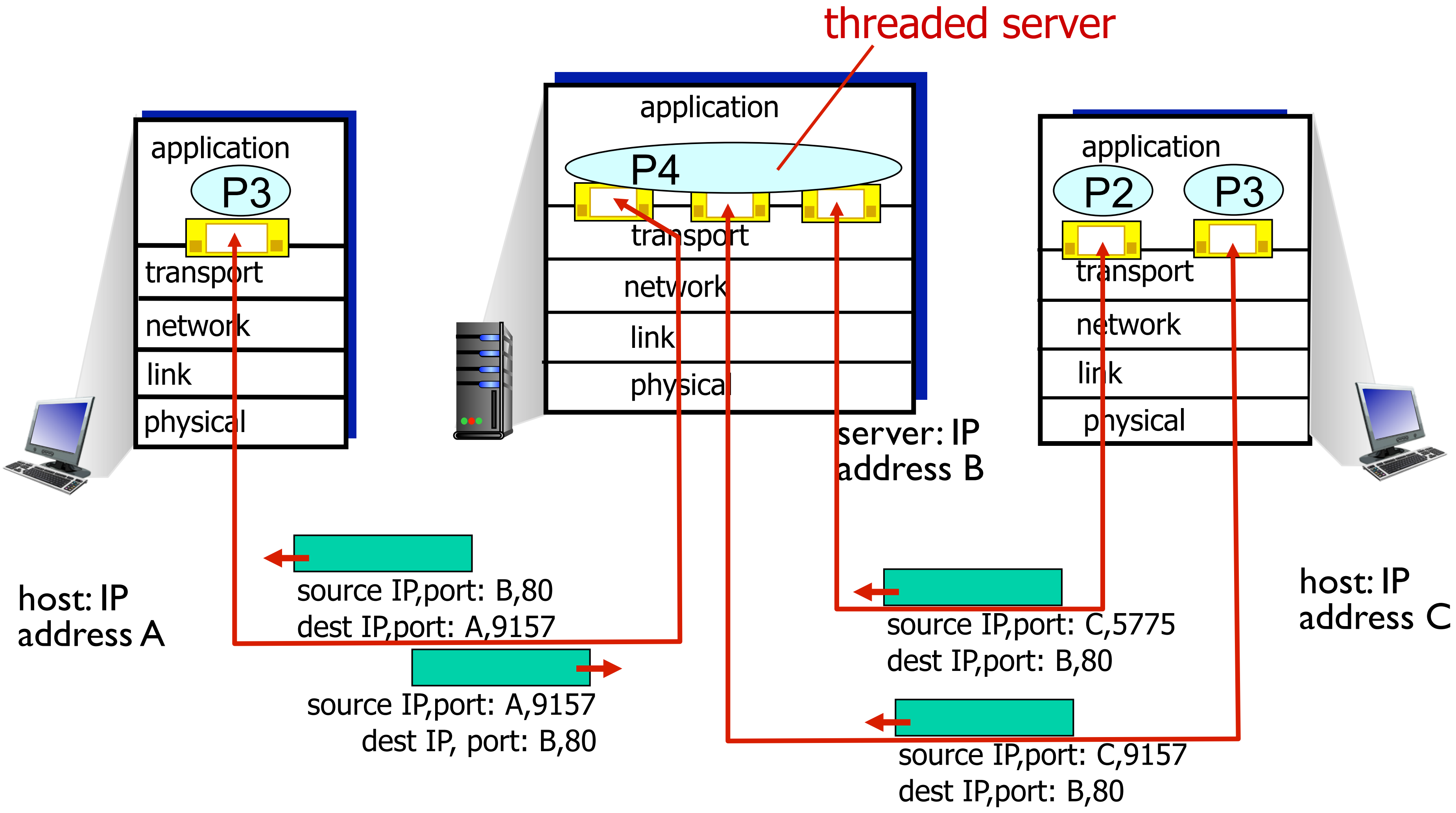
source IP,port: A,9157
dest IP, port: B,80

source IP,port: C,5775
dest IP,port: B,80

source IP,port: C,9157
dest IP,port: B,80

three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

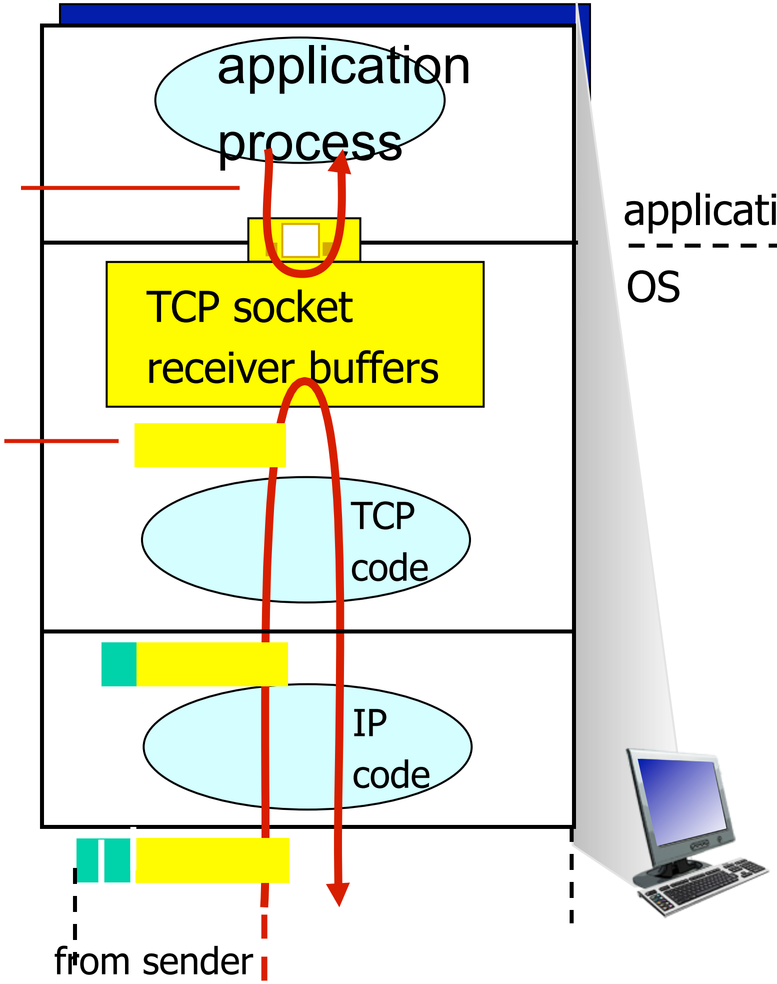
Connection-oriented demux: example



TCP flow control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)

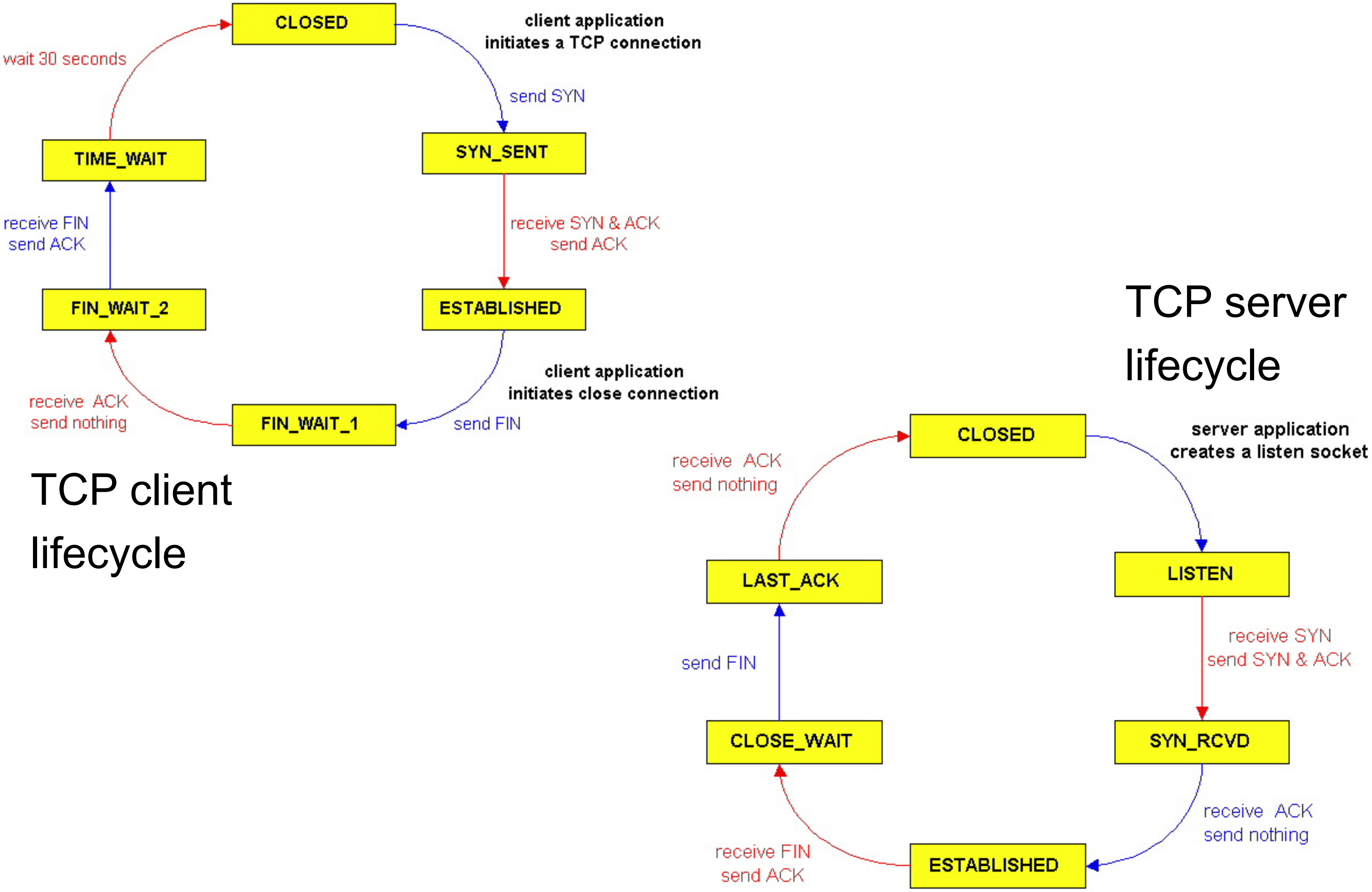


receiver protocol stack

flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

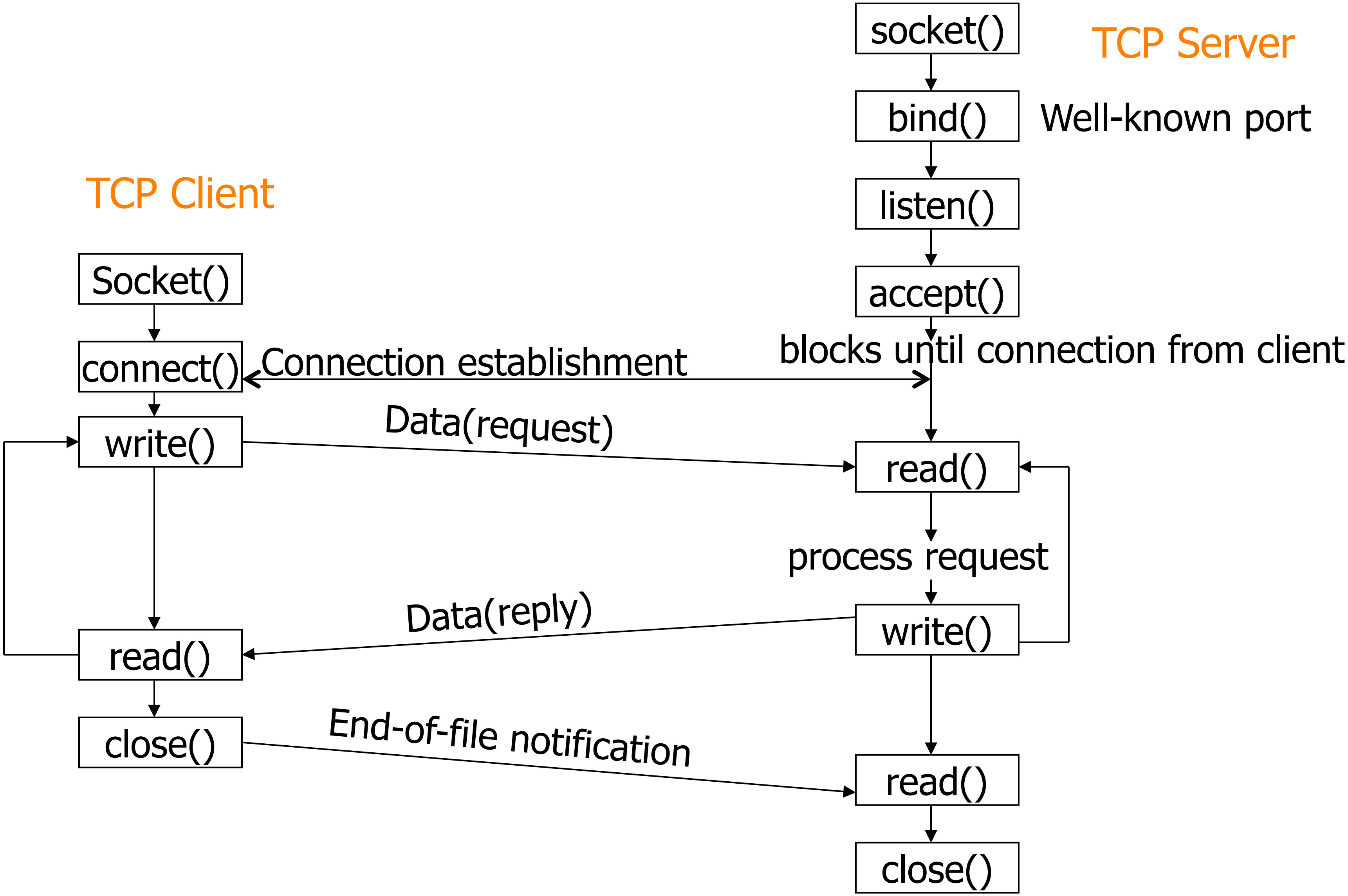
TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle

Socket programming *with TCP*



Socket programming *with TCP*

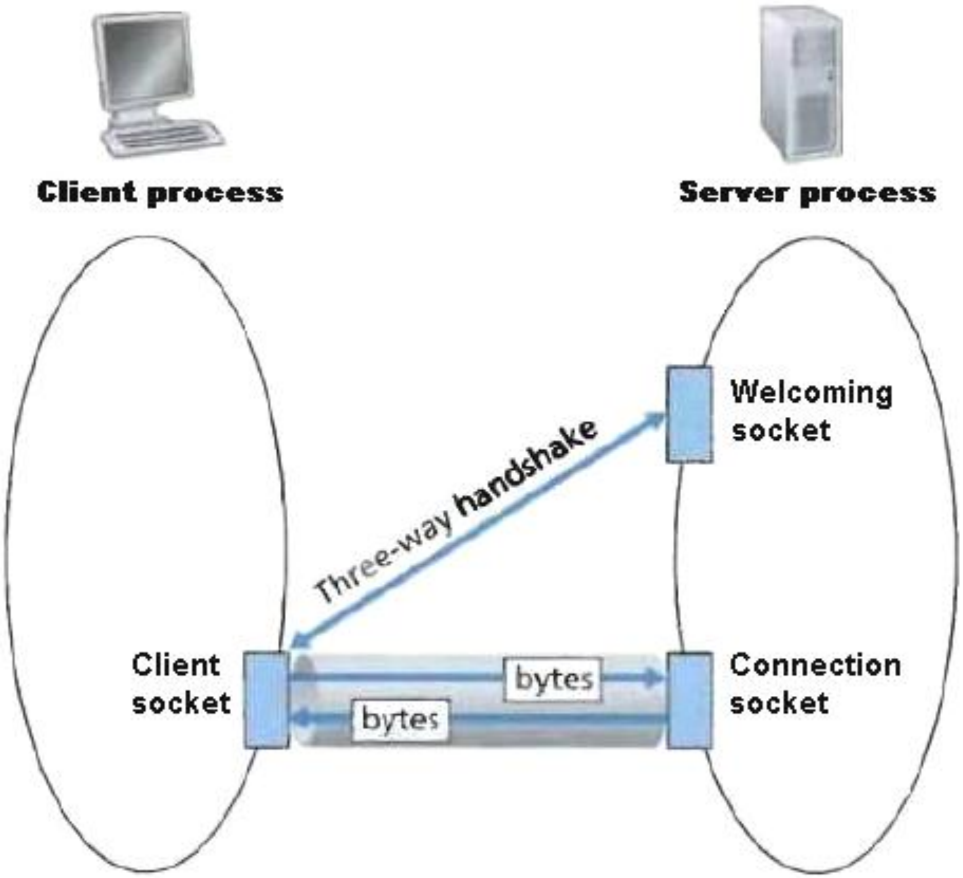
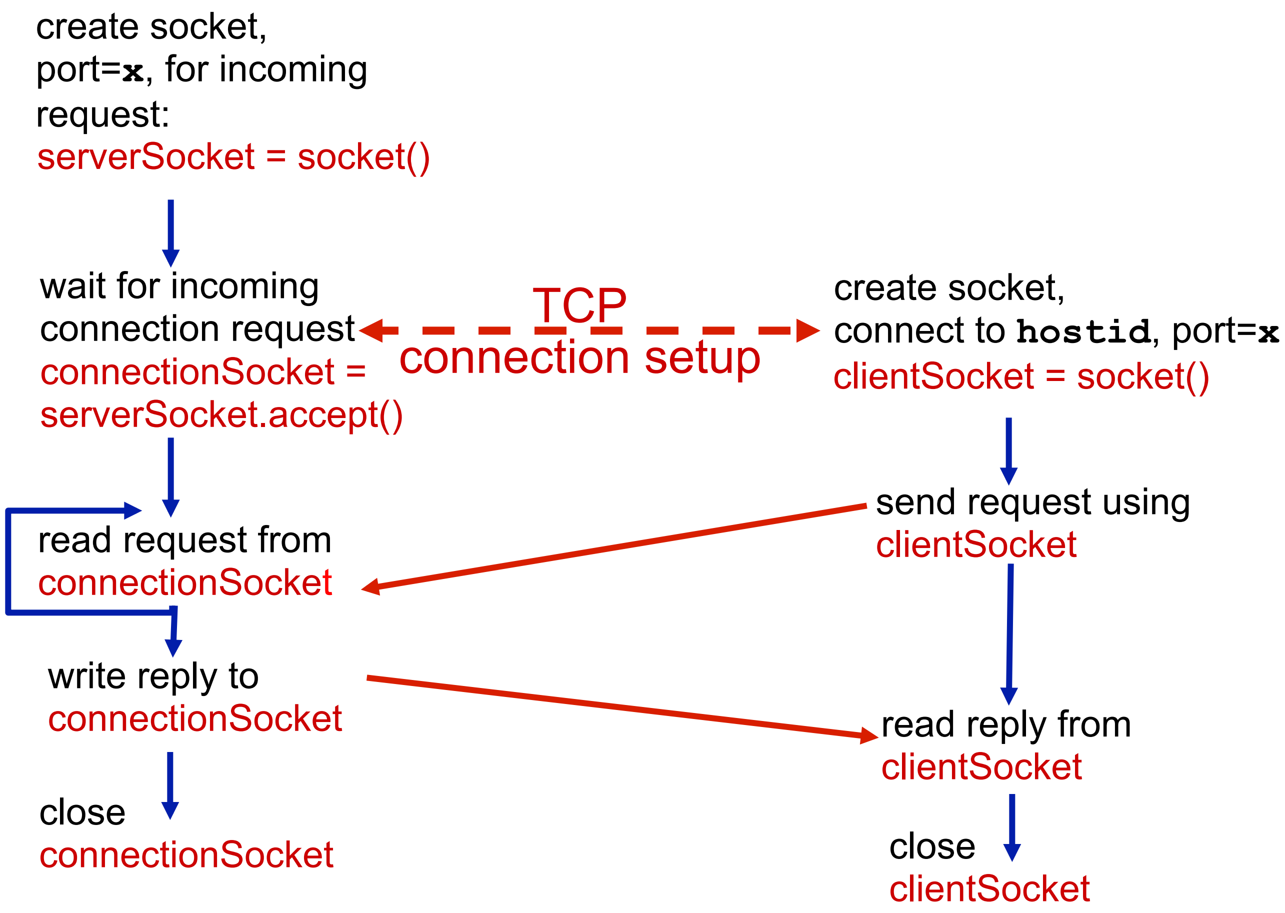


Figure 2. Client-socket, welcoming socket, and connection socket

Client/server socket interaction: TCP

server (running on **hostid**)

client



create socket,
port=**x**, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

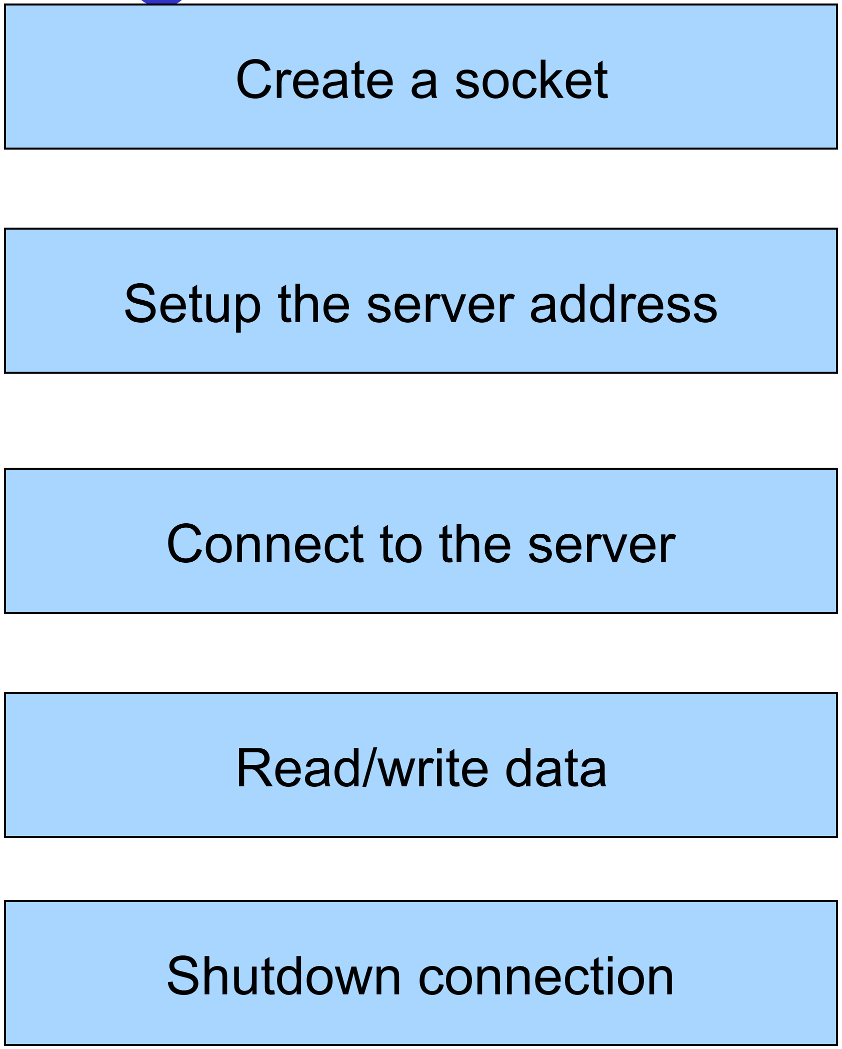
write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

Client – high level view



Create a socket

Setup the server address

Connect to the server

Read/write data

Shutdown connection

Example app: TCP client

Python TCPClient

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, 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: Java client (TCP)

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

This package defines Socket() and ServerSocket() classes

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

create input stream

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

create clientSocket object of type Socket, connect to server

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

server port #

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

create output stream attached to socket

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

Example: Java client (TCP), cont.

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');
```

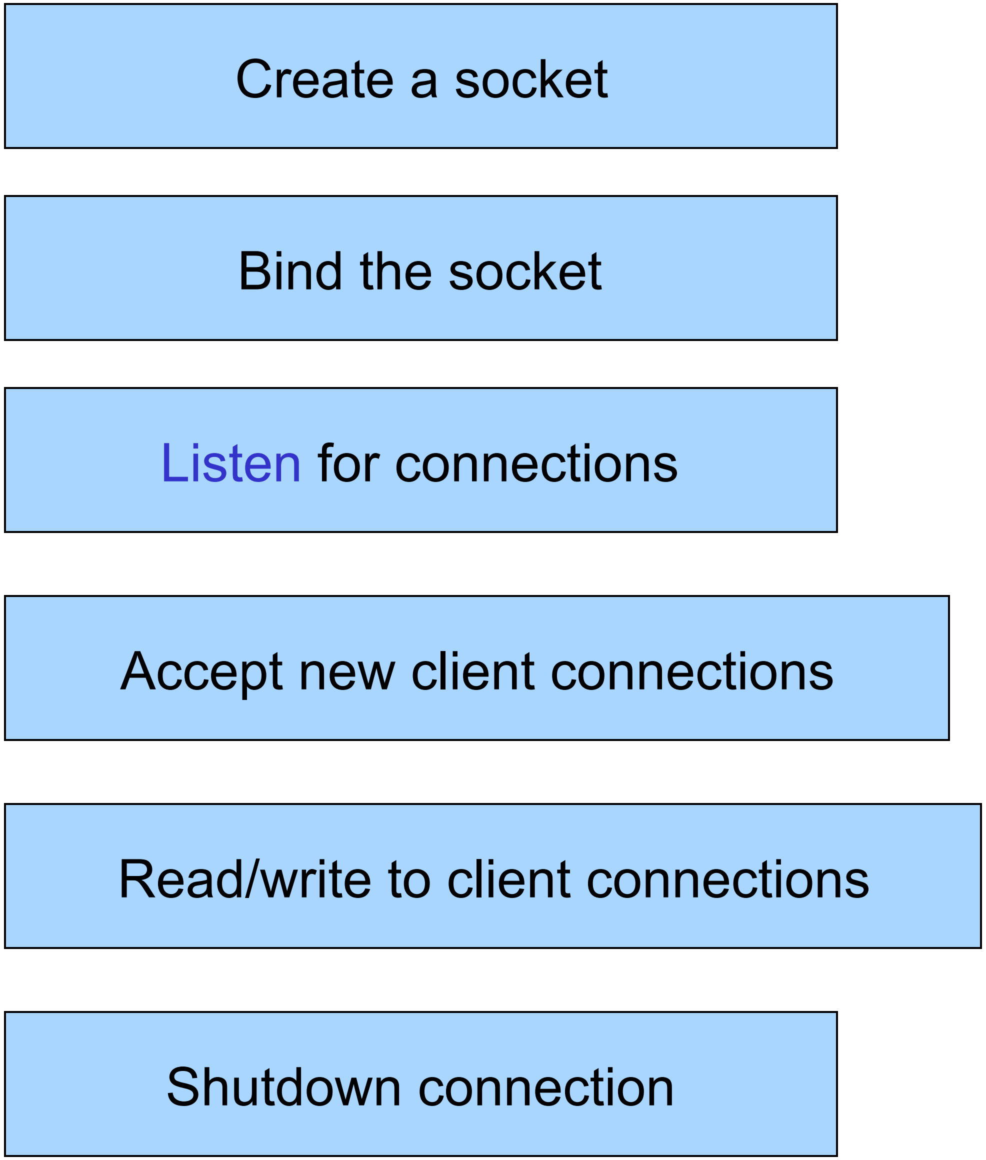
read line from server

```
 modifiedSentence = inFromServer.readLine();

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

close socket
(clean up behind yourself!)

```
 clientSocket.close();

   }
}
```

Server – high level view



Create a socket

Bind the socket

Listen for connections

Accept new client connections

Read/write to client connections

Shutdown connection

Example app: TCP server

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)

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()
```

Example: Java server (TCP)

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

class TCPServer {

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

create welcoming socket at port 6789

```
     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {
```

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

```
        Socket connectionSocket = welcomeSocket.accept();
```

create input stream, attached to socket

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

Example: Java server (TCP), cont

create output stream, attached to socket

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

read in line from socket

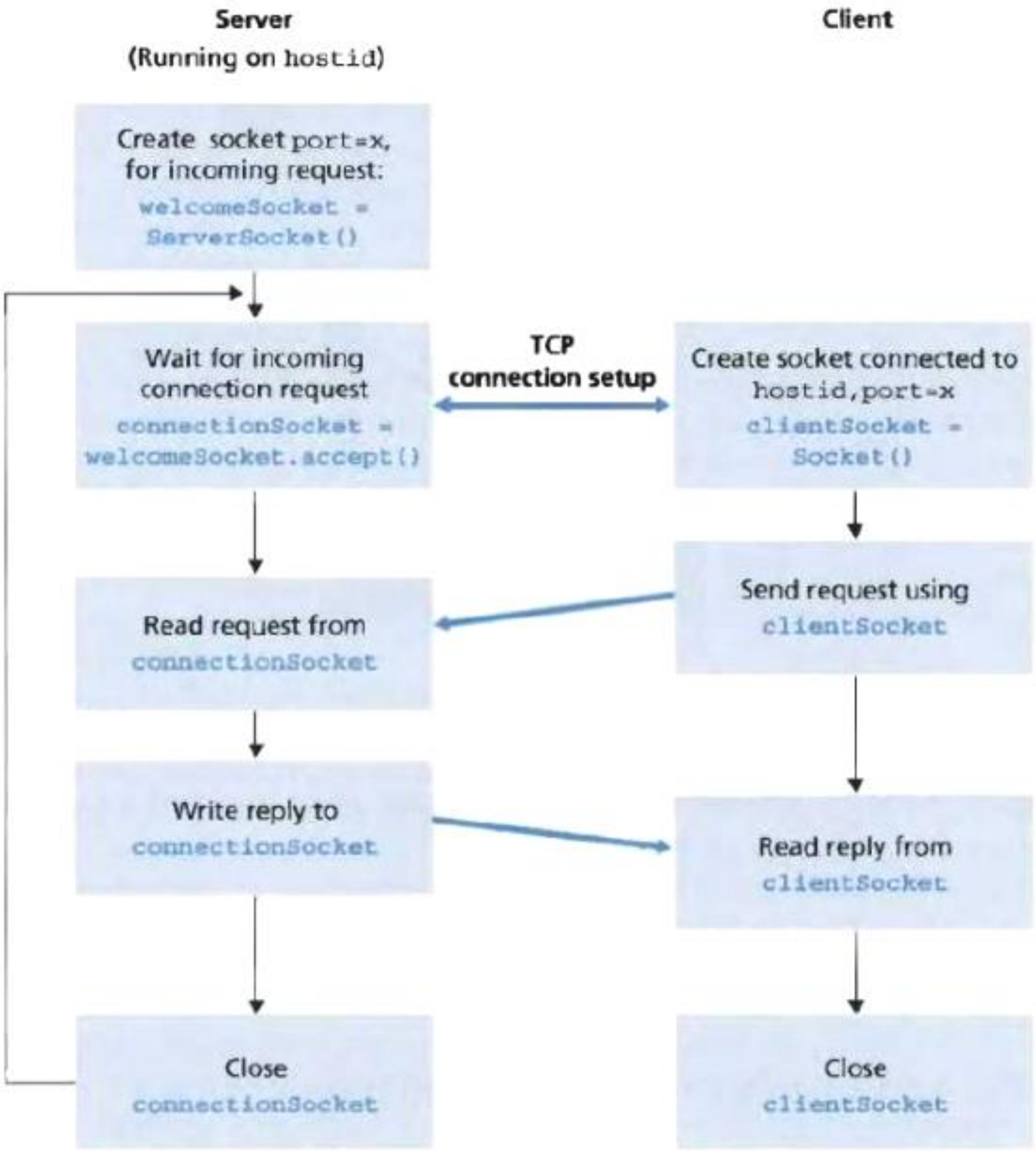
```
clientSentence = inFromClient.readLine();
```

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

write out line to socket

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

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



Server

(Running on hostid)

Create socket port=x, for incoming request:

welcomeSocket = ServerSocket()

Wait for incoming connection request

connectionSocket = welcomeSocket.accept()

Read request from connectionSocket

Write reply to connectionSocket

Close connectionSocket

TCP connection setup

Client

Create socket connected to hostid,port=x

clientSocket = Socket()

Send request using clientSocket

Read reply from clientSocket

Close clientSocket

Figure 1. The client-server application, using connection-oriented transport services