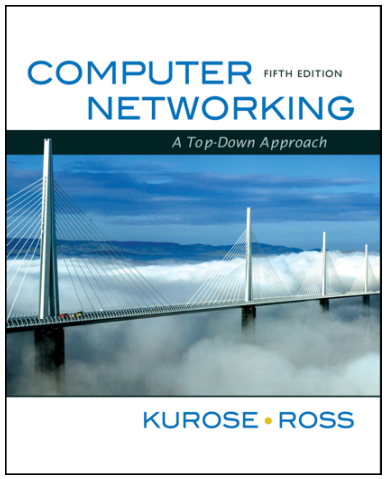
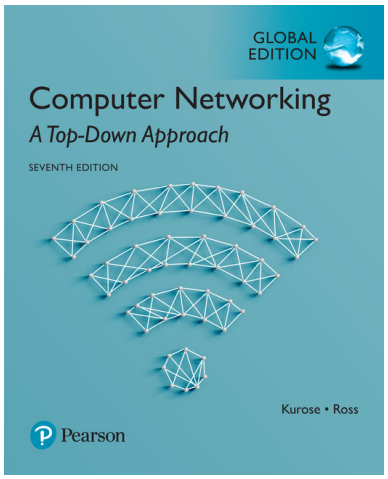
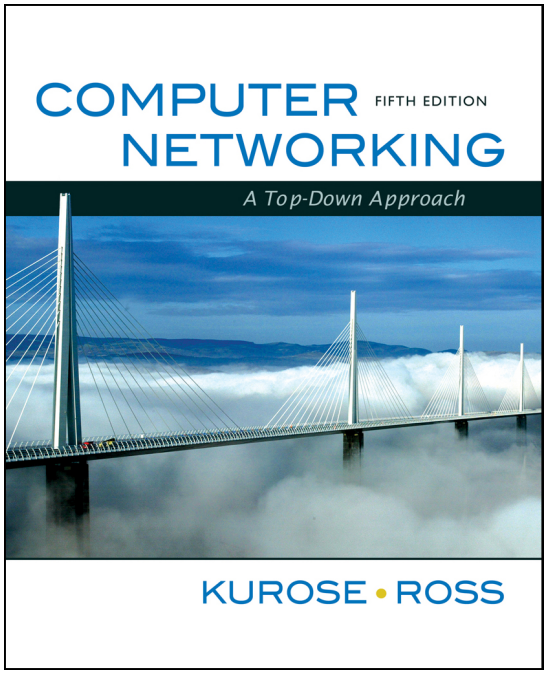
Chapter 2
Socket Programming





Material from 5th and 7th Edition of Kurose-Ross,
Computer Networking: A Top Down Approach

Chapter 2
Application Layer



A note on the use of these ppt slides:

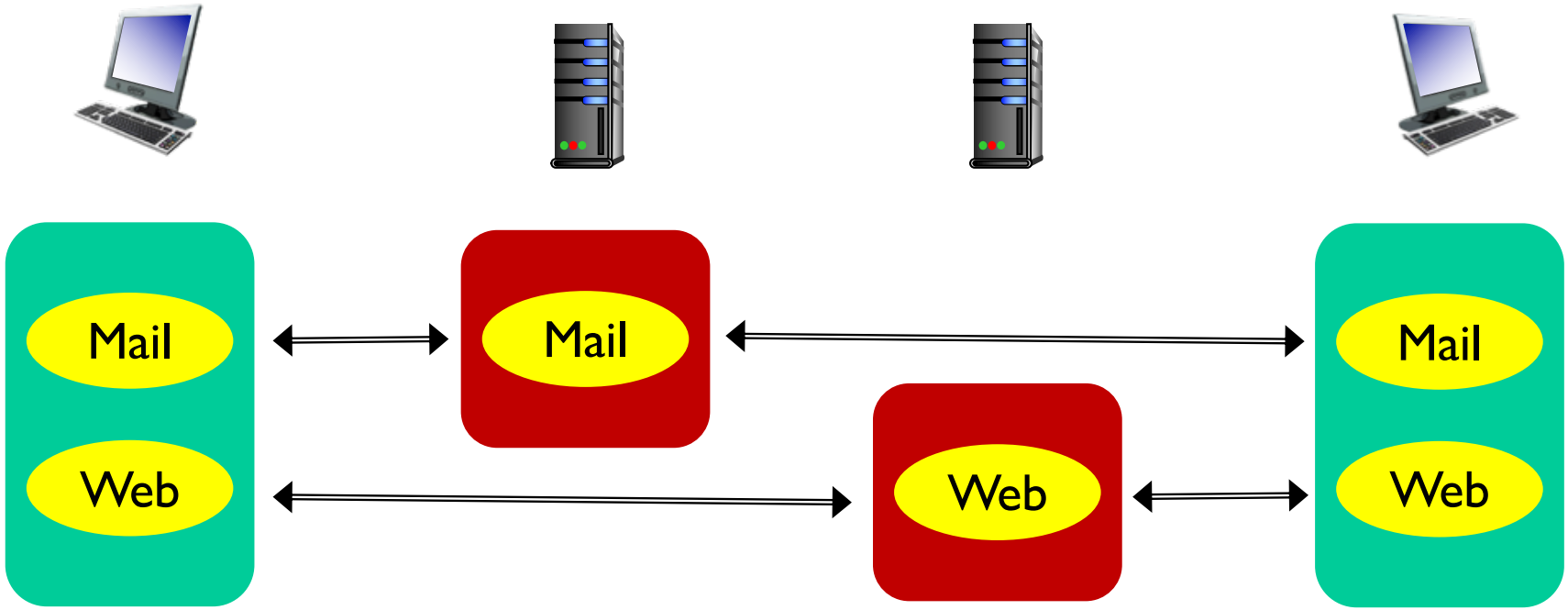
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach, 5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

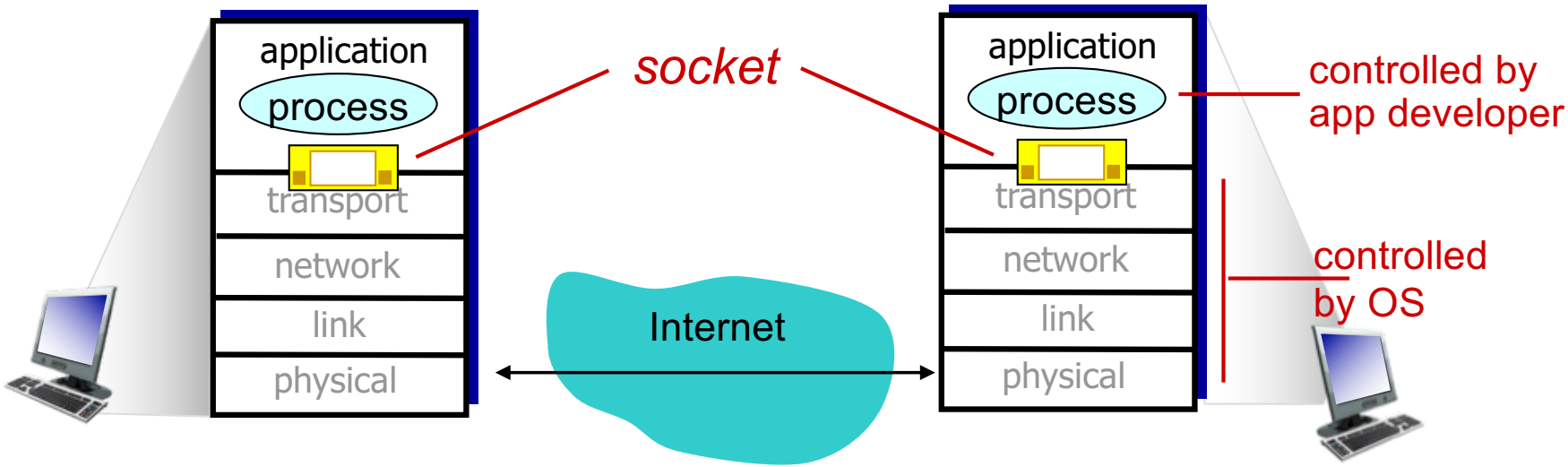
Many-to-many Communication



- Many client processes on same host, communicating with different server processes
- A server process communicates with many client processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

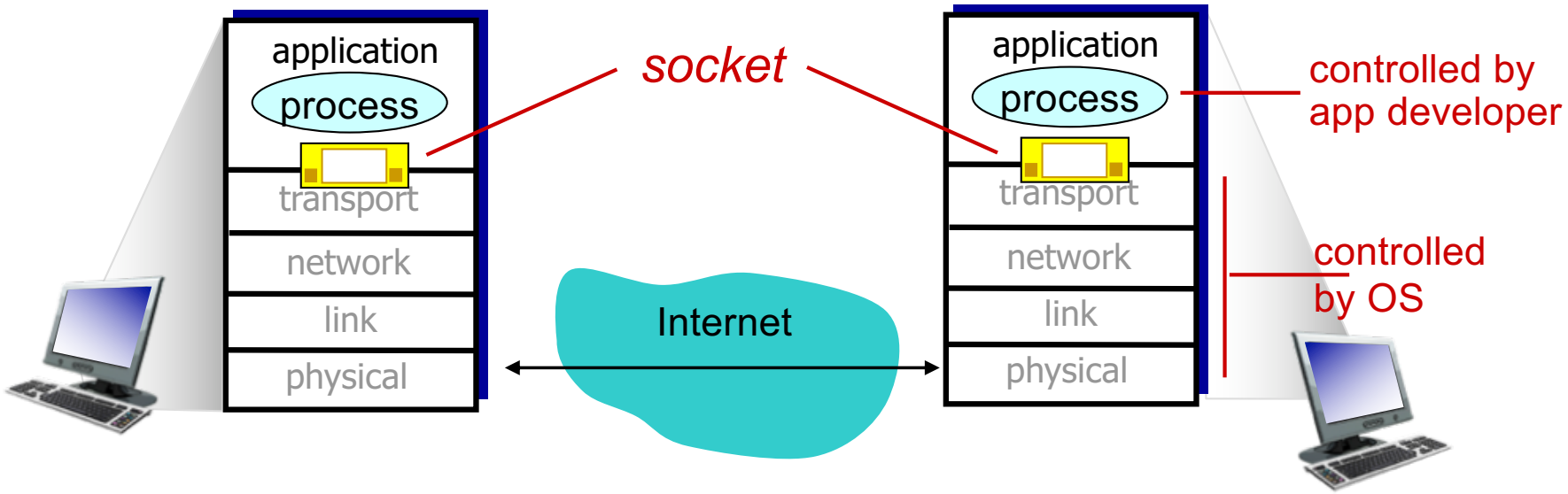
2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

Socket programming

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

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



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 data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

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:

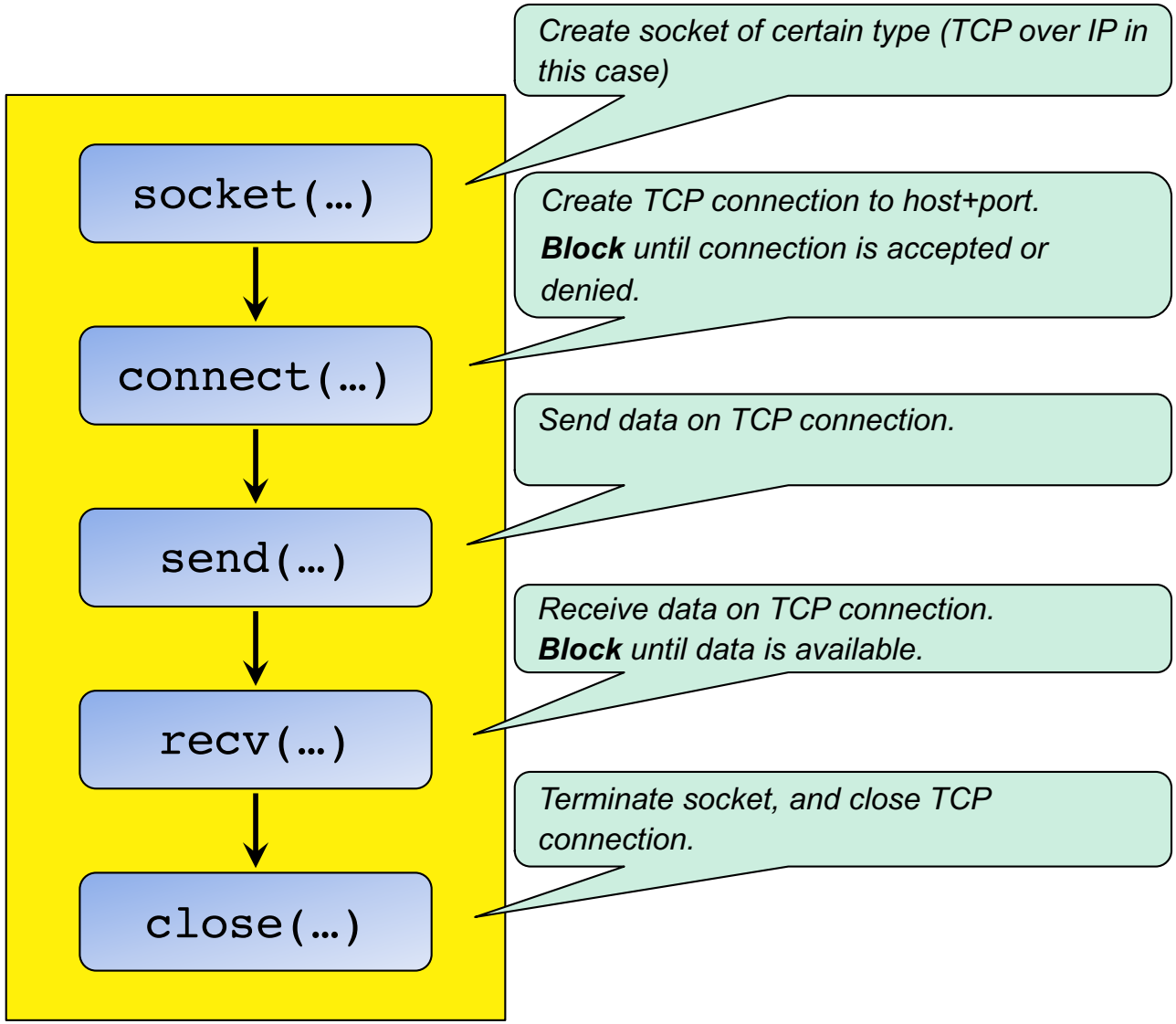
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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 (more in Chap 3)

application viewpoint:

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

TCP Client


socket(…)
connect(…)
send(…)
recv(…)
close(…)
Create socket of certain type (TCP over IP in this case)
Create TCP connection to host+port.
Block until connection is accepted or denied.
Send data on TCP connection.
Receive data on TCP connection.
Block until data is available.
Terminate socket, and close TCP connection.

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 no.

create input stream

create client socket, connect to server

create output stream attached to socket

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);
```

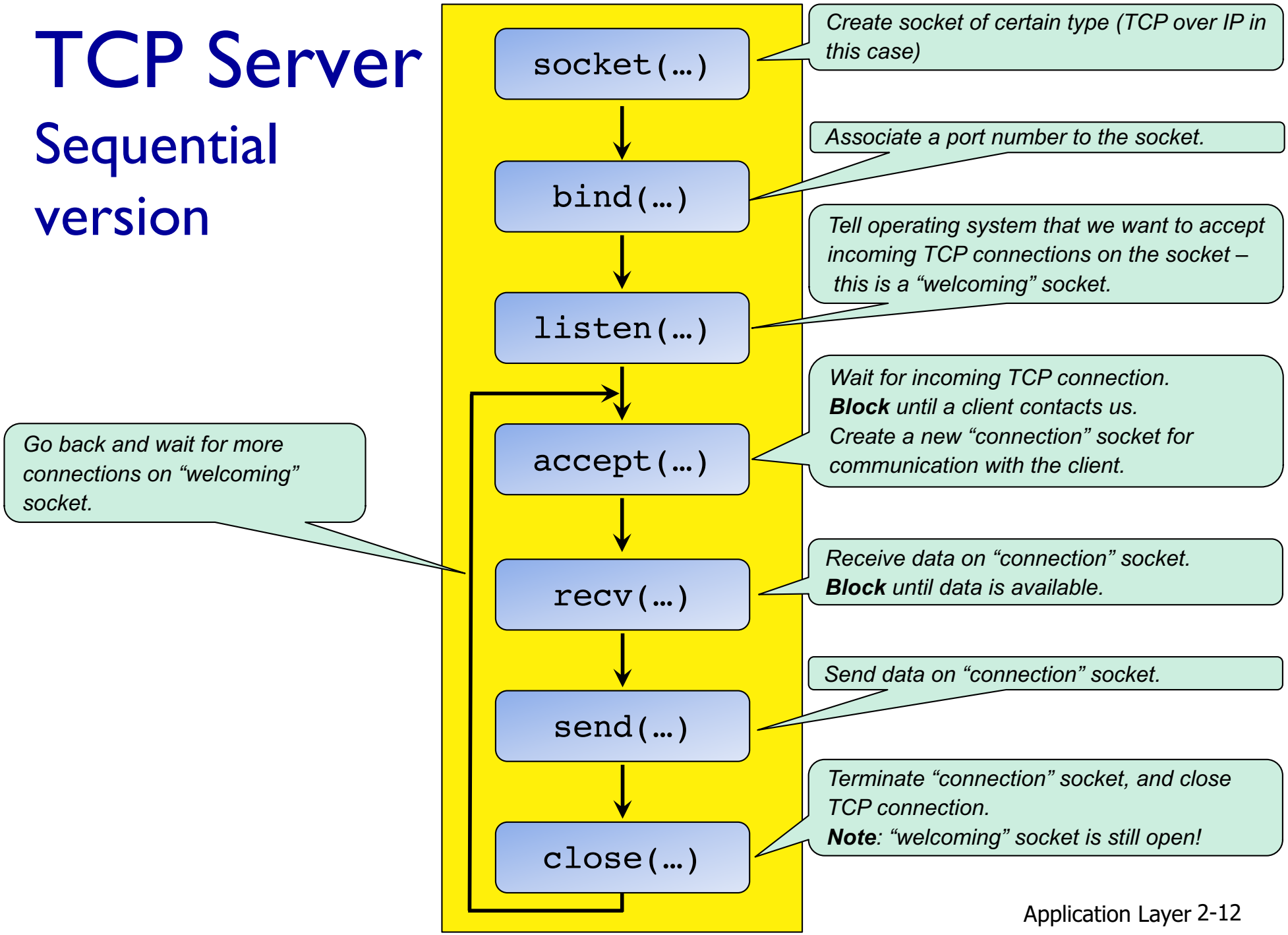
close socket (clean up behind yourself!) →

```
clientSocket.close();

    }
}
```

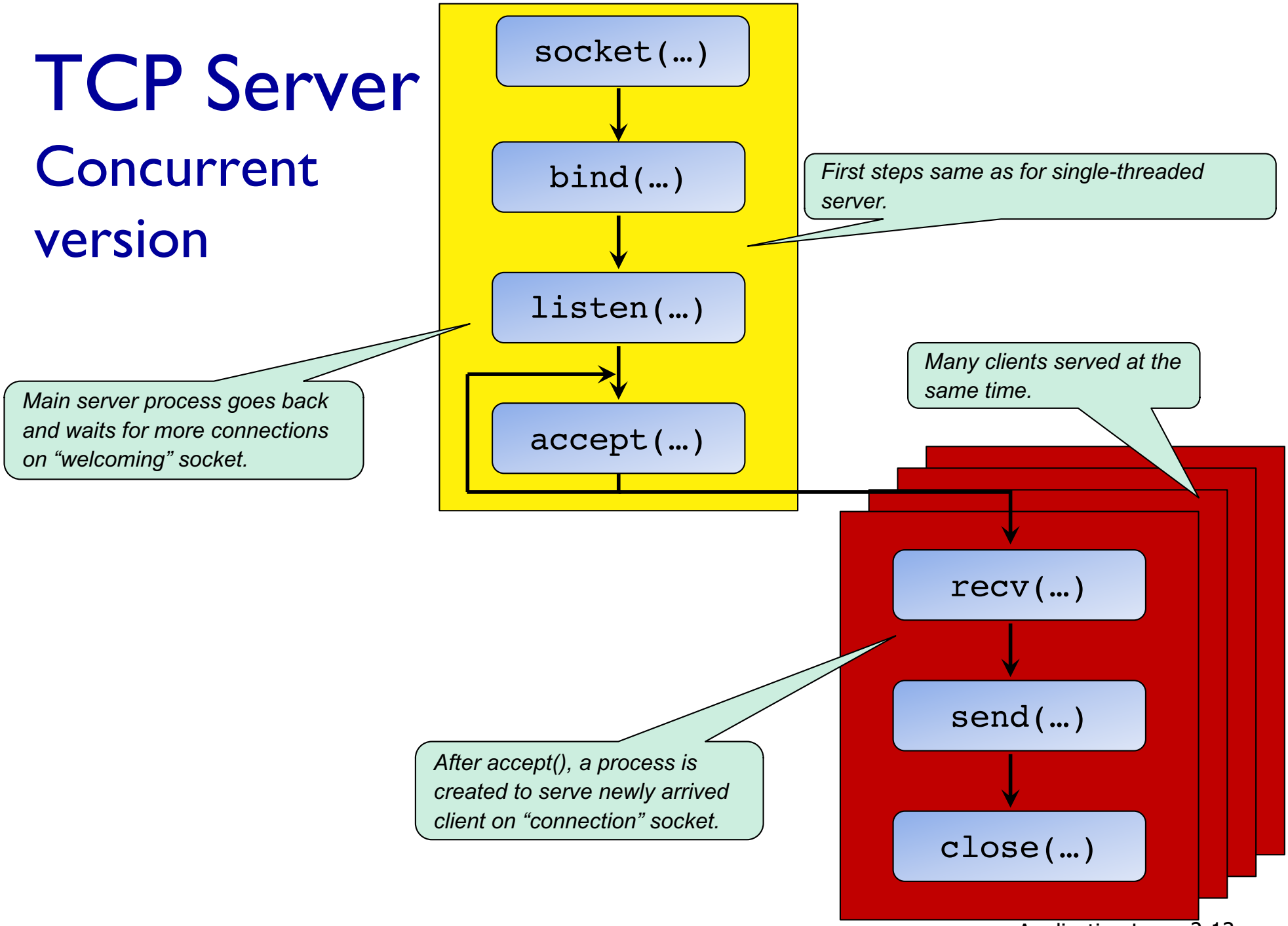
TCP Server

Sequential version



TCP Server

Concurrent version



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, includes bind() + listen()

Wait for client to contact, create *new* socket on return

Create input stream attached to socket

Example: Java server (TCP), cont

create output stream attached to socket

read line from socket

write line to socket

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

                clientSentence = inFromClient.readLine();

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

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

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

Sequential or concurrent server?

Socket programming *with UDP*

UDP: no "connection" between client & server

- no handshaking before sending data
  - no need for connect(), listen(), accept()
- sender explicitly attaches IP destination address and port number to each packet
- receiver extracts sender IP address and port number 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

Example: Java client (UDP)

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

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

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

      DatagramSocket clientSocket = new DatagramSocket();

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

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

      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
```

create input stream

create client socket

translate hostname to IP address using DNS

Example: Java client (UDP), cont.

create datagram with data, length of data, IP address, 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), cont

```
                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);
            }
        }
    }
```

get IP address and port number of sender

create datagram to send to client

write datagram to socket

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

Chapter 2: summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming:

  TCP, UDP sockets