

# Chapter 2

## Application Layer

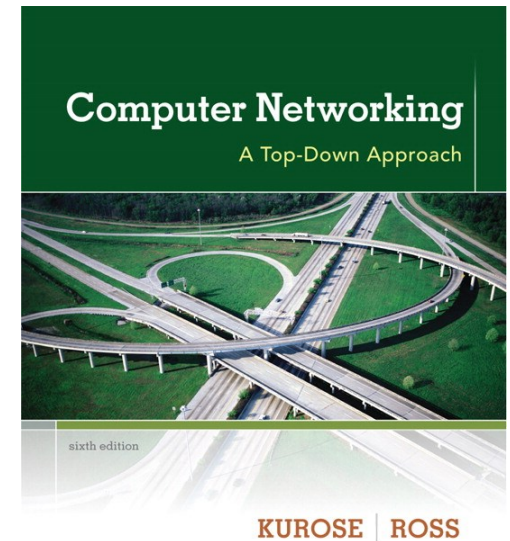
### A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and 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) that you mention their source (after all, we'd like people to use our book!)
- ❖ If you post any slides 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

© All material copyright 1996-2012  
J.F Kurose and K.W. Ross, All Rights Reserved



## Computer Networking: A Top Down Approach

6<sup>th</sup> edition

Jim Kurose, Keith Ross

Addison-Wesley

March 2012

# Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.7 socket programming with UDP and TCP

# Chapter 2: application layer

## our goals:

- ❖ conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- ❖ learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- ❖ creating network applications
  - socket API

# Some network apps

- ❖ e-mail
- ❖ web
- ❖ text messaging
- ❖ remote login
- ❖ P2P file sharing
- ❖ multi-user network games
- ❖ streaming stored video (YouTube, Hulu, Netflix)
- ❖ voice over IP (e.g., Skype)
- ❖ real-time video conferencing
- ❖ social networking
- ❖ search
- ❖ ...
- ❖ ...

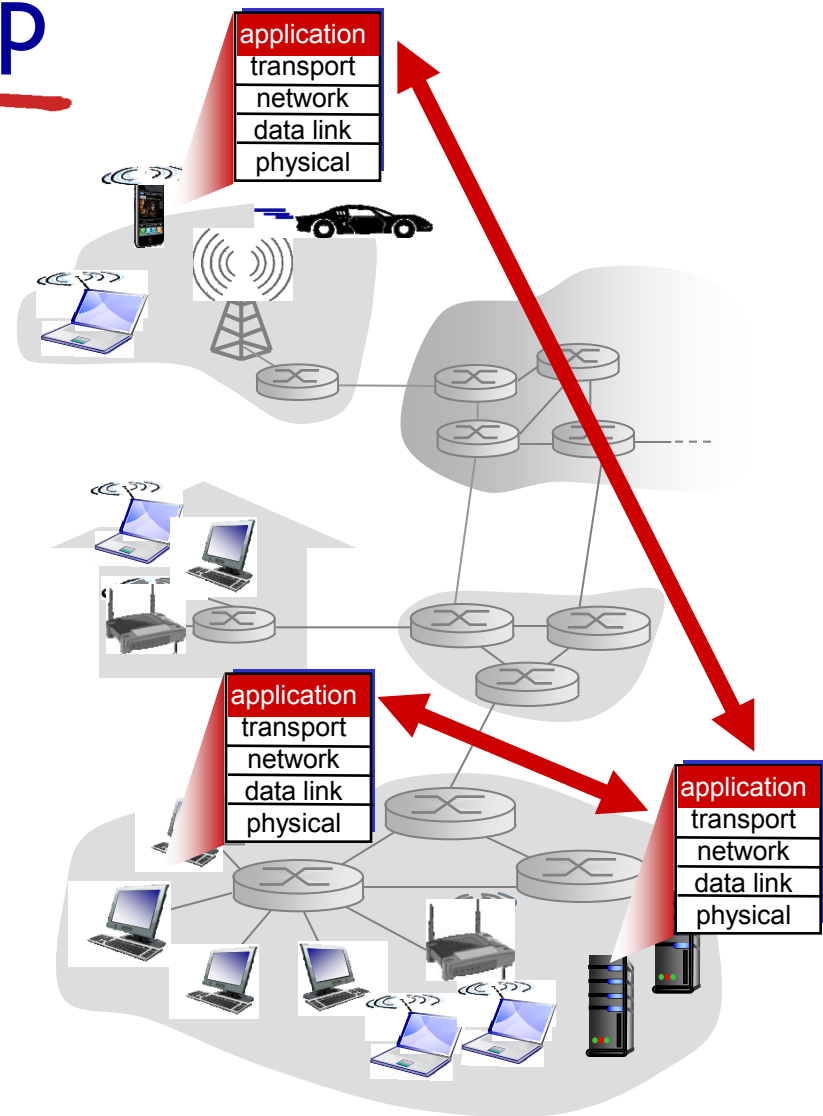
# Creating a network app

write programs that:

- ❖ run on (different) *end systems*
- ❖ communicate over network
- ❖ e.g., web server software communicates with browser software

no need to write software for  
network-core devices

- ❖ network-core devices do not run user applications
- ❖ applications on end systems allows for rapid app development, propagation

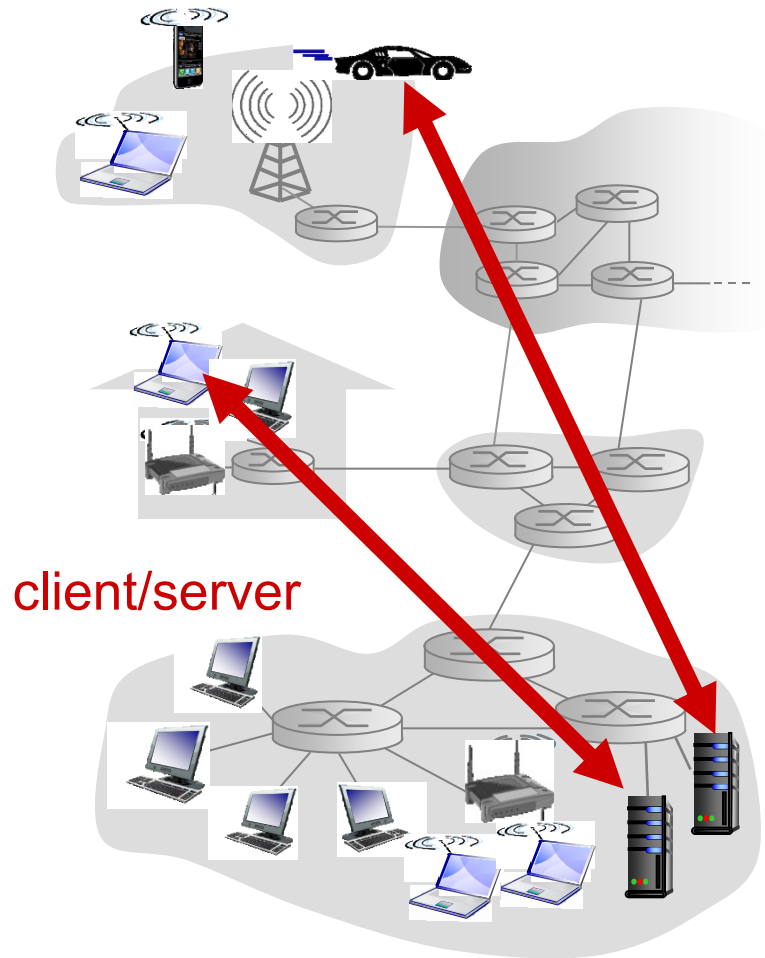


# Application architectures

possible structure of applications:

- ❖ client-server
- ❖ peer-to-peer (P2P)

# Client-server architecture



## server:

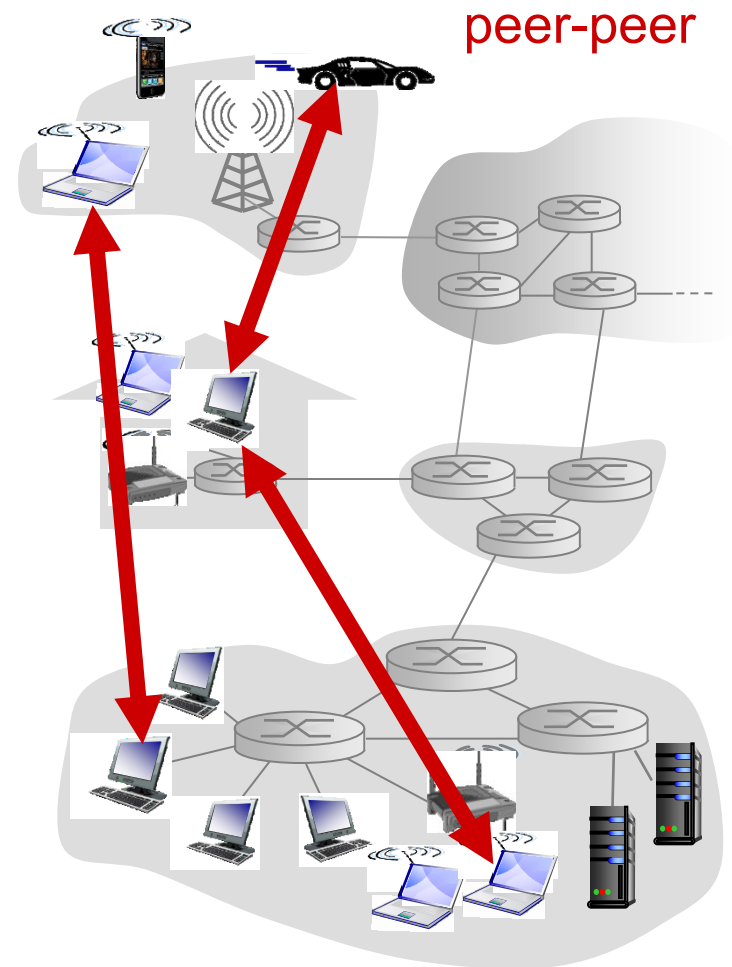
- ❖ always-on host
- ❖ permanent IP address
- ❖ data centers for scaling

## clients:

- ❖ communicate with server
- ❖ may be intermittently connected
- ❖ may have dynamic IP addresses
- ❖ do not communicate directly with each other

# P2P architecture

- ❖ no always-on server
- ❖ arbitrary end systems directly communicate
- ❖ peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- ❖ peers are intermittently connected and change IP addresses
  - complex management





# 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**

clients, servers

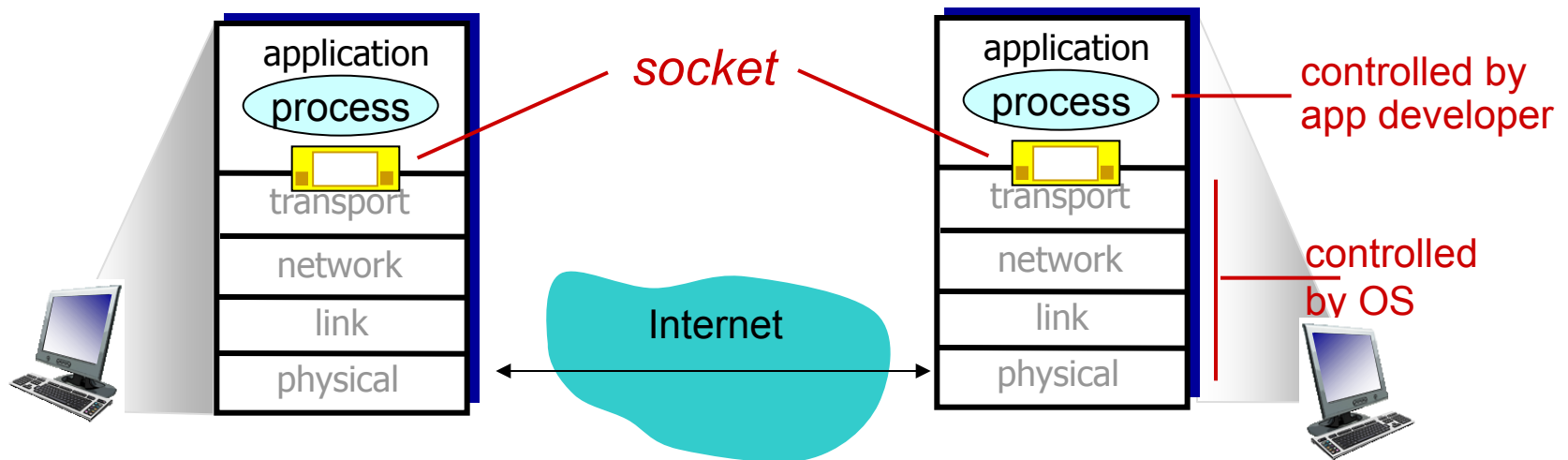
*client process*: process that initiates communication

*server process*: process that waits to be contacted

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



# 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,  
...

# Transport service requirements: common apps

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video: 10kbps-5Mbps	yes, 100' s msec
stored audio/video	loss-tolerant	same as above	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100' s msec yes and no

# Internet transport protocols services

## TCP service:

- ❖ *reliable transport* between sending and receiving process
- ❖ *flow control*: sender won't overwhelm receiver
- ❖ *congestion control*: throttle sender when network overloaded
- ❖ *does not provide*: timing, minimum throughput guarantee, security
- ❖ *connection-oriented*: setup required between client and server processes

## UDP service:

- ❖ *unreliable data transfer* between sending and receiving process
- ❖ *does not provide*: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup,

Q: why bother? Why is there a UDP?

# Internet apps: application, transport protocols

<b>application</b>	<b>application layer protocol</b>	<b>underlying transport protocol</b>
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP



# Securing TCP

## TCP & UDP

- ❖ no encryption
- ❖ cleartext passwds sent into socket traverse Internet in cleartext

## SSL

- ❖ provides encrypted TCP connection
- ❖ data integrity
- ❖ end-point authentication

## SSL is at app layer

- ❖ Apps use SSL libraries, which “talk” to TCP

## SSL socket API

- ❖ cleartext passwds sent into socket traverse Internet encrypted
- ❖ See Chapter 7

# Chapter 2: outline

## 2.1 principles of network applications

- app architectures
- app requirements

## 2.2 Web and HTTP

## 2.7 socket programming with UDP and TCP

# Web and HTTP

*First, a review...*

- ❖ *web page* consists of *objects*
- ❖ object can be HTML file, JPEG image, Java applet, audio file,...
- ❖ web page consists of *base HTML-file* which includes *several referenced objects*
- ❖ each object is addressable by a *URL*, e.g.,

`www.someschool.edu/someDept/pic.gif`

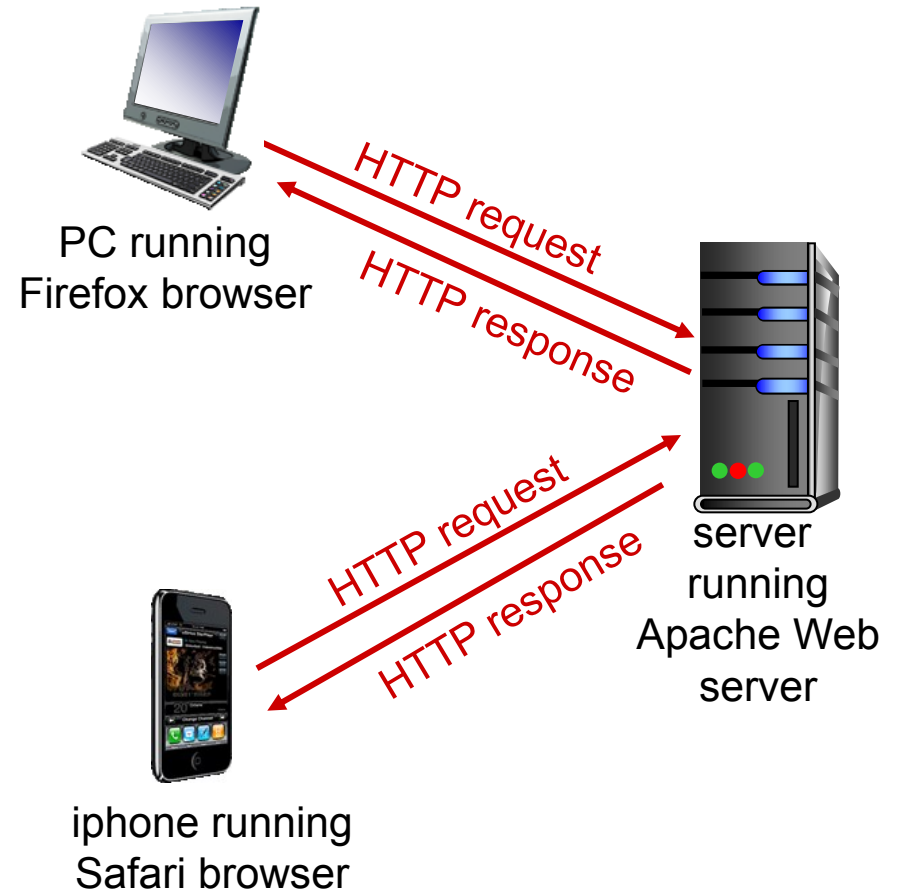
host name

path name

# HTTP overview

## HTTP: hypertext transfer protocol

- ❖ Web's application layer protocol
- ❖ client/server model
  - **client**: browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - **server**: Web server sends (using HTTP protocol) objects in response to requests



# HTTP overview (continued)

## *uses TCP:*

- ❖ client initiates TCP connection (creates socket) to server, port 80
- ❖ server accepts TCP connection from client
- ❖ HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- ❖ TCP connection closed

## *HTTP is “stateless”*

- ❖ server maintains no information about past client requests

## *aside* protocols that maintain “state” are complex!

- ❖ past history (state) must be maintained
- ❖ if server/client crashes, their views of “state” may be inconsistent, must be reconciled

# HTTP connections

## *non-persistent HTTP*

- ❖ at most one object sent over TCP connection
  - connection then closed
- ❖ downloading multiple objects required multiple connections

## *persistent HTTP*

- ❖ multiple objects can be sent over single TCP connection between client, server

# Non-persistent HTTP

suppose user enters URL:

`www.someSchool.edu/someDepartment/home.index`

(contains text,  
references to 10  
jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80

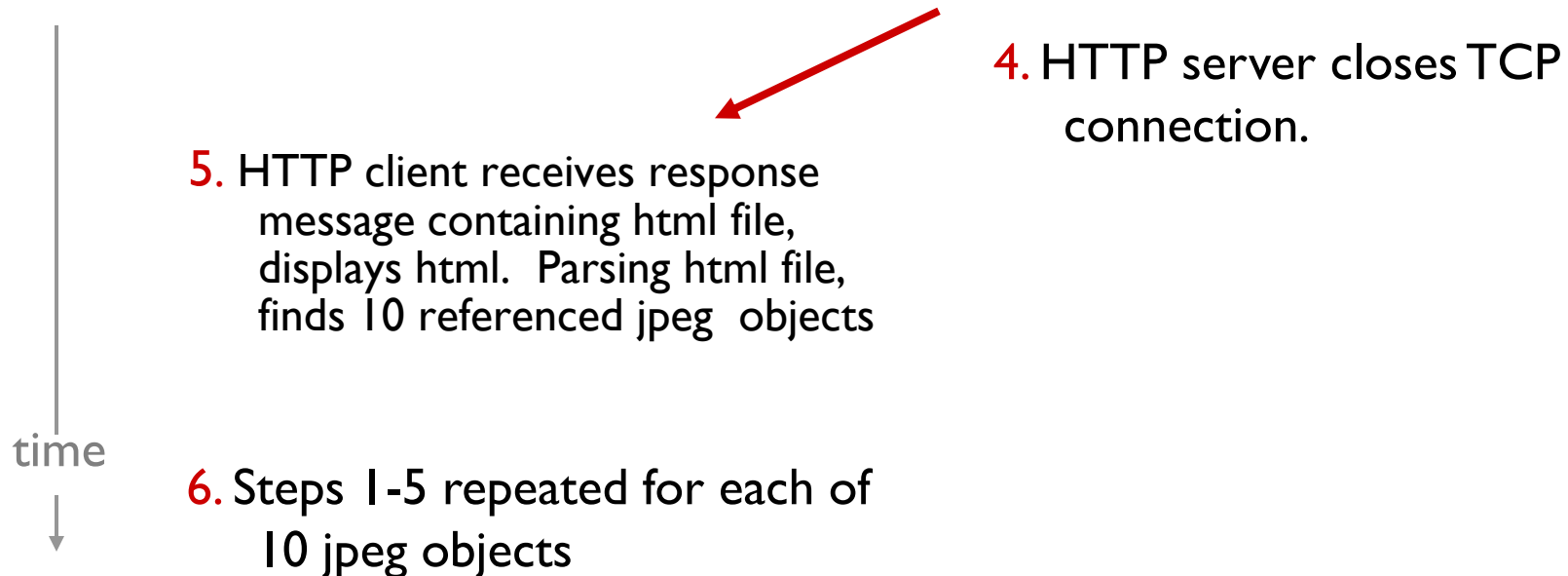
1b. HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80. “accepts” connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time  
↓

# Non-persistent HTTP (cont.)



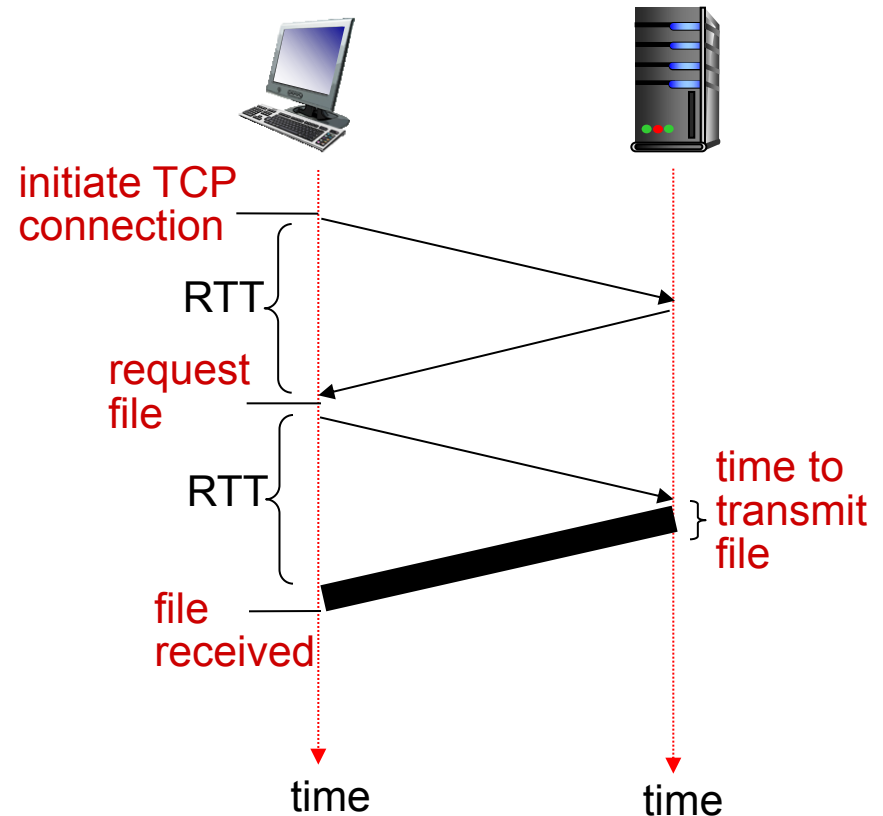


# Non-persistent HTTP: response time

**RTT (definition):** time for a small packet to travel from client to server and back

**HTTP response time:**

- ❖ one RTT to initiate TCP connection
- ❖ one RTT for HTTP request and first few bytes of HTTP response to return
- ❖ file transmission time
- ❖ non-persistent HTTP response time =  
 $2\text{RTT} + \text{file transmission time}$



# Persistent HTTP

## *non-persistent HTTP issues:*

- ❖ requires 2 RTTs per object
- ❖ OS overhead for *each* TCP connection
- ❖ browsers often open parallel TCP connections to fetch referenced objects

## *persistent HTTP:*

- ❖ server leaves connection open after sending response
- ❖ subsequent HTTP messages between same client/server sent over open connection
- ❖ client sends requests as soon as it encounters a referenced object
- ❖ as little as one RTT for all the referenced objects

# HTTP request message

- ❖ two types of HTTP messages: *request, response*
- ❖ **HTTP request message:**
  - ASCII (human-readable format)

request line  
(GET, POST,  
HEAD commands)

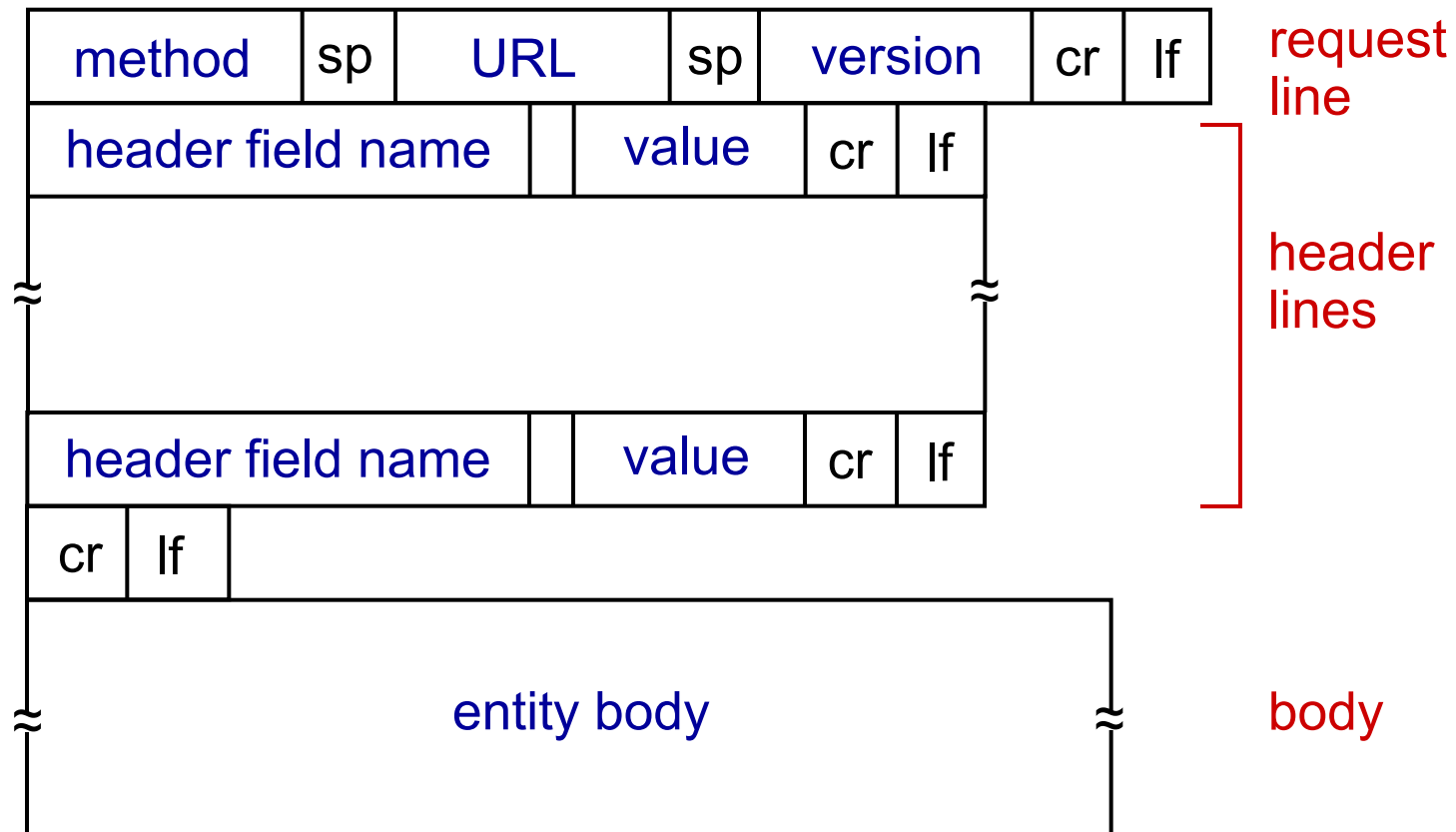
header  
lines

carriage return,  
line feed at start  
of line indicates  
end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return character  
line-feed character

# HTTP request message: general format



# Uploading form input

## POST method:

- ❖ web page often includes form input
- ❖ input is uploaded to server in entity body

## URL method:

- ❖ uses GET method
- ❖ input is uploaded in URL field of request line:

`www.somesite.com/animalsearch?monkeys&banana`

# Method types

## HTTP/1.0:

- ❖ GET
- ❖ POST
- ❖ HEAD
  - asks server to leave requested object out of response

## HTTP/1.1:

- ❖ GET, POST, HEAD
- ❖ PUT
  - uploads file in entity body to path specified in URL field
- ❖ DELETE
  - deletes file specified in the URL field

# HTTP response message

status line  
(protocol  
status code  
status phrase)

header  
lines

data, e.g.,  
requested  
HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
1\r\n
\r\n
data data data data data ...
```

# HTTP response status codes

- ❖ status code appears in 1st line in server-to-client response message.
- ❖ some sample codes:

## **200 OK**

- request succeeded, requested object later in this msg

## **301 Moved Permanently**

- requested object moved, new location specified later in this msg (Location:)

## **400 Bad Request**

- request msg not understood by server

## **404 Not Found**

- requested document not found on this server

## **505 HTTP Version Not Supported**



# Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

opens TCP connection to port 80  
(default HTTP server port) at cis.poly.edu.  
anything typed in sent  
to port 80 at cis.poly.edu

2. type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1  
Host: cis.poly.edu
```

by typing this in (hit carriage  
return twice), you send  
this minimal (but complete)  
GET request to HTTP server

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

# Chapter 2: outline

## 2.1 principles of network applications

- app architectures
- app requirements

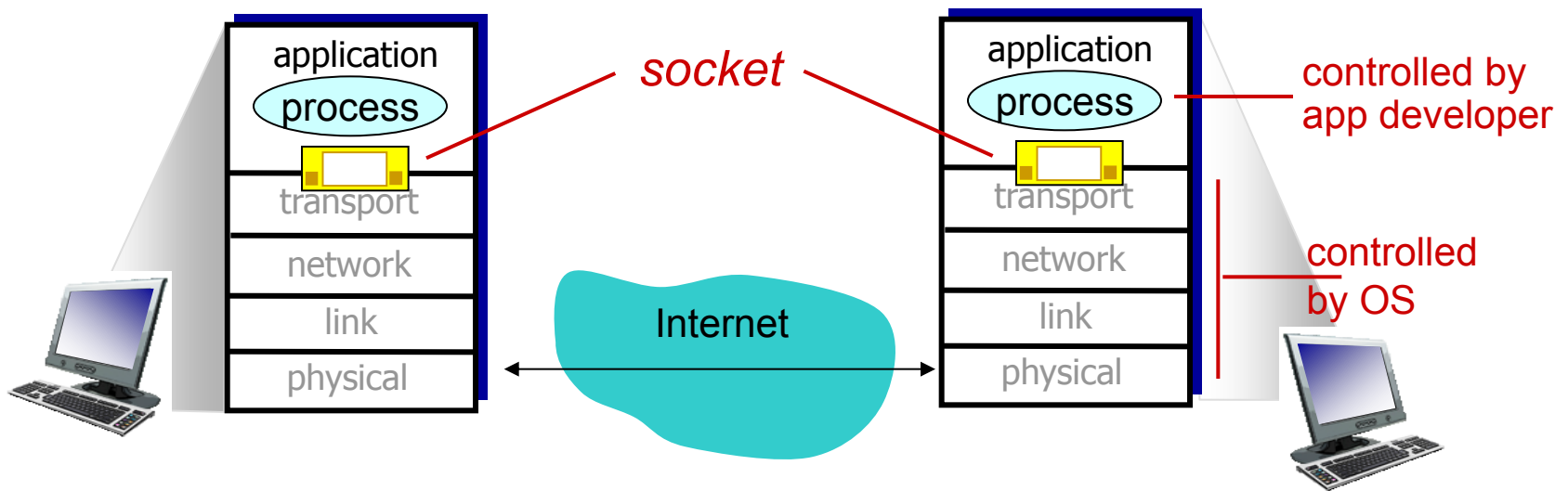
## 2.2 Web and HTTP

## 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 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 *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

# 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 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()
```

# Example app: UDP server

## *Python UDPServer*

```
from socket import *  
serverPort = 12000  
  
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)  
bind socket to local port number 12000 → serverSocket.bind(("", serverPort))  
print "The server is ready to receive"  
  
loop forever → while 1:  
    Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)  
    modifiedMessage = message.upper()  
    send upper case string back to this client → serverSocket.sendto(modifiedMessage, clientAddress)
```



# 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)

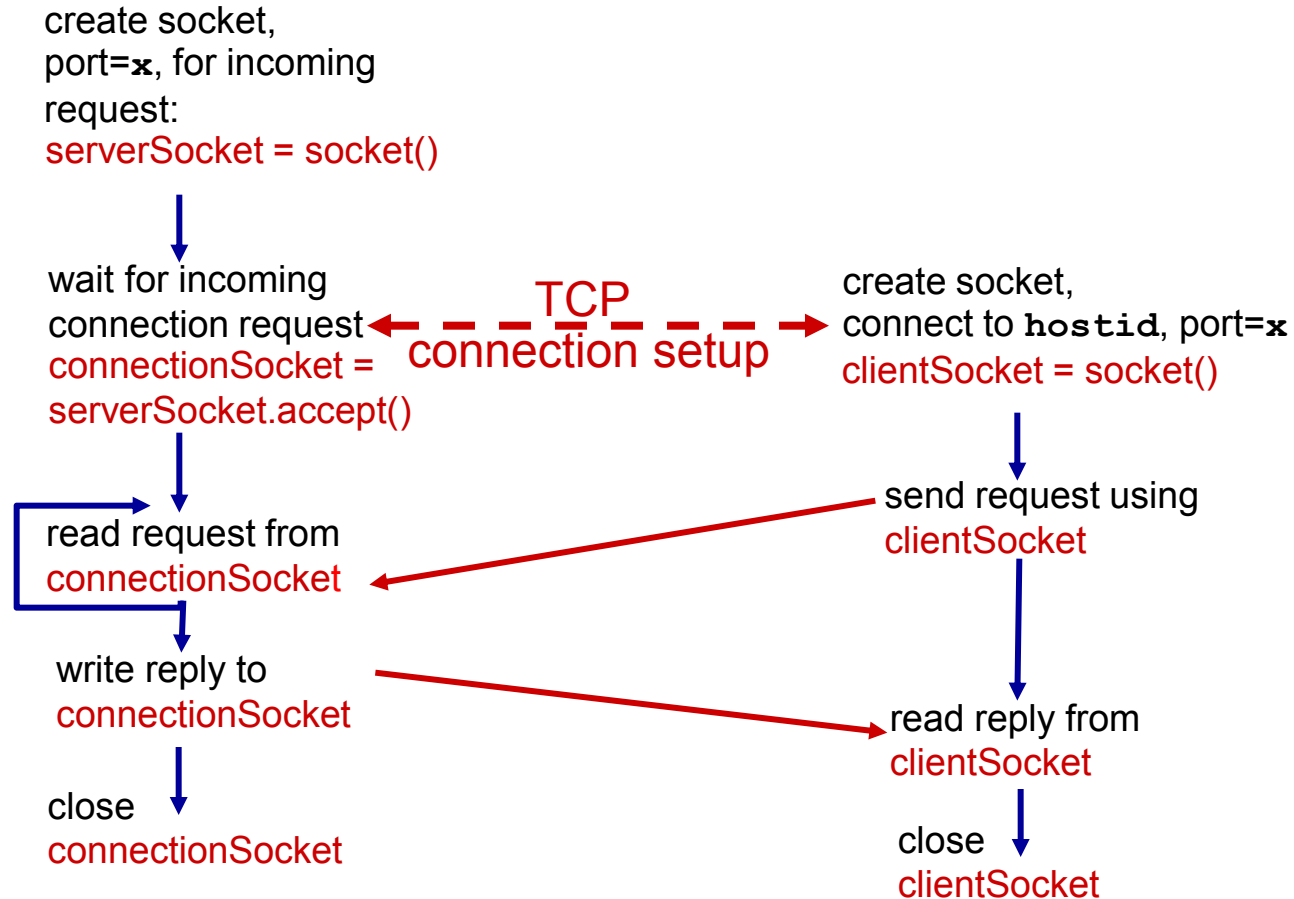
## **application viewpoint:**

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

# Client/server socket interaction: TCP

## server (running on `hostid`)

## client



# Example app:TCP client

## *Python TCPClient*

```
from socket import *
```

```
serverName = 'servername'
```

```
serverPort = 12000
```

create TCP socket for  
server, remote port 12000

```
→ clientSocket = socket(AF_INET, SOCK_STREAM)
```

```
clientSocket.connect((serverName,serverPort))
```

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

No need to attach server  
name, port

```
→ clientSocket.send(sentence)
```

```
modifiedSentence = clientSocket.recv(1024)
```

```
print 'From Server:', modifiedSentence
```

```
clientSocket.close()
```

# Example app: TCP server

## *Python TCPServer*

create TCP welcoming socket	→	<pre>from socket import * serverPort = 12000 serverSocket = socket(AF_INET, SOCK_STREAM) serverSocket.bind(('', serverPort)) serverSocket.listen(1) print 'The server is ready to receive'</pre>
server begins listening for incoming TCP requests	→	<pre>while 1:</pre>
loop forever	→	<pre>    connectionSocket, addr = serverSocket.accept()      sentence = connectionSocket.recv(1024)     capitalizedSentence = sentence.upper()     connectionSocket.send(capitalizedSentence)     connectionSocket.close()</pre>
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 <i>not</i> welcoming socket)	→	

# 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
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, DHT
- ❖ socket programming: TCP, UDP sockets

# Chapter 2: summary

*most importantly: learned about protocols!*

- ❖ typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- ❖ message formats:
  - headers: fields giving info about data
  - data: info being communicated

## *important themes:*

- ❖ control vs. data msgs
  - in-band, out-of-band
- ❖ centralized vs. decentralized
- ❖ stateless vs. stateful
- ❖ reliable vs. unreliable msg transfer
- ❖ “complexity at network edge”