

# Data Communication and Computer Networks

## 2. Application Layer PART-A

***Dr. Aiman Hanna***

Department of Computer Science & Software Engineering  
Concordia University, Montreal, Canada

These slides has mainly been extracted, modified and updated from original slides of :  
Computer Networking: A Top Down Approach, 6th edition Jim Kurose, Keith Ross  
Addison-Wesley, 2013

Additional materials have been extracted, modified and updated from:  
Understanding Communications and Networking, 3e by William A. Shay 2005

Copyright © 1996-2013 J.F Kurose and K.W. Ross  
Copyright © 2005 William A. Shay  
Copyright © 2019 Aiman Hanna  
All rights reserved

# Some network applications

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

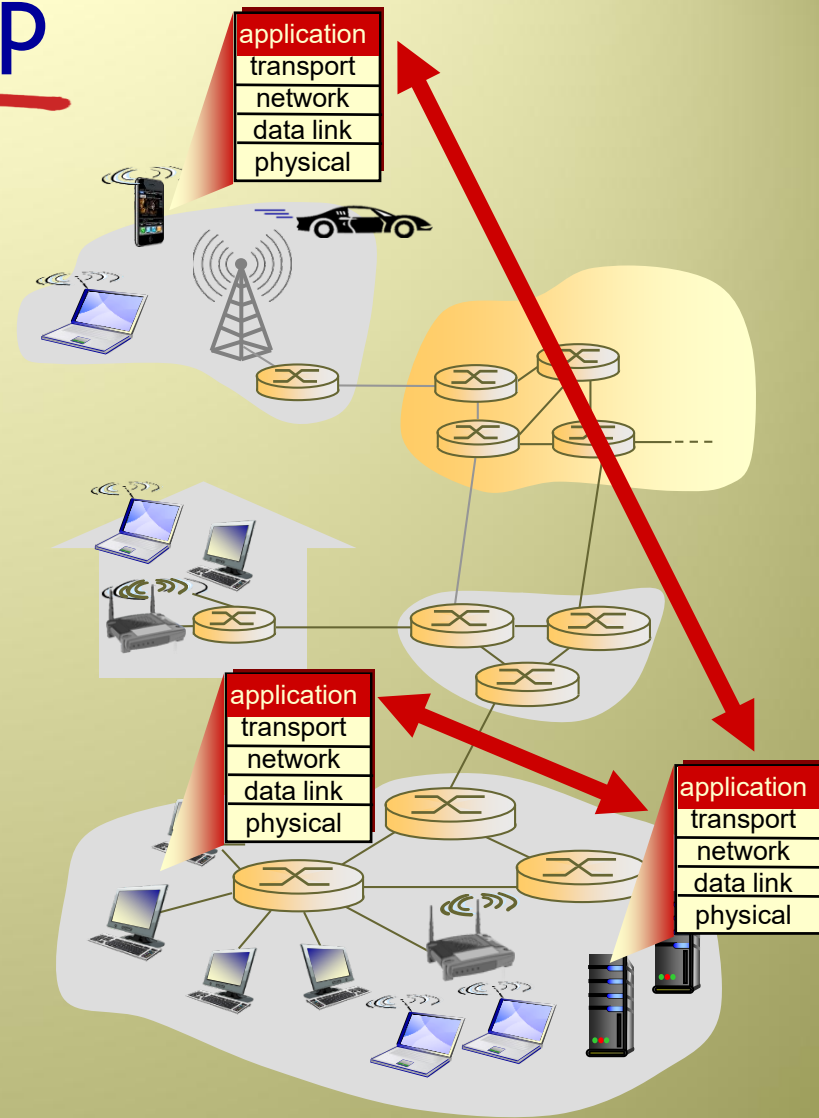
# Creating a network app

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. Why?
- ❖ applications on end systems allows for rapid app development, propagation

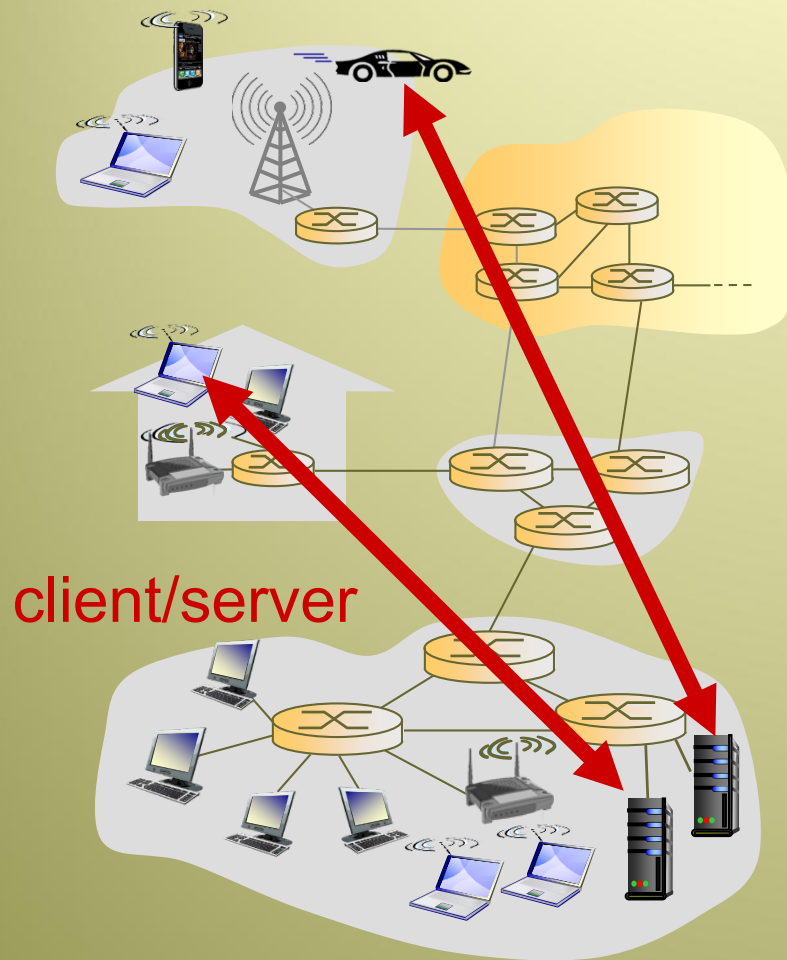


# Application architectures

possible structure of applications:

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

# Client-server architecture



## server:

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

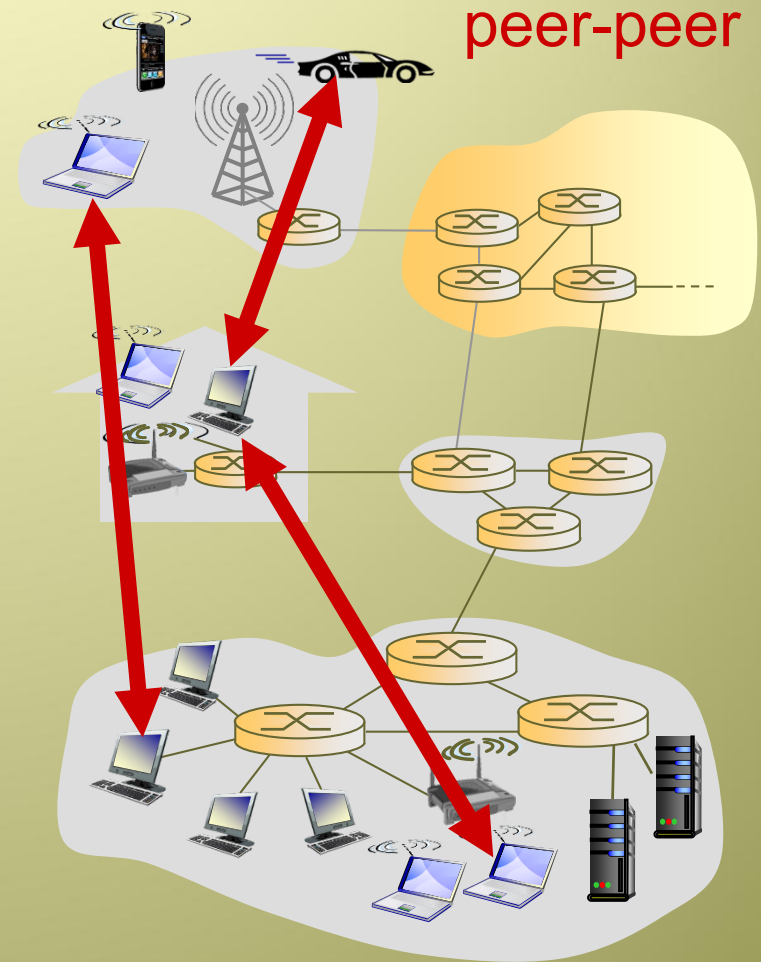
## clients:

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

- ❖ famous client-server apps include Web, FTP, e-mail, Telnet, ...

# 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
- ❖ examples of P2P applications include Skype, IPTV, BitTorrent,...



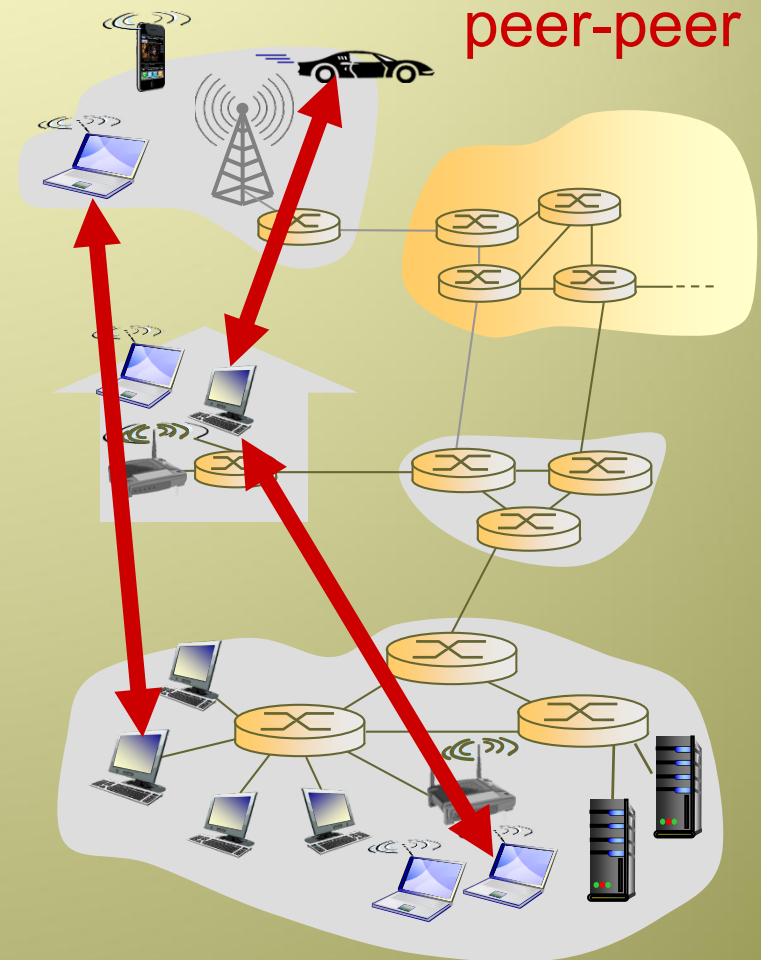
# P2P architecture

## ❖ Cost-effective

- no need for server infrastructure and server bandwidth

## ❖ However, future P2P applications face 3 major challenges:

- *ISP-Friendly* : residential ISP (DSL, Cable, ...) are asymmetric
- *Security*: a challenge, since P2P apps are highly distributed and naturally open
- *Incentive*: users must volunteer bandwidth, storage space and computational resources to P2P apps



# 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

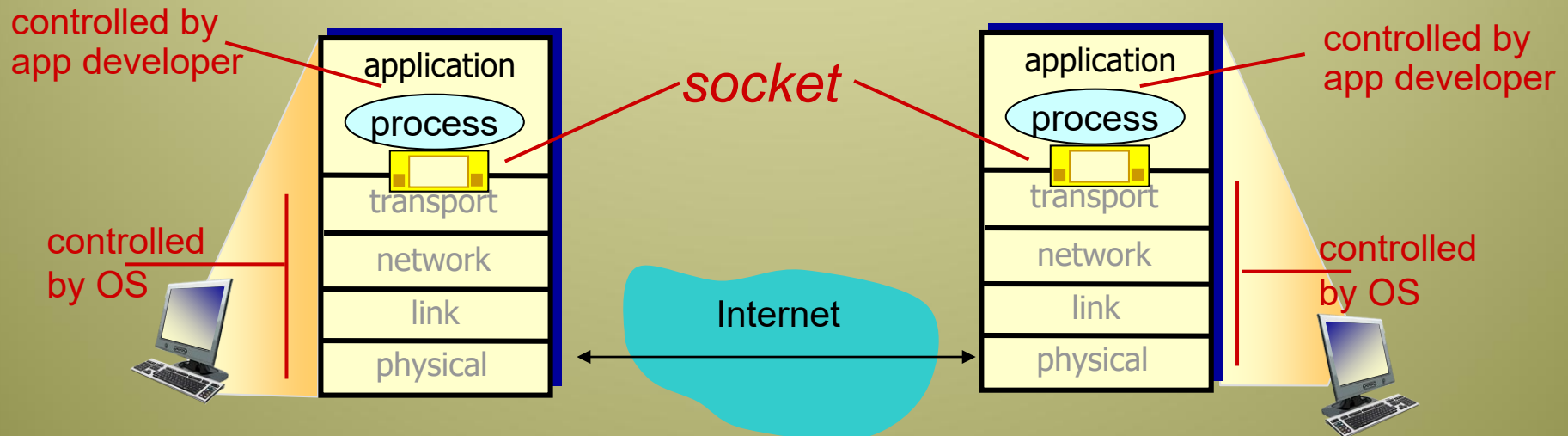
*server process*: process that waits to be contacted

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



# Sockets

- ❖ process sends/receives messages to/from its **socket** (a software interface; an **API**)
- ❖ 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 receiving process (more precisely, the receiving socket)?
  - A: no, *many* network processes can be running on same host
- ❖ *identifier* includes both **IP address** and **port numbers** associated with process on host.
- ❖ example port numbers:
  - HTTP (web) server: 80
  - SMTP (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 / reliability

- ❖ 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) are **bandwidth-sensitive**; they require minimum amount of throughput to be “effective”
- ❖ other apps (e.g. file transfer, e-mail,) are “**elastic** apps”; they can make use of whatever throughput they get

## security

- ❖ encryption, data integrity, end-to-end authentication,

...

# Transport service requirements: common apps

application	data loss	throughput	time sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Real-time audio/video (i.e. Internet telephony, videoconferencing)	Loss-tolerant	Audio: 5kbps-1Mbps Video: 10 Kbps – 5 Mbps	Yes; 100s ms
Streaming stored audio/video	Loss-tolerant	Same as above;	Yes; few seconds
Interactive games	Loss-tolerant	few kbps – 10 Kbps	Yes; 100s ms
Text/Instant messaging	No loss	Elastic	Yes and No

# Internet transport protocols services

---

## TCP service:

- ❖ *connection-oriented*: setup required between client and server processes
- ❖ *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

## 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

Application	Application layer protocol	Underlying transport protocol
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, Dailymotion, ...)	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], proprietary (e.g., Skype)	TCP or UDP (Why?)

# Securing TCP

## TCP & UDP

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

## Secure Socket Layer: **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
- ❖ More details shortly...



# Services **not** provided by Internet transport protocols

The services: 1) reliability, 2) throughput,  
3) timing, 4) security

- ❖ Which of these services are provided by TCP and UDP?
  - 1) Reliability (by TCP),
  - 4) Security (by SSL-enhanced-TCP)
  - What about throughput and timing?!!!
- ❖ NO! these services are not provided by today's Internet transport protocols.
- ❖ Then, we cannot run time-sensitive applications (i.e. Internet telephony) over the Internet. Correct?
- ❖ No; that is incorrect! How then!

# Application-layer protocols

- ❖ application-layer protocols define how application processes pass messages to each other
- ❖ in particular, they define:
  - Types of exchanged messages: i.e. *request* and *response* messages
  - Syntax of message types: i.e. what the different fields in the message are
  - Semantics of the fields: the meaning of the information in the fields
  - Rules of communication: i.e. when and how a process can send a message (or respond to a message)
- ❖ some public application-layer protocols are specified in RFCs; i.e. RFC 2616 for HTTP
  - for instance, if a browser developer follows the rules of RFC2616, the browser will be able to retrieve web pages from any server that followed the rules of HTTP RFC

# Network Applications

- ❖ there are MANY network applications (possibly new ones are being written as you read these slides!)
- ❖ we will here look at 5 of the most important and pervasive network applications:
  - Web and HTTP
  - FTP
  - Electronic Mail
  - DNS
  - P2P

# 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`

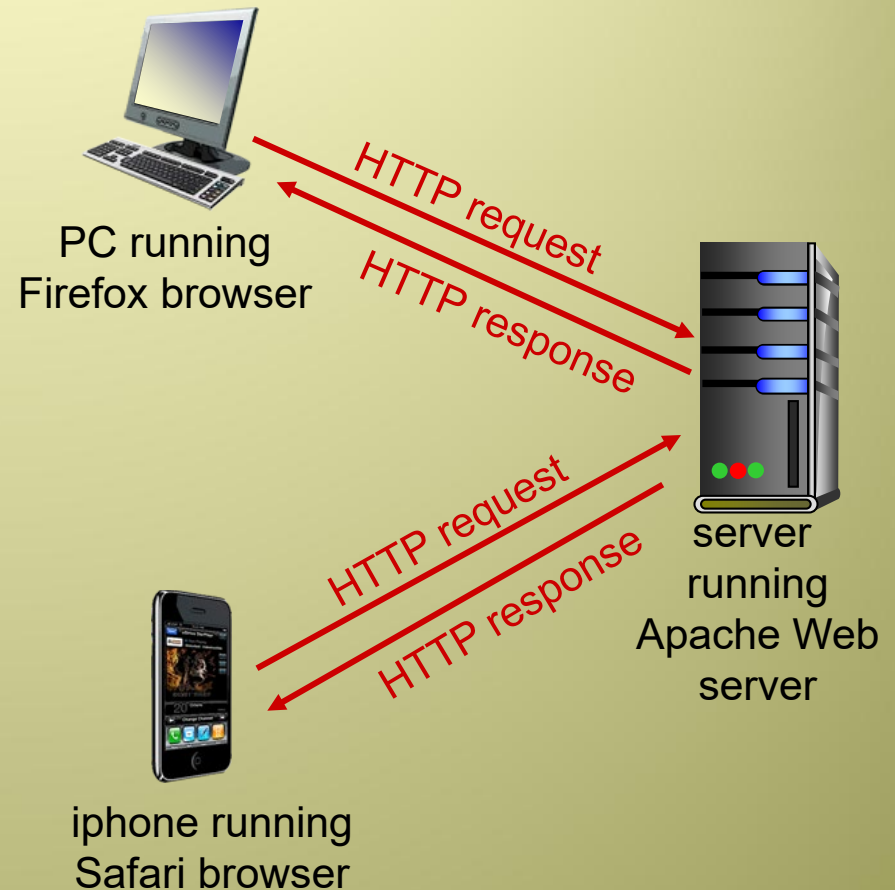
host name

path name

# HTTP overview

## HTTP: hypertext transfer protocol

- ❖ the 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 is then closed
- ❖ downloading multiple objects requires multiple connections

## *persistent HTTP*

- ❖ multiple objects can be sent over single TCP connection between client, server
- ❖ HTTP default mode uses persistent connection, but this can be re-configured

# 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`

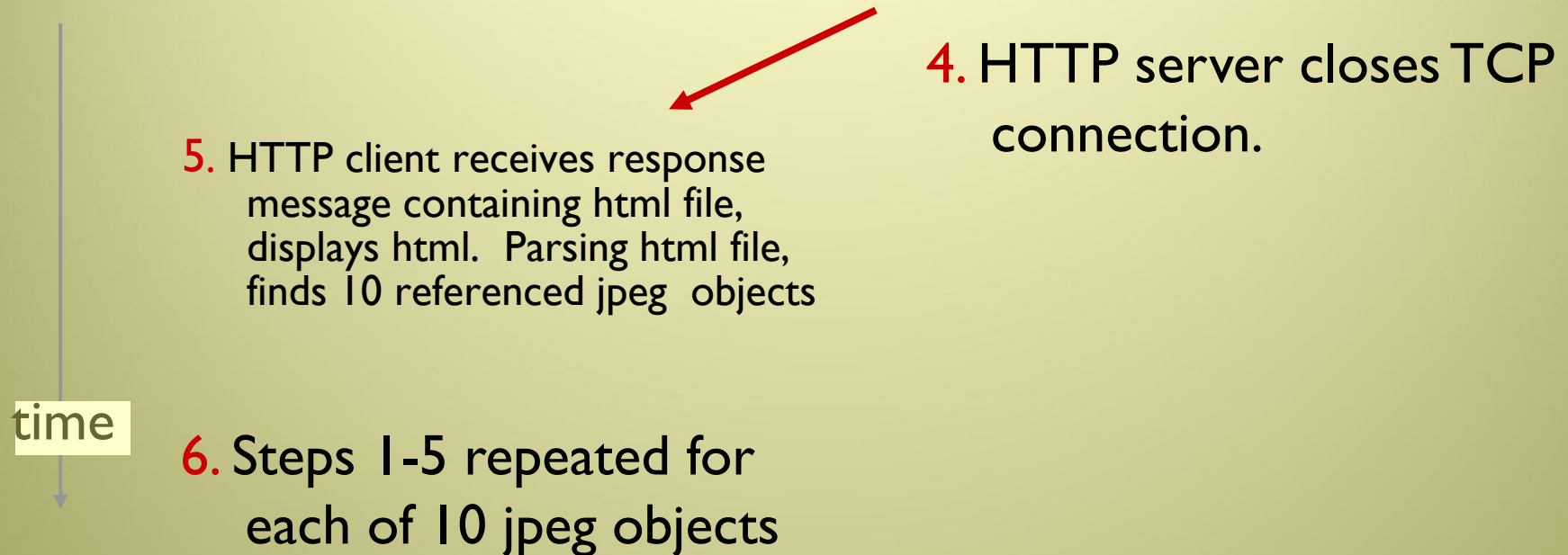
3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time





# Non-persistent HTTP (cont.)



**Q:** how these 10 jpeg objects were obtained? Have they been retrieved in a serial fashion, or using parallel connections?

# Non-persistent HTTP: response time

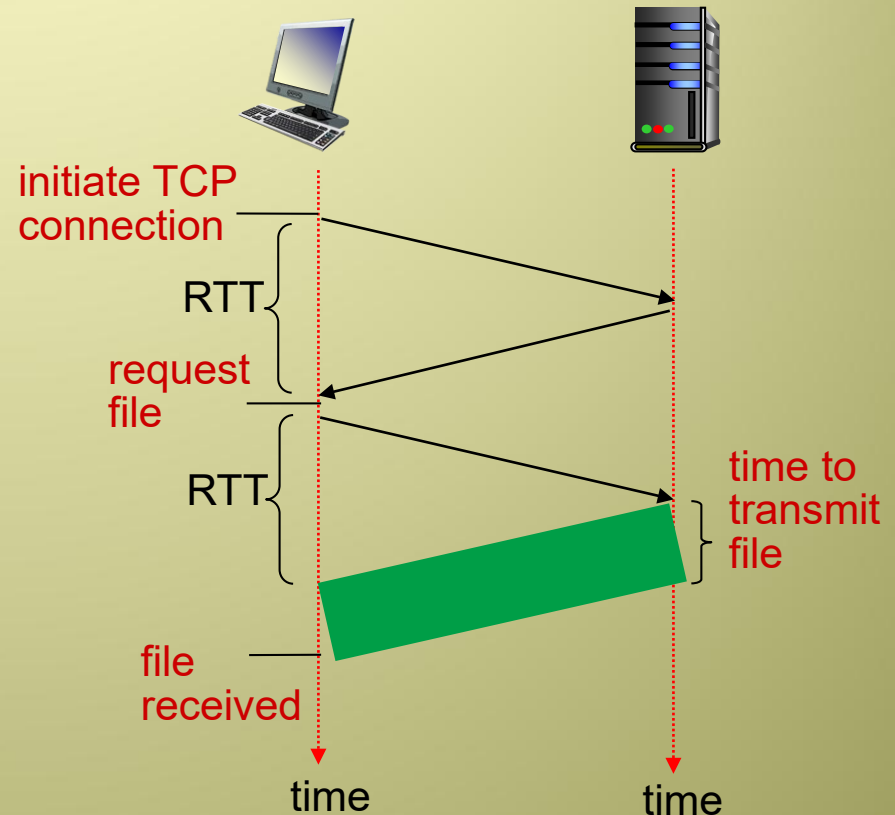
**RTT (definition):** time for a small packet to travel from client to server and back

**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 =**

*$2RTT + \text{file transmission time}$*



# 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 (connection is only closed when it is not used for certain, configurable, amount of time)
- ❖ 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

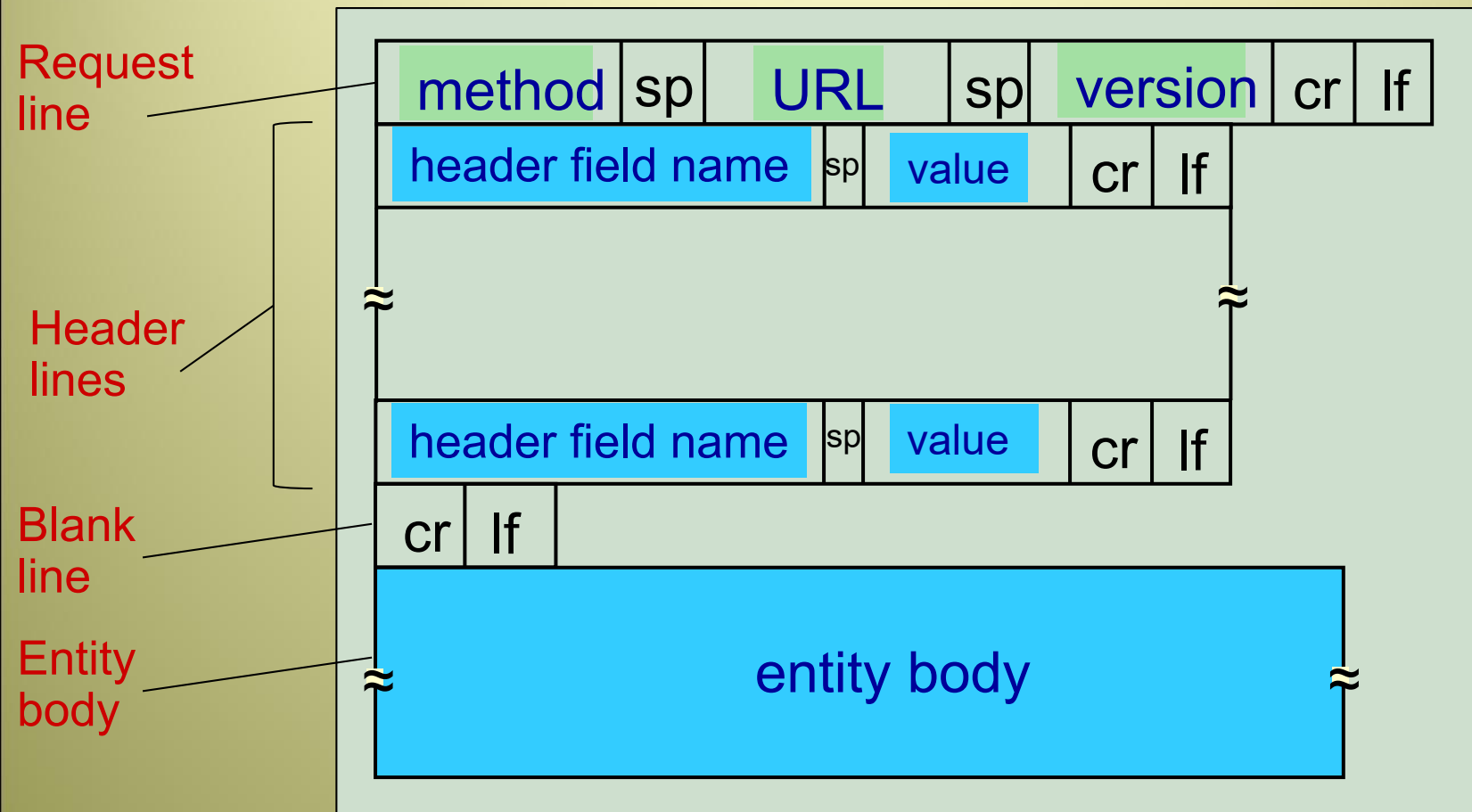
```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n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

HTTP version #

carriage return character

line-feed character

# HTTP request message: general format



General format of an HTTP request message

# Uploading form input

## POST method:

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

## URL method:

- ❖ a request with a form does not necessary use POST
- ❖ GET method can be used with input data
- ❖ input is uploaded in URL field of request line:

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

# Method types

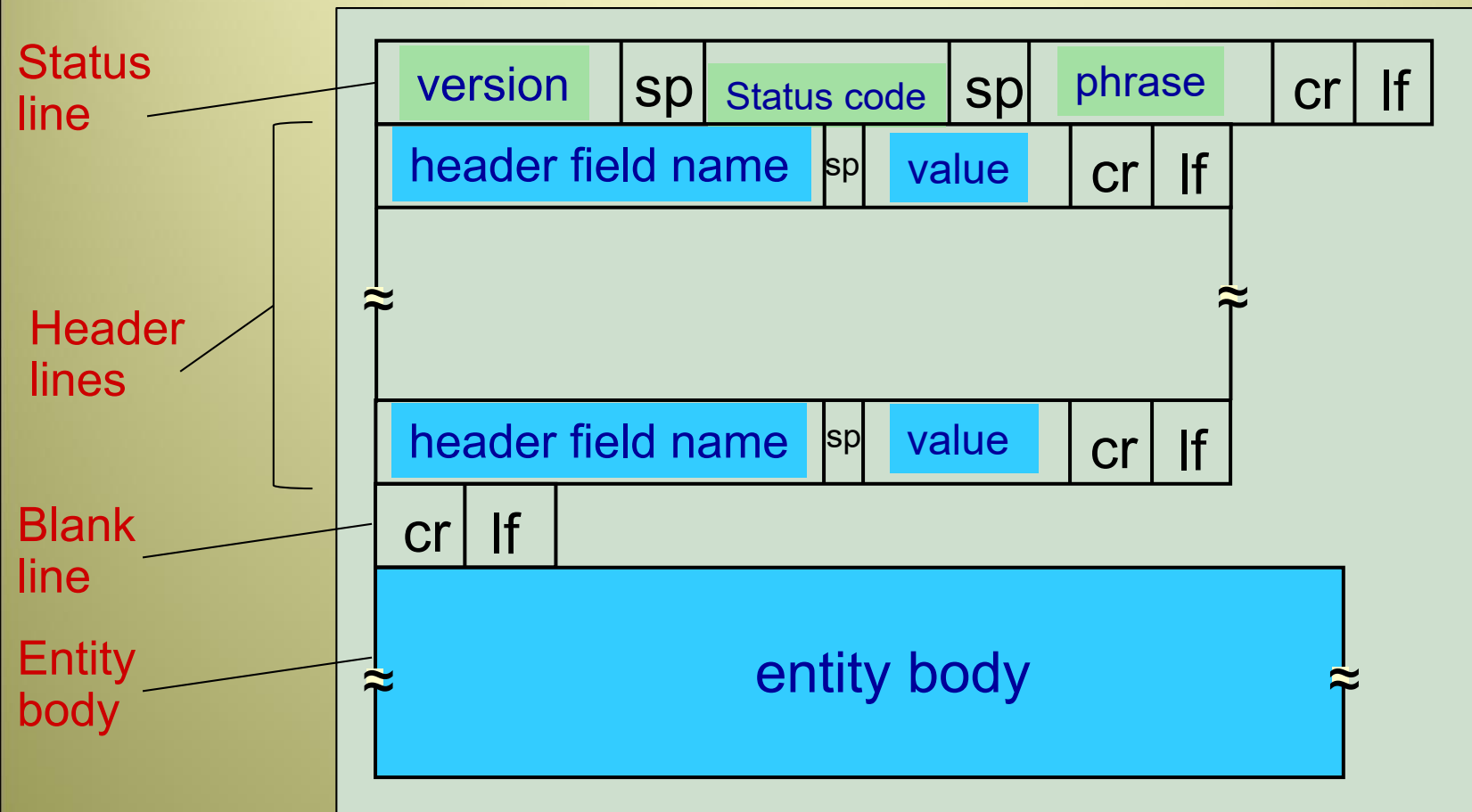
## HTTP/1.0:

- ❖ 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 a web server object specified in the URL field

# HTTP response message: general format



General format of an HTTP response message



# HTTP response message

status line

HTTP/1.1 200 OK\r\n

Date: Sun, 14 Sep 2014 22:07:50 GMT\r\n

Server: Apache/2.0.52 (CentOS)\r\n

Last-Modified: Tue, 30 Oct 2007 17:00:02  
GMT\r\n

header  
lines

ETag: "17dc6-a5c-bf716880"\r\n

Accept-Ranges: bytes\r\n

Content-Length: 2652\r\n

Keep-Alive: timeout=10, max=50\r\n

Connection: Keep-Alive\r\n

Content-Type: text/html; charset=ISO-8859-  
1\r\n\r\n

data, e.g.,  
requested  
HTML file

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 does not exist on this server

## **505 HTTP Version Not Supported**

- requested HTTP version is not supported by the server

# 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

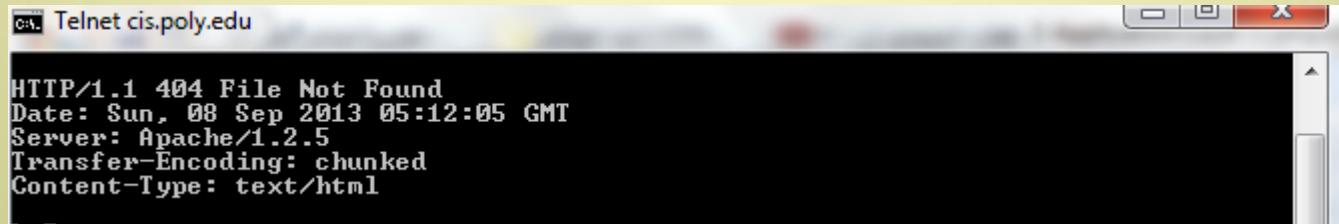
2. type in a GET HTTP request:

```
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!

# Trying out HTTP (client side) for yourself



```
CA: Telnet cis.poly.edu

HTTP/1.1 404 File Not Found
Date: Sun, 08 Sep 2013 05:12:05 GMT
Server: Apache/1.2.5
Transfer-Encoding: chunked
Content-Type: text/html
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>
e>FILE NOT FOUND</title>
    <link rel="stylesheet" type="text/css" href="/templa
te/css/cis_style.css">
    <meta http-equiv="Content-Type" content="text/html; ch
arset=ISO-8859-1">
  </head>
  <body class="std">
    <table width="650" cellpadding="0
```

```
<font color="#ff0000">
<p>You are here because the URL you are looking for does not exist. We are sorry for this inconvenience. Please <a href="/webmaster/">contact the webmaster</a> so that this problem can be corrected.</p>
<p>Note: Course pages are maintained by faculty members teaching those courses; comments/concerns relating to course pages should be directed to them.</p>
</font>
  </td>
</tr>
</table>
```

# User-server state: cookies

- ❖ HTTP servers are stateless
  - simplifies server design
  - increases performance
  - However, it is often desirable for a website to identify users

- ❖ many Web sites use cookies

## *four components:*

- 1) cookie header line in HTTP *response* message
- 2) cookie header line in next HTTP *request* message
- 3) cookie file kept on user's host, managed by user's browser
- 4) back-end database at the Web site

## **example:**

- ❖ Susan always access Internet from PC
- ❖ visits specific e-commerce site for first time
- ❖ when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

# Cookies: keeping “state” (cont.)

client



server



cookie file



ebay 8734  
amazon 1678

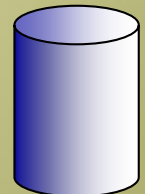
usual http request msg

Amazon server  
creates ID  
1678 for user

usual http response  
**set-cookie: 1678**

create  
entry

backend  
database



usual http request msg  
**cookie: 1678**

cookie-  
specific  
action

access

usual http response msg

one week later:



ebay 8734  
amazon 1678

usual http request msg  
**cookie: 1678**

cookie-  
specific  
action

access

usual http response msg

# Cookies (continued)

## *what cookies can be used for:*

- ❖ authorization
- ❖ shopping carts
- ❖ recommendations
- ❖ user session state (Web e-mail)
- ❖ ...

aside

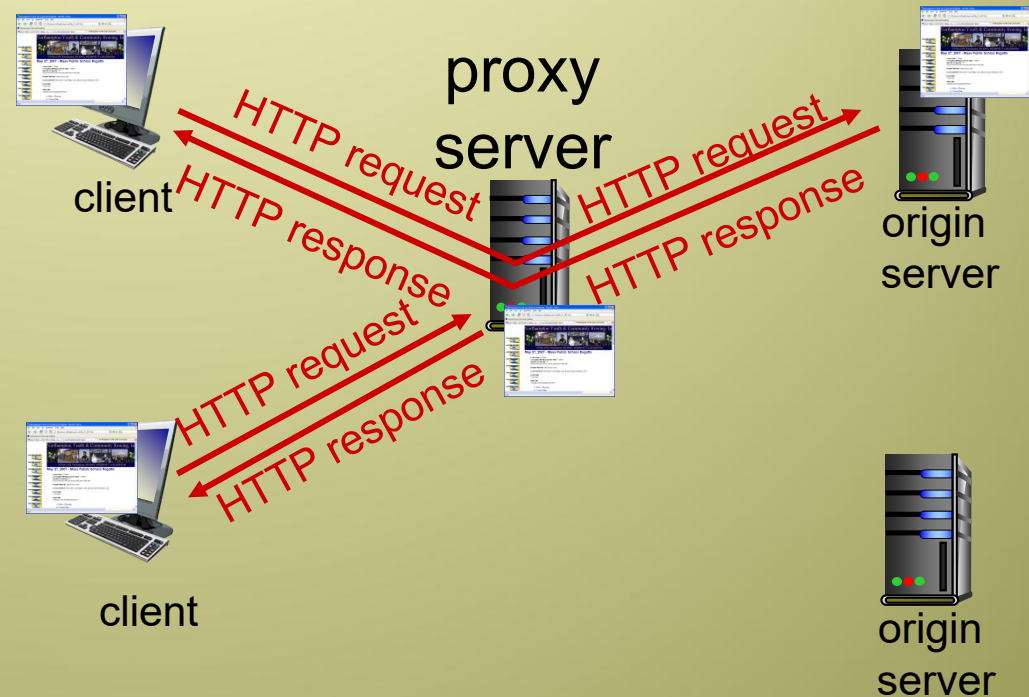
## *cookies and privacy:*

- ❖ cookies are somehow controversial
- ❖ cookies permit sites to learn a lot about you
- ❖ you may supply name, e-mail, .... to sites

# Web caches (proxy servers)

**goal:** satisfy client request without involving origin server

- ❖ user sets browser: Web accesses via cache
- ❖ browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else: cache requests object from origin server, then returns object to client





# More about Web caching

- ❖ cache acts as both client and server
  - server for original requesting client
  - client to origin server
- ❖ typically cache is installed by ISP (university, company, residential ISP)

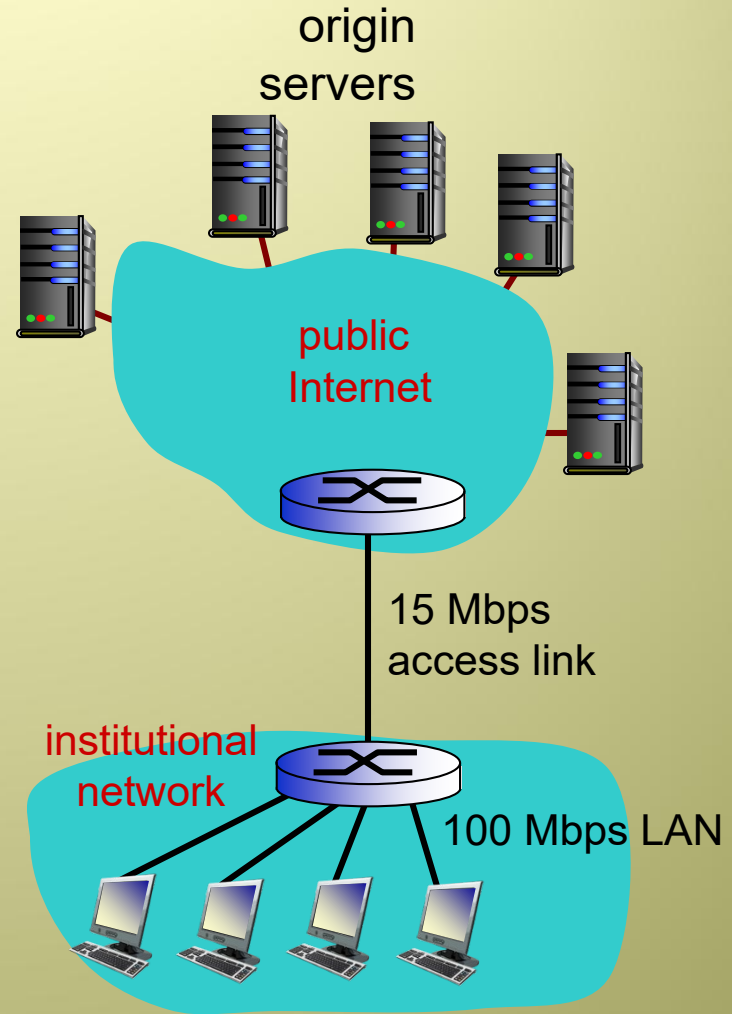
## *why Web caching?*

- ❖ reduce response time for client request
- ❖ reduce traffic on an institution's access link
- ❖ reduce traffic in the Internet as a whole
- ❖ Internet dense with caches: enables “poor” content providers to effectively deliver content (so too does P2P file sharing)

# Caching example:

## *assumptions:*

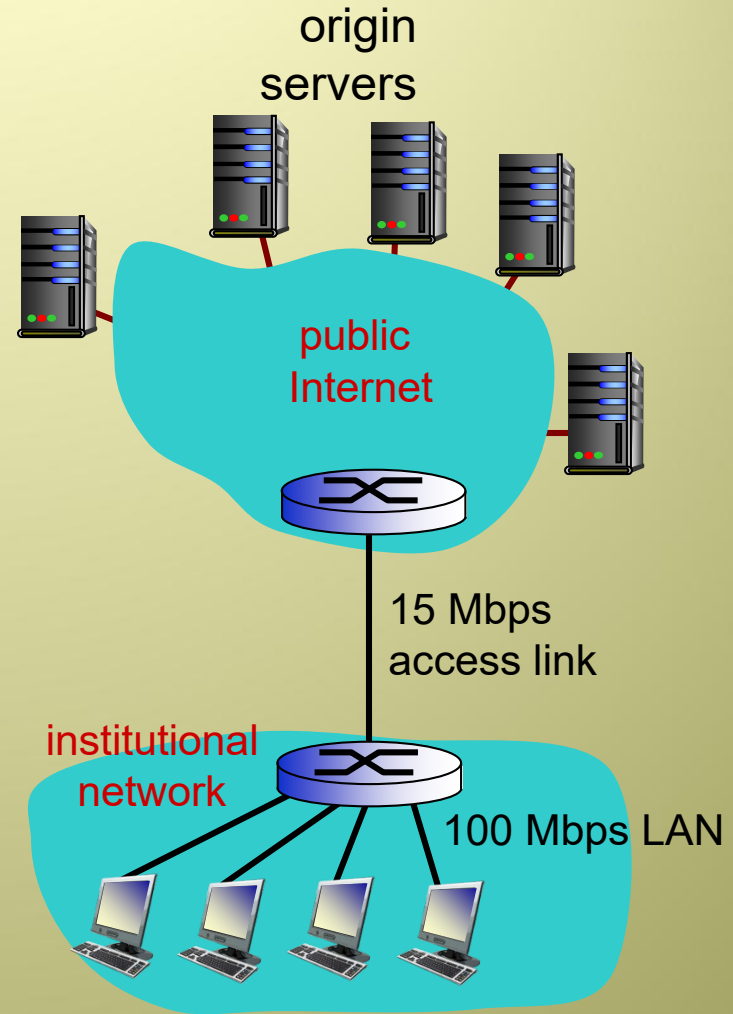
- ❖ **100 Mbps** LAN
- ❖ **15 Mbps** access between institution router and Internet router
- ❖ avg object size: **1 Mbits**
- ❖ avg request rate from browsers to origin servers: **15 request/sec**
- ❖ HTTP requests are too small; hence represent **negligible** load on both institutional and public links
- ❖ Internet delay (RTT from Internet-side router to origin servers) is, in avg: **2 sec**



# Caching example:

- ❖ Total response time (from time browser requests an object and it receives it) is:
  - ❖ LAN delay
  - +
  - ❖ access delay between the two routers
  - +
  - ❖ Internet delay

*Q: what is the total response time of this network?*



# Caching example:

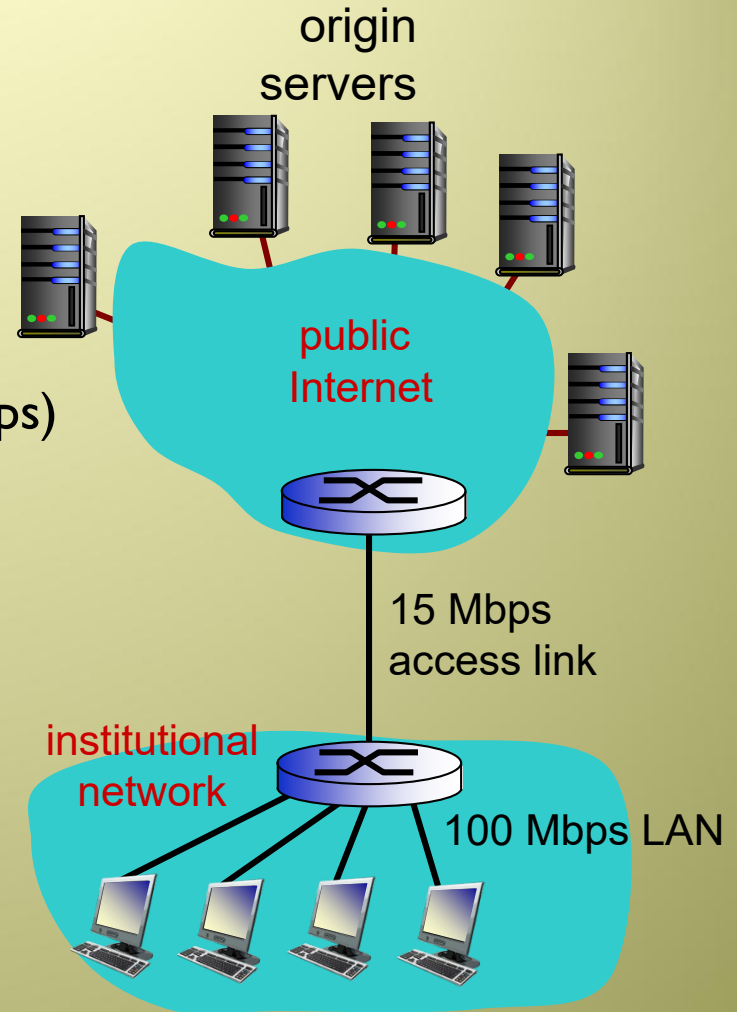
*traffic intensity ( $L\lambda/R$ ) on the LAN is:*

$$(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (100 \text{ Mbps}) = 0.15$$

*traffic intensity on access link (between the two routers) is:*

$$(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (15 \text{ Mbps}) = 1$$

- ❖ traffic intensity of 15% on LAN  
→ small delay ( $\mu$ secs to 10s of msec)
- ❖ traffic intensity of 1 on access link  
→ delay grows without bound  
order of minutes, if not more!
- ❖ **total delay** = Internet delay + access delay + LAN delay  
= 2 sec + minutes +  $\mu$ secs → **TERRIBLE!**



# Caching example:

*Solution 1 (expensive solution):*

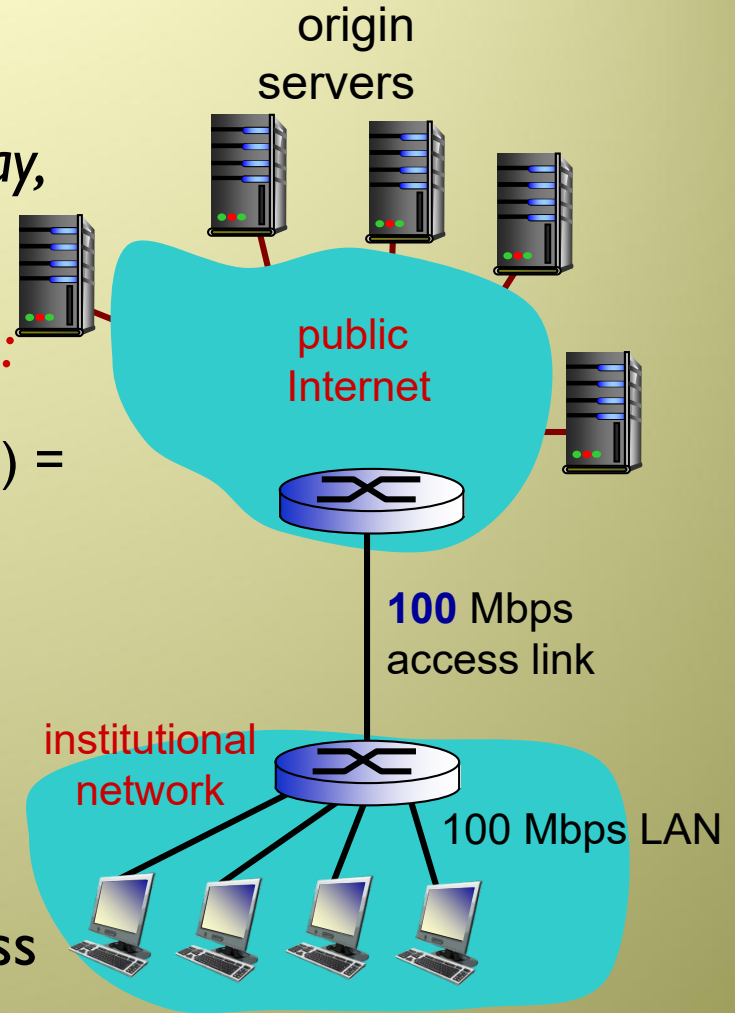
- ❖ Increase access rate from 15 Mbps to, say, 100 Mbps

*traffic intensity on LAN and access link is:*

$$(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (100 \text{ Mbps}) = 0.15$$

- ❖ traffic intensity of 15%  
→ small delay ( $\mu$ secs to 10s of msec)

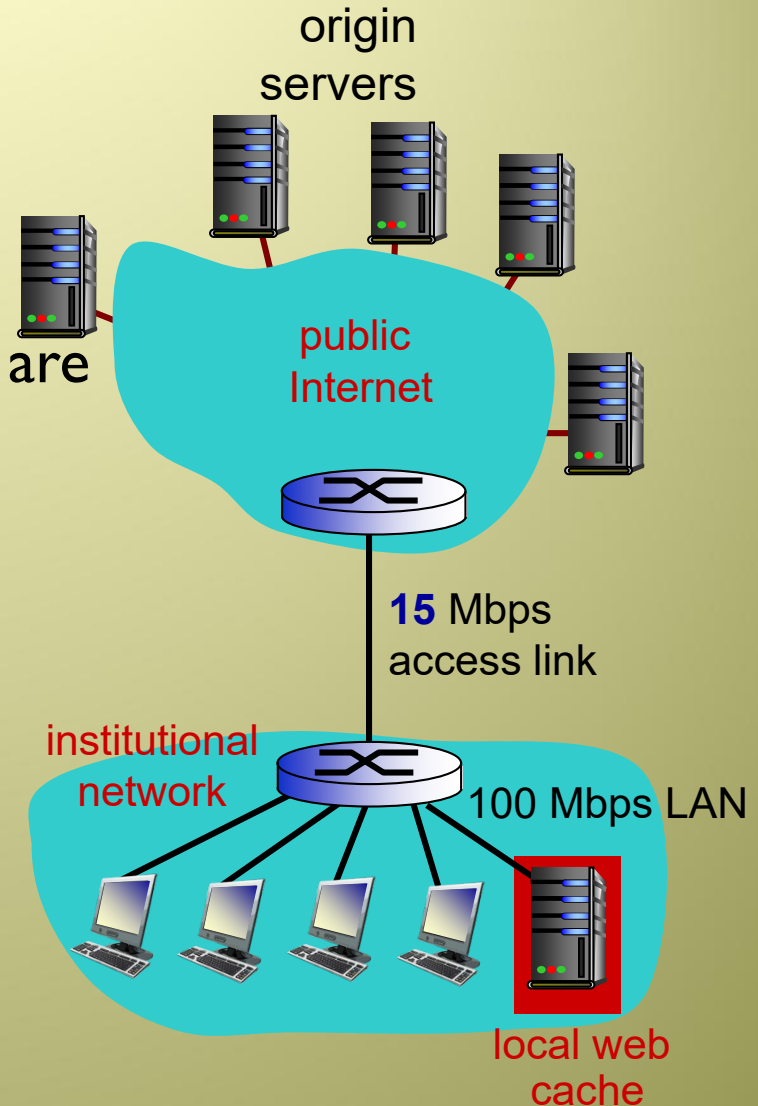
- ❖ **total delay** = Internet delay + access delay + LAN delay  
= 2 sec + msecs +  $\mu$ secs  
 $\approx$  **2 seconds**



# Caching example:

*Solution 2 (more cost-effective solution):*

- ❖ keep access link at 15 Mbps
- ❖ install a web cache
- ❖ **hit-rate:** the fraction of request that are satisfied by the cache
  - typically range from 20% to 70%
- ❖ for illustration, assume
  - 40% hit rate, and
  - requests from cache server take an average of **10 msec**



*Q: what is the total response time of this network?*

# Caching example:

*Solution 2 (more cost-effective solution):*

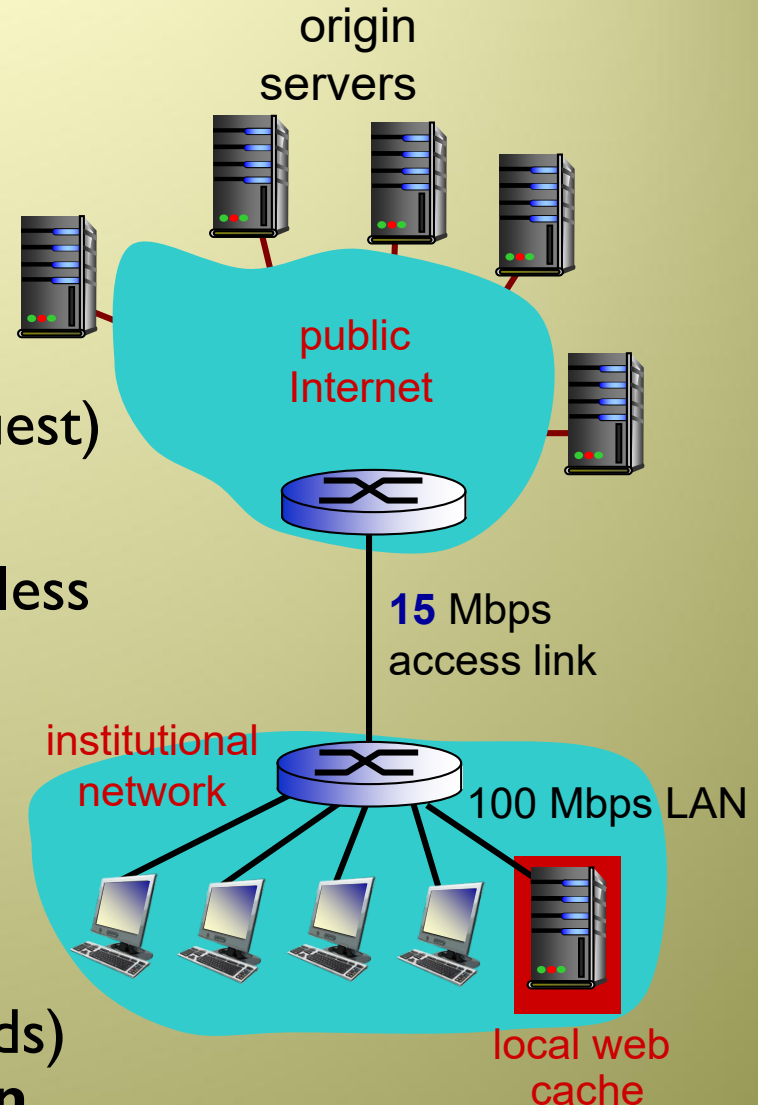
- ❖ 40% hits → 60% misses
- ❖ traffic intensity between two routers is:

$$0.6 \cdot [(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (15 \text{ Mbps})] = 0.6$$

➔ 60% traffic intensity (often 80% or less is okay) results in a negligible delay compared to the 2 seconds internet delay

- ❖ **total delay =**

$$0.4 \cdot (0.01 \text{ seconds}) + 0.6 \cdot (2.01 \text{ seconds}) \approx 1.2 \text{ seconds} \rightarrow \text{even better than the expensive Solution 1}$$



# Caching

- ❖ although caching has clear and obvious advantages, it introduced a very serious problem!
- ❖ what if the copy in the cache is stale?
  - cached copy may have been modified in the web server after being cached by the proxy server
- ❖ a mechanism is hence needed by HTTP to verify against that
  - the **conditional GET**



# Conditional GET

- ❖ **Goal:** don't send object if cache has up-to-date cached version

- no object transmission delay
- lower link utilization

- ❖ **cache:** specify date of cached copy in HTTP request

**If-modified-since:**  
**<date>**

- ❖ **server:** response contains no object if cached copy is up-to-date:

**HTTP/1.0 304 Not Modified**

client



server



HTTP request msg  
**If-modified-since: <date>**

object  
not  
modified  
before  
<date>

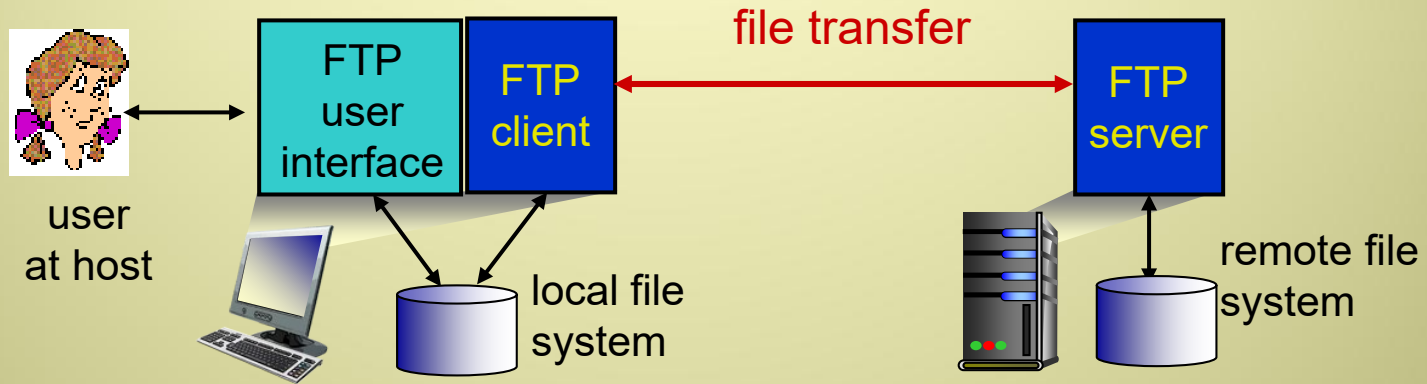
HTTP response  
**HTTP/1.0  
304 Not Modified**

HTTP request msg  
**If-modified-since: <date>**

object  
modified  
after  
<date>

HTTP response  
**HTTP/1.0 200 OK  
<data>**

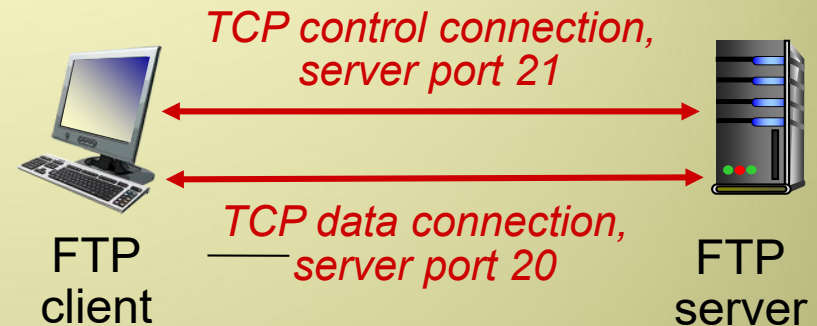
# FTP: the file transfer protocol



- ❖ transfer file to/from remote host
- ❖ client/server model
  - **client**: side that initiates transfer (either to/from remote)
  - **server**: remote host
- ❖ ftp: RFC 959
- ❖ ftp server: port 21

# FTP: separate control, data connections

- ❖ FTP client contacts FTP server at port 21, using TCP
- ❖ client authorized over **control connection**
- ❖ client browses remote directory, sends commands over control connection
- ❖ when server receives file transfer command, **server** opens 2<sup>nd</sup> **parallel TCP data connection** (for file) to client
- ❖ after transferring one file, server closes data connection



- ❖ server opens another TCP **data connection** to transfer another file
- ❖ control connection: *“out of band”*
- ❖ FTP server maintains “state”: current directory, earlier authentication

# FTP commands, responses

## *sample commands:*

- ❖ sent as ASCII text over control channel
- ❖ **USER** *username*
- ❖ **PASS** *password*
- ❖ **LIST** return list of file in current directory
- ❖ **RETR** *filename* retrieves (gets) file
- ❖ **STOR** *filename* stores (puts) file onto remote host

## *sample return codes*

- ❖ status code and phrase (as in HTTP)
- ❖ 331 Username OK, password required
- ❖ 125 data connection already open; transfer starting
- ❖ 425 Can't open data connection
- ❖ 452 Error writing file

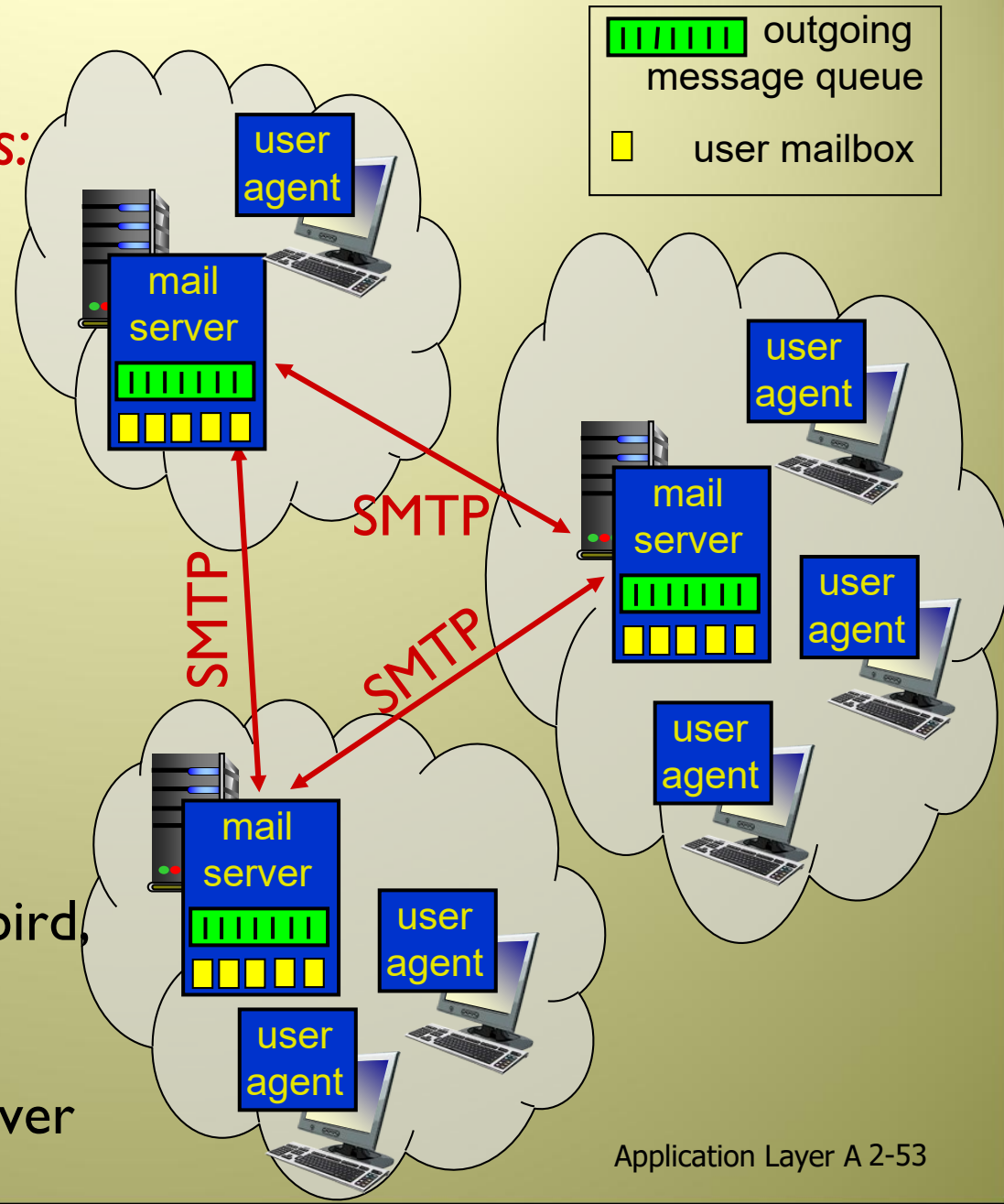
# Electronic mail

## *Three major components:*

- ❖ user agents
- ❖ mail servers
- ❖ simple mail transfer protocol: SMTP

## *User Agent*

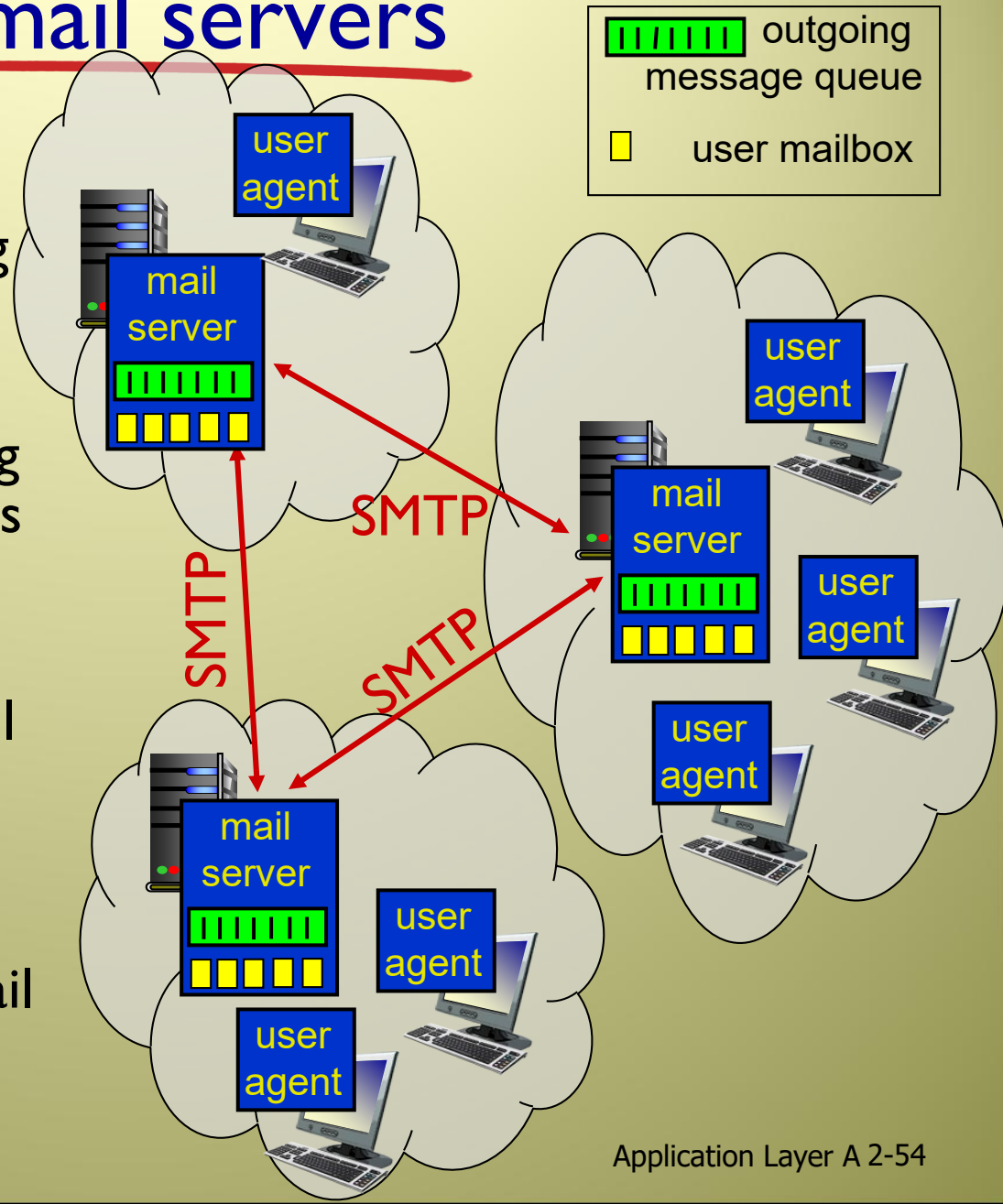
- ❖ a.k.a. “mail reader”
- ❖ composing, editing, reading, forwarding, replying mail messages
- ❖ e.g., Outlook, Thunderbird, iPhone mail client
- ❖ outgoing, incoming messages stored on server



# Electronic mail: mail servers

## mail servers:

- ❖ *mailbox* contains incoming messages for user
- ❖ *message queue* of outgoing (to be sent) mail messages
- ❖ *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - “server”: receiving mail server

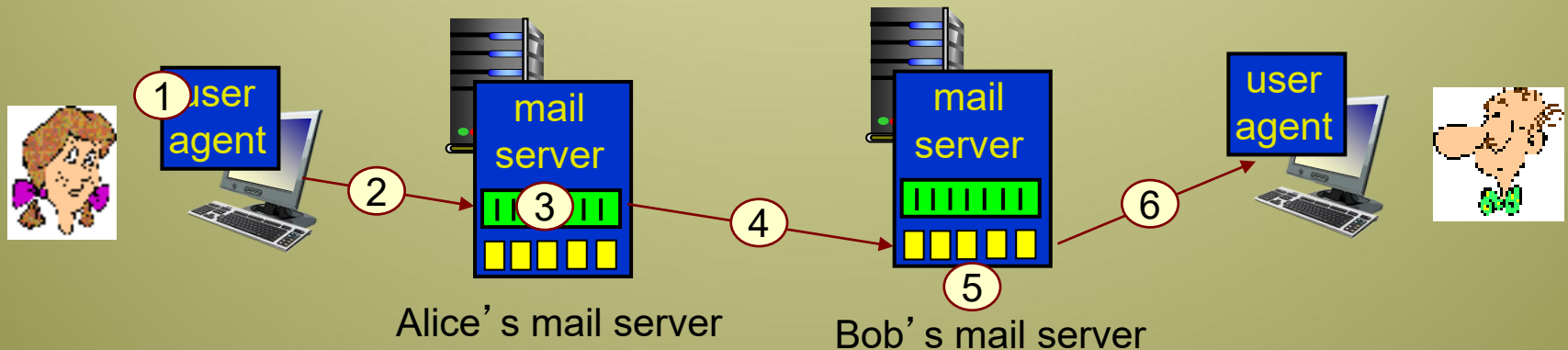


# Simple Mail Transfer Protocol: SMTP [RFC 5321]

- ❖ the principle application-layer protocol for e-mail
- ❖ uses TCP to reliably transfer email message from client to server, port 25
- ❖ **direct transfer**: sending server to receiving server
- ❖ three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- ❖ command/response interaction (like HTTP, FTP)
  - **commands**: ASCII text
  - **response**: status code and phrase
- ❖ messages must be in **7-bit ASCII**

# Scenario: Alice sends message to Bob

- 1) Alice uses UA to compose message "to" bob@someschool.edu
- 2) Alice's UA sends message to her mail server; message placed in message queue
- 3) client side of SMTP opens TCP connection with Bob's mail server
- 4) SMTP client sends Alice's message over the TCP connection
- 5) Bob's mail server places the message in Bob's mailbox
- 6) Bob invokes his user agent to read message





# Sample SMTP interaction

```
S: 220 sunset.edu
C: HELO jupiter.fr
S: 250 Hello jupiter.fr, pleased to meet you
C: MAIL FROM: <alice@jupiter.fr>
S: 250 alice@jupiter.fr... Sender ok
C: RCPT TO: <bob@sunset.edu>
S: 250 bob@sunset.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Hi Bob,
C: How is it in beautiful Paris?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 sunset.edu closing connection
```

# Try SMTP interaction for yourself:

- ❖ `telnet servername 25`
- ❖ see 220 reply from server
- ❖ enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

These above steps lets you send email without using email client (reader)

[Click here for another example](#)

[Click here to view SMTP codes](#)

# SMTP vs. HTTP

SMTP	HTTP
File transfer: mail server to mail server	File transfer: from a web server to a web client
Persistent TCP connection	Either persistent or non-persistent TCP connection
<b>Push</b> protocol	<b>Pull</b> protocol
Messages (headers & body) must be in ASCII i.e. special French characters, images, etc. must be encoded into ASCII	No such restriction
All objects (i.e. text and images) are placed in one message (multi-parts of the message)	Each object has its own HTTP response message

# Mail message format

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

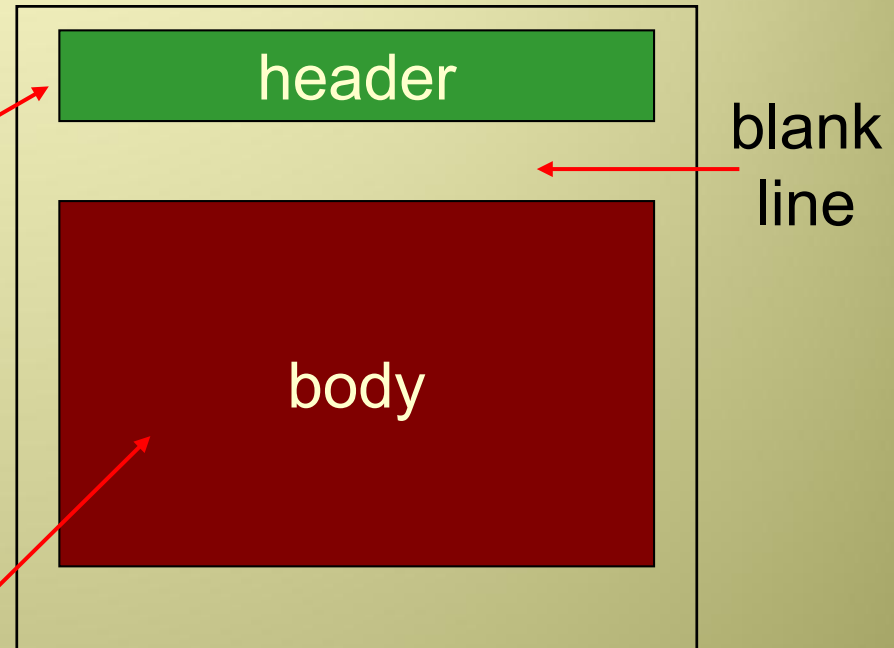
❖ header lines, e.g.,

- **To:**
- **From:**
- Subject:

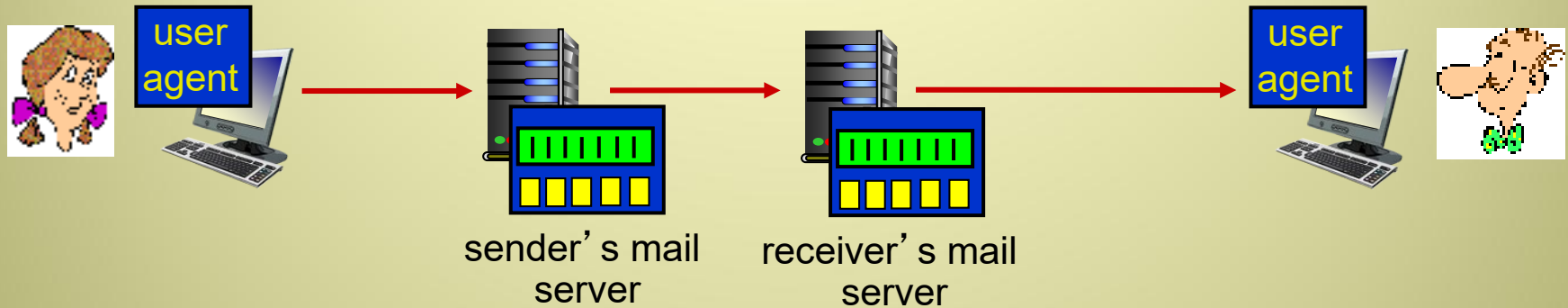
*different from* SMTP MAIL  
**FROM, RCPT TO:**  
commands!

❖ Body: the “message”

- ASCII characters only



# Mail access protocols

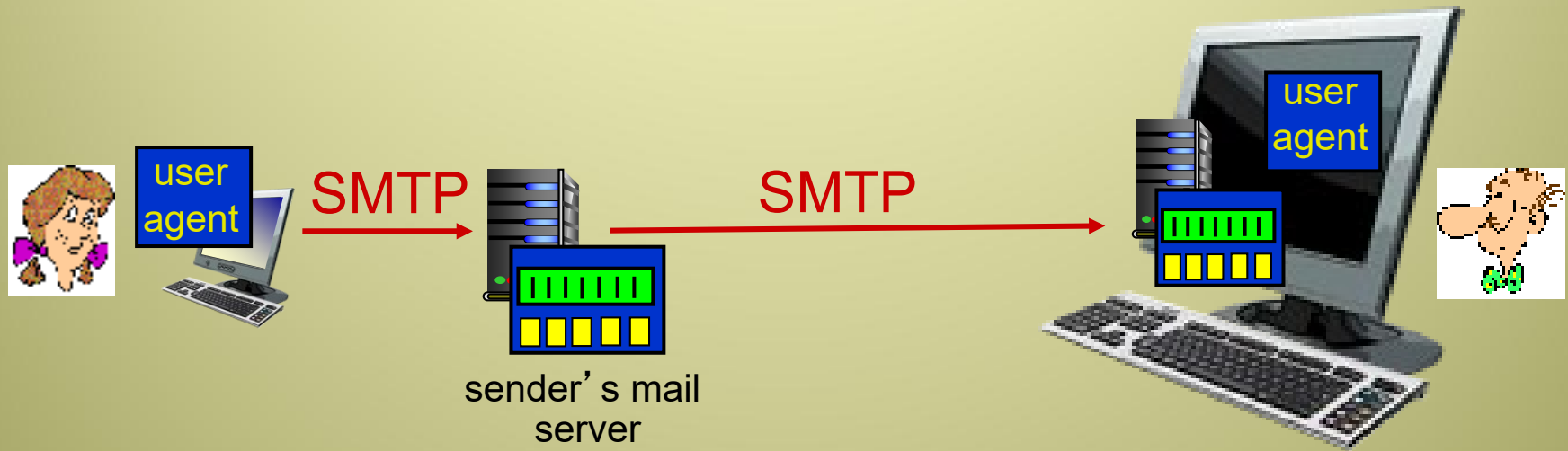


❖ Q: Why the two-step procedure?

# Mail access protocols

❖ Q: Why have a receiver server?

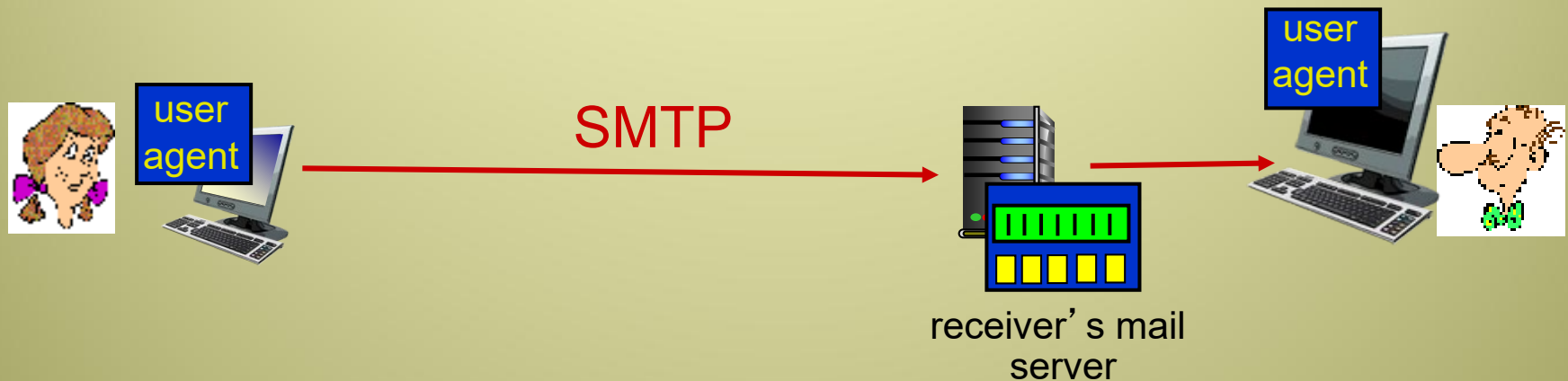
Just install it in the receiver's local PC, where receiver executes his e-mail agent anyways!



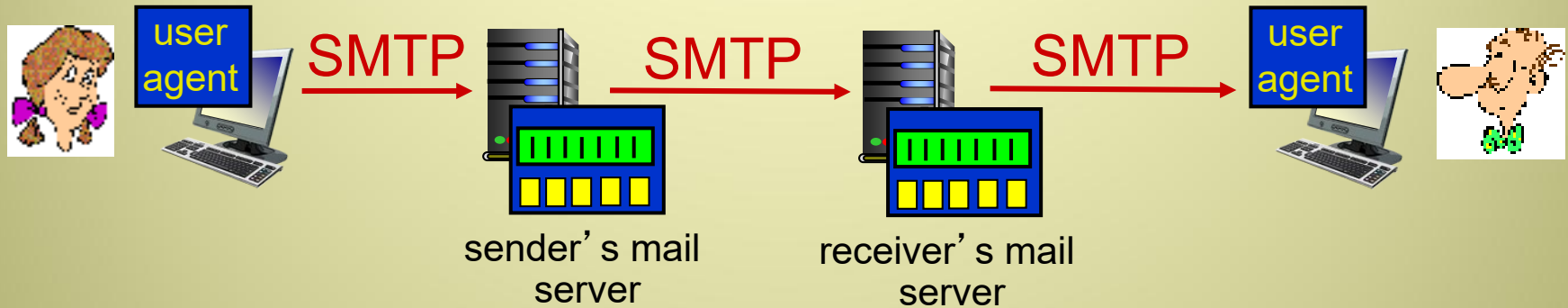
# Mail access protocols

❖ Q: Why have a sender server?

Just let the sender's user agent place the message directly on the receiver mail server



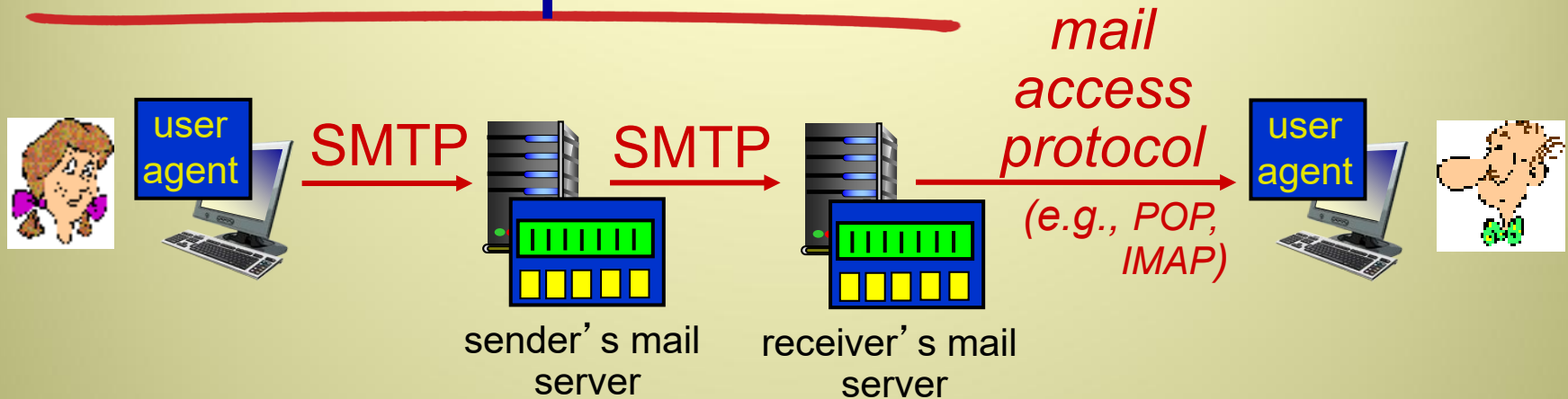
# Mail access protocols



❖ **OK:** We need both of them, but will it work?



# Mail access protocols



- ❖ **SMTP**: delivery/storage to receiver's server
- ❖ mail access protocol: retrieval from server
  - **POP**: Post Office Protocol [RFC 1939]: authorization, download
  - **IMAP**: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - **HTTP**: gmail, Hotmail, Yahoo! Mail, etc.

# POP3 protocol

- ❖ extremely simple mail access protocol, but
- ❖ limited !
  - User agent opens a TCP connection with the mail server on port 110
  - User progress through 3 phases:
    - authorization
      - Sends username/password
    - transaction, and
      - retrieves & manipulate message (i.e. mark message for deletion, obtain statistics, ...)
    - update
      - after clients “quits”, server updates what is needed (i.e. delete messages that were marked for deletion).

# POP3 protocol

## *authorization phase*

- ❖ client commands:
  - **user**: declare username
  - **pass**: password
- ❖ server responses
  - **+OK**
  - **-ERR**

## *transaction phase, client:*

- ❖ **list**: list message numbers and there sizes
- ❖ **retr**: retrieve message by number
- ❖ **dele**: delete
- ❖ **quit**

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

# IMAP

---

- ❖ with POP3 user can manage downloaded messages into folders in local machine
- ❖ search can be done on that file system
- ❖ This hierarchy however is on the local machine only
  - cannot be seen on the server, or from different machines, which may be very inconvenient

## IMAP

- ❖ keeps all messages in one place: at server
- ❖ allows user to create folders and organize messages in them
- ❖ keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

# IMAP

---

- ❖ messages are assigned into folders
  - new messages are associated with INBOX folder
- ❖ Protocol provide commands to
  - allow user to create folders and move messages to them,
  - remotely, search folders for messages matching some criteria
  - delete, manipulate messages, ...
- ❖ unlike, POP3, IMAP maintains state
  - i.e. created folders keep their names across multiple IMAP sessions
- ❖ permits user to obtain components of a message
  - i.e. obtain only the header or just a part of a multipart **MIME** message
  - this is useful, especially under low bandwidth connections

# Webmail via HTTP

---

- ❖ users can access e-mail through web browsers
  - Hotmail introduced web-based e-mails in mid 1990s
- ❖ user agent is an ordinary web browser
- ❖ browsers communicate with their remote mailboxes via HTTP
- ❖ when a user accesses his/her mailbox, the e-mail messages are sent from the server to the browser via HTTP **instead of POP3 or IMAP**
- ❖ when a user wishes to send an e-mail, the e-mail is sent from the browser to the user's mail server through HTTP **instead of SMTP**