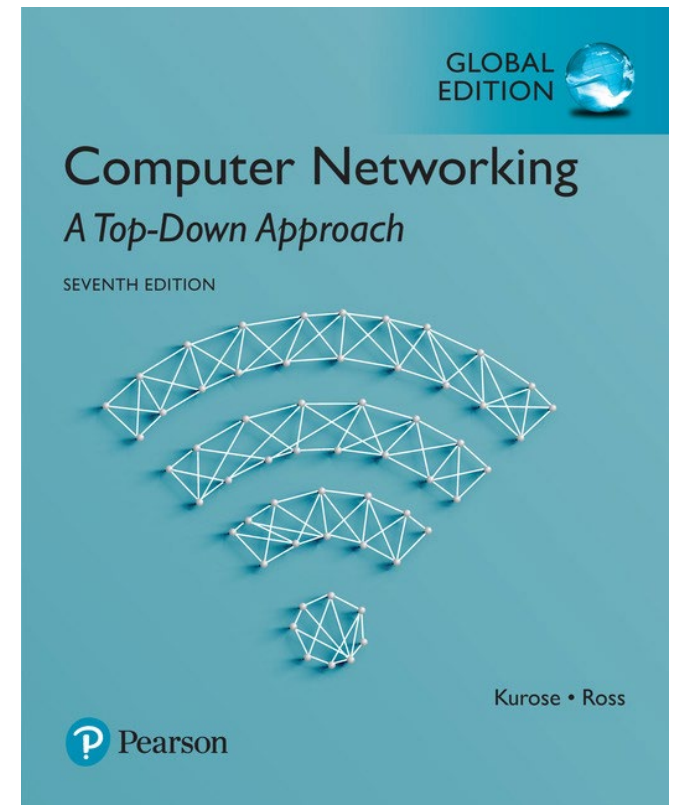


Application Layer



Computer Networking: A Top Down Approach

7th Edition, Global Edition
Jim Kurose, Keith Ross
Pearson
April 2016

Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 media streaming and content distribution networks

2.5 socket programming with UDP and TCP

Application layer

our goals:

- conceptual, implementation aspects of network application protocols
 - transport-layer service models
 - client-server paradigm
 - peer-to-peer paradigm
 - content distribution networks
- learn about protocols by examining popular application-level protocols
 - HTTP
 - FTP
 - SMTP / POP3 / IMAP
- 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
- ...
- ...

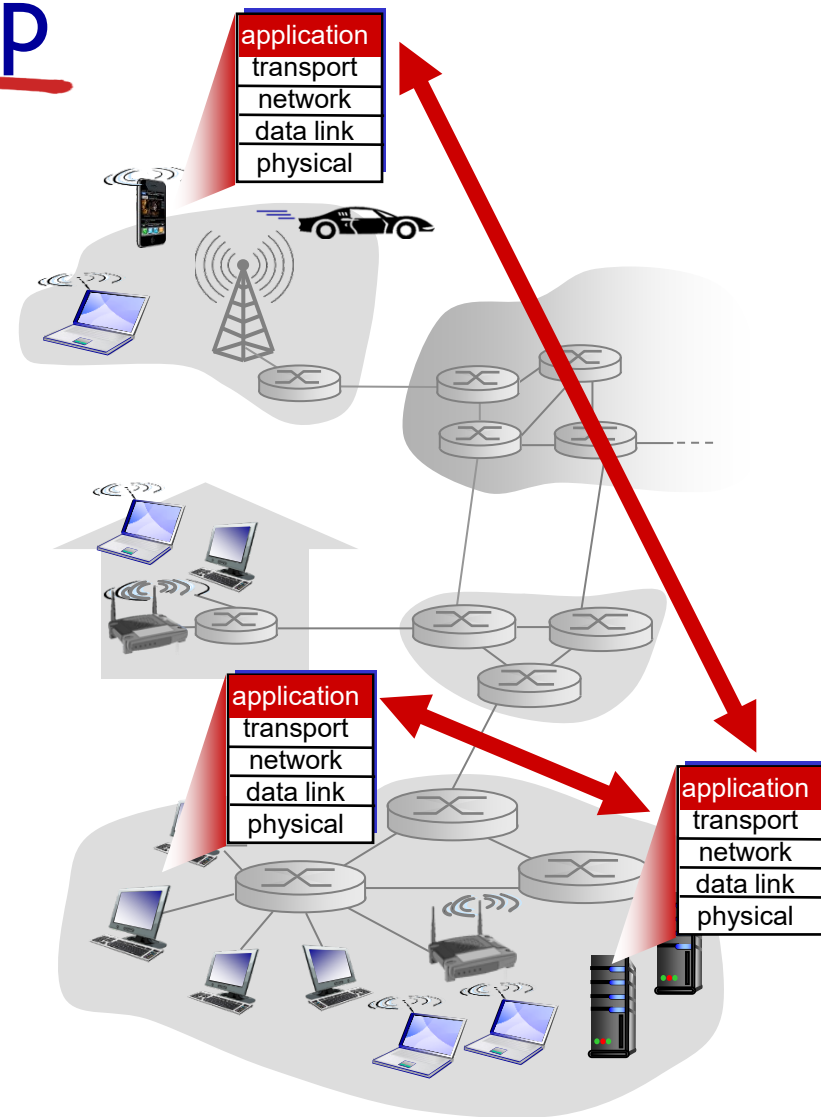
Creating a network app

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 allow for rapid app development & propagation

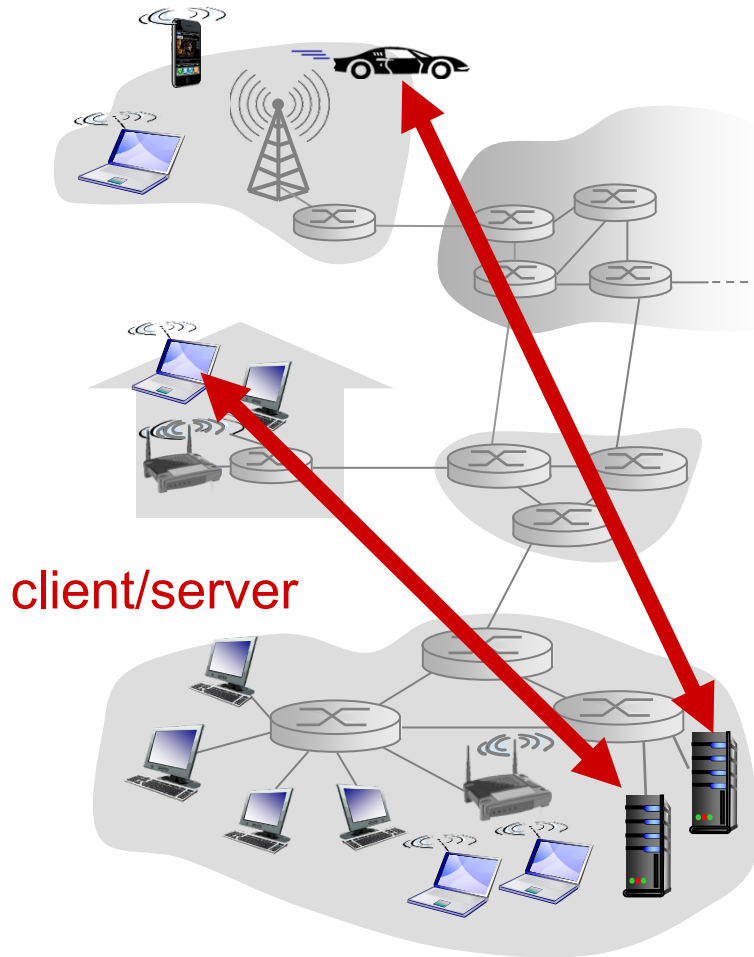


Application architectures

possible structure of applications:

- client-server
 - Including data centers / cloud computing
- peer-to-peer (P2P)

Client-server architecture



server:

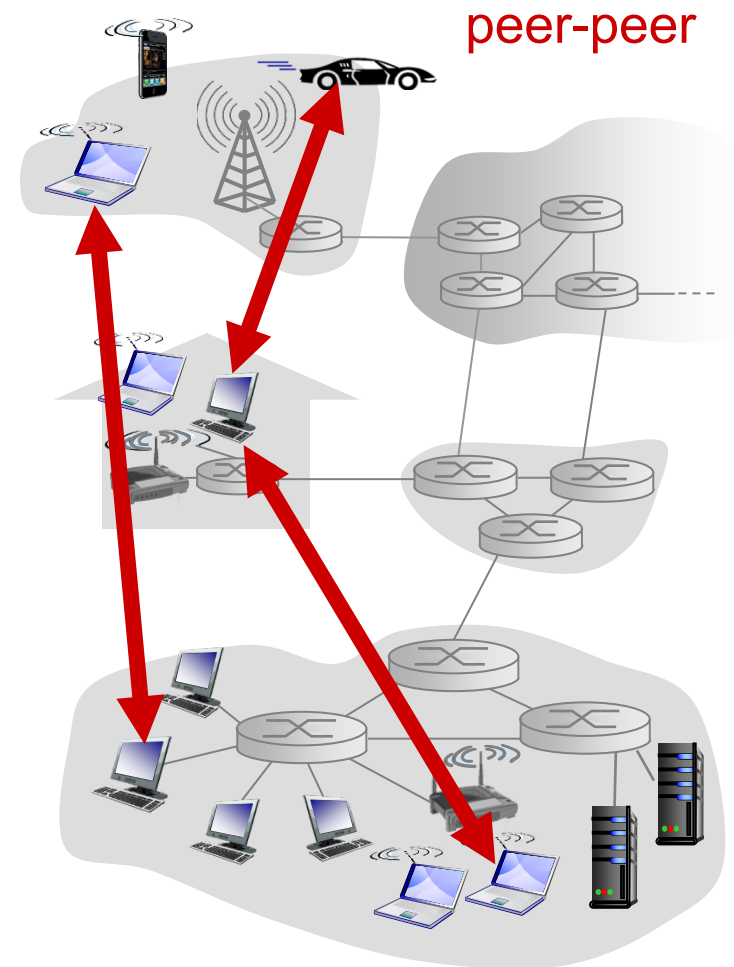
- always-on host
- permanent IP address
- data centers for scaling

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, not only new service demands
- peers are intermittently connected and change IP addresses
 - complex management



Processes communicating

process: program running within a host

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

clients, servers

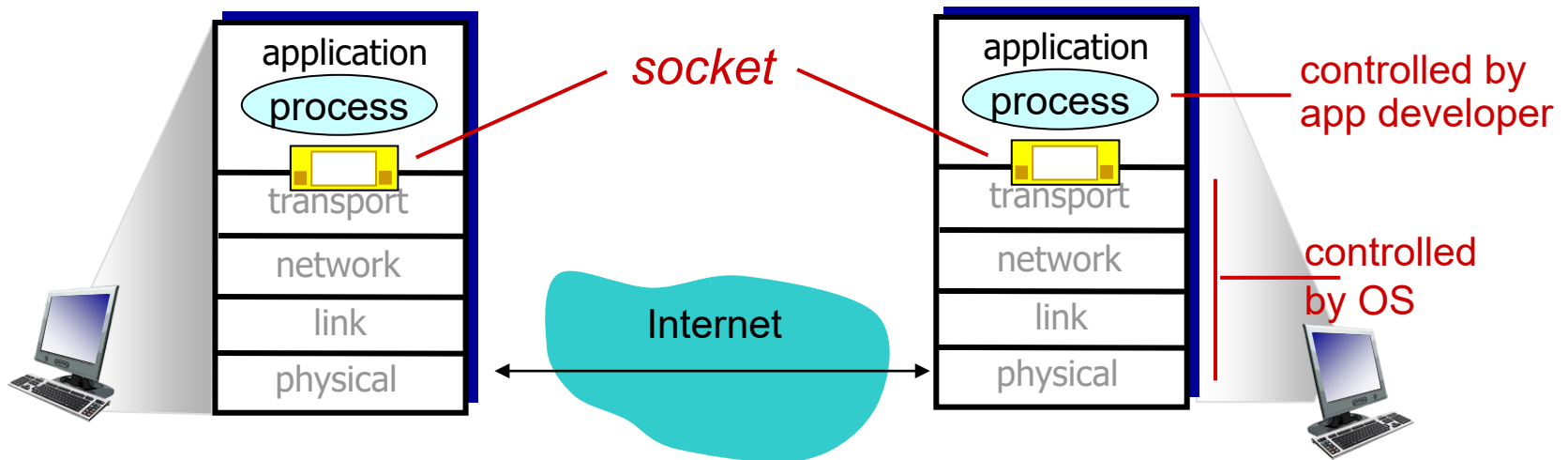
client process: process that initiates communication

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 pushes message out of the 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.
- examples for 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 allocated
- **message semantics**
 - meaning of information in fields
- **rules** for when and how processes send & respond to messages

open protocols:

- defined in RFCs
- allow 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	latency
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 processes
- *flow control*: sender won't overwhelm receiver
- *congestion control*: slow down 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 processes
- *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
remote terminal access	e-mail	SMTP [RFC 2821]	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

Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 media streaming and content distribution networks

2.5 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`

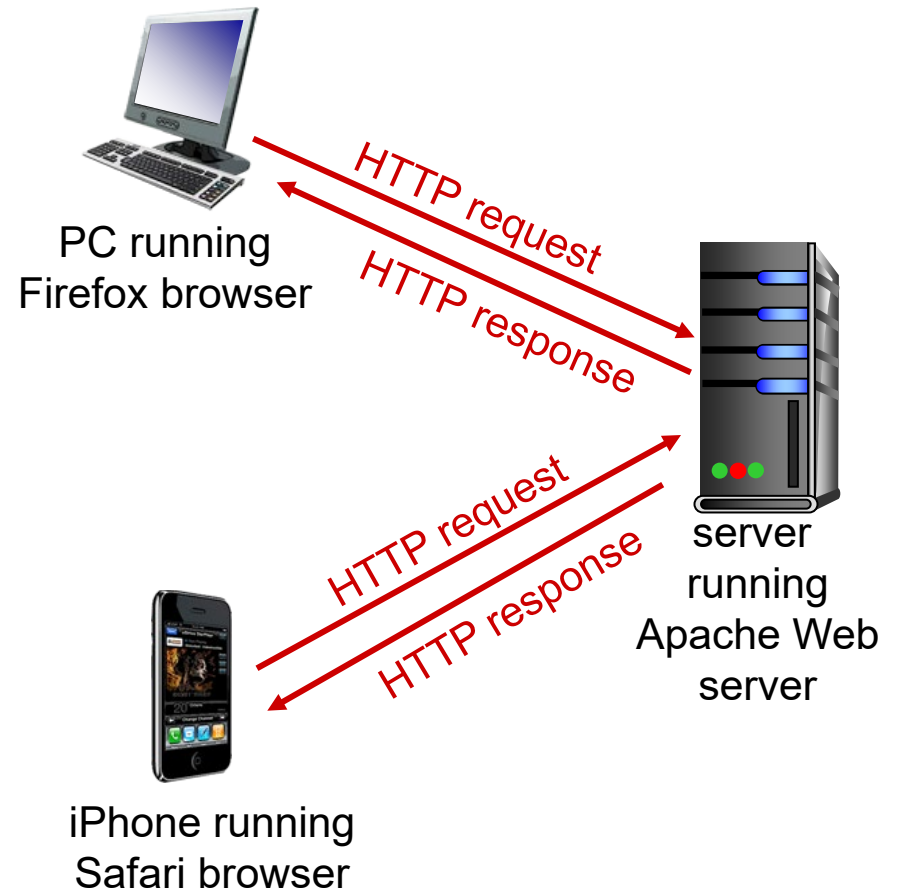
host name

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 fixed up

HTTP connections

non-persistent HTTP

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

persistent HTTP

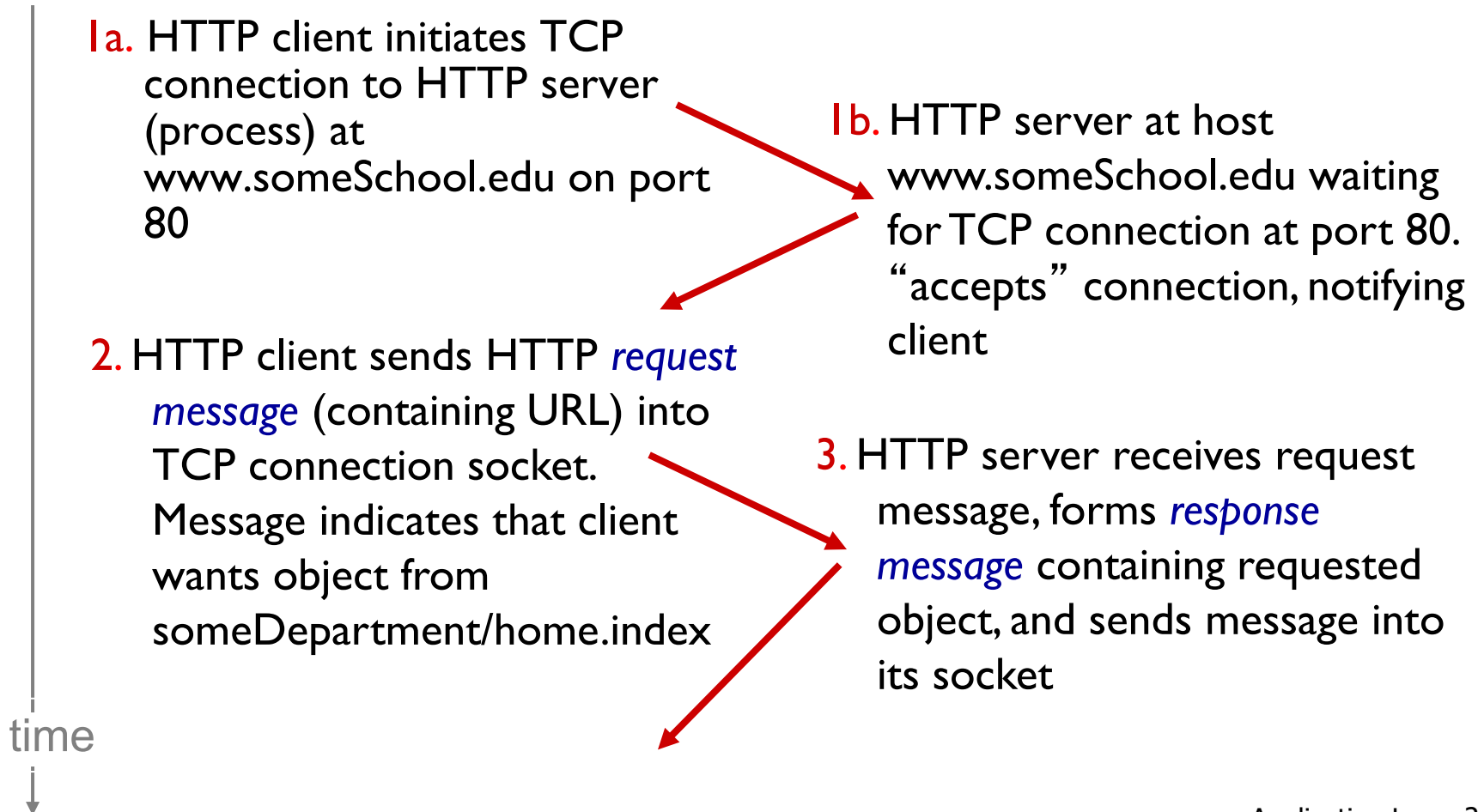
- multiple objects can be sent over single TCP connection between client and server

Non-persistent HTTP

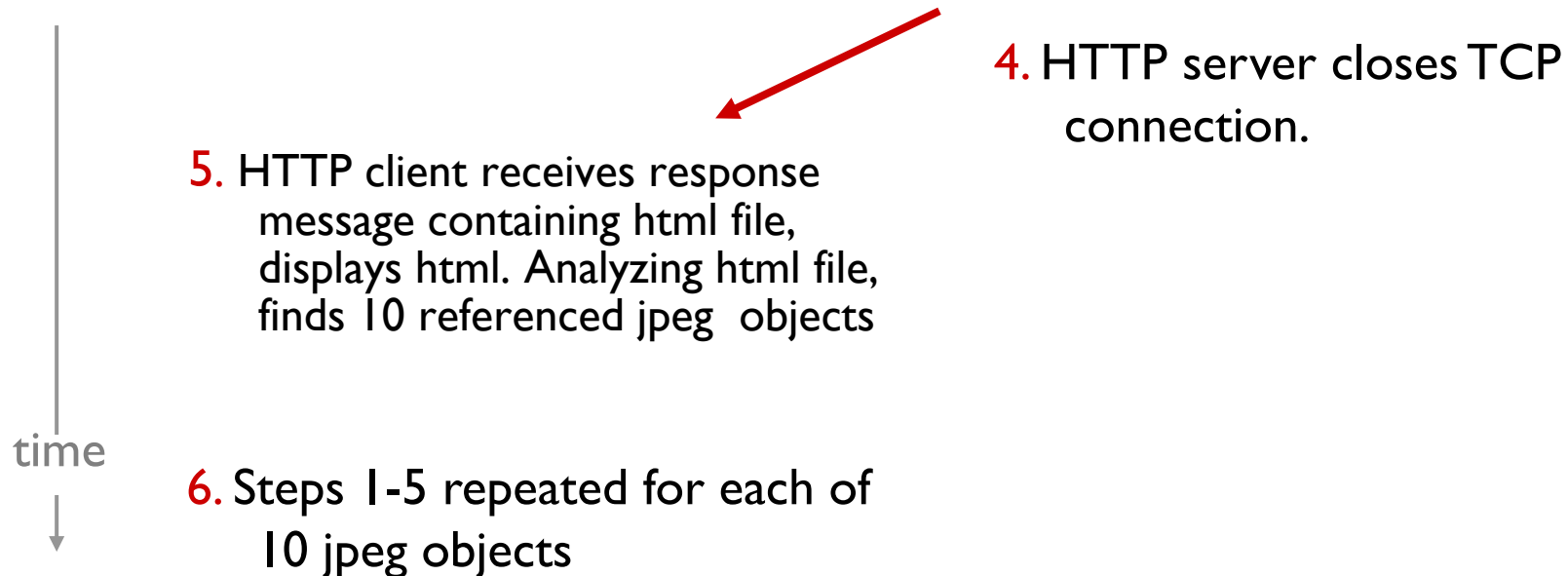
suppose user enters URL:

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

(contains text,
references to 10
jpeg images)



Non-persistent HTTP (cont.)

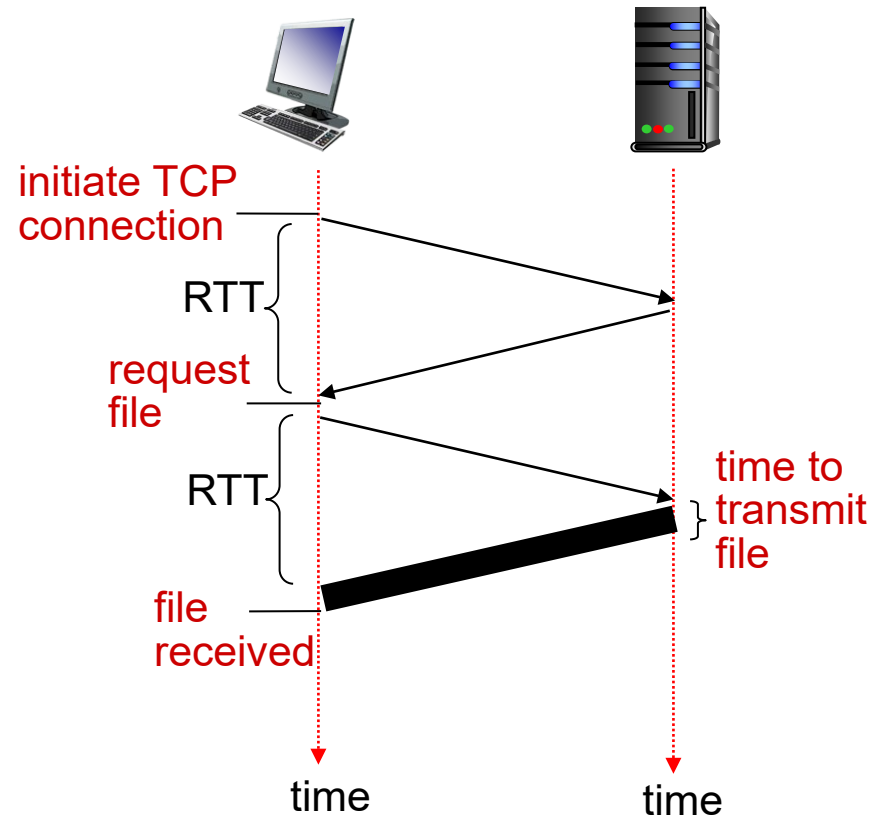


Non-persistent HTTP: response time

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 (with pipelining):

- 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 (pipelining)
- 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

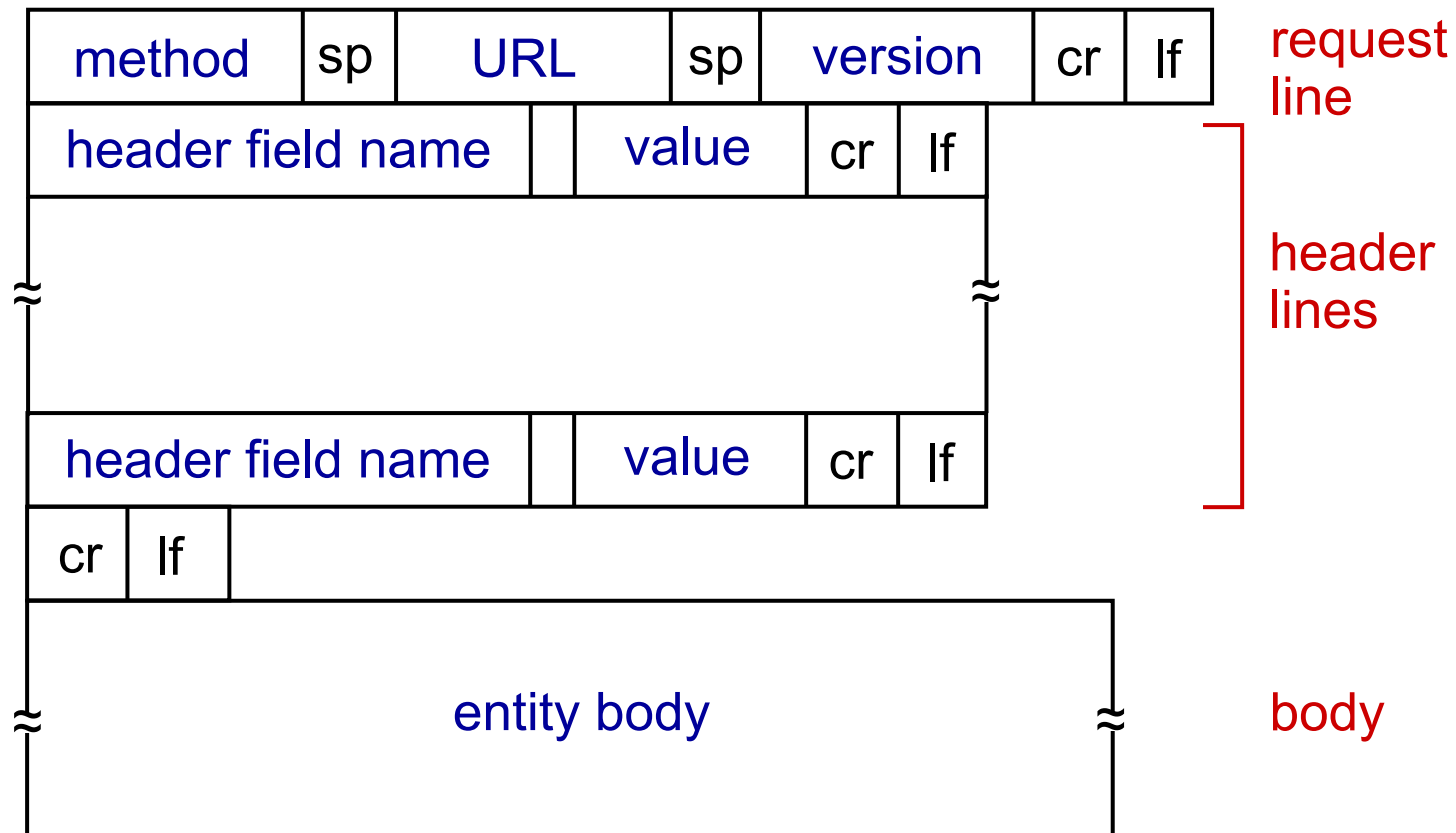
carriage return,
line feed at start
of line indicates
end of header lines

```
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

* Check out the online interactive exercises for more
examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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 ...
```

* Check out the online interactive exercises for more
examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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

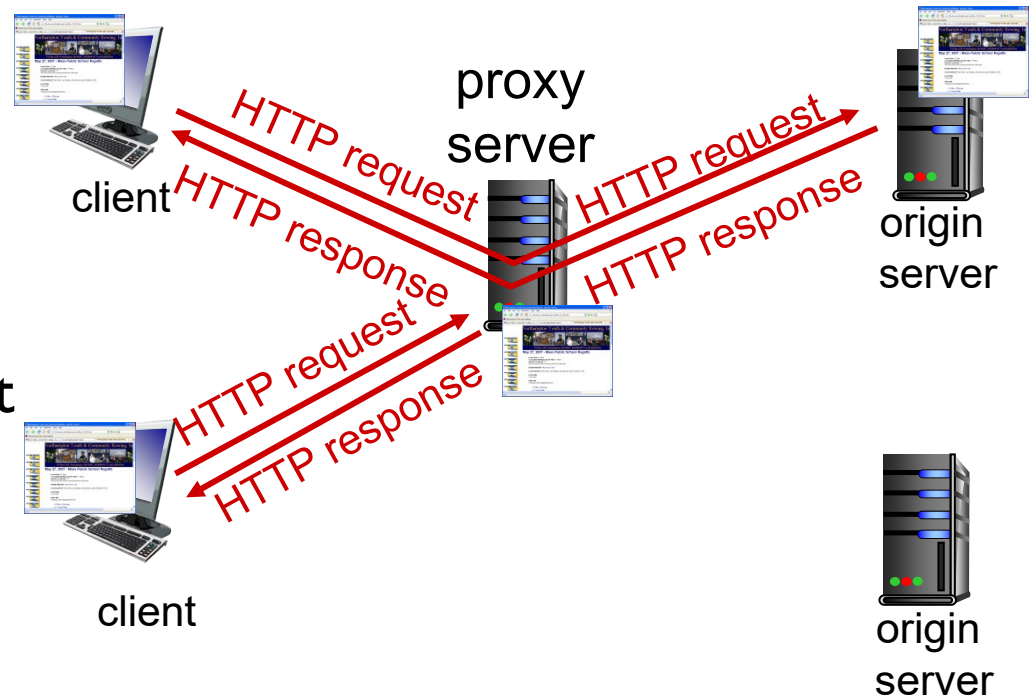
- requested document not found on this server

505 HTTP Version Not Supported

Web caches (proxy server)

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 requests object from origin server, then returns object to client
 - object in cache: cache returns object



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
- Internet dense with caches: enables “poor” content providers to effectively deliver content (so too does P2P file sharing)

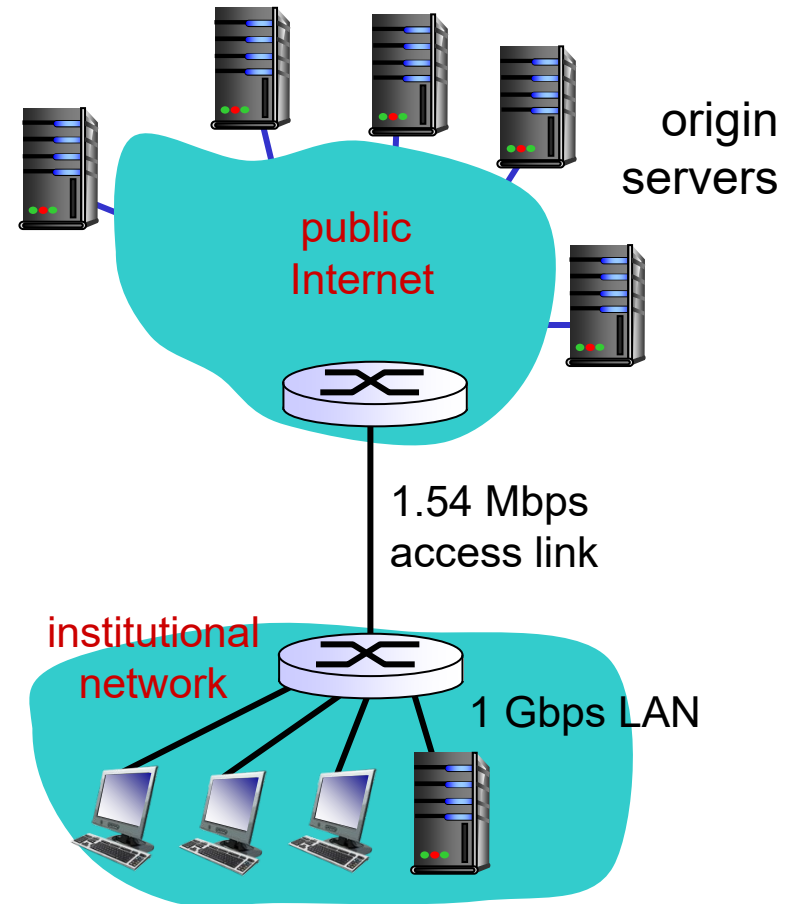
Caching example:

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

- LAN utilization: 0.15% *problem!*
- access link utilization = ~99%
- total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs



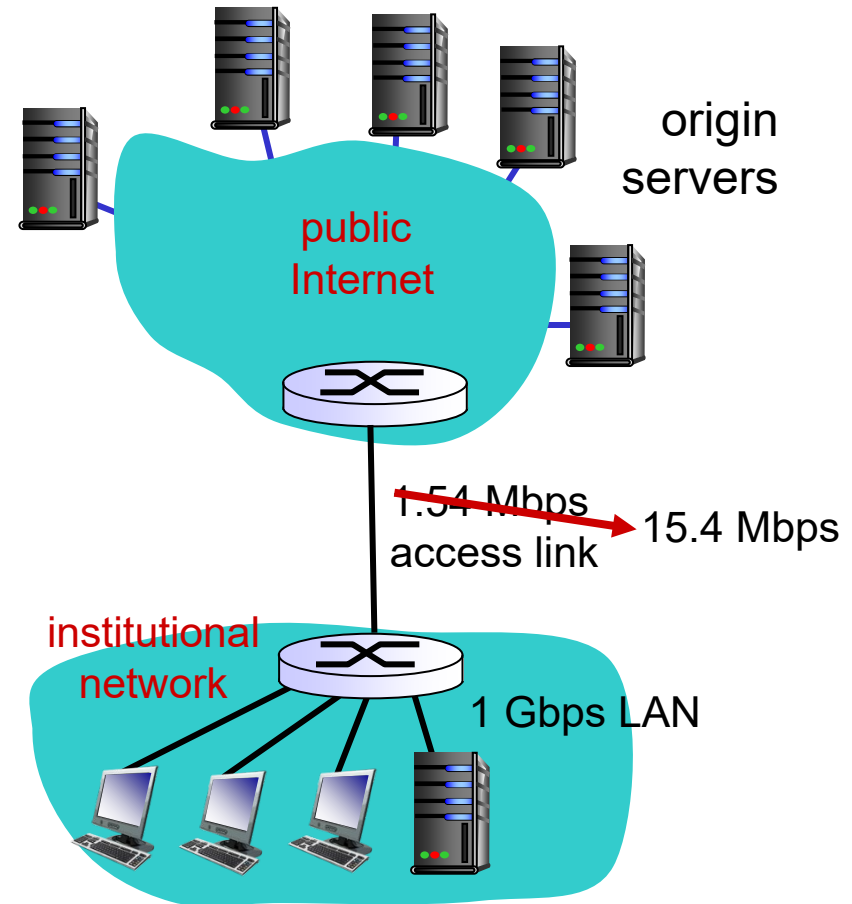
Caching example: faster access link

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: ~~1.54 Mbps~~ → 15.4 Mbps

consequences:

- LAN utilization: 0.15%
- access link utilization = ~~99%~~ → ~9.9%
- total delay = Internet delay + access delay + LAN delay
= 2 sec + ~~minutes~~ → msec



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

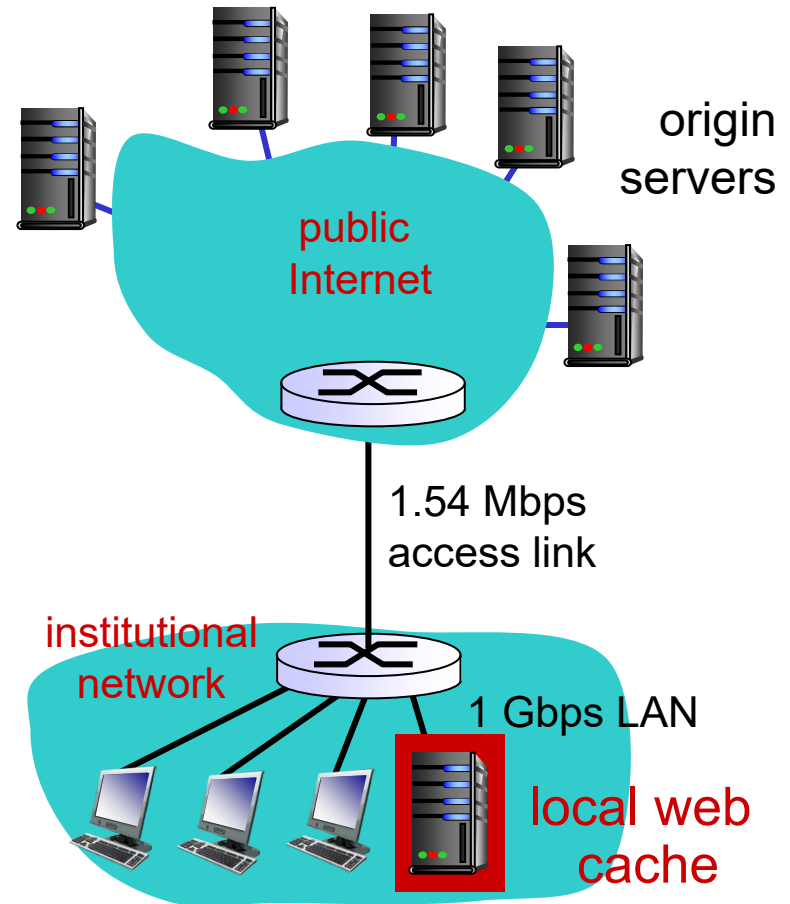
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

- LAN utilization: 0.15%
- access link utilization = ?
- total delay = ?

How to compute link utilization, & delay?

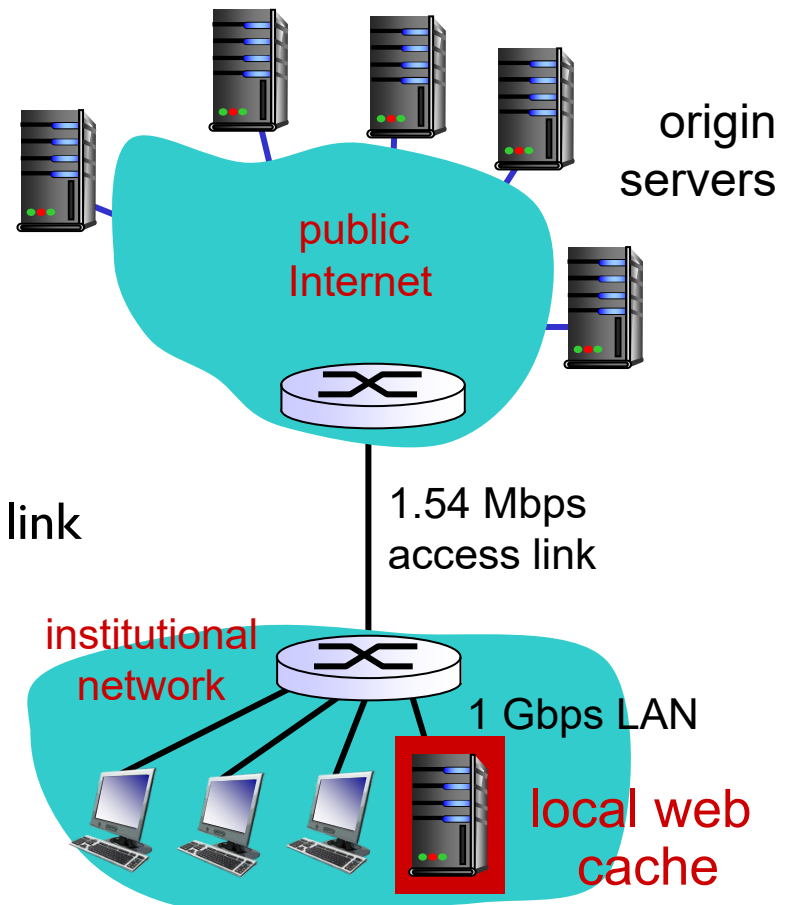
Cost: web cache (cheap!)



Caching example: install local cache

Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
 - 40% requests satisfied at cache, 60% requests satisfied at origin servers
- access link utilization:
 - 60% of requests use access link
- data rate to browsers over access link
 $= 0.6 * 1.50 \text{ Mbps} = 0.9 \text{ Mbps}$
 - utilization $= 0.9 / 1.54 = 0.58$
- total delay
 - $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
 - $= 0.6 (2 + \text{negligible}) + 0.4 (\text{negligible})$
 $= \sim 1.2 \text{ secs}$
 - less than with 15.4 Mbps link (and cheaper too!)



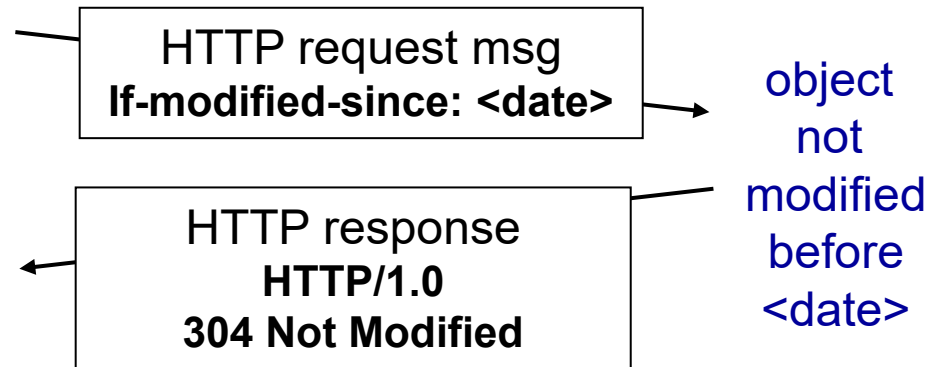
Conditional GET

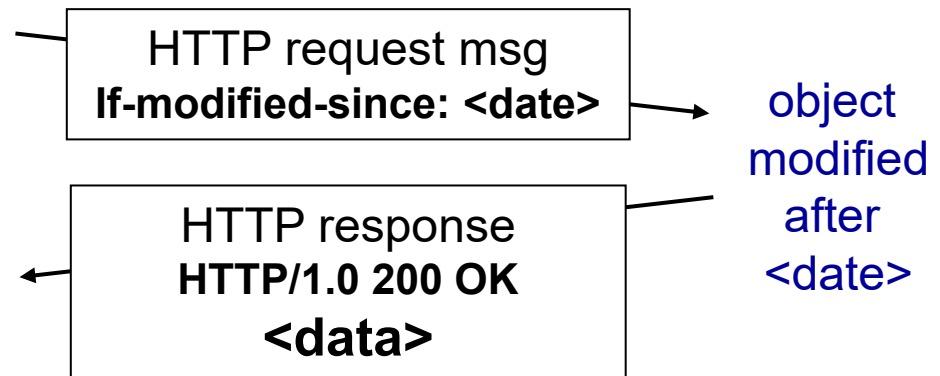
- **Goal:** don't send object if cache has up-to-date cached version
 - no object transmission delay
 - lower link utilization
- **cache:** specify date of cached copy in HTTP request
If-modified-since: <date>
- **server:** response contains no object if cached copy is up-to-date:
HTTP/1.0 304 Not Modified

client



server





User-server state: cookies

many Web sites use cookies

four components:

- 1) cookie header line of 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 Web site

example:

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

Cookies: keeping “state” (cont.)

Susan
(client)



Amazon
(server)



cookie file



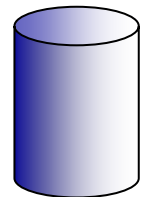
ebay 8734
amazon 1678

usual http request msg

Amazon server
creates ID
1678 for user

create
entry

backend
database



usual http response
set-cookie: 1678

usual http request msg
cookie: 1678

cookie-
specific
action

access

usual http response msg

access

cookie-
specific
action

usual http request msg
cookie: 1678

usual http response msg



ebay 8734
amazon 1678

one week later:

Cookies (continued)

what cookies can be used for:

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

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

how to keep “state”:

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry “state”

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 media streaming and content distribution networks

2.5 socket programming with UDP and TCP

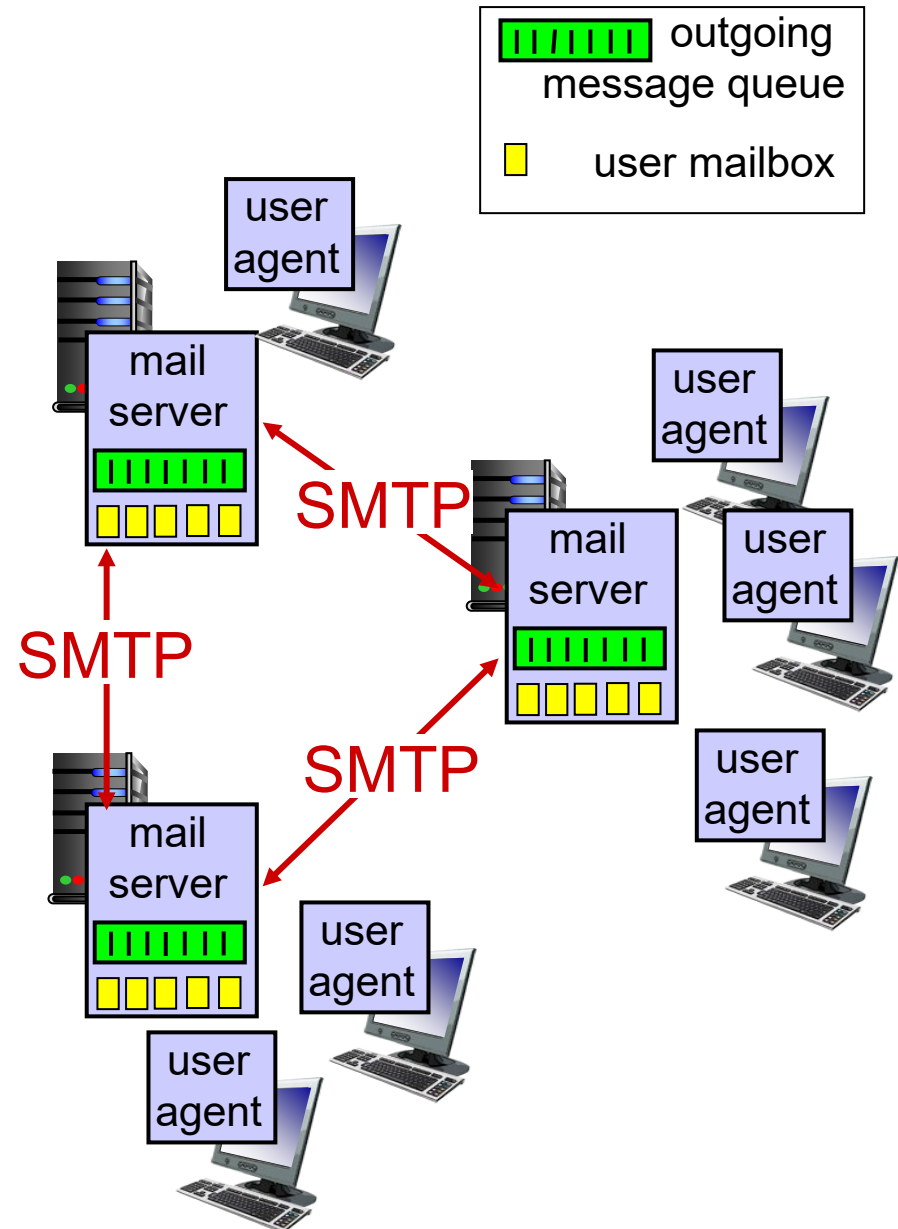
Electronic mail

Three major components:

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

User Agent

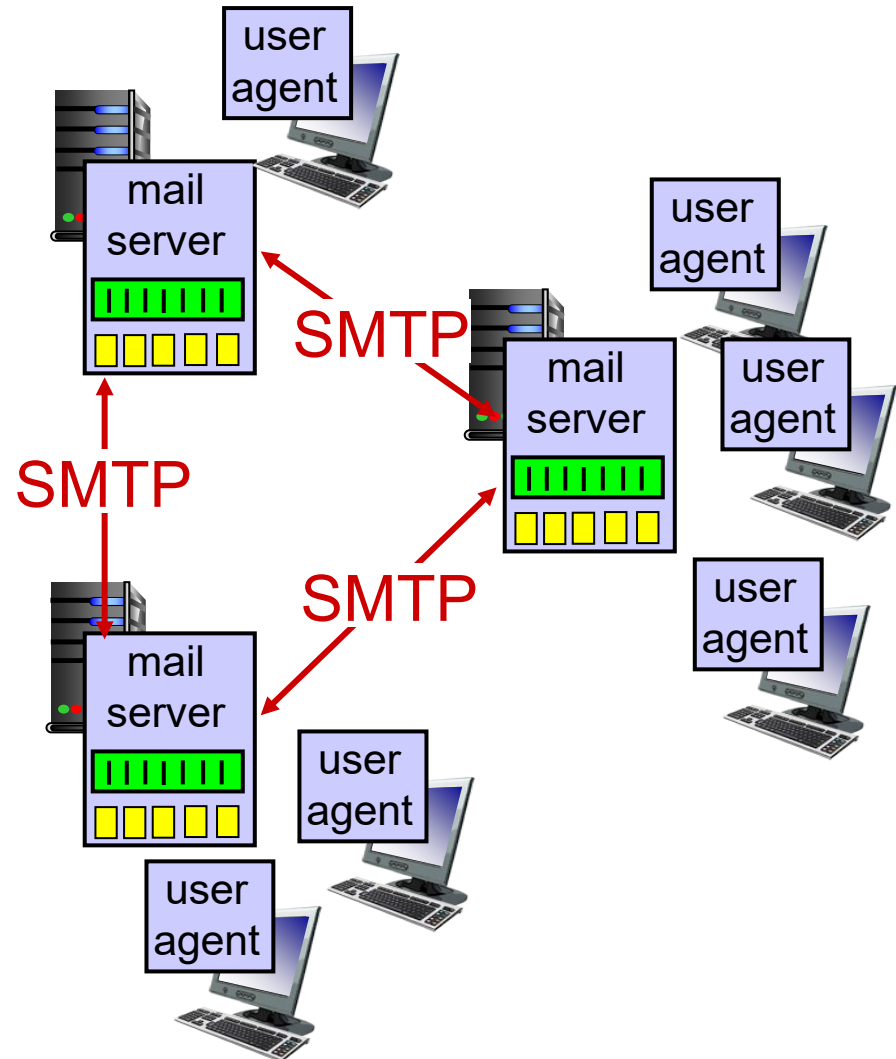
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



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

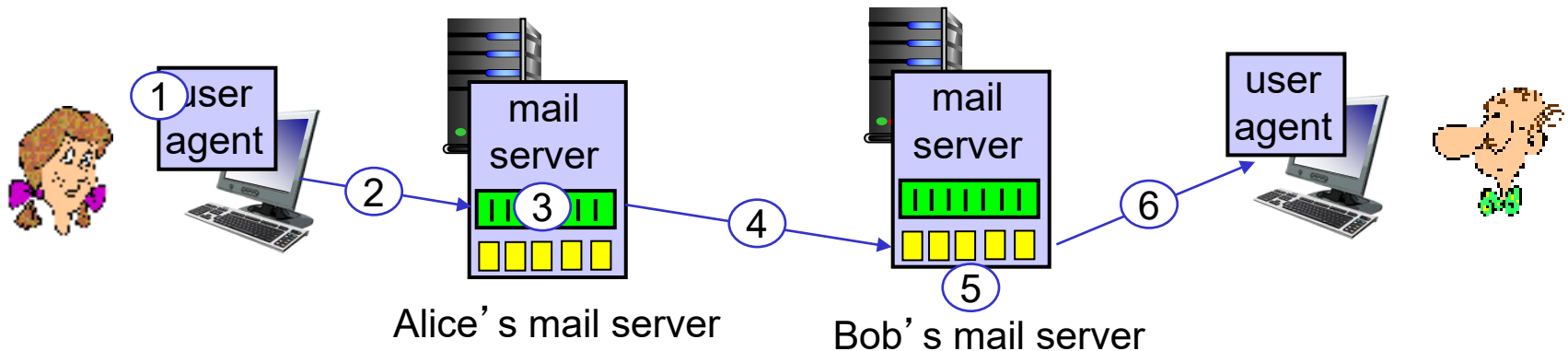


Electronic Mail: SMTP [RFC 2821]

- 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)
 - **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 commands his user agent to read message



Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes and phrase
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

Mail message format

SMTP: protocol for exchanging email messages

RFC 822: standard for text message format:

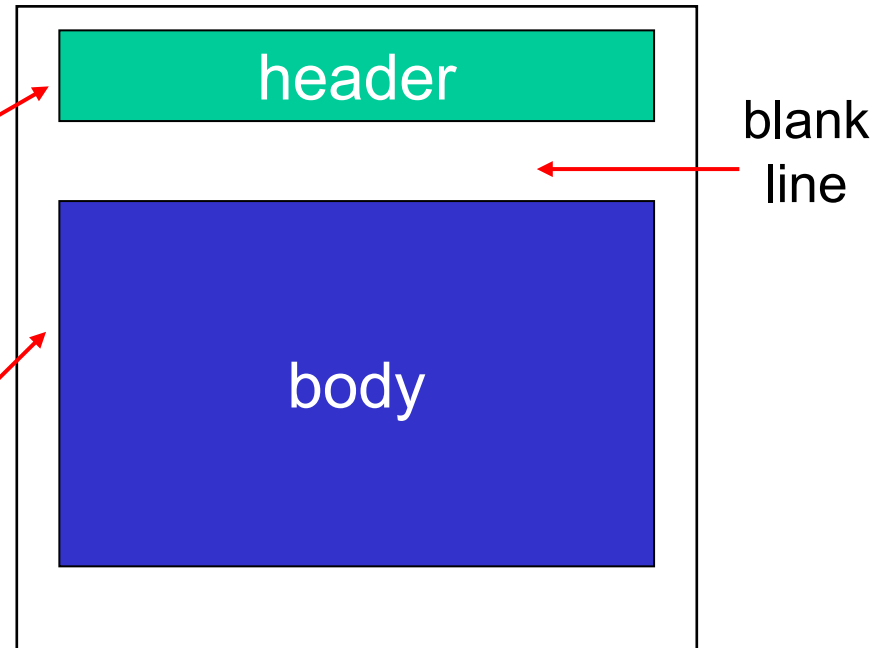
- header lines, e.g.,

- To:
- From:
- Subject:

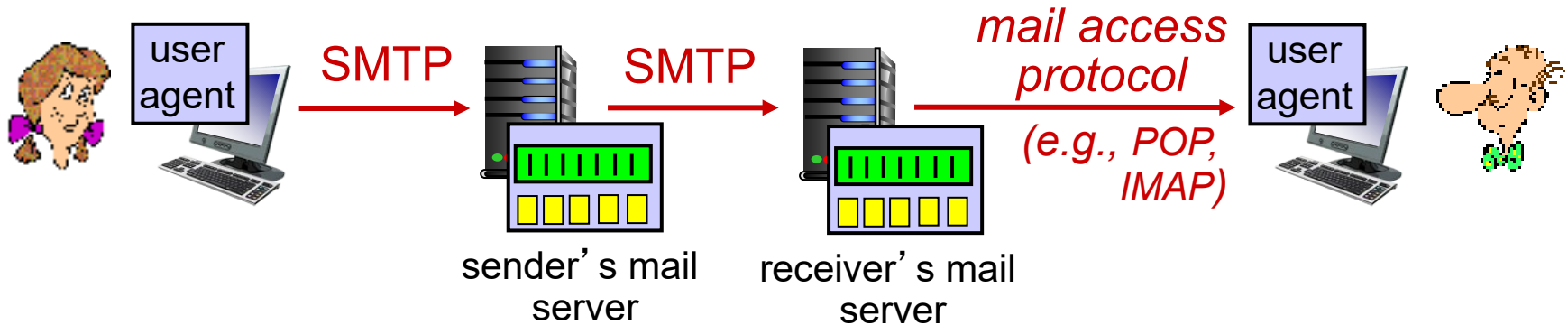
different from SMTP MAIL
FROM, RCPT TO:
commands!

- Body: the “message”

- ASCII characters only



Mail access protocols



- **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 messages on server
 - **HTTP:** gmail, Hotmail, Yahoo! Mail, etc.

POP3 protocol

authorization phase

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

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

transaction phase, client:

- **list**: list message numbers
- **retr**: retrieve message by number
- **dele**: delete
- **quit**

```
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
```

POP3 (more) and IMAP

more about POP3

- previous example uses POP3 “download and delete” mode
 - Bob cannot re-read e-mail if he changes client
- POP3 “download-and-keep”: copies of messages on different clients
- POP3 is stateless across sessions

IMAP

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

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 media streaming and content distribution networks (CDNs)

2.5 socket programming with UDP and TCP

Media Streaming and CDNs: context

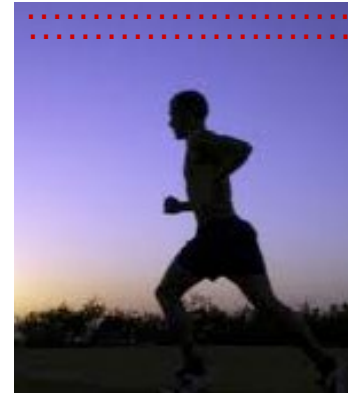
- video traffic: major consumer of Internet bandwidth
 - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
 - ~2B YouTube users, ~200M Netflix users
- challenge: scale - how to reach ~2B users?
 - single mega-video server won't work (why?)
- challenge: heterogeneity
 - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution*: distributed, application-level infrastructure



Multimedia: video

- video: sequence of images displayed at constant rate
 - e.g., 24 images/sec
- digital image: array of pixels
 - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
 - spatial (within image)
 - temporal (from one image to next)

spatial coding example: instead of sending N values of same color (all purple), send only two values: color value (*purple*) and number of repeated values (N)



frame i

temporal coding example: instead of sending complete frame at $i+1$, send only differences from frame i

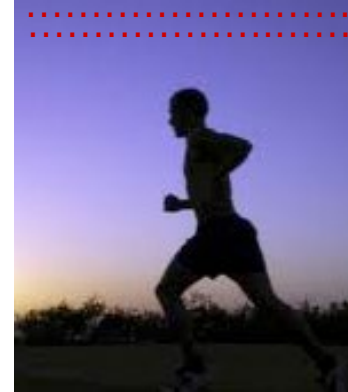


frame $i+1$

Multimedia: video

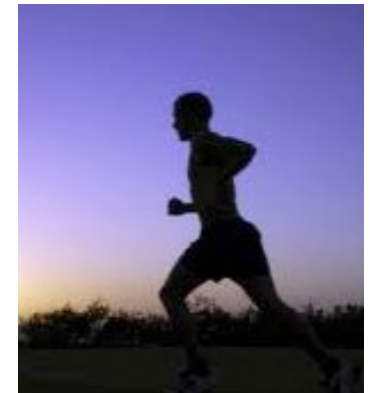
- **CBR (constant bit rate):** video encoding rate fixed
- **VBR: (variable bit rate):** video encoding rate changes as amount of spatial, temporal coding changes
- **examples:**
 - MPEG I (CD-ROM) 1.5 Mbps
 - MPEG2 (DVD) 3-6 Mbps
 - MPEG4 (often used in Internet) < 1 Mbps

spatial coding example: instead of sending N values of same color (all purple), send only two values: color value (*purple*) and number of repeated values (N)



frame i

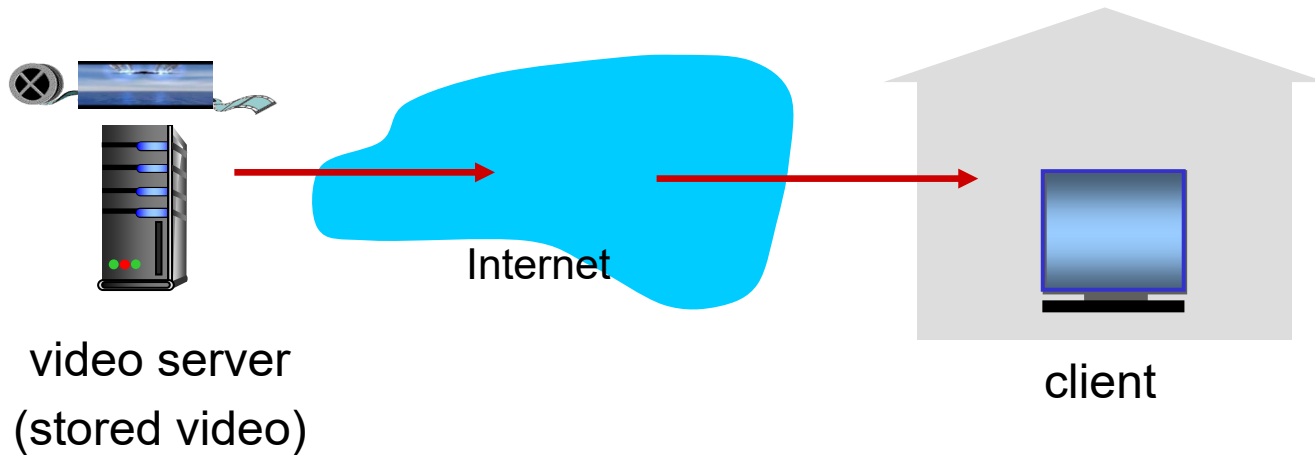
temporal coding example: instead of sending complete frame at $i+1$, send only differences from frame i



frame $i+1$

Streaming stored video:

simple scenario:



Streaming multimedia: DASH

- *DASH*: *D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server*:
 - divides video file into multiple chunks
 - each chunk stored, encoded at different rates
 - *manifest file*: provides URLs for different chunks
- *client*:
 - periodically measures server-to-client bandwidth
 - consulting manifest, requests one chunk at a time
 - chooses maximum coding rate sustainable given current bandwidth
 - can choose different coding rates at different points in time (depending on available bandwidth at time)

Streaming multimedia: DASH

- *DASH: Dynamic, Adaptive Streaming over HTTP*
- “intelligence” at client: client determines
 - *when* to request chunk (so that buffer starvation, or overflow does not occur)
 - *what encoding rate* to request (higher quality when more bandwidth available)
 - *where* to request chunk (can request from URL server that is “close” to client or has high available bandwidth)

Content distribution networks

- *challenge*: how to stream media content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- *option 1*: single, large “mega-server”
 - single point of failure
 - point of network congestion
 - long path to distant clients
 - multiple copies of video sent over outgoing link

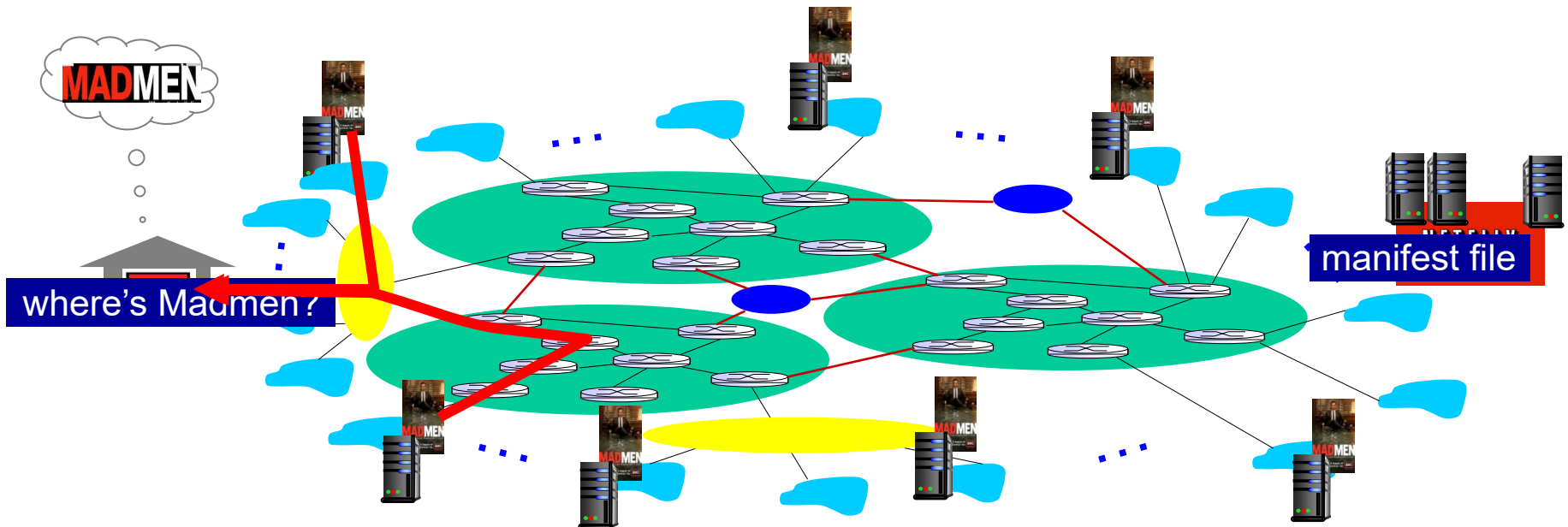
....quite simply: this solution *doesn't scale*

Content distribution networks

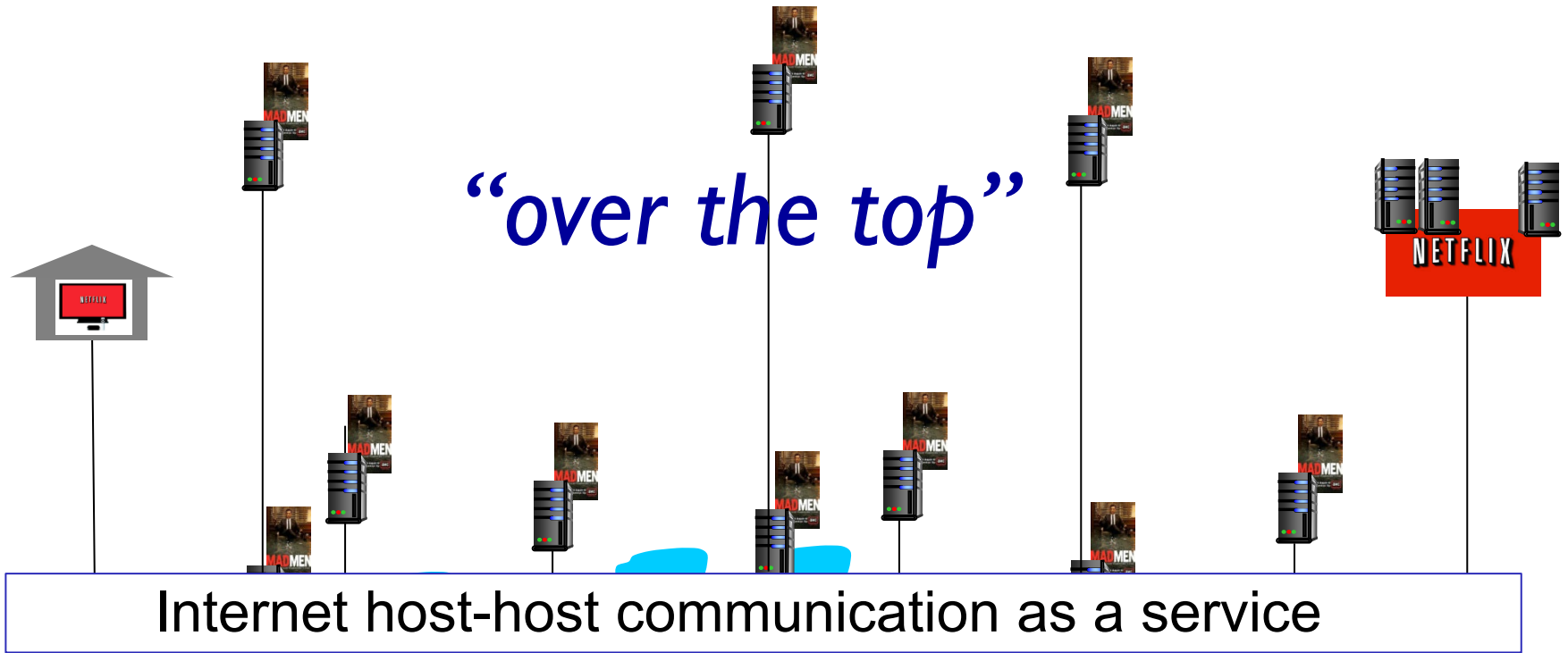
- *challenge*: how to stream media content (selected from millions of videos) to hundreds of thousands of simultaneous users?
- *option 2*: store/serve multiple copies of videos at multiple geographically distributed sites (*CDN*)
 - *enter deep*: push CDN servers deep into many access networks
 - close to users
 - used by Akamai, 1700 locations
 - *bring home*: smaller number (10's) of larger clusters in POPs near (but not within) access networks
 - used by Limelight

Content Distribution Networks (CDNs)

- CDN: stores copies of media content at CDN nodes
 - e.g. Netflix stores copies of MadMen
- subscriber requests media content from CDN
 - directed to nearby copy, retrieves content
 - may choose different copy if network path congested



Content Distribution Networks (CDNs)



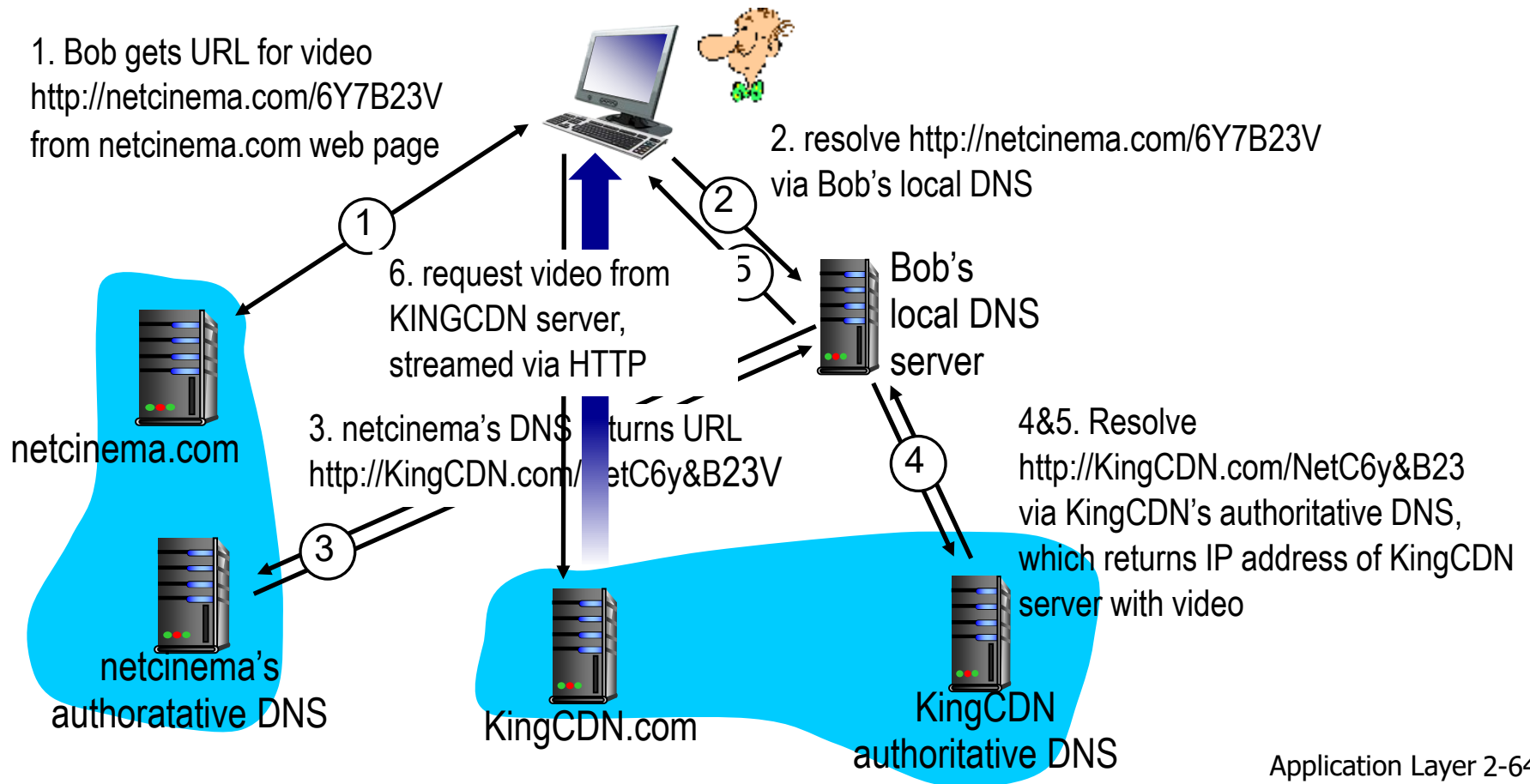
“over the top” challenges: coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

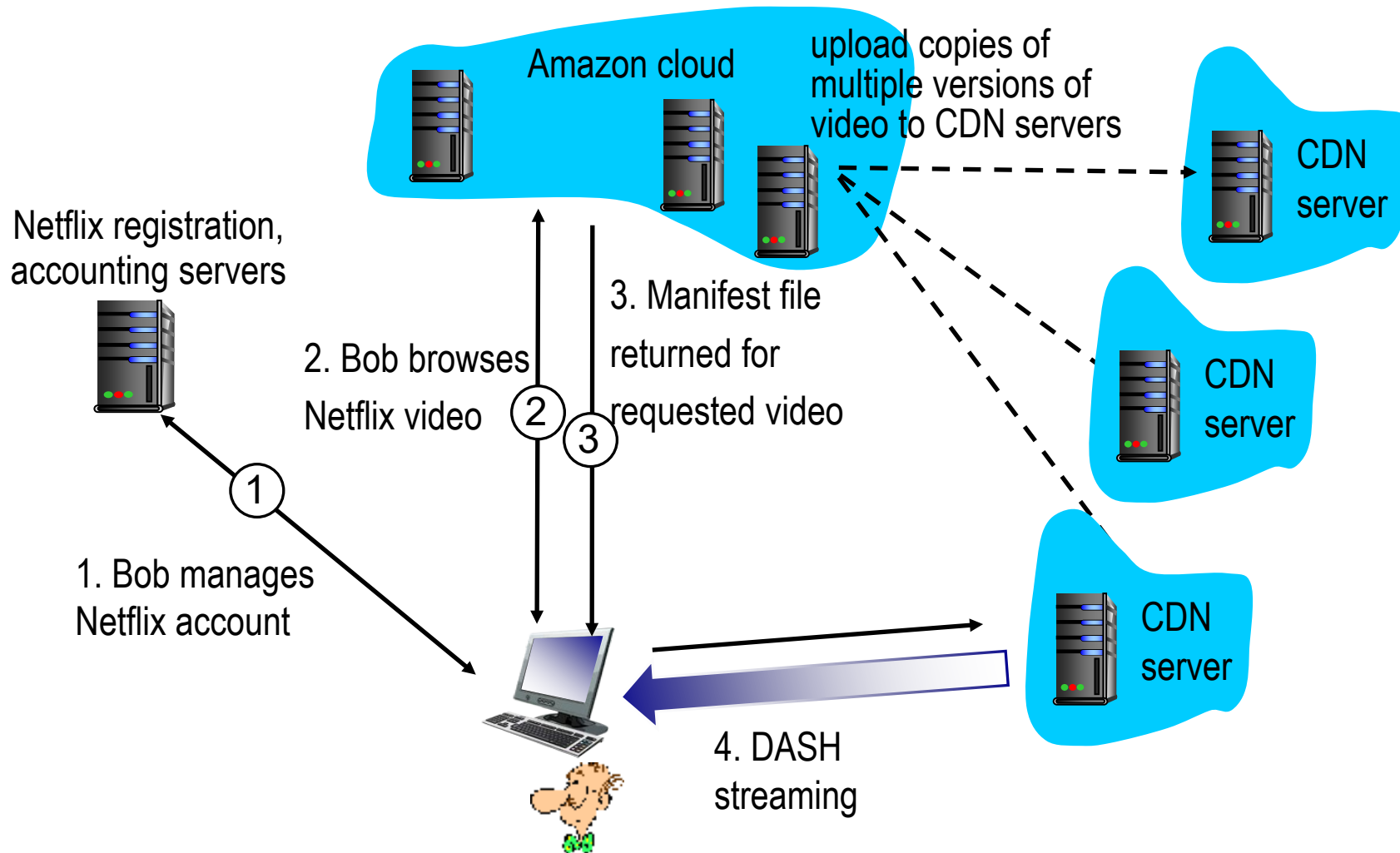
CDN content access: a closer look

Bob (client) requests video `http://netcinema.com/6Y7B23V`

- video stored in CDN at `http://KingCDN.com/NetC6y&B23V`



Case study: Netflix



Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

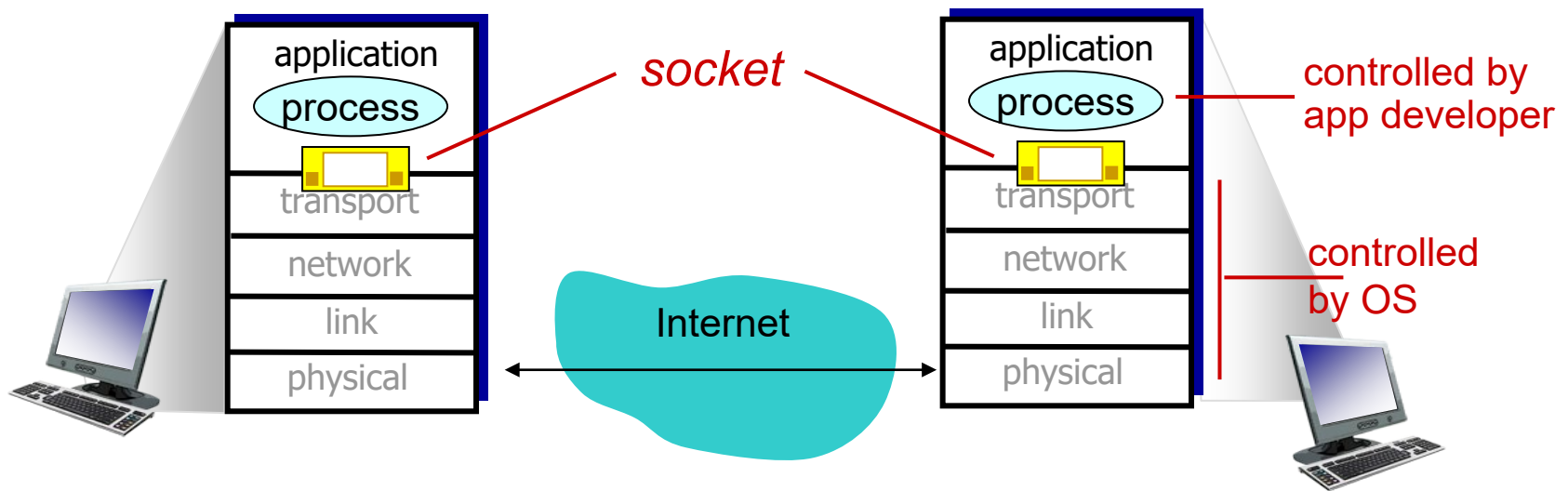
2.4 media streaming and content distribution networks

2.5 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* UDP

UDP: no “connection” between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver 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(AF_INET,
SOCK_DGRAM)

get user keyboard
input

message = raw_input('Input lowercase sentence:')

Attach server name, port to
message; send into socket

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

read reply characters from
socket into string

modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)

print out received string
and close socket

print modifiedMessage.decode()
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 True:  
    Read from UDP socket into  
    message, getting client's  
    address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)  
    modifiedMessage = message.decode().upper()  
    send upper case string  
    back to this client → serverSocket.sendto(modifiedMessage.encode(),  
                                                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

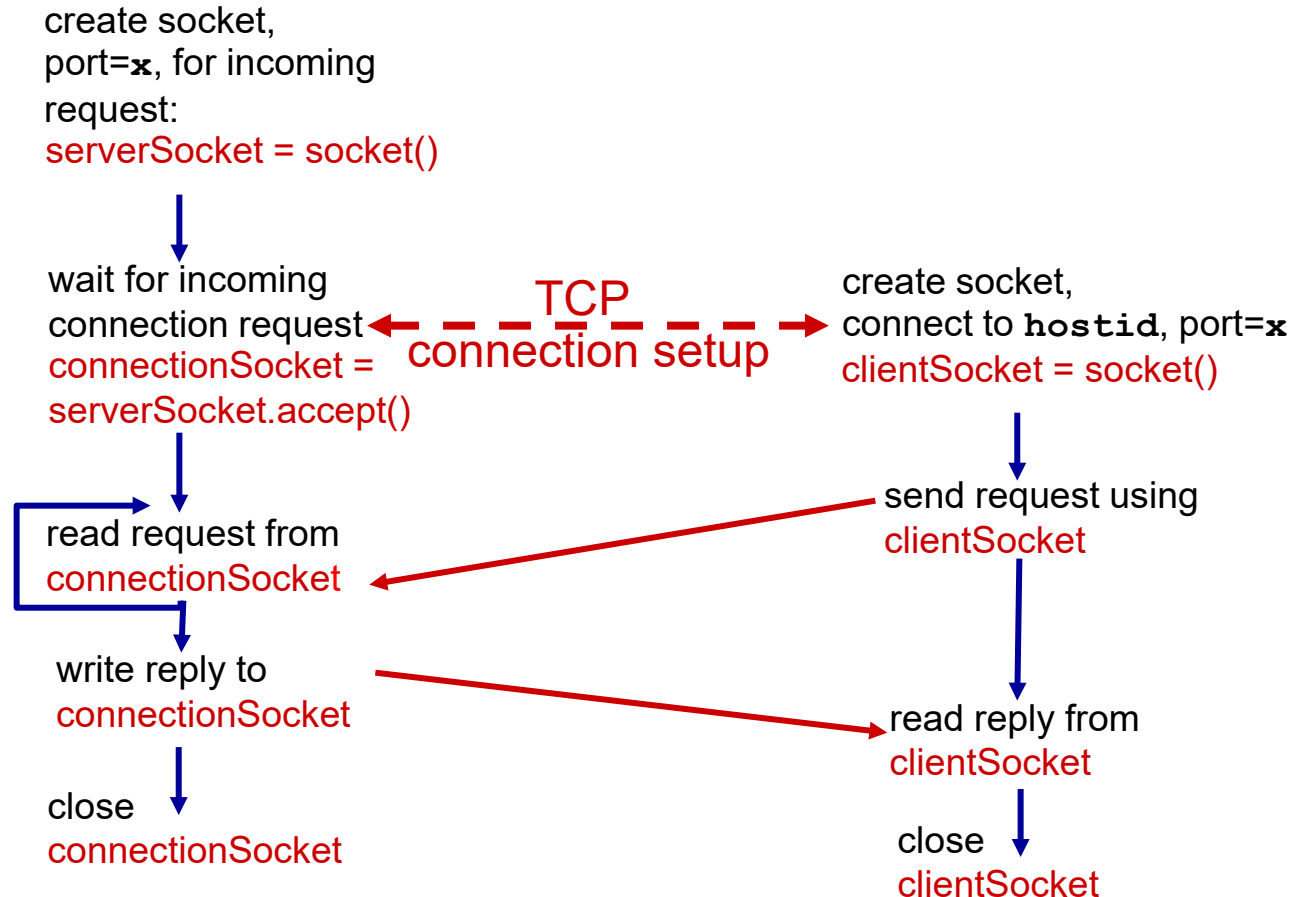
application viewpoint:

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.encode())
```

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

```
print ('From Server:', modifiedSentence.decode())
```

```
clientSocket.close()
```

Example app: TCP server

Python TCPServer

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)

```
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 True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                           encode())
    connectionSocket.close()
```

Summary

our study of network apps now complete!

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

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(payload) being communicated

important themes:

- messages vs. control
 - in-band, vs. out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable message transfer
- “complexity at network edge”