Computer Networks and Applications

COMP 3331/COMP 9331

Week 2

Application Layer (Principles, Web, Email)

Chapter 2: Sections 2.1, 2.2, 2.4

2. Application Layer: outline

2. Application layer

- ❖ 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
  - SMTP, IMAP
  - DNS
- ❖ programming network applications
  - socket API

Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing

- voice over IP (e.g., Skype)
- real-time video conferencing
- Internet search
- remote login
- …

Q: your favorites?

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
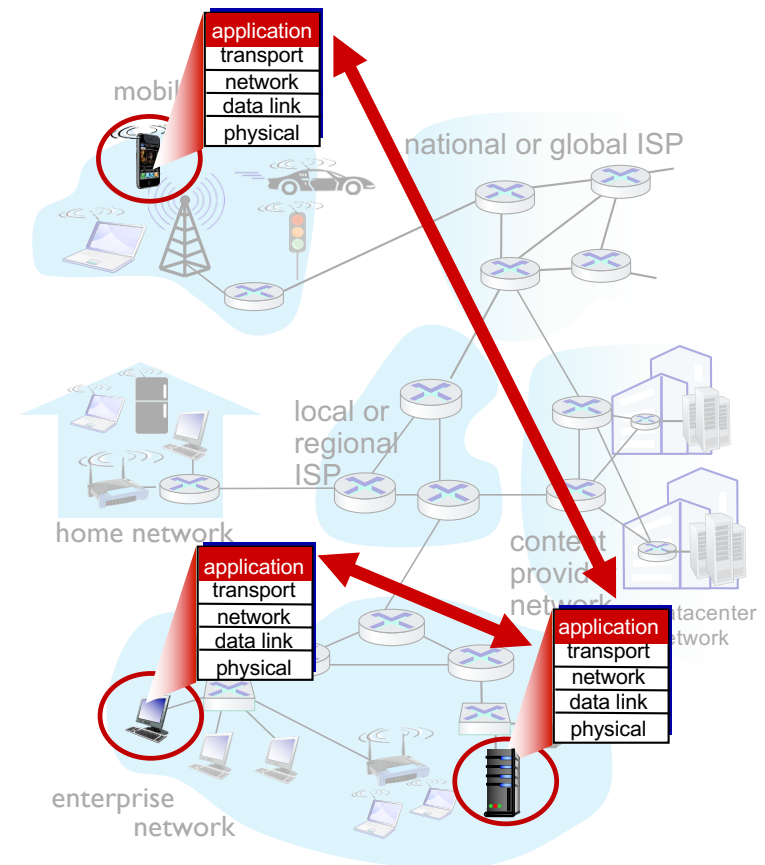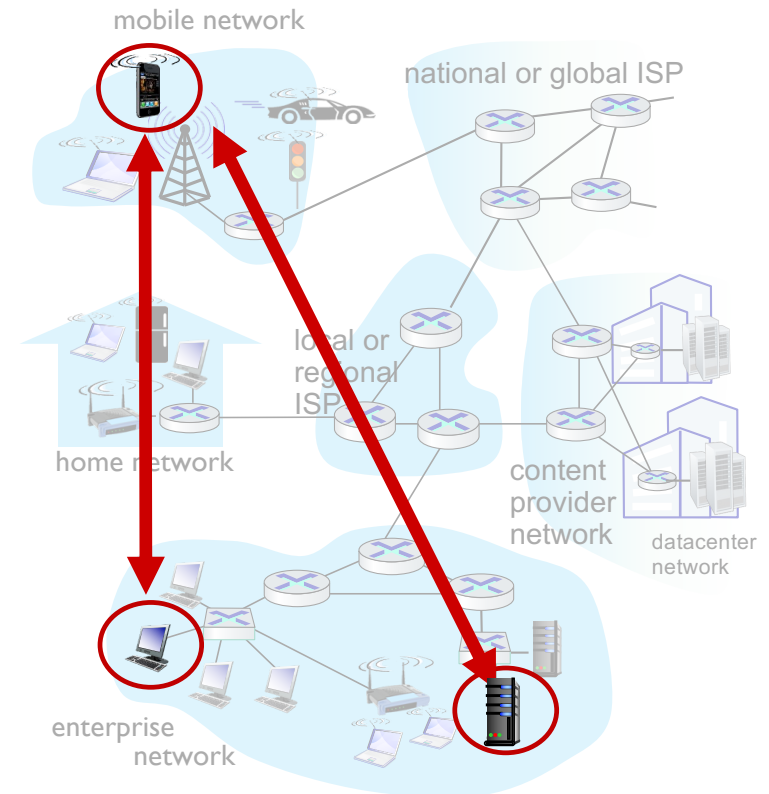
Client-server paradigm

server:
- always-on host
- permanent IP address
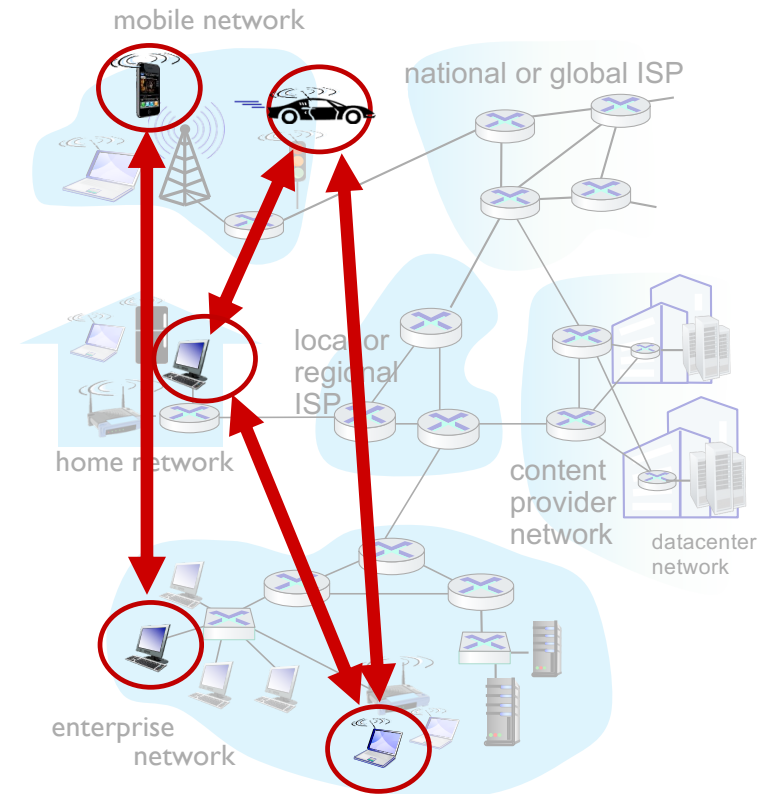- often in data centers, for scaling

clients:
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP

mobile network

national or global ISP

local or regional ISP

home network

content provider network

datacenter network

enterprise network

Peer-peer 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
- example: P2P file sharing, blockchain

mobile network

national or global ISP

local or regional ISP

home network

content provider network

datacenter network

enterprise network

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

- note: 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 the door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side
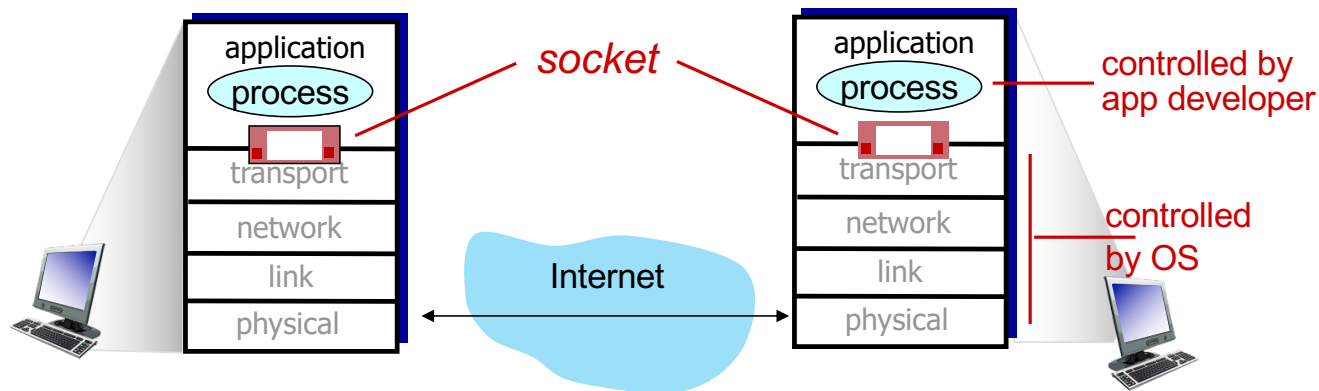
Addressing processes

- 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…

An application-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, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP, WebRTC

proprietary protocols:
- e.g., Skype, Zoom, Teams

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/download	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, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	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?*

NOTE: More on transport layer later

Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Securing TCP

Vanilla TCP & UDP sockets:
- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)
- provides encrypted TCP connections
- data integrity
- end-point authentication

TLS implemented in application layer
- apps use TLS libraries, that use TCP in turn

TLS socket API
- cleartext sent into socket traverse Internet *encrypted*
- see Chapter 8

Quiz: Transport

Pick the true statement

A. TCP provides reliability and guarantees a minimum bandwidth

B. TCP provides reliability while UDP provides bandwidth guarantees

C. TCP provides reliability while UDP does not

D. Neither TCP nor UDP provides reliability

2. Application Layer: outline

The Web – History

Tim Berners-Lee

❖ World Wide Web (WWW): a distributed database of "pages" linked through Hypertext Transport Protocol (HTTP)
- First HTTP implementation - 1990
  - Tim Berners-Lee at CERN
- HTTP/0.9 – 1991
  - Simple GET command for the Web
- HTTP/1.0 –1992
  - Client/Server information, simple caching
- HTTP/1.1 – 1996
- HTTP2.0 - 2015

http://info.cern.ch/hypertext/WWW/TheProject.html

19

2021

This Is What Happens In An Internet Minute

facebook.
1.4 Million Scrolling

21.1 Million Texts Sent

Linked in
9,132 Connections Made

You Tube
500 Hours Content Uploaded

NETFLIX
28,000 Subscribers Watching

Google play / *App Store*
414,764 Apps Downloaded

$1.6 Million Spent Online

695,000 Stories Shared

3.4 Million Snaps Created

60 SECONDS

200,000 People Tweeting

69 Million Messages Sent
Facebook Messenger / WhatsApp

2 Million Swipes
tinder

3 Million Images Viewed

197.6 Million Emails Sent

imgur
932 Smart Audio Devices Shipped
amazon echo / Google Home

5,000 Downloads
Tik Tok

2 Million Views
twitch

Created By:
@LoriLewis
@OfficiallyChadd

20

Web and HTTP

First, a quick review…

- web page consists of *objects,* each of which can be stored on different Web servers

- object can be HTML file, JPEG image, Java applet, audio file,…

- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL,* e.g.,

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

        host name   path name

Uniform Resource Locator (URL)

```
protocol://host-name[:port]/directory-path/resource
```

- ❖ *protocol*: http, ftp, https, smtp *etc.*
- ❖ *hostname*: DNS name, IP address
- ❖ *port*: defaults to protocol's standard port; *e.g.,* http: 80 https: 443
- ❖ *directory path*: hierarchical, reflecting file system
- ❖ *resource*: Identifies the desired resource

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 request →

HTTP response →

HTTP request →

HTTP response →

PC running
Firefox browser

iPhone running
Safari browser

server running
Apache Web
server

HTTP overview (continued)

HTTP 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

24

HTTP request message

- two types of HTTP messages: *request, response*

- HTTP request message:
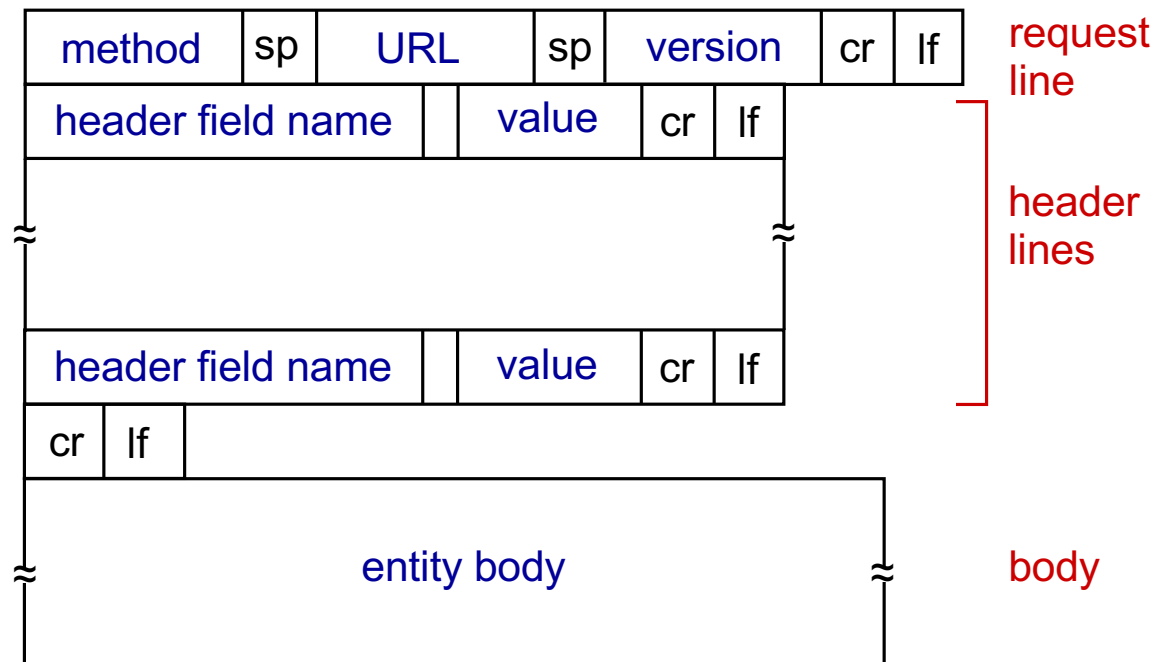 - ASCII (human-readable format)

request line (GET, POST, HEAD commands)

carriage return character
line-feed character

```
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
```

header lines

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

HTTP request message: general format

method	sp	URL	sp	version	cr	lf	request line

header field name		value	cr	lf

header lines

header field name		value	cr	lf

cr	lf

entity body	body

Other HTTP request messages

POST method:
- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):
- include user data in URL field of HTTP GET request message (following a '?'):

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

HEAD method:
- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

PUT method:
- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of PUT HTTP request message

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 message

301 Moved Permanently
- requested object moved, new location specified later in this message (in Location: field)

400 Bad Request
- request msg not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

HTTP is all text

❖ Makes the protocol simple
  - Easy to delineate messages (\r\n)
  - (relatively) human-readable
  - No issues about encoding or formatting data
  - Variable length data

❖ Not the most efficient
  - Many protocols use binary fields
    - Sending "12345678" as a string is 8 bytes
    - As an integer, 12345678 needs only 4 bytes
  - Headers may come in any order
  - Requires string parsing/processing

❖ Non-text content needs to be encoded

Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

- no notion of multi-step exchanges of HTTP messages to complete a Web "transaction"
  - no need for client/server to track "state" of multi-step exchange
  - all HTTP requests are independent of each other
  - no need for client/server to "recover" from a partially-completed-but-never-completely-completed transaction

a stateful protocol: client makes two changes to X, or none

lock data record
X

OK

update X ←X'

OK

t'

update X ←X"

OK

unlock X

OK

time   time
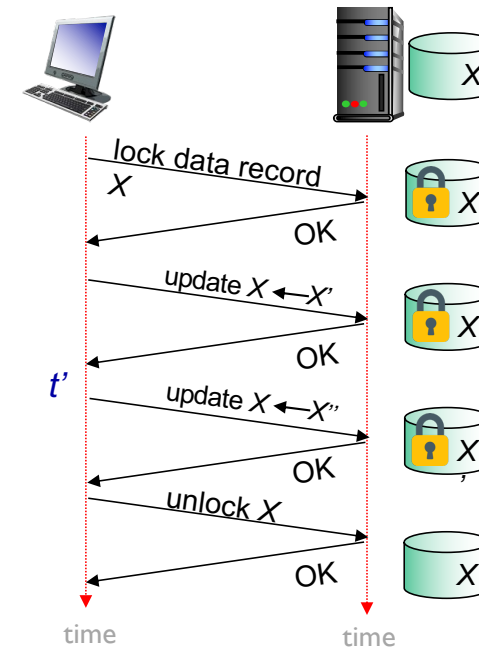
Q: what happens if network connection or client crashes at *t'*?

Maintaining user/server state: cookies

Web sites and client browser use *cookies* to maintain some state between transactions

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 uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
    - unique ID (aka "cookie")
    - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

Maintaining user/server state: cookies

client

server

ebay 8734

cookie file

ebay 8734
amazon 1678

usual HTTP request msg

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

usual HTTP request msg
cookie: 1678

usual HTTP response msg

one week later:

ebay 8734
amazon 1678

usual HTTP request msg
cookie: 1678

usual HTTP response msg

Amazon server
creates ID
1678 for user

create
entry

backend
database

cookie-
specific
action

access

access

cookie-
specific
action

time

time

HTTP cookies: comments

What cookies can be used for:

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

Challenge: How to keep state:

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

cookies and privacy:

- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

The Dark Side of Cookies

- ❖ Cookies permit sites to learn a lot about you

- ❖ You may supply name and e-mail to sites (and more)

- ❖ 3rd party cookies (from ad networks, etc.) can follow you across multiple sites
  - ▪ Ever visit a website, and the next day ALL your ads are from them ?
    - • Check your browser's cookie file (cookies.txt, cookies.plist)
    - • Do you see a website that you have never visited

- ❖ You COULD turn them off
  - ▪ But good luck doing anything on the Internet !!

Third party cookies

Website: "We use cookies to improve performance."
Me: Same.

Doubleclick server

Website A

Banner 1 url

Website B

Banner 2 url

Cookie:3445

Create cookie for doubleclick: 3445
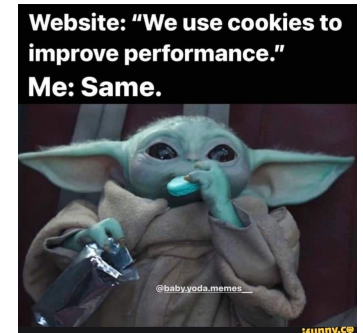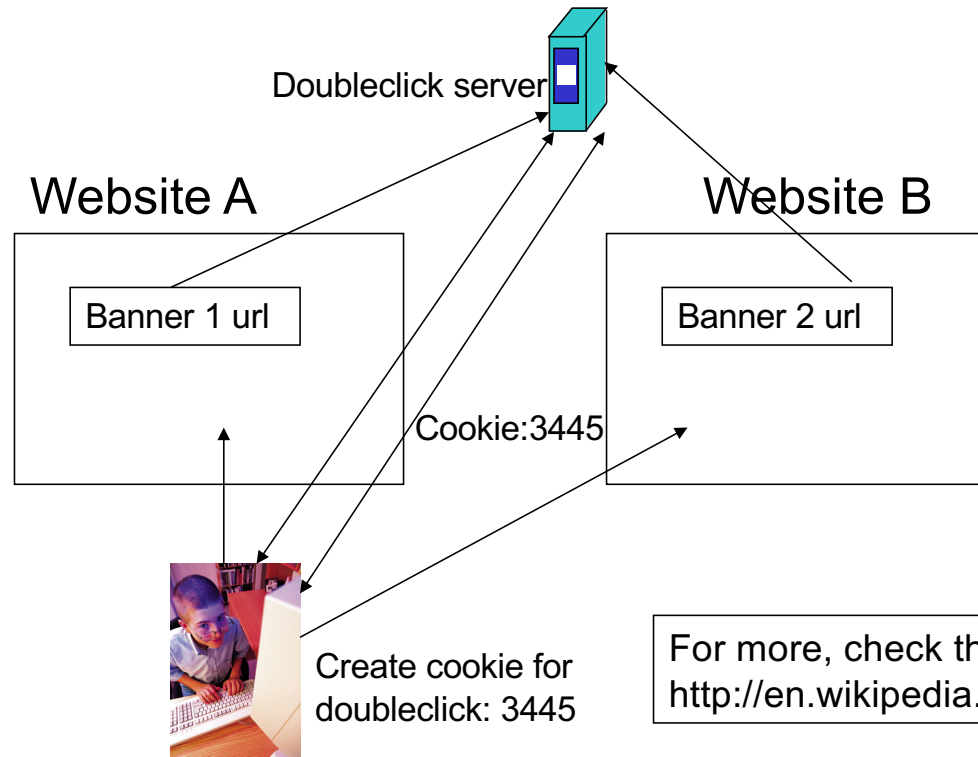
For more, check the following link and follow the references:
http://en.wikipedia.org/wiki/HTTP_cookie

In practice the banner can be a single pixel (invisible to the user)

36

Performance of HTTP

➢ Page Load Time (PLT) is an important metric

  • From click (or typing URL) until user sees page
  • Key measure of web performance

➢ Depends on many factors such as

  • page content/structure,
  • protocols involved and
  • Network bandwidth and RTT

Performance Goals

❖ User
  ▪ fast downloads
  ▪ high availability

❖ Content provider
  ▪ happy users (hence, above)
  ▪ cost-effective infrastructure

❖ Network (secondary)
  ▪ avoid overload

Solutions?

❖ User
 ▪ fast downloads
 ▪ high availability

❖ Content provider
 ▪ happy users (hence, above)
 ▪ cost-effective infrastructure

❖ Network (secondary)
 ▪ avoid overload

Improve HTTP to achieve faster downloads

39

Solutions?

Improve HTTP to achieve faster downloads

❖ User
  ▪ fast downloads
  ▪ high availability

❖ Content provider
  ▪ happy users (hence, above)
  ▪ cost-effective delivery infrastructure

❖ Network (secondary)
  ▪ avoid overload

Caching and Replication

Solutions?

Improve HTTP to achieve faster downloads

- ❖ User
  - fast downloads
  - high availability

- ❖ Content provider
  - happy users (hence, above)
  - cost-effective delivery infrastructure

Caching and Replication

- ❖ Network (secondary)
  - avoid overload

Exploit economies of scale
(Webhosting, CDNs, datacenters)

How to improve Page Load Time (PLT)

➢ Reduce content size for transfer

  • Smaller images, compression

➢ Change HTTP to make better use of available bandwidth

  • Persistent connections and pipelining

➢ Change HTTP to avoid repeated transfers of the same content

  • Caching and web-proxies

➢ Move content closer to the client

  • CDNs

HTTP Performance

❖ Most Web pages have multiple objects
  ▪ *e.g.,* HTML file and a bunch of embedded images

❖ How do you retrieve those objects (naively)?
  ▪ *One item at a time*

❖ New TCP connection per (small) object!

  non-persistent HTTP

  ❖ at most one object sent over TCP connection
    ▪ connection then closed
  ❖ downloading multiple objects required multiple 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 (approximate 3-way handshake)
* ❖ one RTT for HTTP request and first few bytes of HTTP response to return
* ❖ file transmission time
* ❖ non-persistent HTTP response time =
   2RTT+ file transmission time
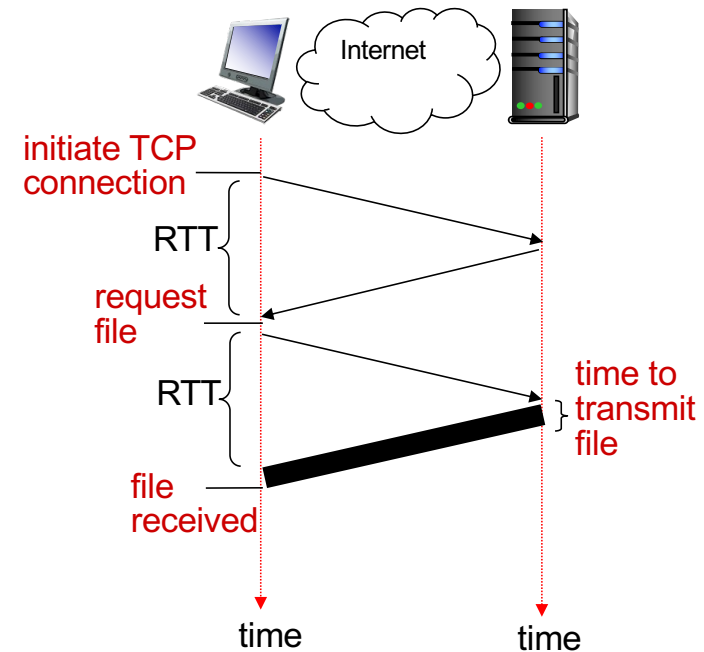
HTTP/1.0

➢ Non-Persistent: One TCP connection to fetch one web resource

➢ Fairly poor PLT

➢ 2 Scenarios
  • Multiple TCP connections setups to the same server
  • Sequential request/responses even when resources are located on different servers

➢ Multiple TCP slow-start phases (more in lecture on TCP)

Connection setup — HTTP Request

HTTP Response

Connection setup

Time
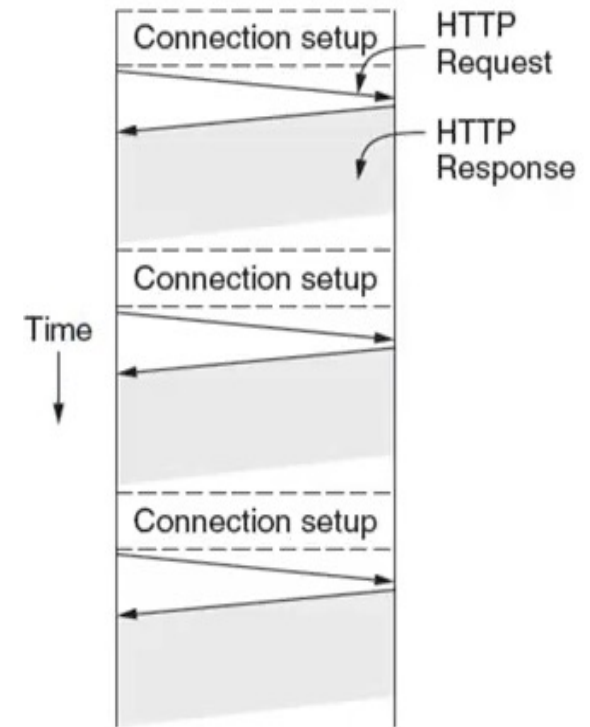
Connection setup

Improving HTTP Performance:
Concurrent Requests & Responses

❖ Use multiple connections *in parallel*

❖ Does not necessarily maintain order of responses

R1
R2
R3
T2
T3
T1
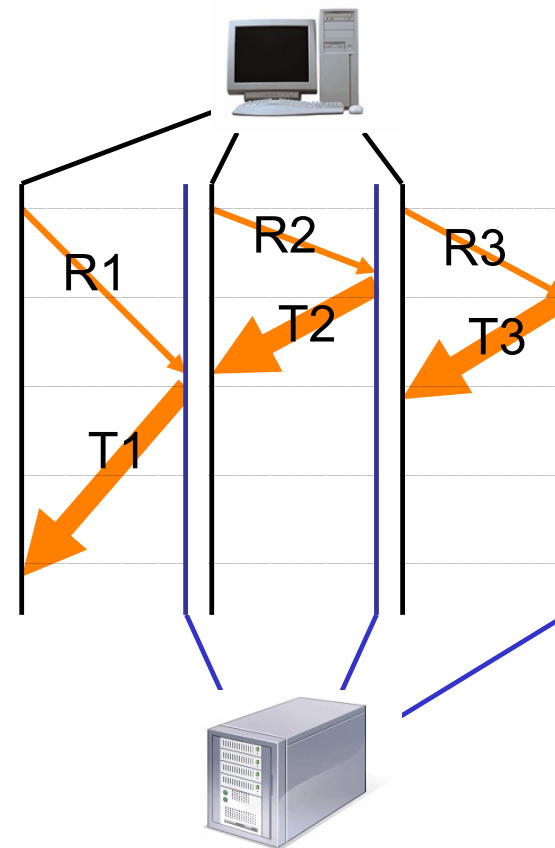
Quiz: Parallel HTTP Connections

What are potential downsides of parallel HTTP connections, i.e., can opening too many parallel connections be harmful and if so in what way?

Answer:

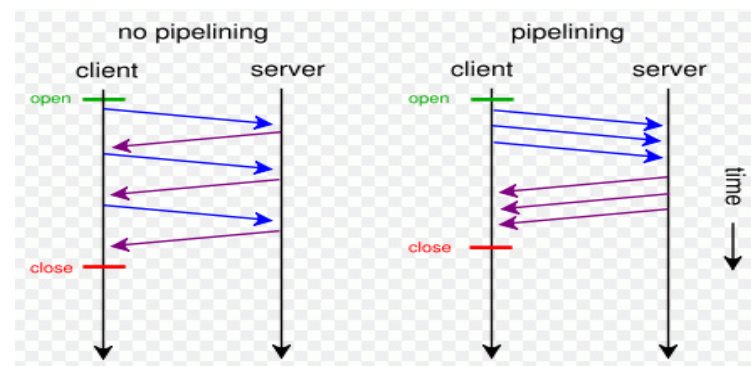# Persistent HTTP (HTTP/1.1)

Persistent HTTP

- ❖ server leaves TCP connection open after sending response
- ❖ subsequent HTTP messages between same client/server are sent over the same TCP connection
- ❖ Allow TCP to learn more accurate RTT estimate (APPARENT LATER IN THE COURSE)
- ❖ Allow TCP congestion window to increase (APPARENT LATER)
- ❖ i.e., leverage previously discovered bandwidth (APPARENT LATER)

Persistent without pipelining:

- ❖ client issues new request only when previous response has been received
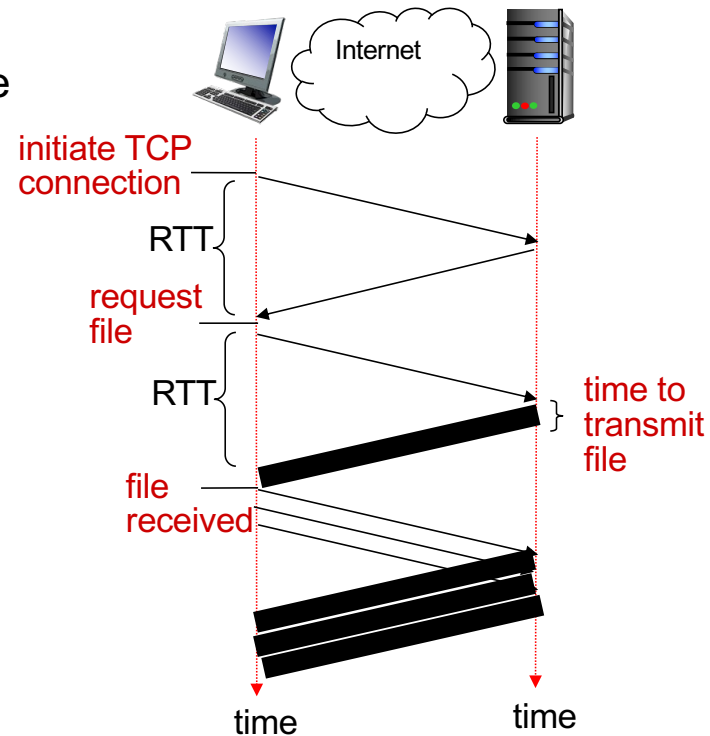- ❖ one RTT for each referenced object

Persistent with pipelining:

- ❖ introduced in HTTP/1.1
- ❖ client sends requests as soon as it encounters a referenced object
- ❖ as little as one RTT for all the referenced objects

HTTP 1.1: response time with pipelining

Website with one
index page and three
embedded objects

How to improve PLT

➢ Reduce content size for transfer

  • Smaller images, compression

➢ Change HTTP to make better use of available bandwidth

  • Persistent connections and pipelining

➢ Change HTTP to avoid repeated transfers of the same content

  • Caching and web-proxies

➢ Move content closer to the client

  • CDNs

Improving HTTP Performance: Caching

➢ **Why does caching work?**
- Exploit *locality of reference*

➢ **How well does caching work?**
- Very well, up to a limit
- Large overlap in content
- But many unique requests

➢ **Trend: increase in dynamic content**
- For example, customization of web pages
- Reduces benefits of caching
- Some exceptions, for example, video content

Web caches (proxy servers)

Goal: satisfy client request without involving origin server

- user configures browser to point to a *Web cache*
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client
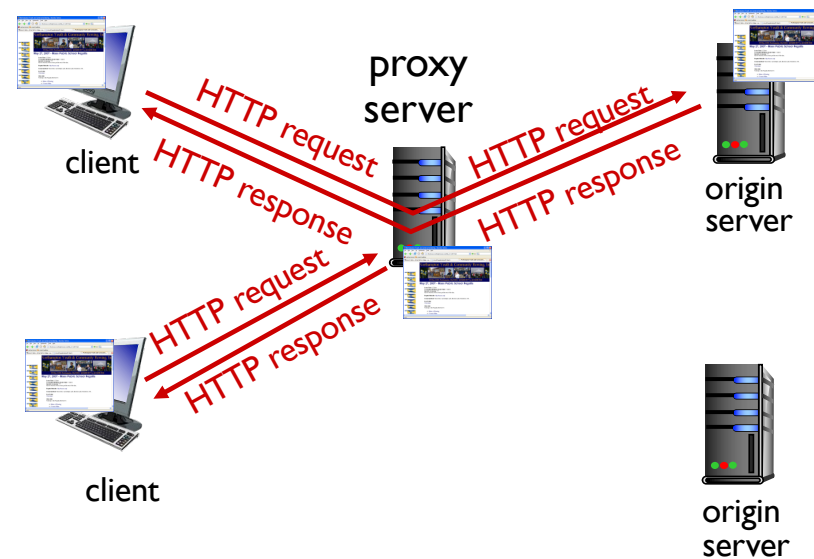
Web caches (proxy servers)

- Web 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
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables "poor" content providers to more effectively deliver content
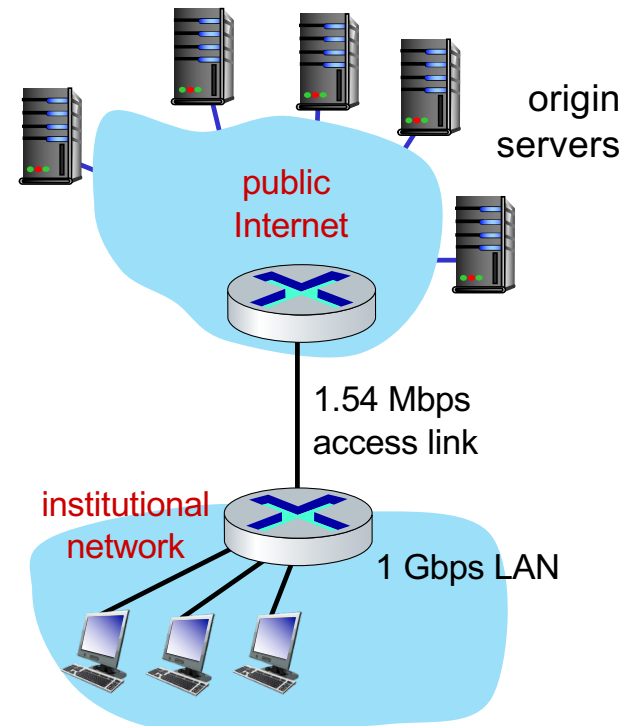
Caching example

Scenario:

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

Performance:

- LAN utilization: .0015
- access link utilization = .97
- end-end delay = Internet delay +
                        access link delay + LAN delay
                   = 2 sec + minutes + usecs

problem: large delays at high utilization!

origin servers

public Internet

1.54 Mbps access link

institutional network
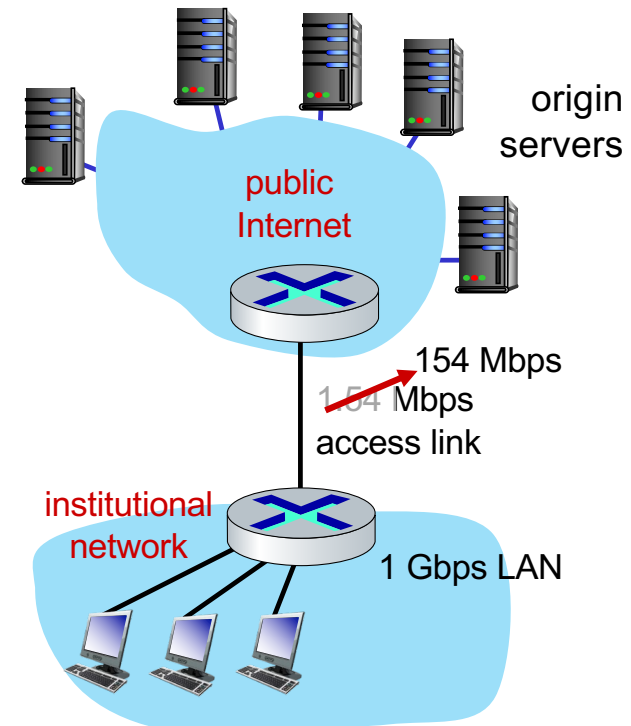
1 Gbps LAN

Caching example: buy a faster access link

Scenario:

154 Mbps

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

Performance:

- LAN utilization: .0015
- access link utilization = .97 → .0097
- end-end delay = Internet delay +
                    access link delay + LAN delay

  = 2 sec + minutes + usecs

Cost: faster access link (expensive!)   msecs

origin servers

public Internet

154 Mbps
1.54 Mbps
access link

institutional network

1 Gbps LAN

55

Caching example: install a web cache

Scenario:

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

Performance:

- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?

How to compute link utilization, delay?

Cost: web cache (cheap!)

origin servers

public Internet

1.54 Mbps access link

institutional network
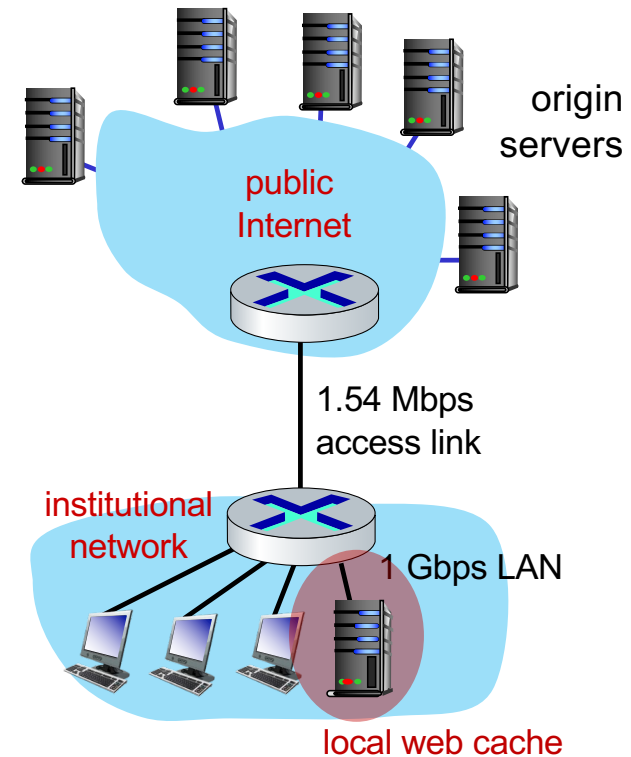
1 Gbps LAN

local web cache
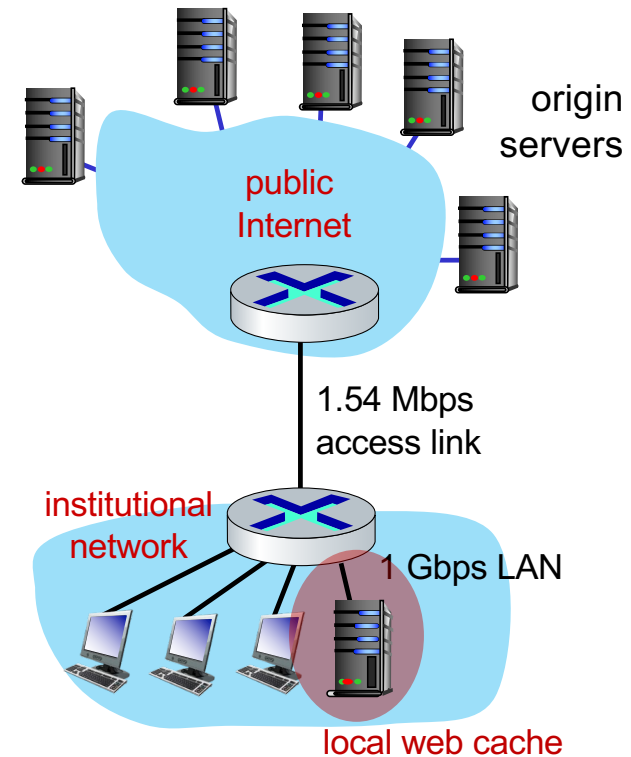
Caching example: install a web cache

Calculating access link utilization, end-end delay with cache:

- suppose cache hit rate is 0.4: 40% requests satisfied at cache; 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link
  = 0.6 * 1.50 Mbps = .9 Mbps
- utilization = 0.9/1.54 = .58

- average end-end delay
  = 0.6 * (delay from origin servers)
    + 0.4 * (delay when satisfied at cache)
  = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs
    approximation

lower average end-end delay than with 154 Mbps link (and cheaper too!)

origin servers

public Internet

1.54 Mbps access link

institutional network

1 Gbps LAN

local web cache

Conditional GET

Goal: don't send object if cache has up-to-date cached version
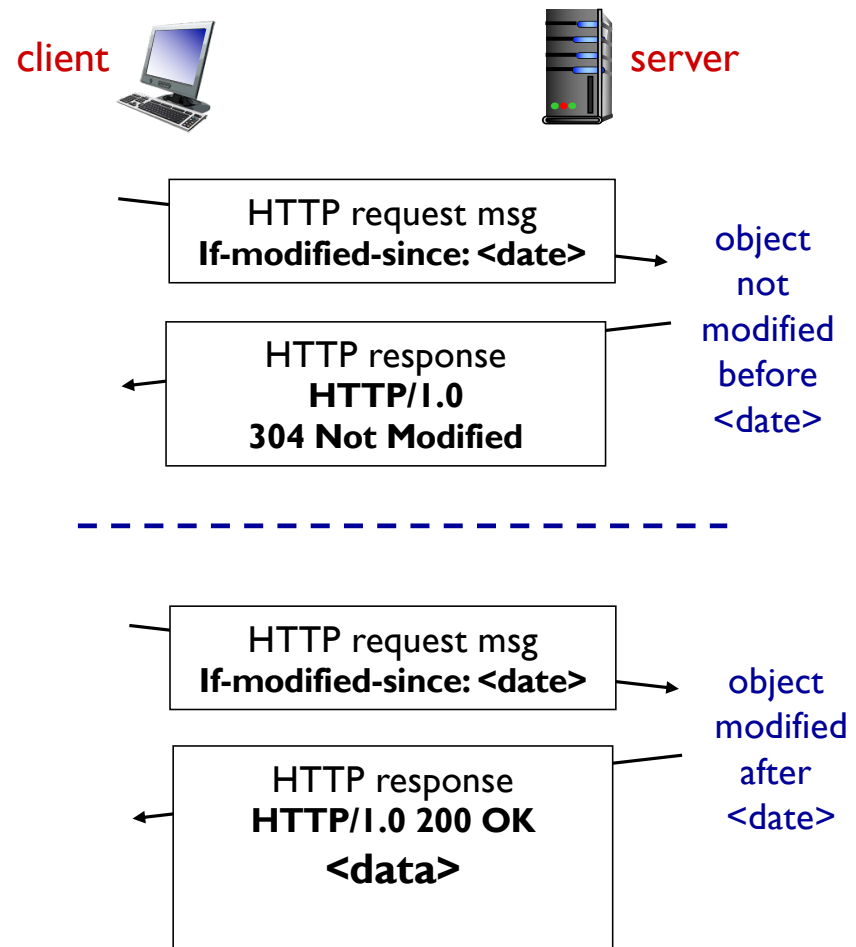 - no object transmission delay
  - lower link utilization

- *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
HTTP response   modified
HTTP/1.0   before
304 Not Modified   <date>

- -

HTTP request msg
If-modified-since: <date>   object
                                            modified
HTTP response   after
HTTP/1.0 200 OK   <date>
<data>

Example Cache Check Request

```
GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
If-Modified-Since: Mon, 29 Jan 2001 17:54:18 GMT
If-None-Match: "7a11f-10ed-3a75ae4a"
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT
    5.0)
Host: www.intel-iris.net
Connection: Keep-Alive
```

Example Cache Check Response

HTTP/1.1 304 Not Modified

Date: Tue, 27 Mar 2001 03:50:51 GMT

Server: Apache/1.3.14 (Unix) (Red-Hat/Linux) mod_ssl/2.7.1
   OpenSSL/0.9.5a DAV/1.0.2 PHP/4.0.1pl2 mod_perl/1.24

Connection: Keep-Alive

Keep-Alive: timeout=15, max=100

ETag: "7a11f-10ed-3a75ae4a"

Etag: Usually used for dynamic content. The value is often a
cryptographic hash of the content.

Improving HTTP Performance: Replication

➢ Replicate popular Web site across many machines
  • Spreads load on servers
  • Places content closer to clients
  • Helps when content isn't cacheable

➢ Problem:
  • Want to direct client to a particular replica
    • Balance load across server replicas
    • Pair clients with nearby servers
  • Expensive

➢ Common solution:
  • DNS returns different addresses based on client's geo-location, server load, *etc.*

More on this later

Improving HTTP Performance: CDN

- ➢ Caching and replication as a service
- ➢ Large-scale distributed storage infrastructure (usually) administered by one entity
  - e.g., Akamai has servers in 20,000+ locations
- ➢ Combination of (pull) caching and (push) replication
  - **Pull:** Direct result of clients' requests
  - **Push:** Expectation of high access rate
- ➢ Also do some processing
  - Handle dynamic web pages
  - Transcoding

What about HTTPS?

➤ HTTP is insecure

➤ HTTP basic authentication: password sent using base64 encoding (can be readily converted to plaintext)

➤ HTTPS: HTTP over a connection encrypted by Transport Layer Security (TLS)

➤ Provides:
  • Authentication
  • Bidirectional encryption

➤ Widely used in place of plain vanilla HTTP

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection

- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (head-of-line (HOL) blocking) behind large object(s)
- loss recovery (retransmitting lost TCP segments) stalls object transmission

HTTP/2: mitigating HOL blocking

HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)
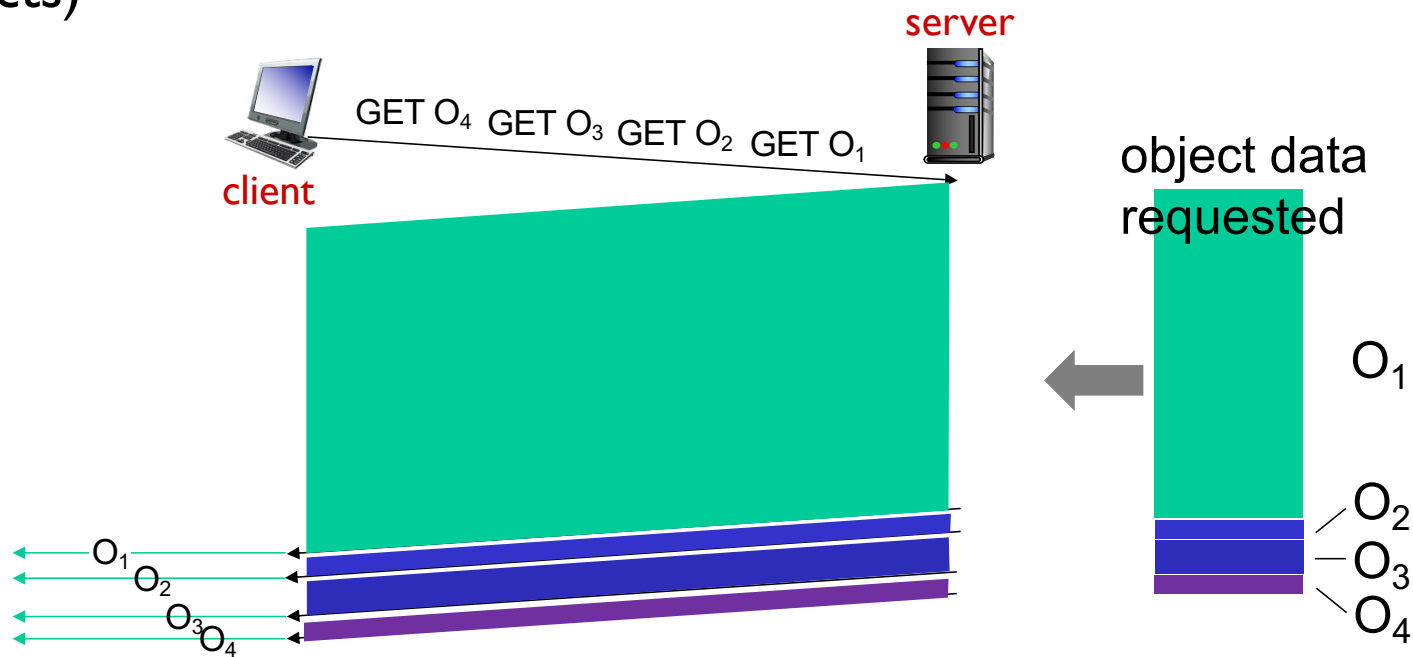
objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:

- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved

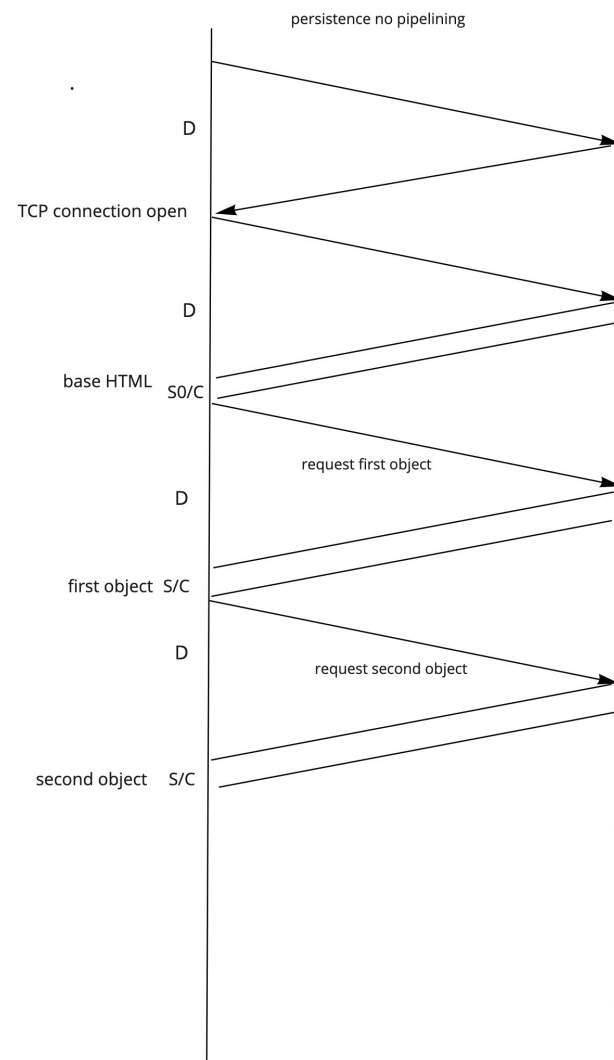O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

Quiz: HTTP (1)

Consider an HTML page with a base file of size S_0 bits and N inline objects each of size S bits. Assume a client fetching the page across a link of capacity C bits/s and RTT of D. How long does it take to download the page using **non-persistent HTTP (without parallelism)**?

A. $D + (S_0 + NS)/C$

B. $2D + (S_0 + NS)/C$

C. $N(D + S/C)$

D. $2D + S_0/C + N(2D + S/C)$

E. $2D + S_0/C + N(D + S/C)$

non-persistence

.

D

TCP connection open

D

base HTML   S0/C
TCP connection close

D

TCP connection open

request first object

D

S/C
first object
TCP connection close

D

TCP connection open

request second object

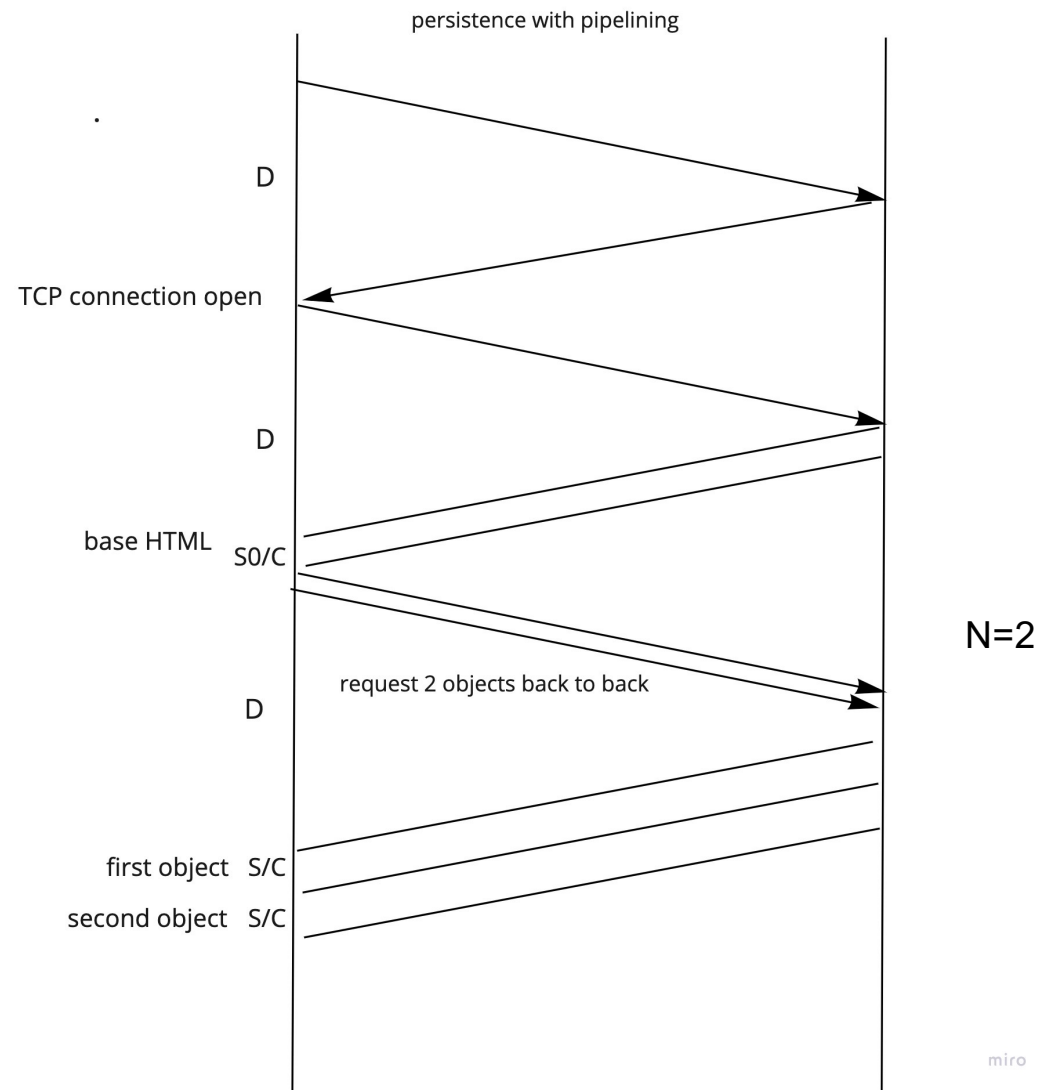D

second object   S/C

TCP connection close

N=2

miro
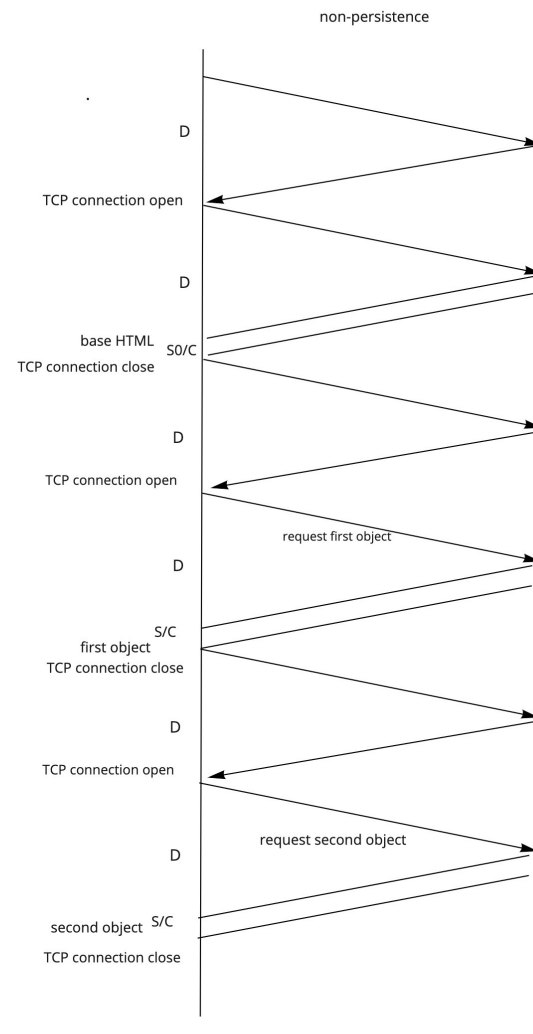
Quiz: HTTP (2)

Consider an HTML page with a base file of size S_0 bits and N inline objects each of size S bits. Assume a client fetching the page across a link of capacity C bits/s and RTT of D. How long does it take to download the page using **persistent HTTP (without parallelism or pipelining)**?

A. $2D + (S_0 + NS)/C$

B. $3D + (S_0 + NS)/C$

C. $N(D + S/C)$

D. $2D + S_0/C + N(2D + S/C)$

E. $2D + S_0/C + N(D + S/C)$

persistence no pipelining

.

D

TCP connection open

D

base HTML   S0/C

request first object

D

first object S/C

D   request second object

second object S/C

N=2

Quiz: HTTP (3)

Consider an HTML page with a base file of size S_0 bits and N inline objects each of size S bits. Assume a client fetching the page across a link of capacity C bits/s and RTT of D. How long does it take to download the page using **persistent HTTP with pipelining**?

A. $2D + (S_0 + NS)/C$

B. $4D + (S_0 + NS)/C$

C. $N(D + S/C)$

D. $3D + S_0/C + NS/C$

E. $2D + S_0/C + N(D + S/C)$

persistence with pipelining

.

D

TCP connection open

D

base HTML   S0/C

N=2

request 2 objects back to back

D

first object   S/C

second object   S/C
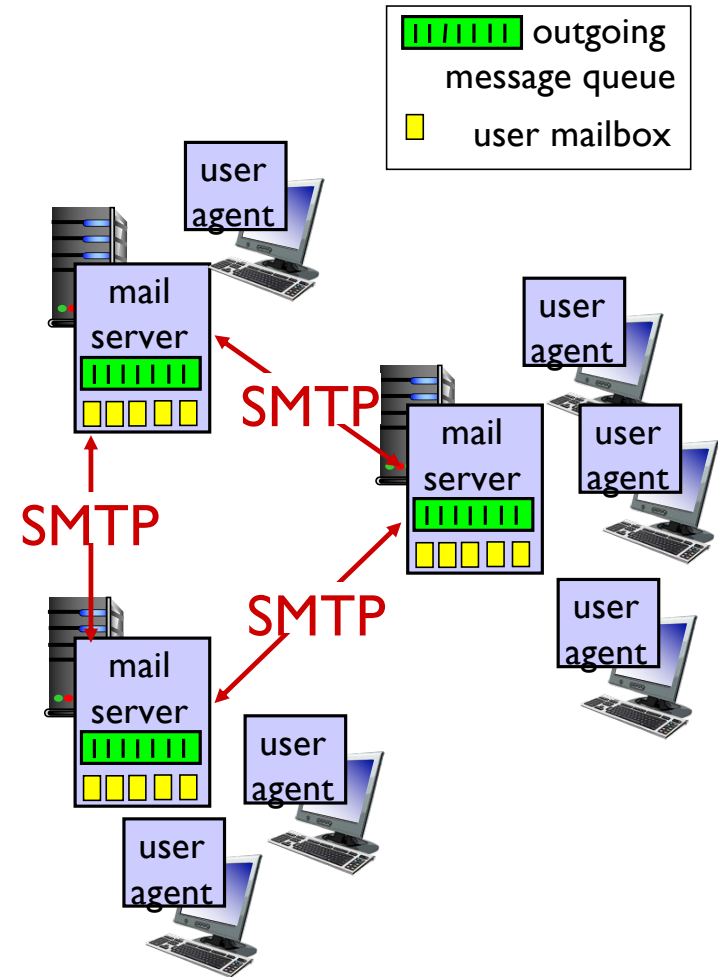
miro

73

2. Application Layer: outline

E-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, iPhone mail client
- outgoing, incoming messages stored on server

E-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
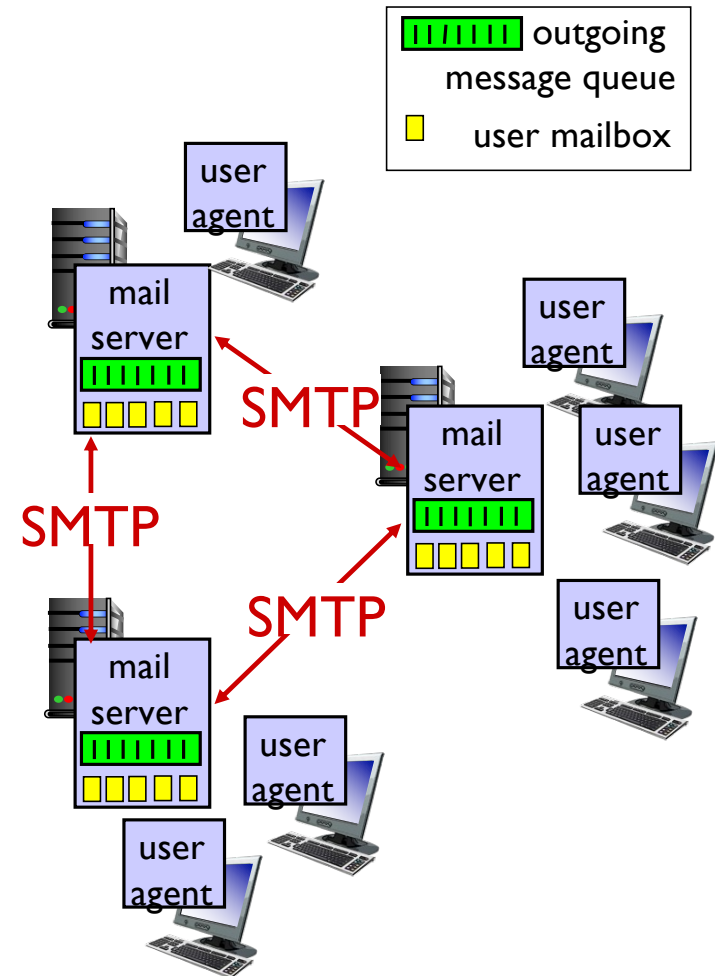
E-mail: the RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
- direct transfer: sending server (acting like client) 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 ASCI

Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail 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

Alice's mail server
Bob's mail server
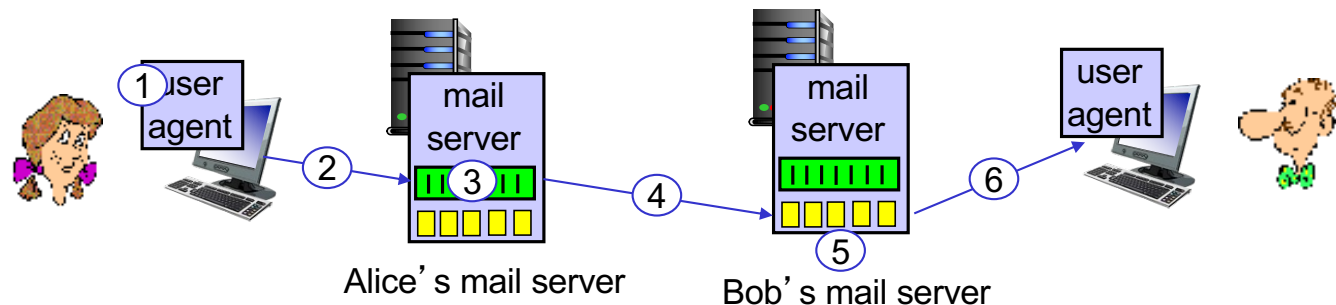
Sample SMTP interaction

```
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
```

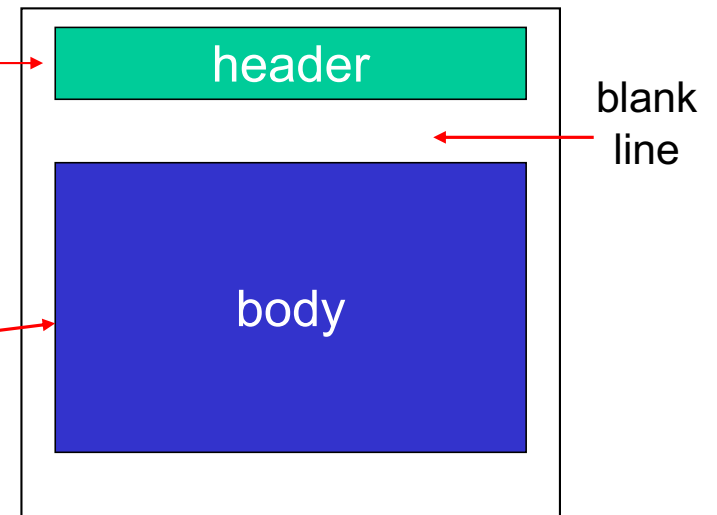# SMTP: closing observations

comparison with HTTP:

- HTTP: pull

- SMTP: push

- both have ASCII command/response interaction, status codes

- HTTP: each object encapsulated in its own response message

- SMTP: multiple objects sent in multipart message

- 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
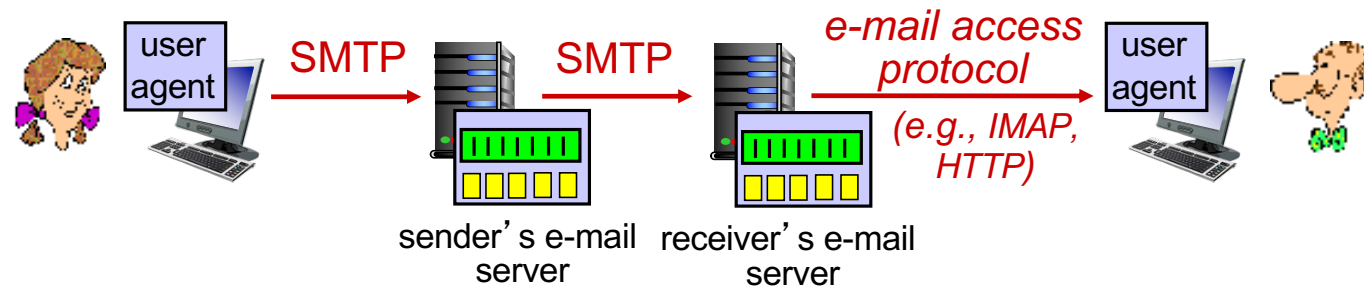
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

RFC 822 defines *syntax* for e-mail message itself (like HTML)

- header lines, e.g.,
  - To:
  - From:
  - Subject:

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!

- Body: the "message" , ASCII characters only

header

body

blank line

Mail access protocols

sender's e-mail server — SMTP — receiver's e-mail server — *e-mail access protocol (e.g., IMAP, HTTP)* — user agent

- **SMTP:** delivery/storage of e-mail messages to receiver's server

- mail access protocol: retrieval from server
  - **IMAP:** Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server

- **HTTP:** gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

Quiz: SMTP

Why do we have Sender's mail server?

- User agent can directly connect with recipient mail server without the need of sender's mail server? What's the catch?

Quiz: SMTP

Why do we have a separate Receiver's mail server?

➢ Can't the recipient run the mail server on own end system?

Summary

❖ **Application Layer (Chapter 2)**
  - Principles of Network Applications
  - HTTP
  - E-mail

❖ **Next:**
  - DNS
  - P2P