

# Lecture # 13

# 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
```

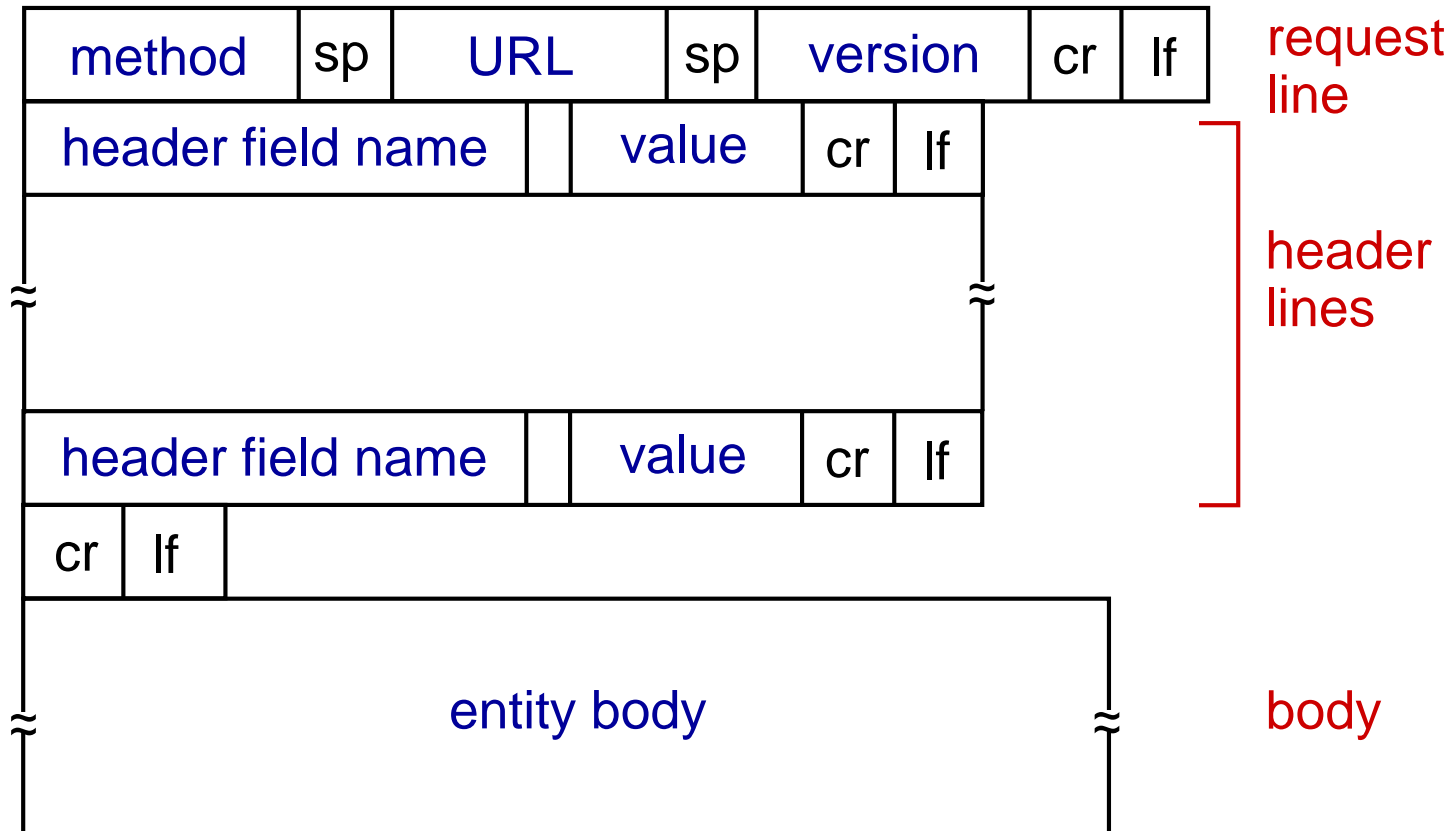
carriage return character

line-feed character

The diagram illustrates the structure of an HTTP request message. It shows a request line followed by several header lines, and then a final line consisting of a carriage return and a line feed. Blue arrows point from descriptive text labels to the corresponding parts of the message. One arrow points from 'request line (GET, POST, HEAD commands)' to the first line of the message. Another arrow points from 'header lines' to the block of lines between the request line and the final separator. A third arrow points from 'carriage return, line feed at start of line indicates end of header lines' to the final '\r\n' line. Two additional arrows point from 'carriage return character' and 'line-feed character' to the '\r' and '\n' characters respectively in the first line of the message.

\* Check out the online interactive exercises for more  
examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](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`

# The GET Method

**GET is used to request data from a specified resource.**

**GET is one of the most common HTTP methods.**

Note that the query string (name/value pairs) is sent in the URL of a GET request:

```
/test/demo_form.php?name1=value1&name2=value2
```

## **Some other notes on GET requests:**

- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests have length restrictions
- GET requests is only used to request data (not modify)

# The POST Method

**POST is used to send data to a server to create/update a resource.**

The data sent to the server with POST is stored in the request body of the HTTP request:

```
POST /test/demo_form.php HTTP/1.1  
Host: w3schools.com  
name1=value1&name2=value2
```

**POST is one of the most common HTTP methods.**

**Some other notes on POST requests:**

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

# 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://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**

# Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

`telnet gaia.cs.umass.edu 80` { opens TCP connection to port 80  
(default HTTP server port)  
at gaia.cs.umass.edu.  
anything typed in will be sent  
to port 80 at gaia.cs.umass.edu

2. type in a GET HTTP request:

`GET /kurose_ross/interactive/index.php HTTP/1.1`  
`Host: gaia.cs.umass.edu` { by typing this in (hit carriage  
return twice), you send  
this minimal (but complete)  
GET request to HTTP server

3. look at response message sent by HTTP server!  
(or use Wireshark to look at captured HTTP request/response)

# 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 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

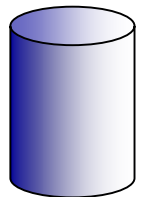
usual http request msg

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

access

cookie-  
specific  
action

one week later:



ebay 8734  
amazon 1678

usual http request msg  
**cookie: 1678**

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

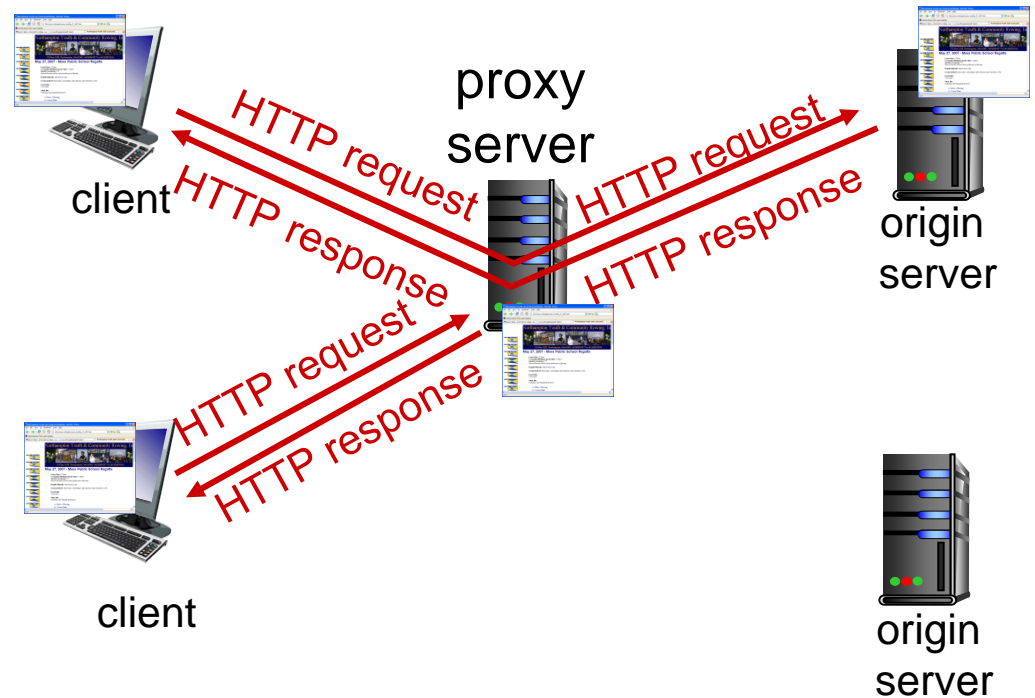
*how to keep “state”:*

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

# 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 returns object
  - else cache requests object from origin server, then returns object to client



# Lecture # 14

# Lecture # 15



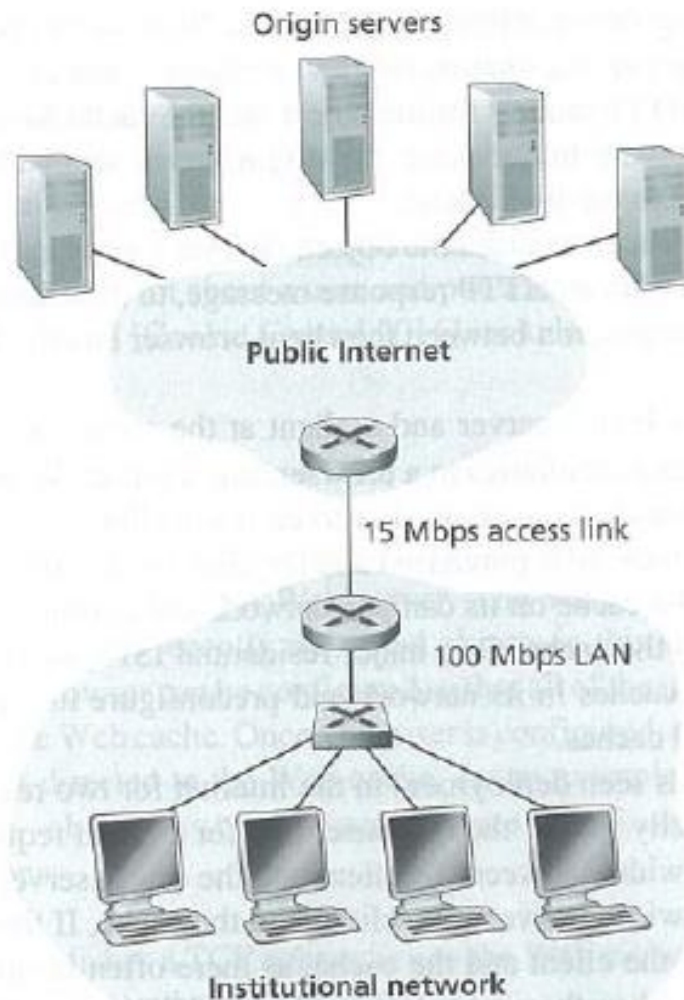
# More about Web caching

- 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)

ple in the context of Figure 2.12. This figure shows two networks—the institutional network and the rest of the public Internet. The institutional network is a high-speed LAN. A router in the institutional network and a router in the Internet are connected by a 15 Mbps link. The origin servers are attached to the Internet but are located all over the globe. Suppose that the average object size is 1 Mbits and that the average request rate from the institution's browsers to the origin servers is 15 requests per second. Suppose that the HTTP request messages are negligibly small and thus create no traffic in the networks or in the access link (from institutional router to Internet router). Also suppose that the amount of time it takes from when the router on the Internet side of the access link in Figure 2.12 forwards an HTTP request (within an IP datagram) until it receives the response (typically within many IP datagrams) is two seconds on average. Informally, we refer to this last delay as the “Internet delay.”



**Figure 2.12** ♦ Bottleneck between an institutional network and the Internet

The total response time—that is, the time from the browser's request of an object until its receipt of the object—is the sum of the LAN delay, the access delay (that is, the delay between the two routers), and the Internet delay. Let's now do a very crude calculation to estimate this delay. The traffic intensity on the LAN (see Section 1.4.2) is

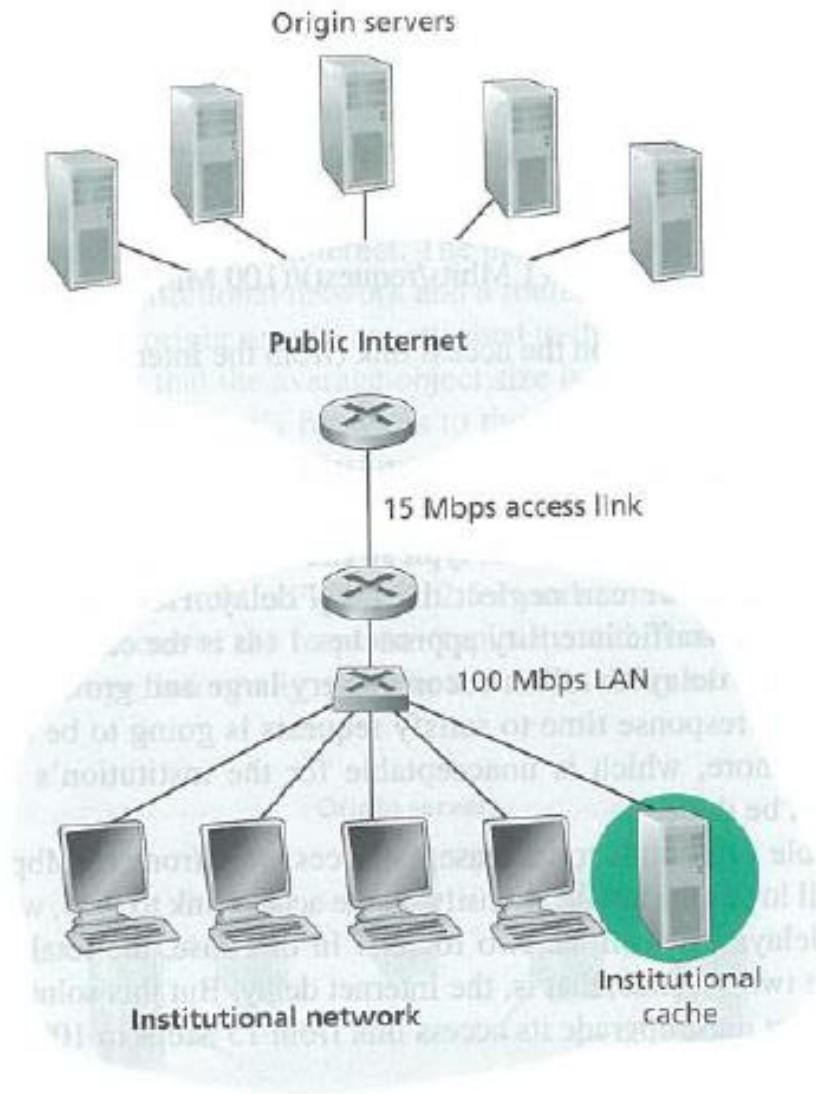
$$(15 \text{ requests/sec}) \cdot (1 \text{ Mbits/request}) / (100 \text{ Mbps}) = 0.15$$

whereas the traffic intensity on the access link (from the Internet router to institution router) is

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

A traffic intensity of 0.15 on a LAN typically results in, at most, tens of milliseconds of delay; hence, we can neglect the LAN delay. However, as discussed in Section 1.4.2, as the traffic intensity approaches 1 (as is the case of the access link in Figure 2.12), the delay on a link becomes very large and grows without bound. Thus, the average response time to satisfy requests is going to be on the order of minutes, if not more, which is unacceptable for the institution's users. Clearly something must be done.

One possible solution is to increase the access rate from 15 Mbps to, say, 100 Mbps. This will lower the traffic intensity on the access link to 0.15, which translates to negligible delays between the two routers. In this case, the total response time will roughly be two seconds, that is, the Internet delay. But this solution also means that the institution must upgrade its access link from 15 Mbps to 100 Mbps, a costly proposition.



**Figure 2.13** ♦ Adding a cache to the institutional network



Now consider the alternative solution of not upgrading the access link but instead installing a Web cache in the institutional network. This solution is illustrated in Figure 2.13. Hit rates—the fraction of requests that are satisfied by a cache—typically range from 0.2 to 0.7 in practice. For illustrative purposes, let's suppose that the cache provides a hit rate of 0.4 for this institution. Because the clients and the cache are connected to the same high-speed LAN, 40 percent of the requests will be satisfied almost immediately, say, within 10 milliseconds, by the cache. Nevertheless, the remaining 60 percent of the requests still need to be satisfied by the origin servers. But with only 60 percent of the requested objects passing through the access link, the traffic intensity on the access link is reduced from 1.0 to 0.6. Typically, a traffic intensity less than 0.8 corresponds to a small delay, say, tens of milliseconds, on a 15 Mbps link. This delay is negligible compared with the two-second Internet delay. Given these considerations, average delay therefore is

$$0.4 \cdot (0.01 \text{ seconds}) + 0.6 \cdot (2.01 \text{ seconds})$$

which is just slightly greater than 1.2 seconds. Thus, this second solution provides an even lower response time than the first solution, and it doesn't require the institution

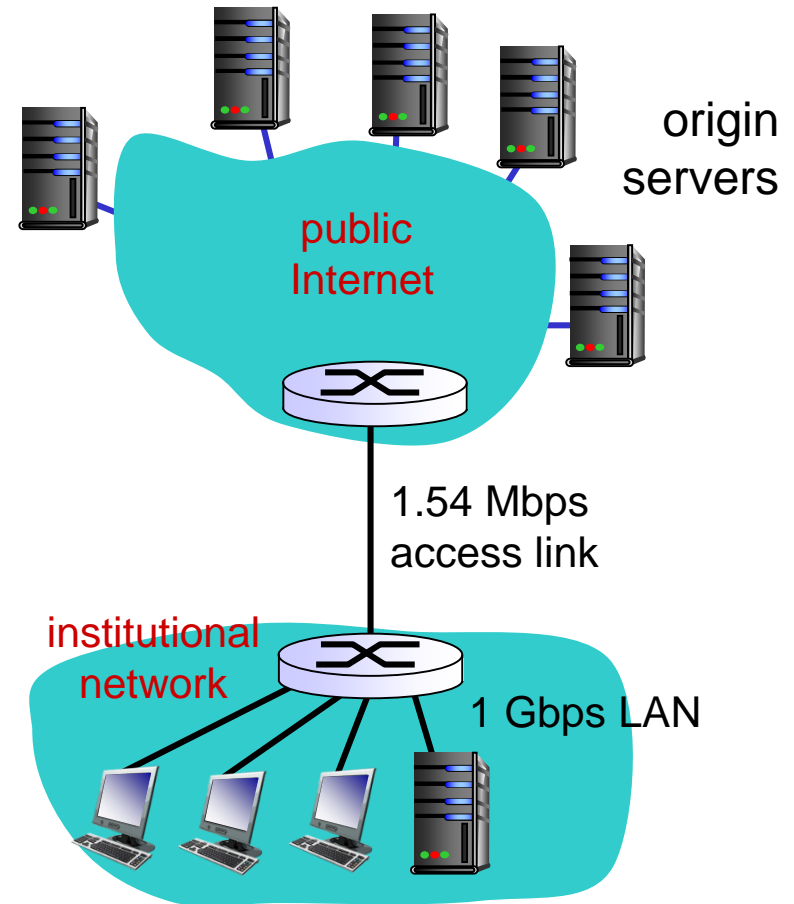
# Caching example:

## *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: 15%
- access link utilization = **99%** *problem!*
- total delay = Internet delay + access delay + LAN delay  
= 2 sec + minutes + usecs





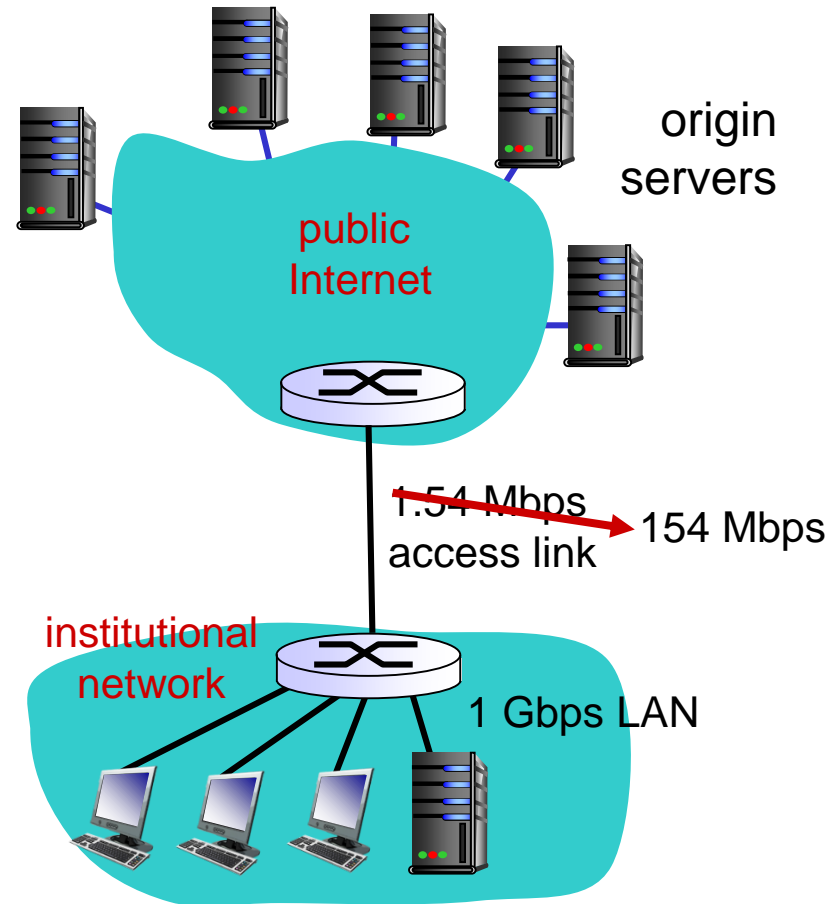
# Caching example: fatter access link

## 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~~ → 154 Mbps

## consequences:

- LAN utilization: 15%
- access link utilization = ~~99%~~ → 9.9%
- total delay = Internet delay + access delay + LAN delay  
= 2 sec + ~~minutes~~ → msecs



**Cost:** increased access link speed (not cheap!)

# Caching example: install local cache

## *assumptions:*

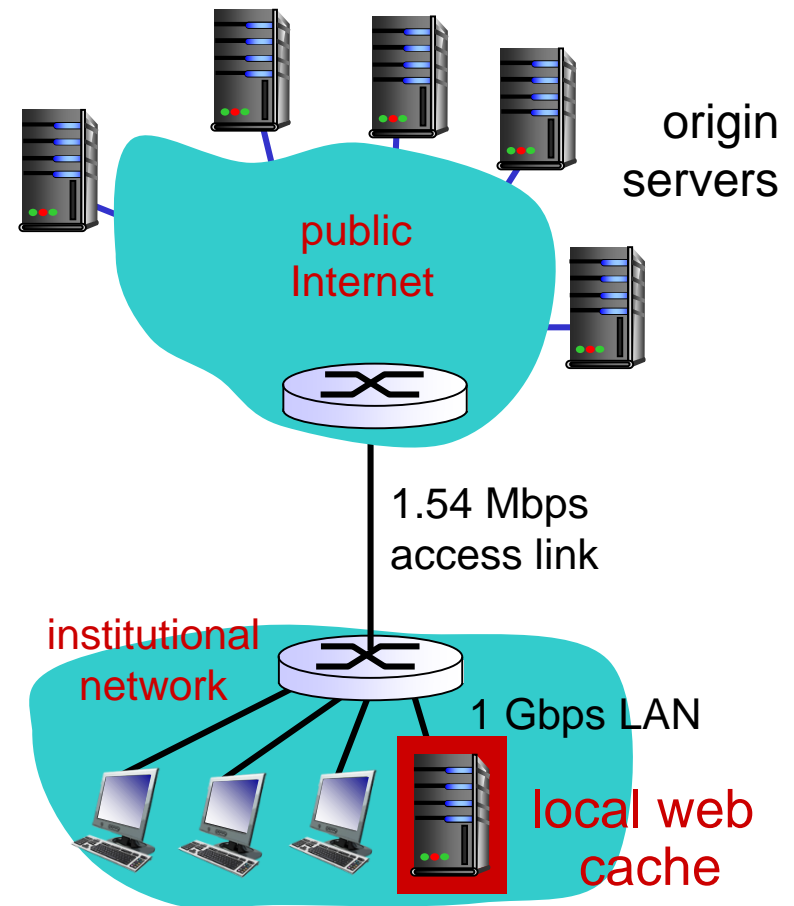
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

## *consequences:*

- LAN utilization: 15%
- access link utilization = ?
- total delay = ?

*How to compute link utilization, delay?*

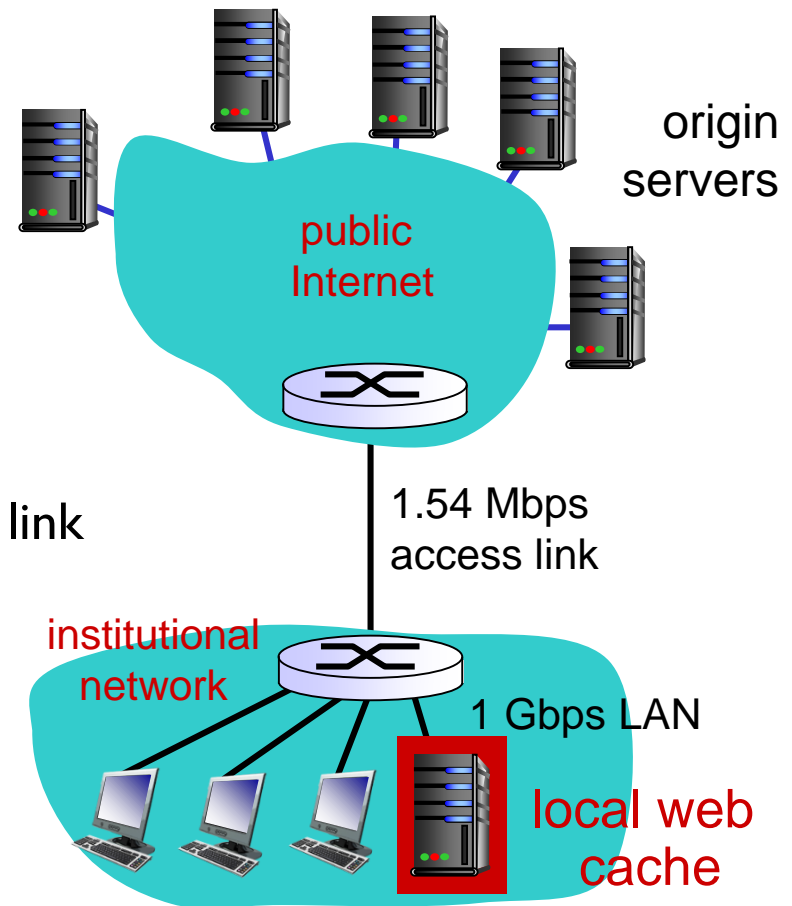
*Cost:* web cache (cheap!)



# 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
- access link utilization:
  - 60% of requests use access link
- data rate to browsers over access link  
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$ 
  - utilization  $= 0.9 / 1.54 = .58$
- total delay
  - $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
  - $= 0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs}$
  - less than with 154 Mbps link (and cheaper too!)



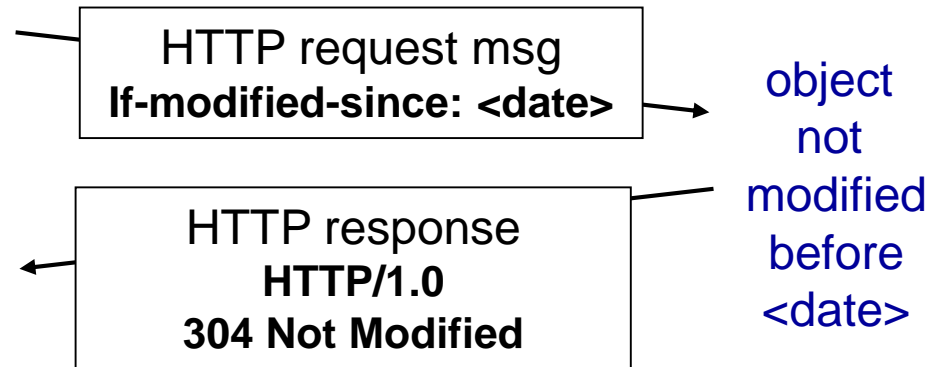
# Conditional GET

- **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

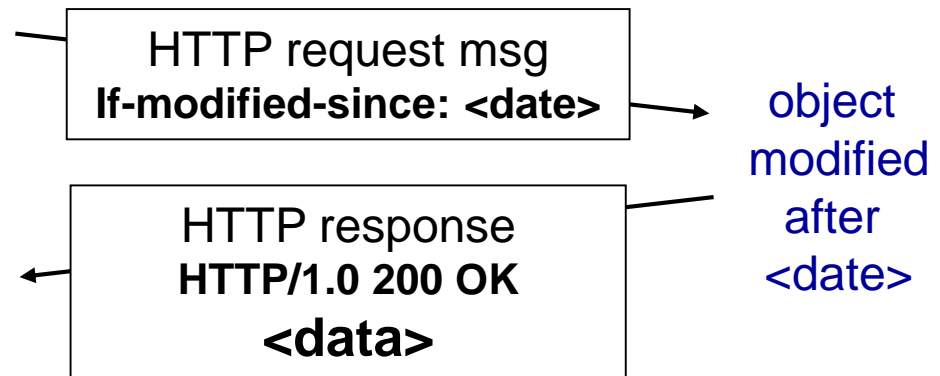
client



server



-----



# Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

# DNS: domain name system

*people*: many identifiers:

- SSN, name, passport #

*Internet hosts, routers*:

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., `www.yahoo.com` - used by humans

Q: how to map between IP address and name, and vice versa ?

## *Domain Name System:*

- *distributed database*  
implemented in hierarchy of many *name servers*
- *application-layer protocol*: hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's “edge”

# DNS: services, structure

## *DNS services*

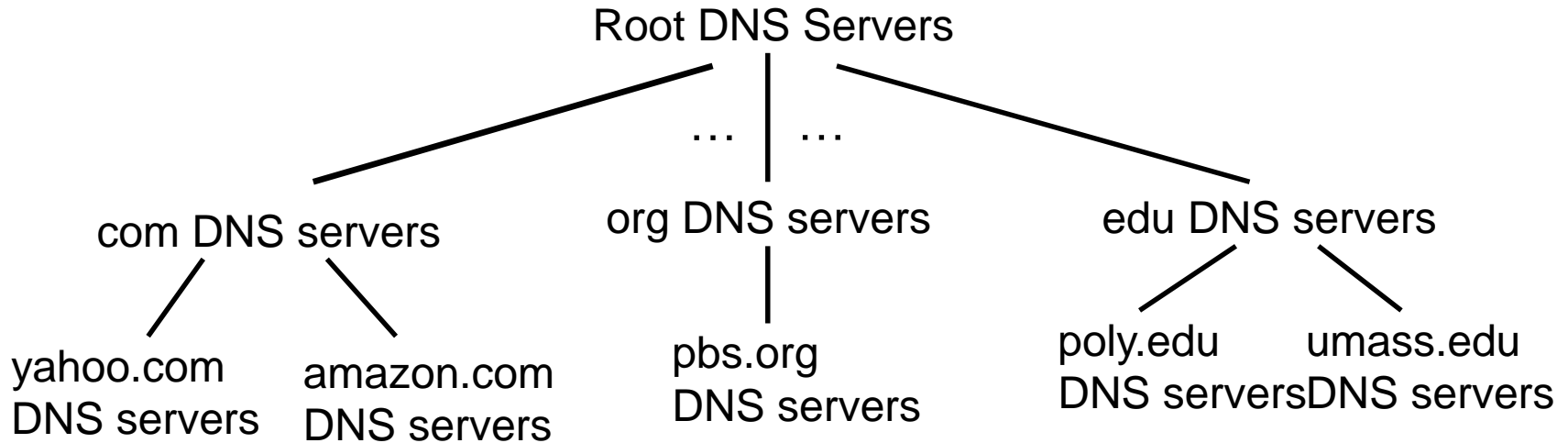
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

## *why not centralize DNS?*

- single point of failure
- traffic volume
- distant centralized database
- maintenance

*A: doesn't scale!*

# DNS: a distributed, hierarchical database



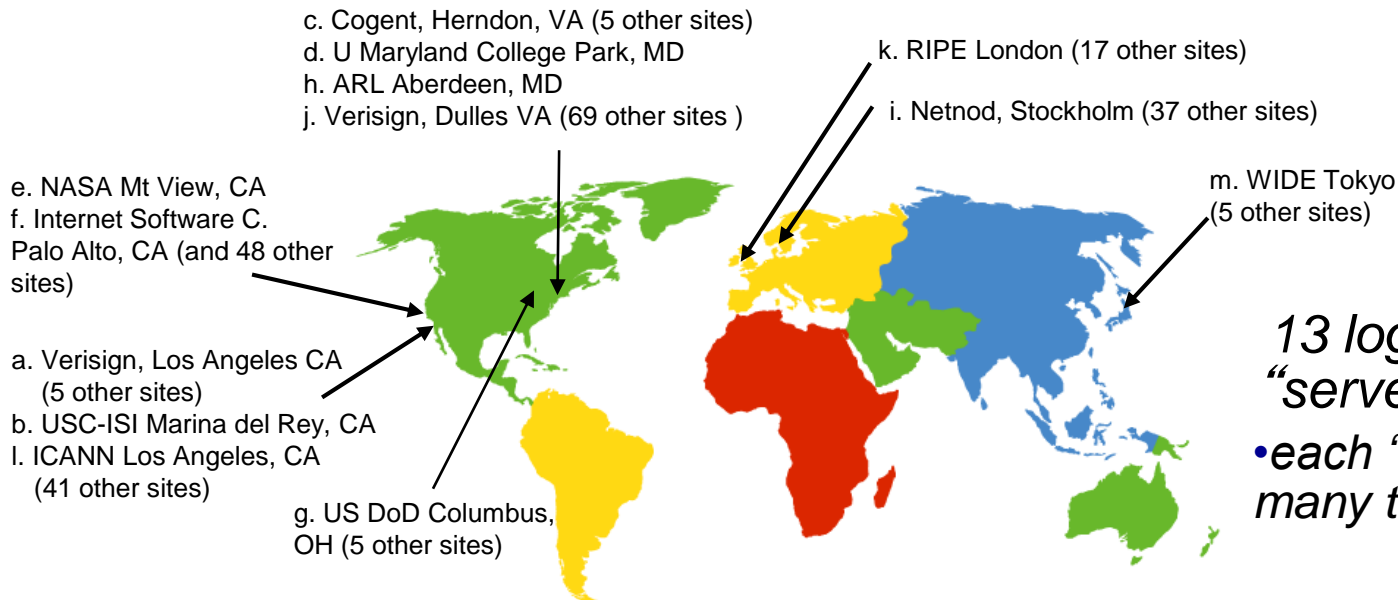
*client wants IP for www.amazon.com; 1<sup>st</sup> approximation:*

- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com



# DNS: root name servers

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server



*13 logical root name  
“servers” worldwide*  
• *each “server” replicated  
many times*

# TLD, authoritative servers

## *top-level domain (TLD) servers:*

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

## *authoritative DNS servers:*

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

# Local DNS name server

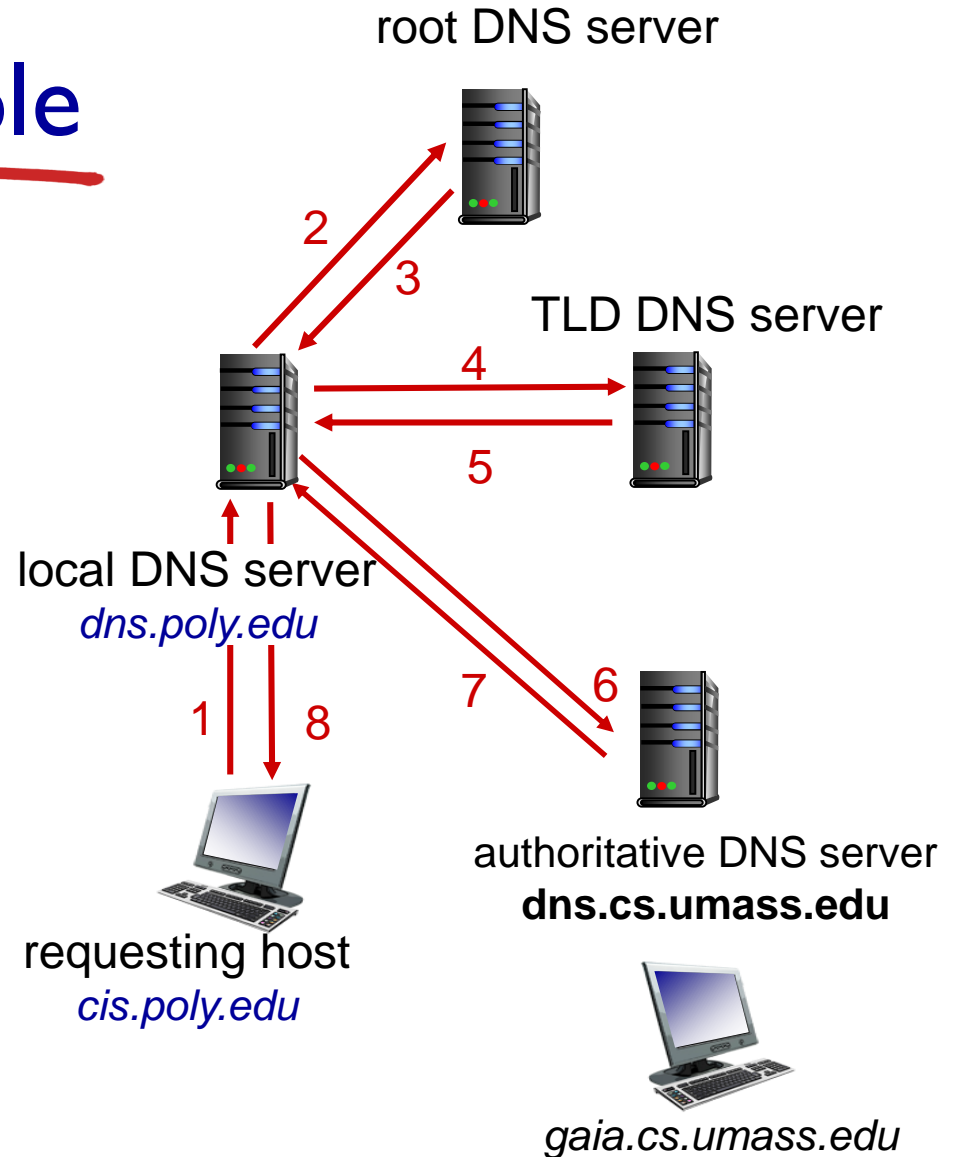
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

# DNS name resolution example

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

## *iterated query:*

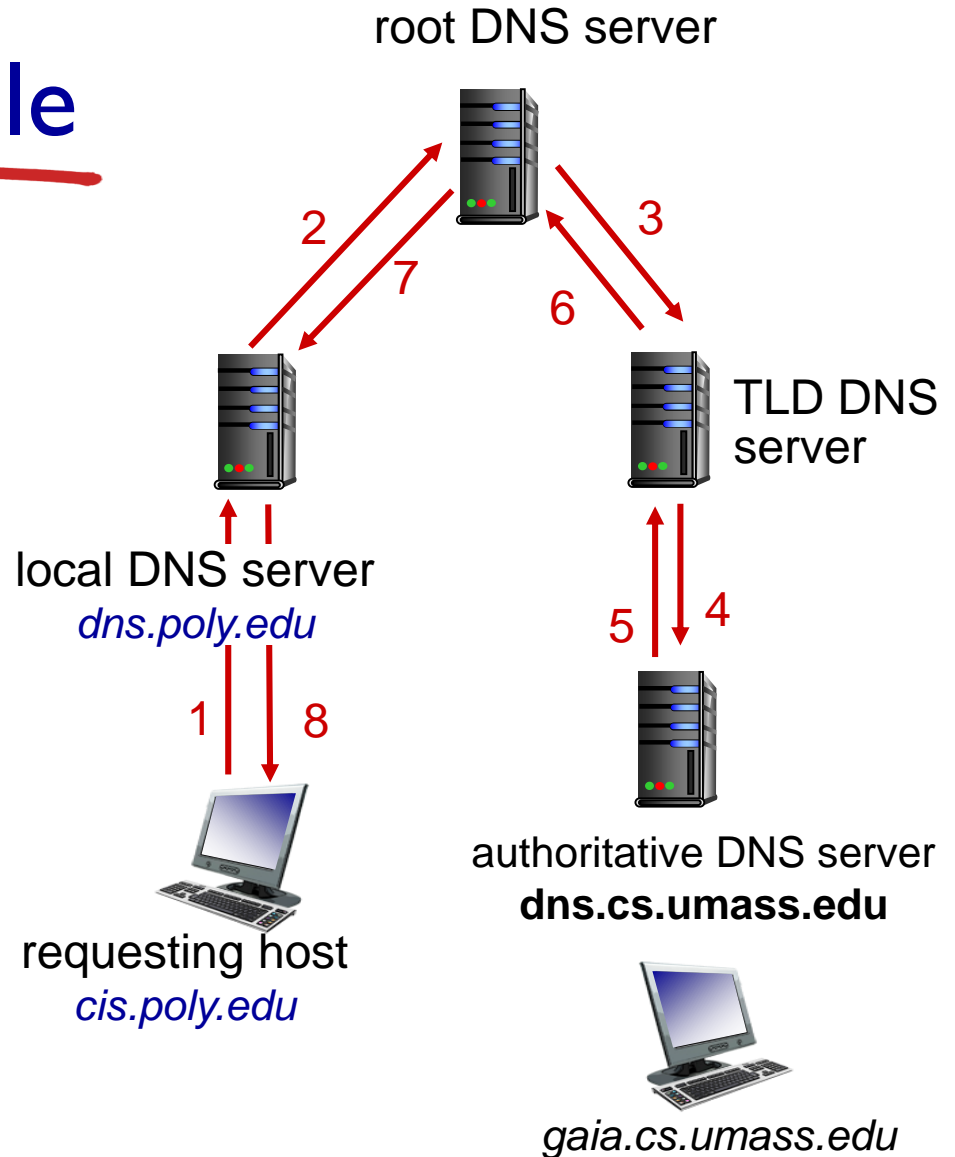
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



# DNS name resolution example

## *recursive query:*

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



# DNS: caching, updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
  - RFC 2136