西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

# Computer Networks

Lecturer: ZHANG Ying

Fall semester 2022

# Chapter 2
# Application Layer

# Chapter Outline

# HTTP connections

*non-persistent HTTP*

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

*persistent HTTP*

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

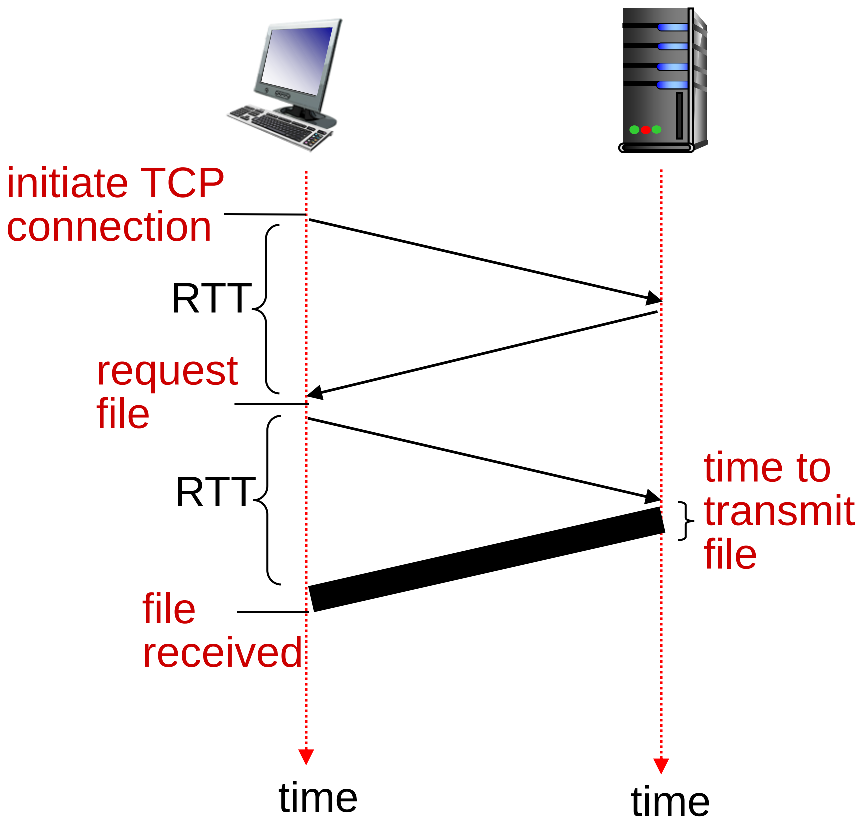
# Non-persistent HTTP: response time

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

HTTP response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2RTT+ file transmission time



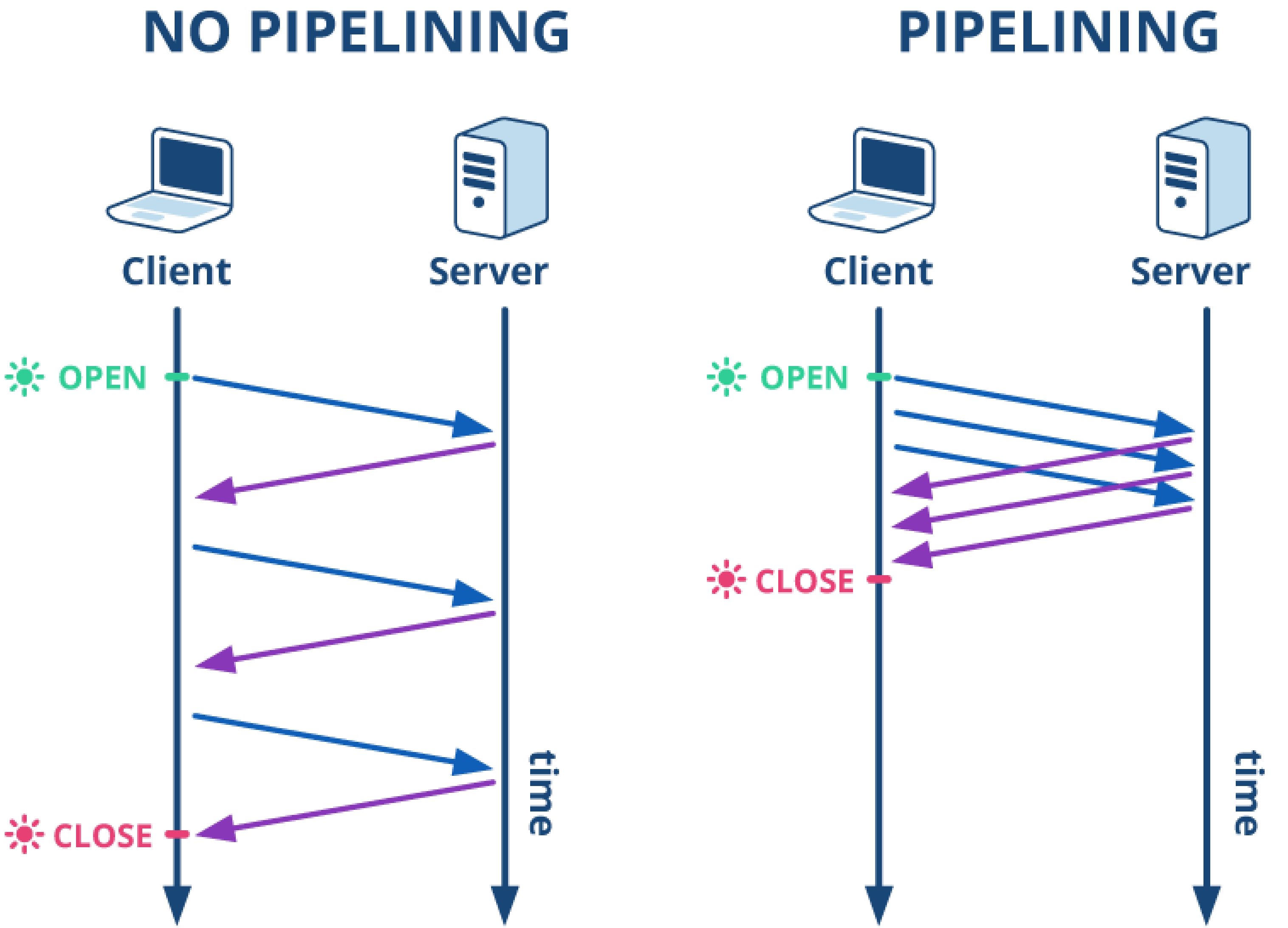
# non-persistent HTTP issues

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

# Persistent HTTP

*persistent HTTP:*

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


NO PIPELINING
PIPELINING
Client
Server
Client
Server
OPEN
CLOSE
OPEN
CLOSE
time
time

# HTTP request message

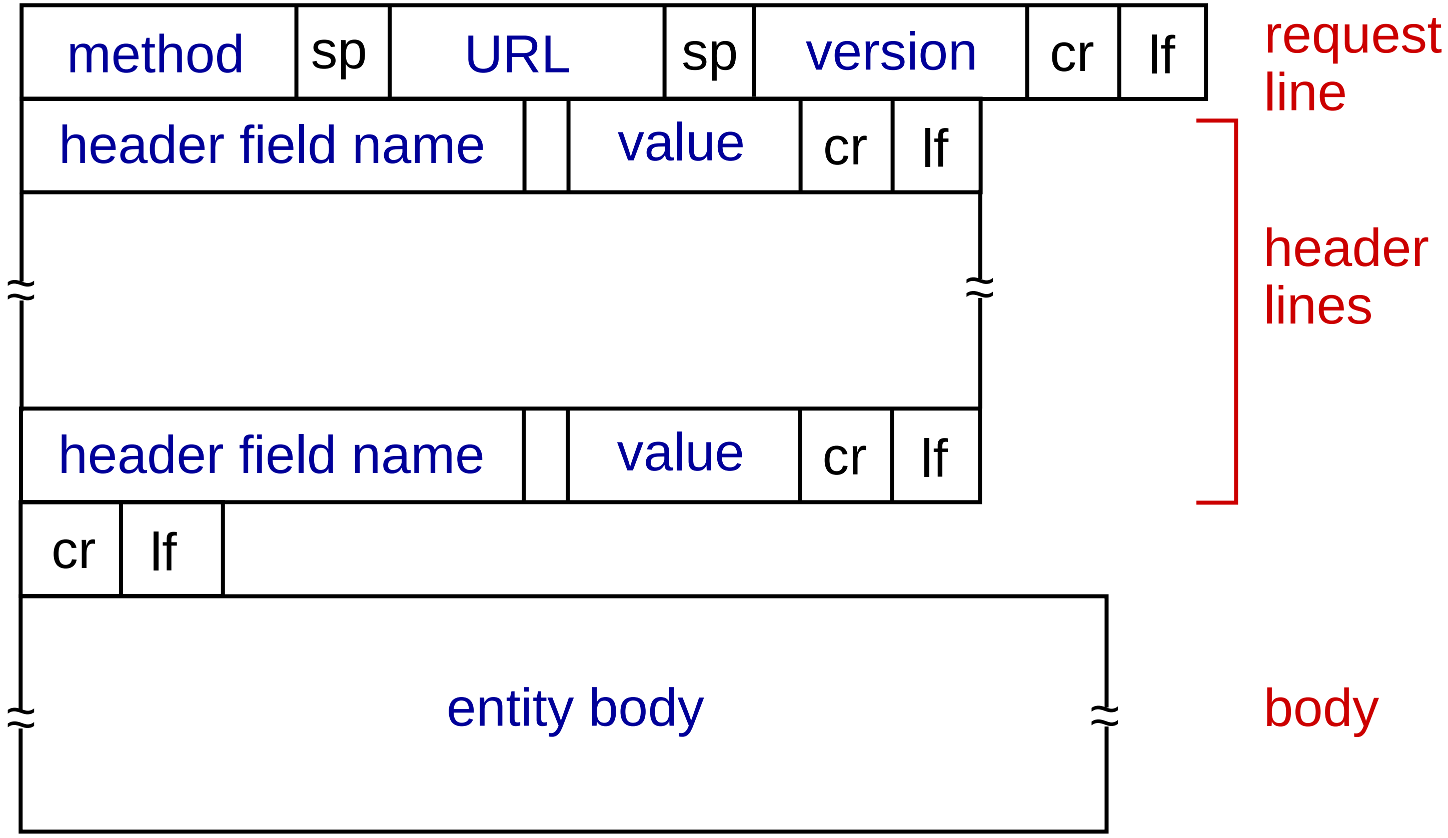
- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)

request line
(GET, POST, HEAD commands)

```
GET /somedir/page.html HTTP/1.1
Connection: close
User-agent: Mozilla/5.0
Accept: text/html, image/gif, image/jpeg
Accept-language:fr
```

header
lines

# HTTP request message: general format




**HTTP Request Methods**
**GET, POST, PUT, DELETE**

# 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 Example

status line (protocol status code status phrase)

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 2018 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 2018 09:23:24 GMT
Content-Length: 6821
Content-Type: text/html
```

*data data data data data ...*

header lines

data, e.g., requested HTML file

# HTTP response status codes

- status code indicates whether a specific HTTP request has been successfully completed

1. Informational responses ( 100 – 199 )
2. Successful responses ( 200 – 299 )
3. Redirection messages ( 300 – 399 )
4. Client error responses ( 400 – 499 )
5. Server error responses ( 500 – 599 )

# 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`**

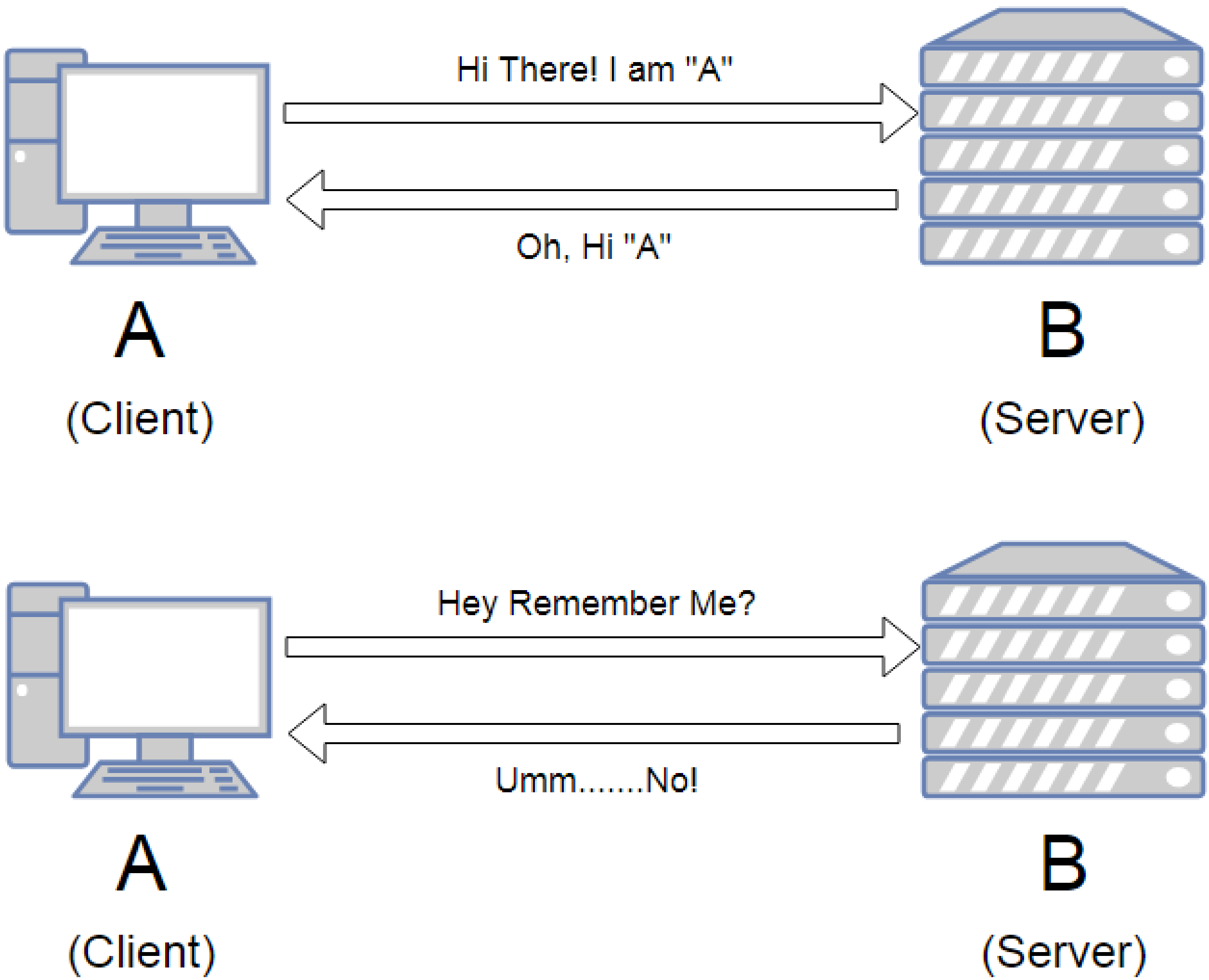
- request msg not understood by server

**`404 Not Found`**

- requested document not found on this server

**`505 HTTP Version Not Supported`**

# http: a stateless protocol



# User-server state: cookies

many Web sites use cookies

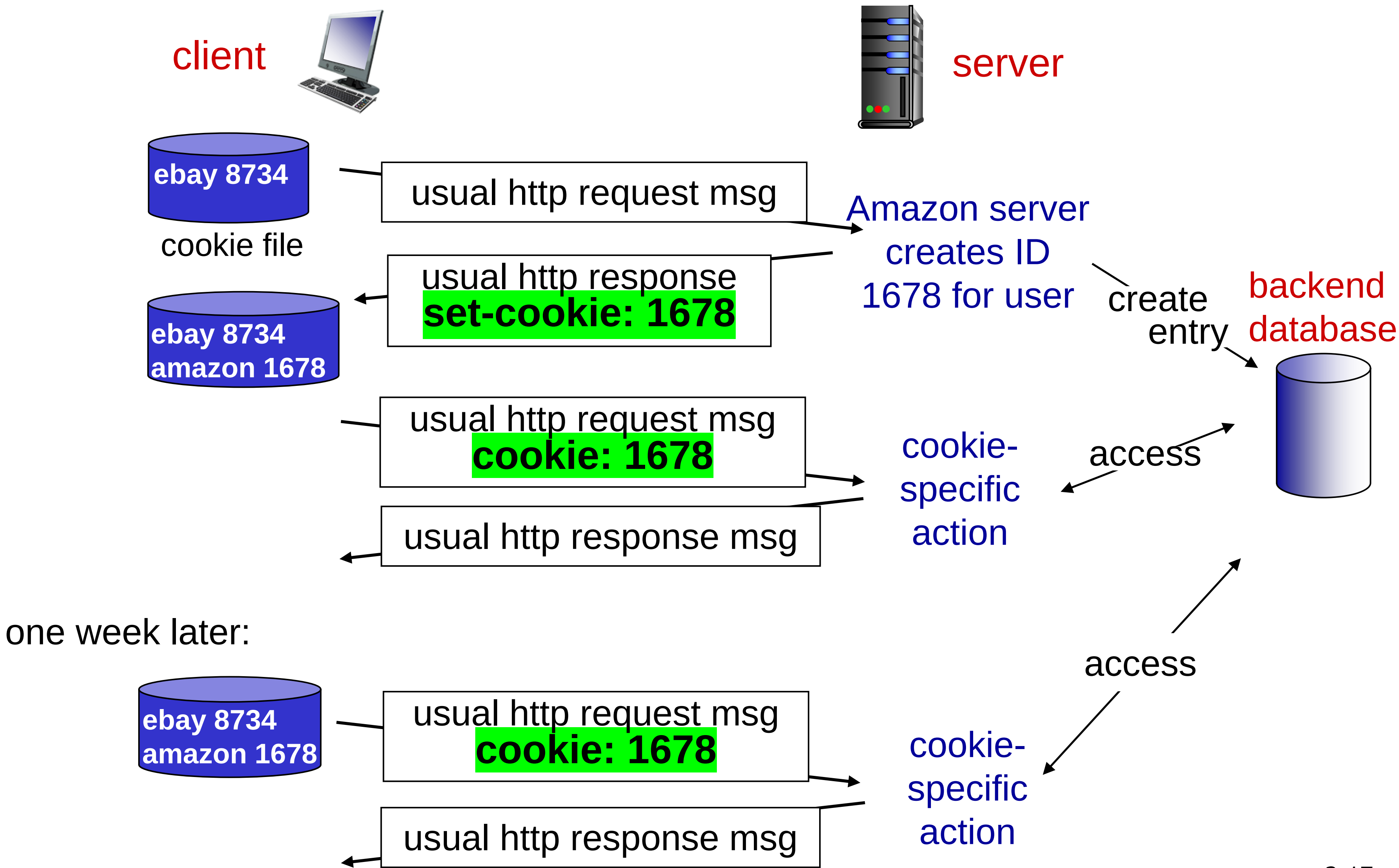
*four components:*

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.)



# Cookies (continued)

*what cookies can be used for:*

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


aside

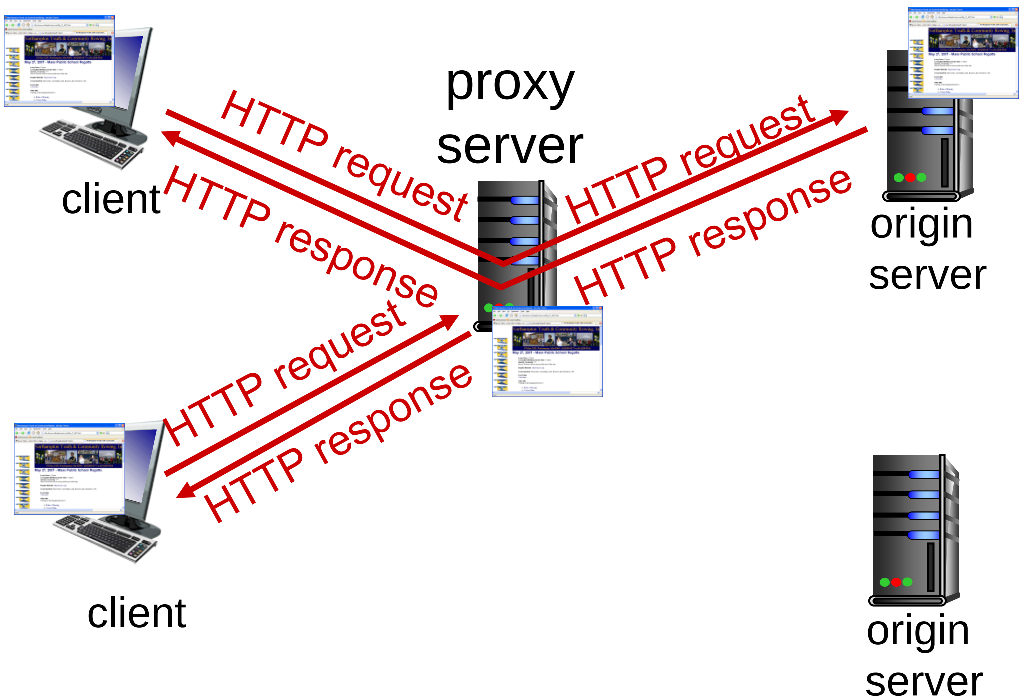
*cookies and privacy:*

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

# 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



# 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)

# More about Web caching

*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)

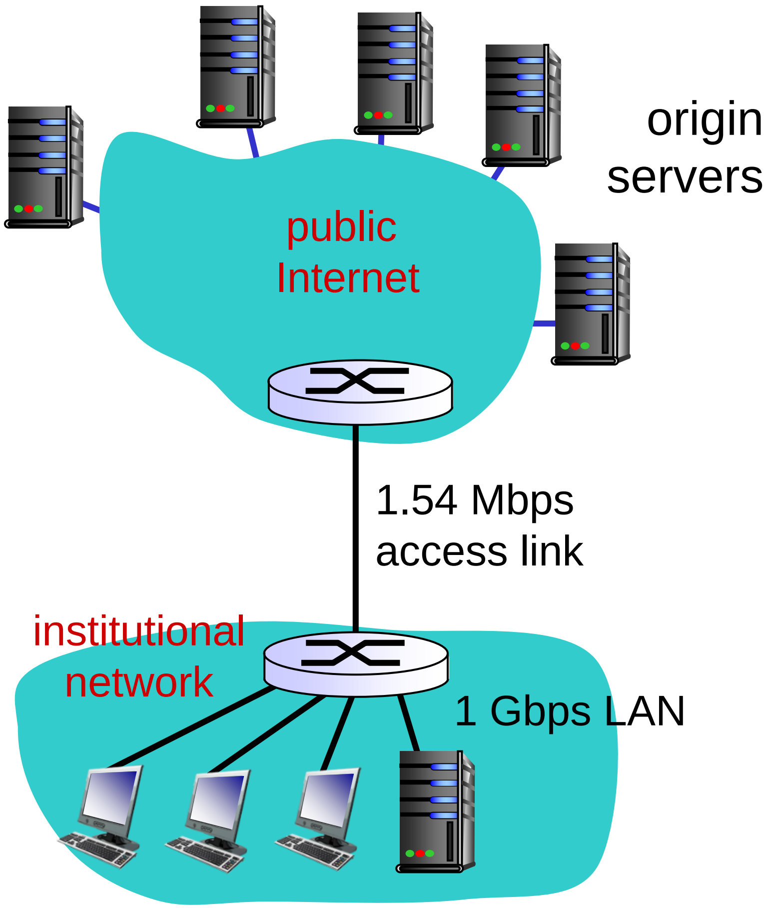
# Caching example:

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



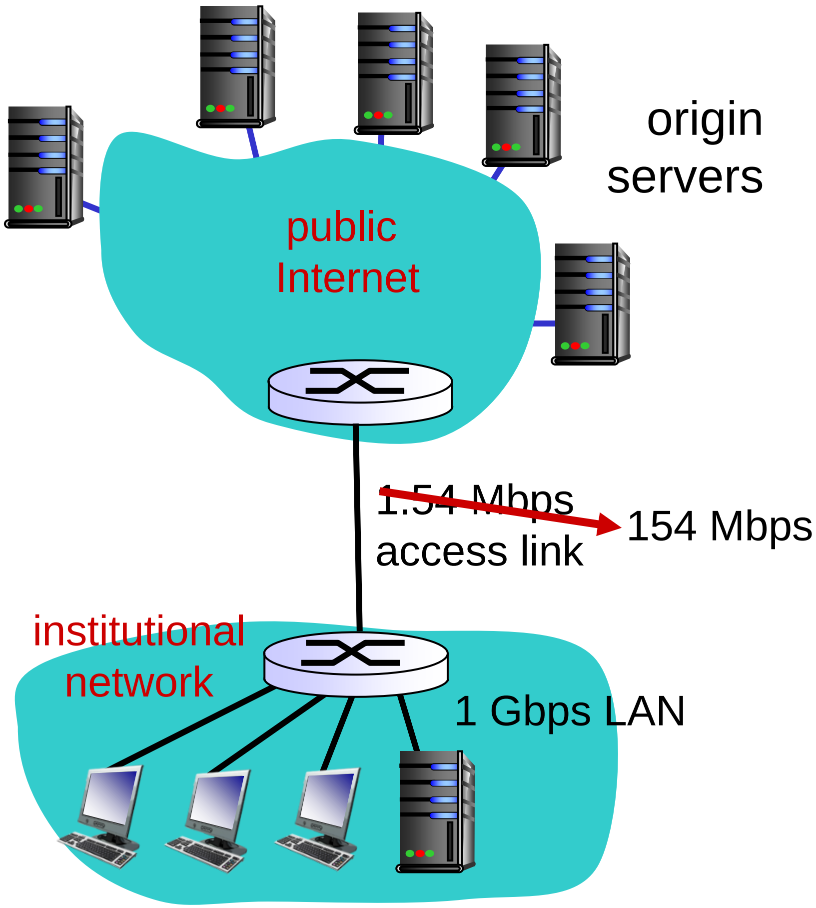
# Option1 - fatter access link

*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 + usecs




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

# Option 2: install local cache

*assumptions:*

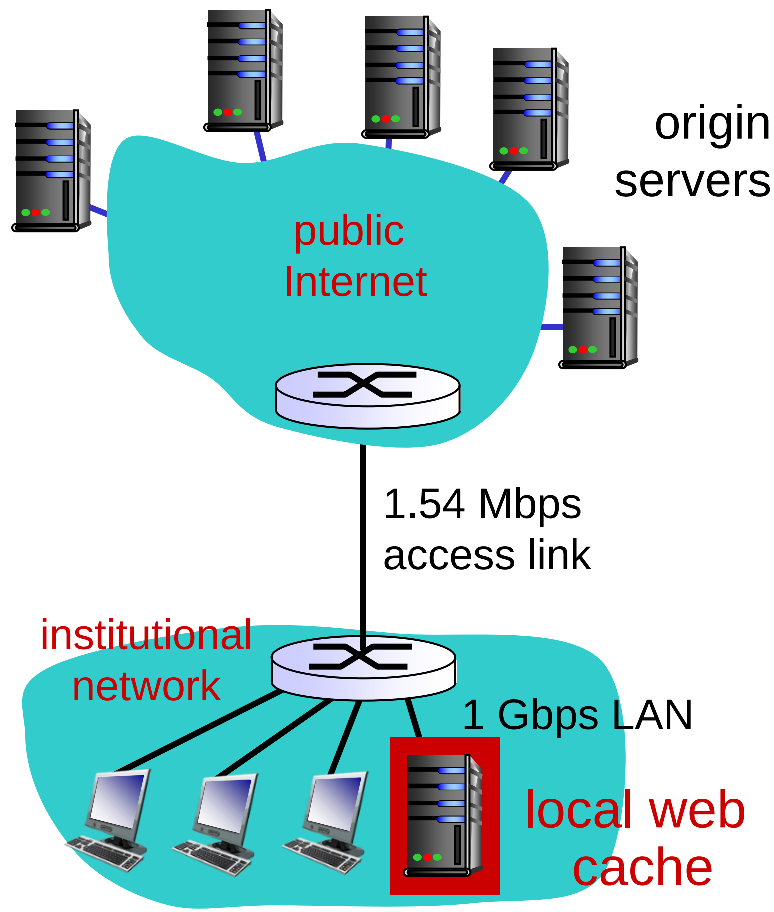
- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

*consequences:*

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

*How to compute link utilization, delay?*

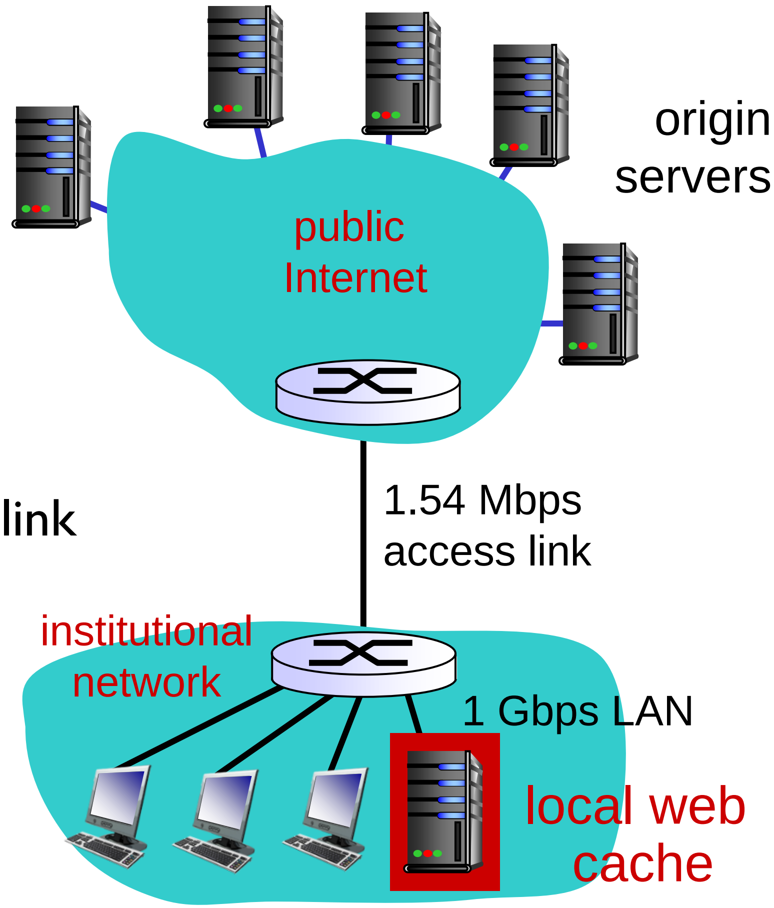
*Cost:* web cache (cheap!)



# Option 2: 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 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)



# Chapter Outline

# DNS: domain name system

*people:* many identifiers:

- SSN, name, passport #

*Internet hosts, routers:*

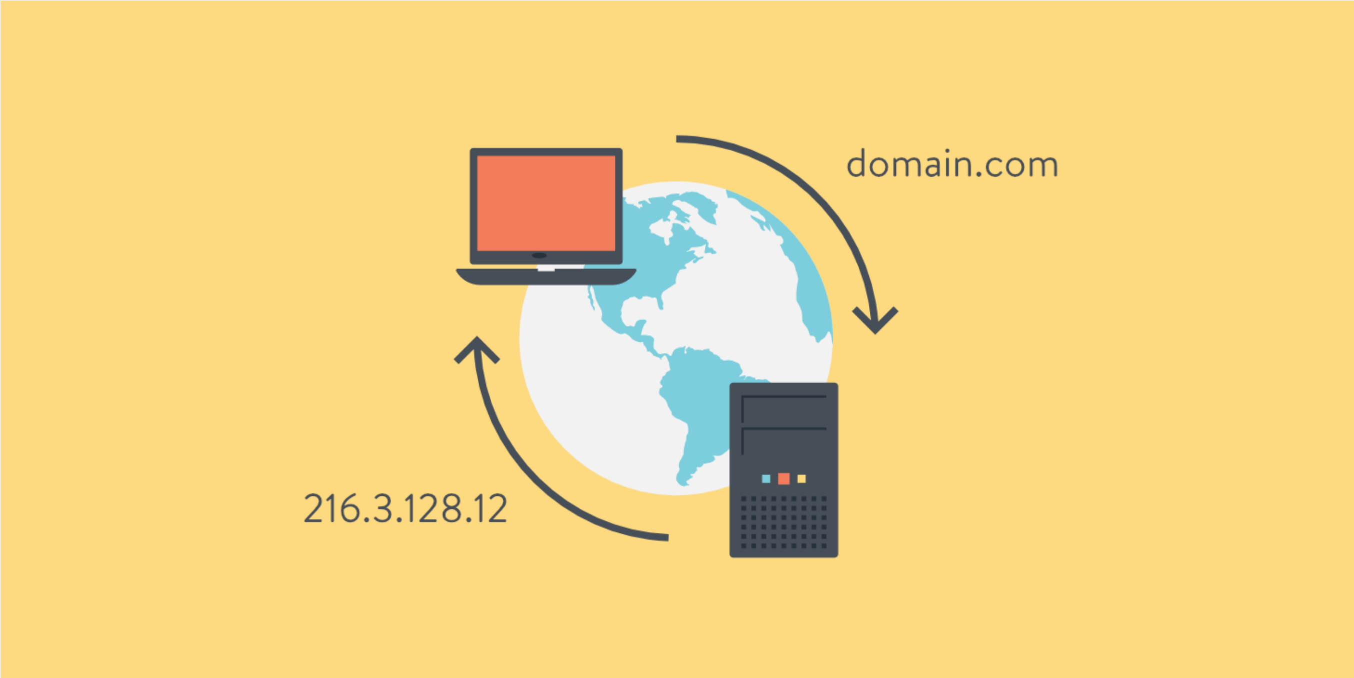
- "name", e.g., www.youtube.com - used by humans
- IP address (32 bit) - used for addressing datagrams

e.g., 121.7.106.83

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

# DNS: domain name system

DNS: map between IP address and name



# DNS: domain name system

*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

*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

# DNS: structure

*why not centralize DNS?*

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

*A: doesn't scale!*

# DNS: a distributed, hierarchical database