

Topic 6 – Applications

- Infrastructure Services (DNS)
 - Now with added security...
- Traditional Applications (web)
 - Now with added QUIC
- P2P Networks
 - Every device serves

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 (e.g., Zoom)
 - Internet search
 - remote login
 - ...
- Q: *your* favorites?

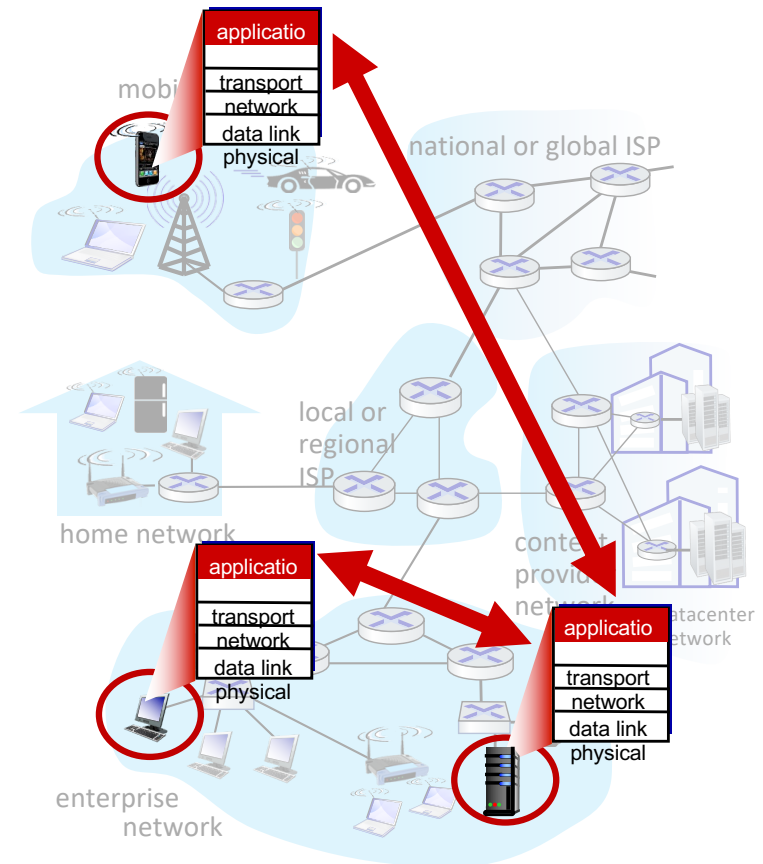
Creating a network app

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



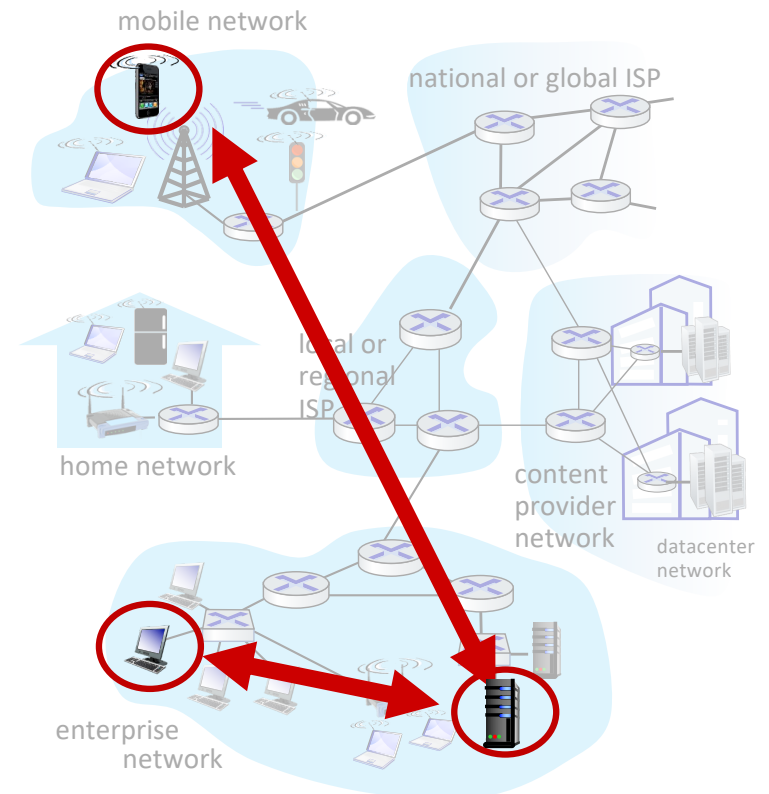
Client-server paradigm

server:

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

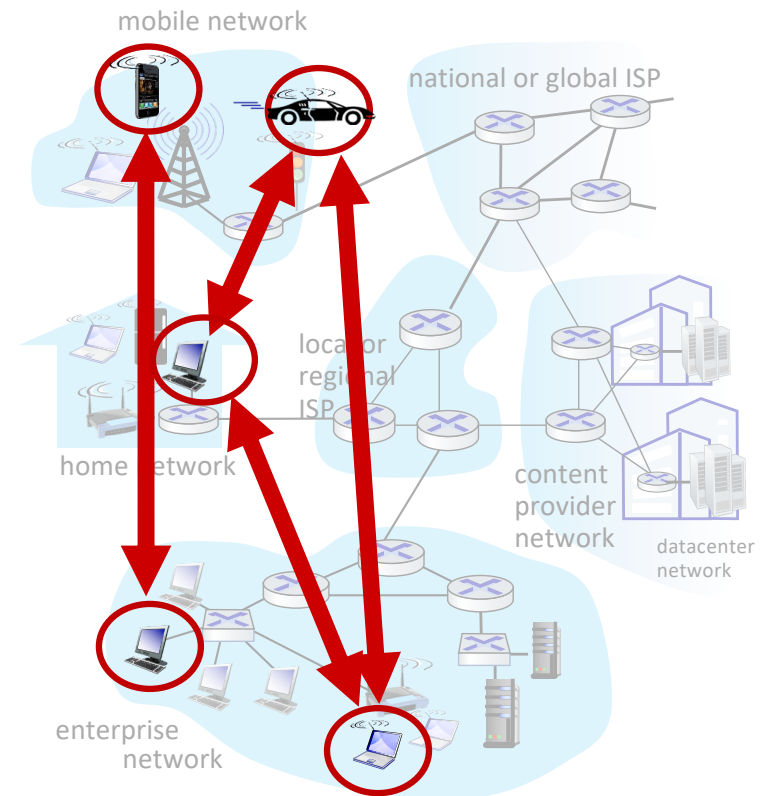
clients:

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



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



An application-layer protocol defines:

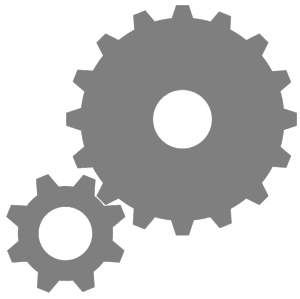
- **types of messages exchanged**,
 - e.g., request, response
- **message syntax**:
 - what fields in messages & how fields are delineated
- **message semantics**
 - meaning of information in fields
- **rules** for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom



Relationship Between Names&Addresses

- Addresses can **change** underneath
 - Move `www.bbc.co.uk` to `212.58.246.92`
 - Humans/Apps should be unaffected
- Name could map to **multiple** IP addresses
 - `www.bbc.co.uk` to multiple replicas of the Web site
 - Enables
 - Load-balancing
 - Reducing latency by picking nearby servers
- **Multiple names** for the same address
 - E.g., aliases like `www.bbc.co.uk` and `bbc.co.uk`
 - Mnemonic stable name, and dynamic canonical name
 - Canonical name = actual name of host

DNS: Domain Name System

people: many identifiers:

- NI #, name, passport #

Internet hosts, routers:

- IP address (32 bit or 128bit) - used for addressing datagrams
- “name”, e.g., cam.ac.uk- used by humans

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

Domain Name System (DNS):

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, DNS 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

Q: Why not centralize DNS?

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

A: doesn't scale!

- Comcast DNS servers alone: 770B DNS queries/day
- Akamai DNS servers alone: 2.6T DNS queries/day

Thinking about the DNS

humongous distributed database:

- ~ billion records, each simple

handles many *trillions* of queries/day:

- *many* more reads than writes
- *performance matters*: almost every Internet transaction interacts with DNS - msec count!

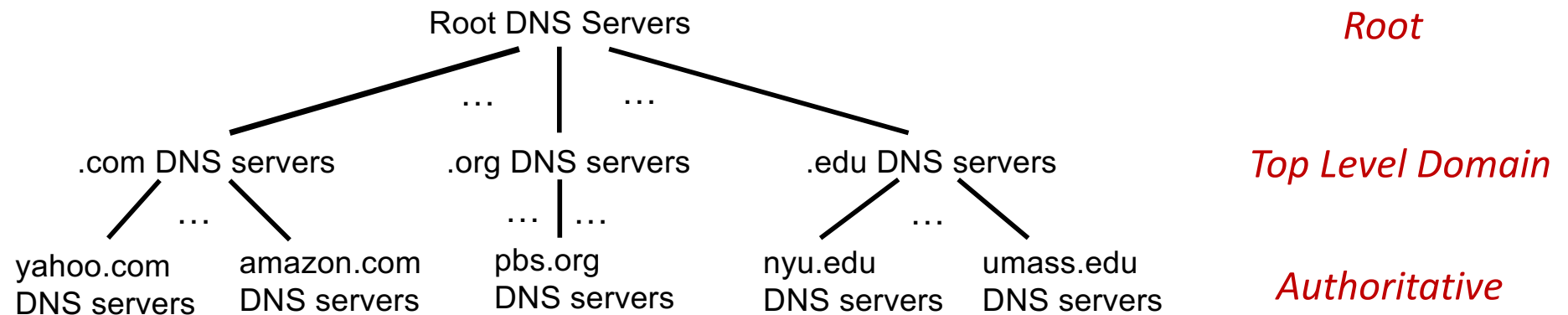
organizationally, physically decentralized:

- millions of different organizations responsible for their records

“bulletproof”: reliability, security



DNS: a distributed, hierarchical database

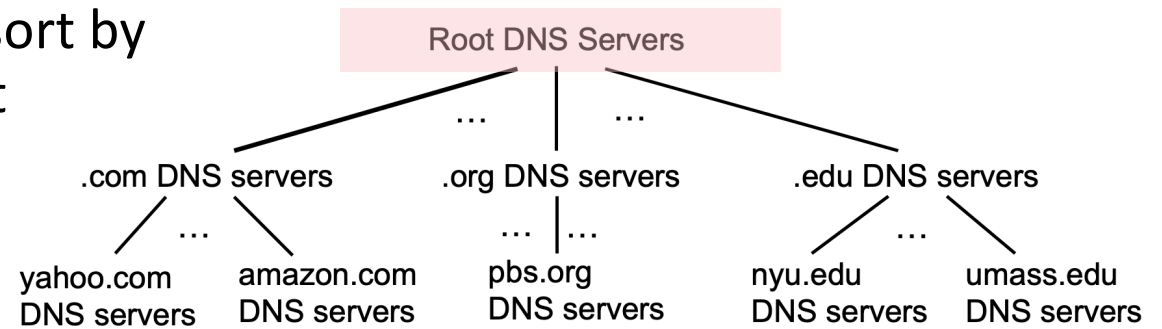


Client wants IP address for `www.amazon.com`; 1st approximation:

- 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

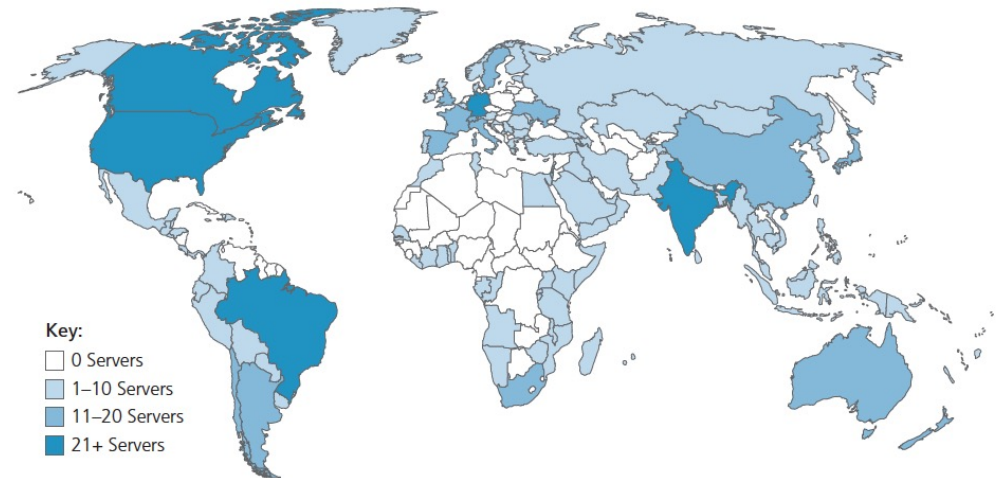
- official, contact-of-last-resort by name servers that can not resolve name



DNS: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
 - Internet couldn't function without it!
 - DNSSEC – provides security (authentication, message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

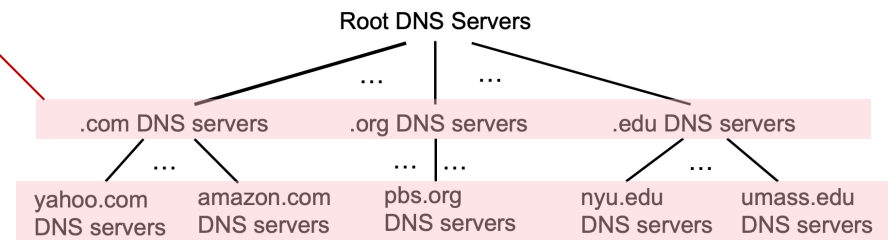
13 logical root name “servers” worldwide each “server” replicated many times (~200 servers in US)



Top-Level Domain, and authoritative servers

Top-Level Domain (TLD) servers:

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

Using DNS

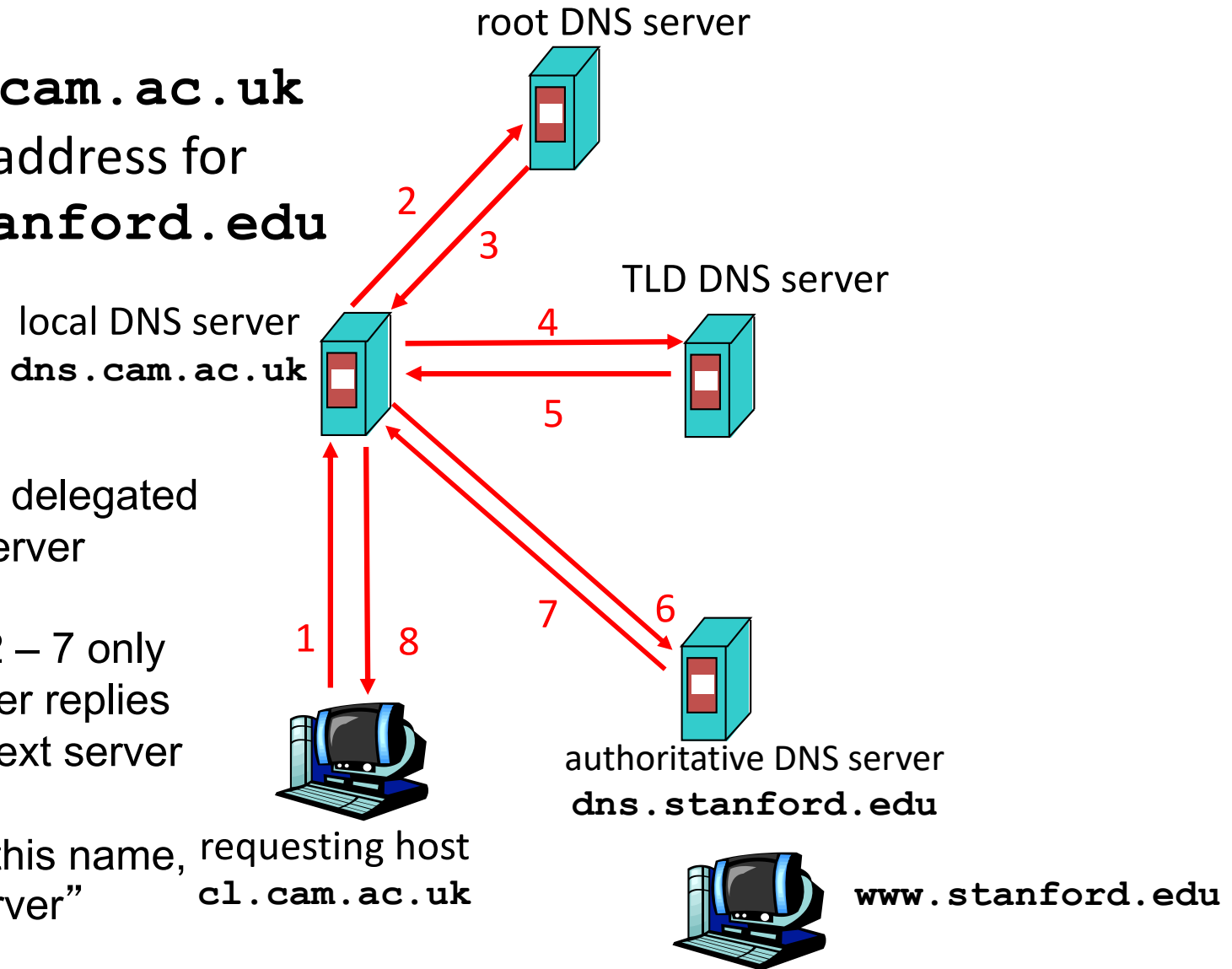
- Two components
 - DNS servers
 - Resolver software on each hosts
- Local DNS server (“default name server”)
 - Usually near the endhosts that use it
 - each ISP has local DNS name server; to find yours:
 - MacOS: % scutil --dns
 - Windows: >ipconfig /all
- Client application
 - Extract server name (e.g., from the URL)
 - Do `gethostbyname()` to trigger resolver code

Local DNS name Servers

- when host makes DNS query, it is sent to its *local* DNS server
 - Local DNS server returns reply, answering:
 - from its local cache of recent name-to-address translation pairs (possibly out of date!)
 - forwarding request into DNS hierarchy for resolution
 - each ISP has local DNS name server; to find yours:
 - MacOS: `% scutil --dns`
 - Windows: `>ipconfig /all`
- local DNS server doesn't strictly belong to hierarchy, acting as they do on behalf of other hosts.

How Does Resolution Happen? (Iterative example)

Host at **cl.cam.ac.uk**
wants IP address for
www.stanford.edu



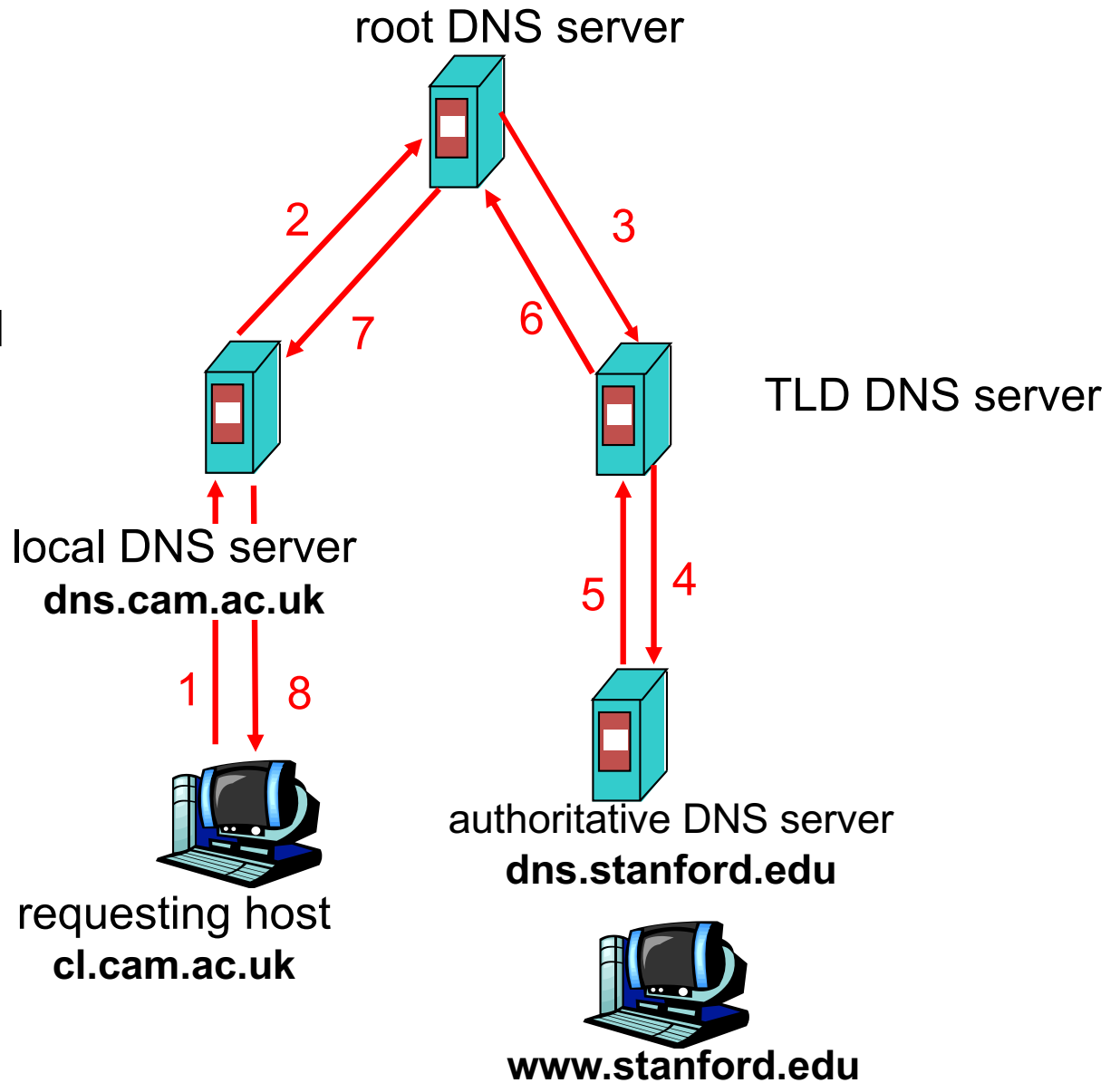
iterated query:

- Host enquiry is delegated to local DNS server
- Consider transactions 2 – 7 only
- contacted server replies with name of next server to contact
- “I don’t know this name, but ask this server”

DNS name resolution **recursive** example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load?



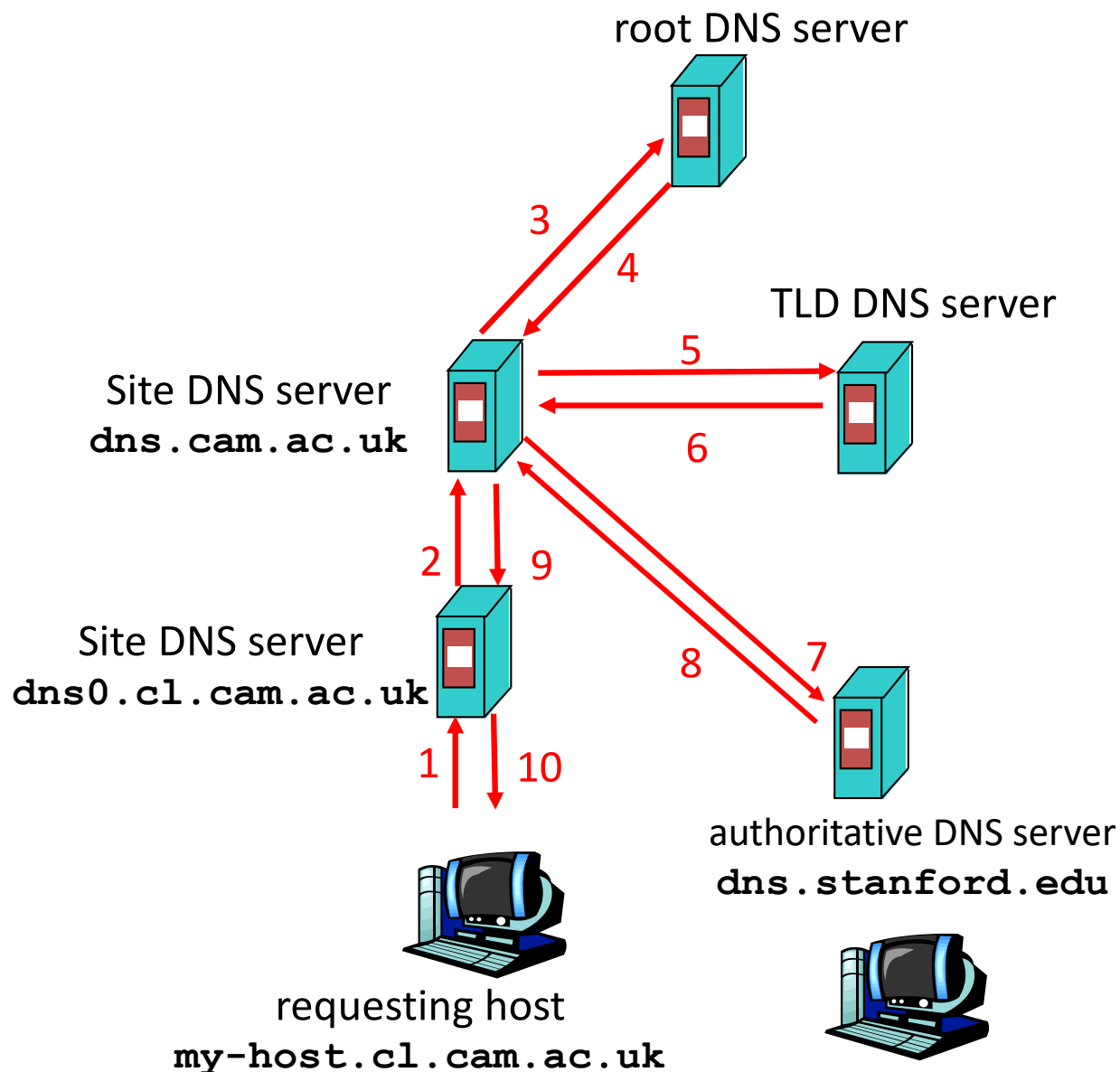
Recursive and Iterative Queries - Hybrid case

recursive query:

- Ask server to get answer for you
- E.g., requests 1,2 and responses 9,10

Iterative query:

- Ask server who to ask next
- E.g., all other request-response pairs



DNS Caching

- Performing all these queries takes time
 - And all this **before** actual communication takes place
 - E.g., 1-second latency before starting Web download
- **Caching** greatly reduces overhead
 - The top-level servers very rarely change
 - Popular sites (e.g., www.bbc.co.uk) visited often
 - Local DNS servers have regularly used information cached
- How DNS caching works
 - DNS servers will cache responses to queries
 - Responses include a “**time to live**” (TTL) field
 - Server deletes cached entry after TTL expires
 - Cached entries may be **out-of-date**
 - if named host changes IP address, may not be known Internet-wide until all TTLs expire!
 - *best-effort name-to-address translation!*

Reliability

- DNS servers are **replicated**
 - Name service available if at least one replica is up
 - Queries can be load-balanced between replicas
- Anycast provides reliability for ROOT servers
- Usually, UDP is used for queries
 - Need reliability: must implement this on top of UDP
 - DNS spec. supports TCP too, but not always available
- Try alternate servers on timeout
 - **Exponential backoff** when retrying same server
- Same identifier for all queries
 - Don't care which server responds

DNS records

DNS: distributed database storing resource records (RR)

RR format: (name, value, type, ttl)

type=A

- name is hostname
- value is IP address

type=NS

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

type=CNAME

- name is alias name for some “canonical” (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

type=MX

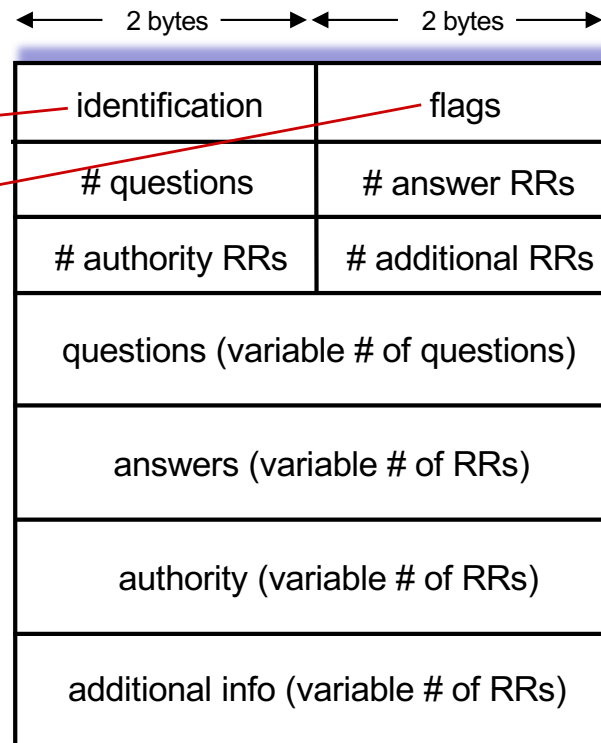
- value is name of SMTP mail server associated with name

DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:

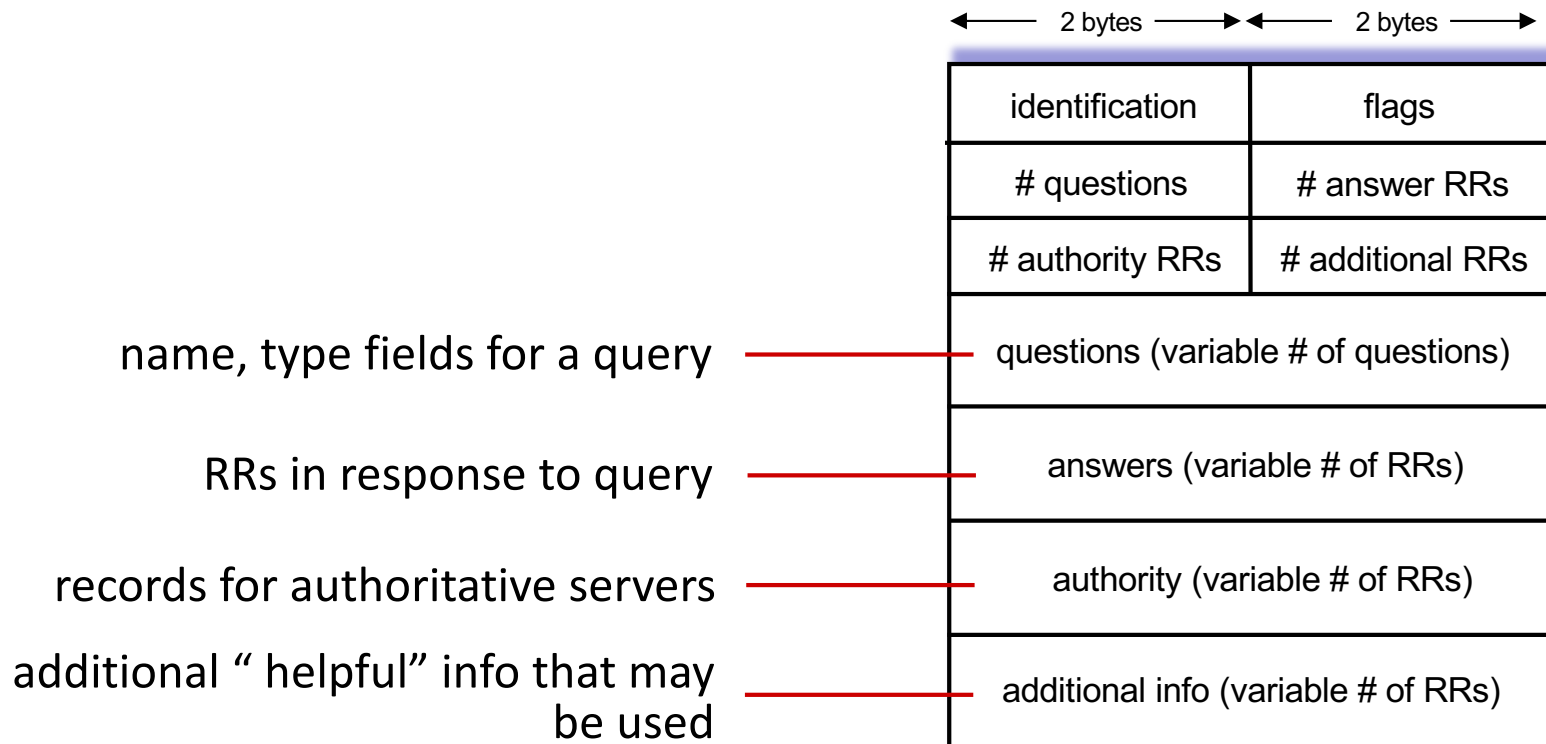
message header:

- **identification:** 16 bit # for query, reply to query uses same #
- **flags:**
 - query or reply
 - recursion desired
 - recursion available
 - reply is authoritative



DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:



Getting your info into the DNS

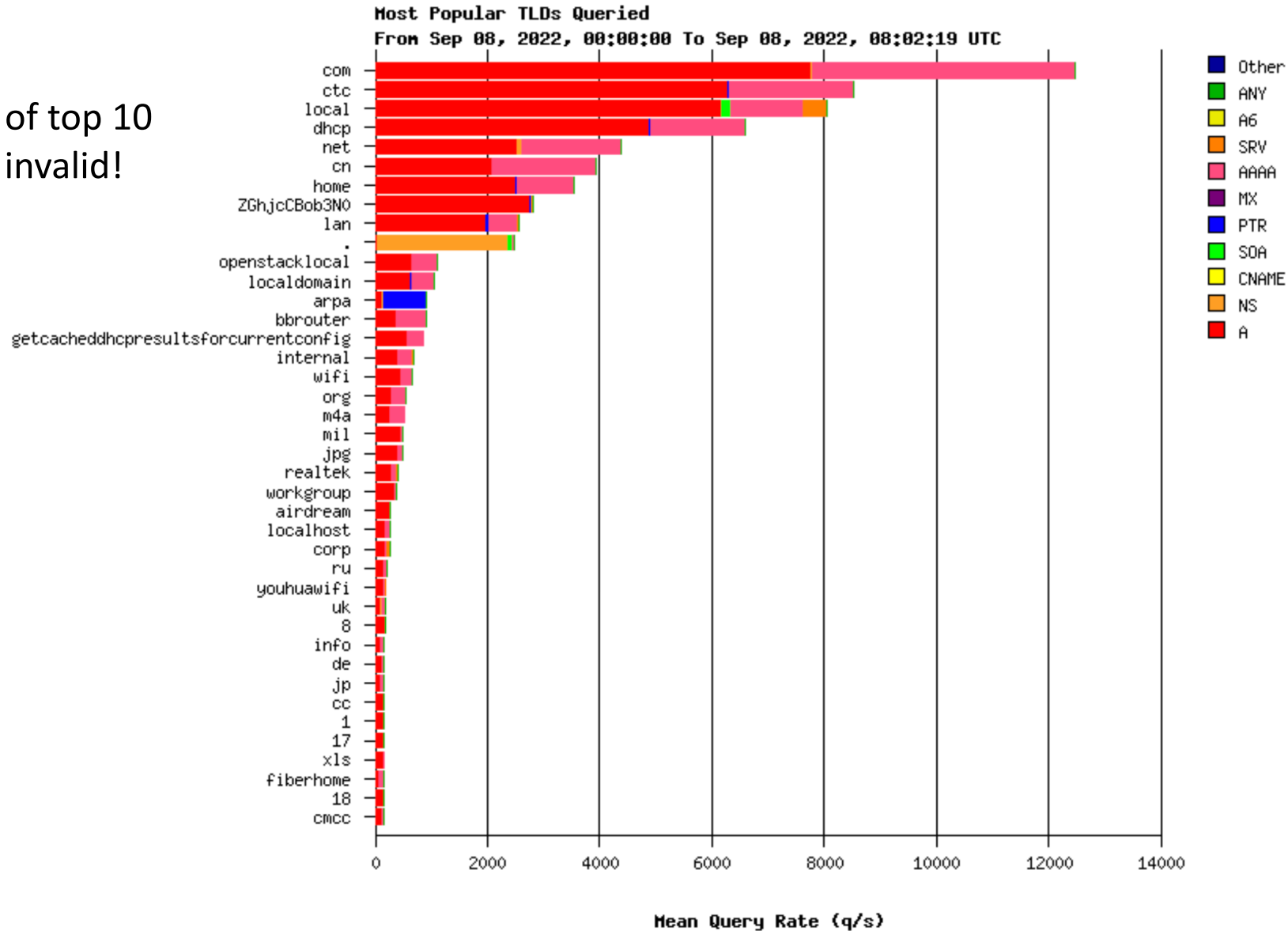
example: new startup “Network Utopia”

- register name networkutopia.com at *DNS registrar* (e.g., Network Solutions)
 - provide names, IP addresses of authoritative name server (primary and secondary)
 - registrar inserts NS, A RRs into .com TLD server:
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server locally with IP address 212.212.212.1
 - type A record for www.networkutopia.com
 - type MX record for networkutopia.com

Most popular TLD

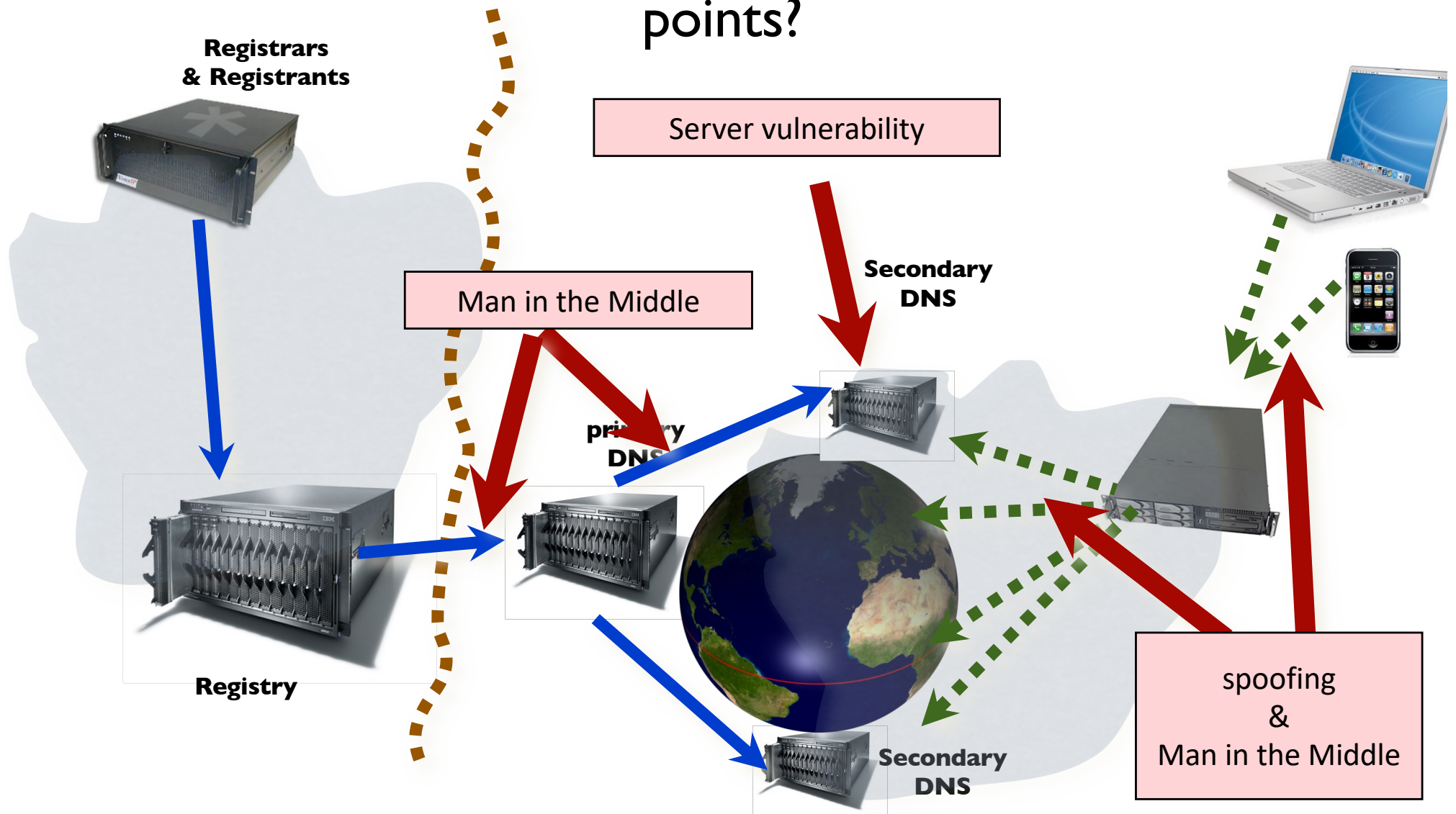
At least WORKGROUP is no longer here!
It was the top invalid TLD for years...

7 of top 10 invalid!



Data flow through the DNS

Where are the vulnerable points?



DNS attack surface

DDoS attacks

- bombard root servers with traffic
 - not successful to date
 - traffic filtering
 - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
 - potentially more dangerous

Spoofing attacks

- intercept DNS queries, returning bogus replies
 - DNS cache poisoning
 - RFC 4033: DNSSEC authentication services

DNS Security

- No way to verify answers
 - Opens up DNS to many potential attacks
 - DNSSEC fixes this
- Most obvious vulnerability: recursive resolution
 - Using recursive resolution, host must trust DNS server
 - When at Starbucks, server is under their control
 - And can return whatever values it wants
- More subtle attack: Cache poisoning
 - Those “additional” records can be anything!

DNSSEC protects all these end-to-end

- provides message authentication and integrity verification through cryptographic signatures
 - You know who provided the signature
 - No modifications between signing and validation
- It does **not** provide authorization
- It does **not** provide confidentiality
- It does **not** provide protection against DDOS

DNSSEC in practice

Problem: Scaling the key signing and key distribution

Solution: Using the DNS to Distribute Keys

- Distribute keys through the DNS hierarchy
 - Use one trusted key to establish authenticity of other keys
 - Building chains of trust from the root down
 - Parents need to sign the keys of their children
- Only the root key needed in ideal world
 - Parents always delegate security to child

On osx "host -av www.cl.cam.ac.uk"

```
% host -va www.cl.cam.ac.uk
Trying "www.cl.cam.ac.uk"
Trying "www.cl.cam.ac.uk"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25214
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 23, AUTHORITY: 0, ADDITIONAL: 0
```

```
;; QUESTION SECTION:
;www.cl.cam.ac.uk. IN ANY
```

```
.. ANSWER SECTION:
```

```
www.cl.cam.ac.uk. 1200 IN RRSIG NSEC 5 5 1200 20230317214336 20230215204336 31575 cl.cam.ac.uk.
h2JCZHfF9m+jqvHQR6z67LDC3g1xKoCRDmrss+LVrAXJqWsi+d8+/Gio /U07C2SXKga3NXj5ByNvqH2HJs6Loc1emEEeiPLSYqk0USGTLRQhLEqy
bHEfeCPV4hJy/NSuRvcxZCqpgEDSbF5K0JzqI6dnPCo0mFdsRa4n90T kDQ=
```

```
www.cl.cam.ac.uk. 1200 IN NSEC www-443-120.cl.cam.ac.uk. A PTR TXT AAAA SSHFP RRSIG NSEC CAA
www.cl.cam.ac.uk. 21600 IN RRSIG SSHFP 5 5 21600 20230306104604 20230204102237 31575 cl.cam.ac.uk.
kfCKxAD9cyLJDj/UEJl7Sr8b55yH8dxFYc+BF9tggcbReo2GNLQel0ZN rB5JAhoewZ9HxLAS05rzCX1BR9AP0H+RK7BNbDp8rvn09G8PWGQcpKHM
BJWb9nj2a/zi360WCUGH/u8GLPw0L7b2P460DYx4wDmL3jNjvw61Ca Y3g=
www.cl.cam.ac.uk. 21600 IN SSHFP 3 2 B7E1DF5B943C481A263307EDEE23F0719858CDF516F2482B54A4B248 0118CAE9
www.cl.cam.ac.uk. 21600 IN SSHFP 2 1 6FB539DBE0E273B56327E619BB1814DB4CE810D8
www.cl.cam.ac.uk. 21600 IN SSHFP 4 2 293F122F4D4970C42B767898C5505C3EB838E0C5BB432EF36AF33C03 1FA792FA
www.cl.cam.ac.uk. 21600 IN SSHFP 4 1 5E0CBE0730922925C446DF1B2DCE336AA7565122
www.cl.cam.ac.uk. 21600 IN SSHFP 1 1 FF45237DF493102CF7478AE0A96DE773FB4877C4
www.cl.cam.ac.uk. 21600 IN SSHFP 1 2 2953D9172EC850D2A46FA0245DFFAE978EE31B3BED233DED77BC937B 115952D7
www.cl.cam.ac.uk. 21600 IN SSHFP 3 1 FD276CF12A0B909533ABFA5931622950308AF099
www.cl.cam.ac.uk. 21600 IN SSHFP 2 2 403A5EE7B8ADD3E16B5973874E54CDFAC82268CC63B4CFD90E74DDC6 4E2EDF6F
```

```
www.cl.cam.ac.uk. 21600 IN RRSIG AAAA 5 5 21600 20230316213444 20230214203939 31575 cl.cam.ac.uk.
eVJM0NwnGPVC9y+96Ijq48feYCDxTLEZ66fcH83a02VFXoCblJkLUCoK e0TeobR+mnLad0XJFUocfjKorIV6s1CNzG90nmV1+dxQD1VBxQzBrV9A
k+JqokUQbkb0UsV4UIWUvRav0M1GccXS5NxzL/HDITMvVXMZx/Citlr lqU=
```

```
www.cl.cam.ac.uk. 21600 IN AAAA 2a05:b400:110::80:14
```

```
www.cl.cam.ac.uk. 21600 IN RRSIG TXT 5 5 21600 20230321232700 20230219230148 31575 cl.cam.ac.uk.
Tjlztn2dsdjr5wGakuPVTy/0V/BBTDEC3K8x7nNnml9dRoy/ncRLWEyA 9XsxENQ20ei7evt6peLFstpvWny6F9nMs+xAfYDiX0PpcJ4pZMNAD0gs
BNXhR2XS0IknnuquUwPLH1WTFZqBd27gIs0QF+79Atj3MHQ5hBZLCS0n EoE=
```

```
www.cl.cam.ac.uk. 21600 IN TXT "pseudo IP address for cl.cam.ac.uk departmental www server (IPv4 and IPv6)"
```

```
www.cl.cam.ac.uk. 21600 IN RRSIG PTR 5 5 21600 20230314163841 20230212160205 31575 cl.cam.ac.uk.
gceJom14zzcCsdyU2ymxuMABhLZ06VDI8J4seDqDt09924AvcpL2P9 5wLDds02a0JGTqbnndIydkFg0A6fJHNMCEBAQr6GVjL/Fg+YLWH8YwLx
Uu9a7FamZvXLk67wC9ij17VZ3V9Co5Kd2kwu0v07Xb50evPKAlL/tL Bdk=
```

```
www.cl.cam.ac.uk. 21600 IN PTR svr-www-00.cl.cam.ac.uk.
```

```
www.cl.cam.ac.uk. 21600 IN RRSIG A 5 5 21600 20230314163841 20230212160205 31575 cl.cam.ac.uk.
j8k+Q8L2C7zzHSvpWpg+1t5WPK9IeTZ9G0vw/0v1pbYVXJfeHuNm9ERM Ff/hEKZm21ooFIBtrbTe/m5b+kBBm20ETDbbGP+na3/DEQyWfP0sHe6j
S2ZS9Ke0XENJK92eA+/dBAWe0vFvTpXrZ/thp61ctqm9b3mR8AbnuNqu uHs=
```

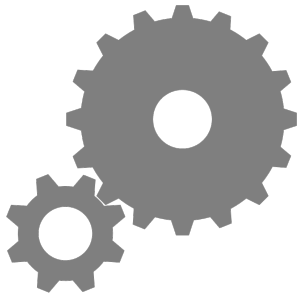
```
www.cl.cam.ac.uk. 21600 IN A 128.232.0.20
```

```
www.cl.cam.ac.uk. 600 IN CAA 0 issuewild ";"
```

```
www.cl.cam.ac.uk. 600 IN CAA 0 issue "quovadisglobal.com"
```

```
www.cl.cam.ac.uk. 600 IN CAA 0 issue "letsencrypt.org"
```

```
www.cl.cam.ac.uk. 600 IN RRSIG CAA 5 5 600 20230324004125 20230222002906 31575 cl.cam.ac.uk.
A930aBg5uKP2l2aYxJ1gbCSnbR/o8n8o0s54fB0SU0kE55YmQWRNkNEW AGuuJltIz0I/LJ9eH4Jf+VL7K01AimzS2ae6GXnXogP3shaz16jh+psX
rRQhKa2S0LcfrJM2j3ltct88AewpLk4nrv5rlvCS2yumGQlvKaMuEaga R14=
```

Why is the web so successful?

- What do the web, youtube, facebook, twitter, instagram, have in common?
 - The ability to self-publish
- Self-publishing that is easy, independent, *free*
- No interest in collaborative and idealistic endeavors
 - People aren't looking for Nirvana (or even Xanadu)
 - People also aren't looking for technical perfection
- Want to make their mark, and find something neat
 - Two sides of the same coin, creates synergy
 - “Performance” more important than dialogue....

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.university.ac.uk/someDept/pic.gif`

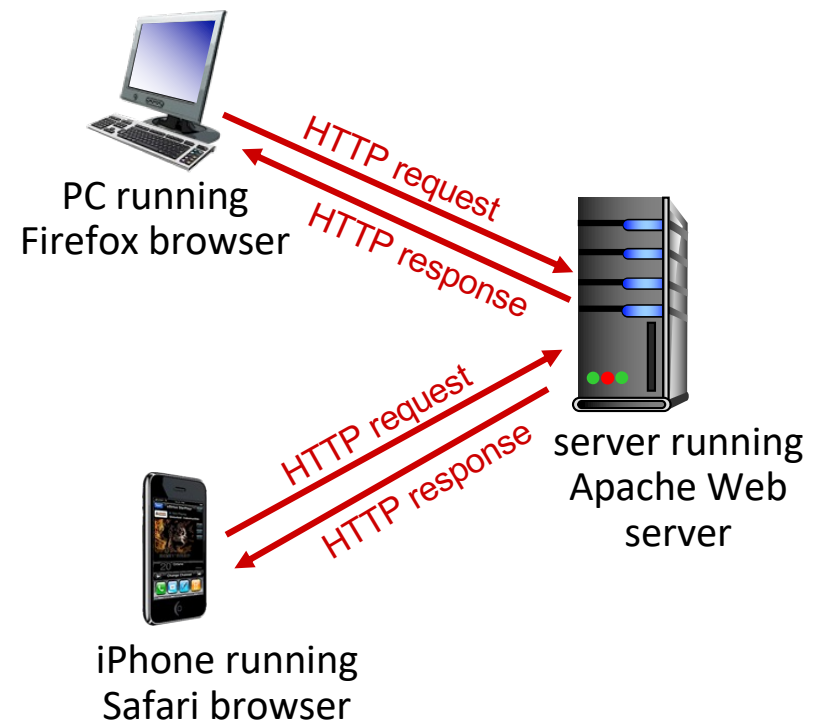
host name

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)

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

Reminder: Distributed Systems are Hard!

protocols that maintain “state” are complex!

- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

HTTP connections: two types

Non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

Persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

Non-persistent HTTP: example

User enters URL: `www.university.ac.uk/someDepartment/home.index`
(containing text, references to 10 jpeg images)



1a. HTTP client initiates TCP connection to HTTP server (process) at `www.university.ac.uk` on port 80



1b. HTTP server at host `www.university.ac.uk` waiting for TCP connection at port 80 “accepts” connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`

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

time



Non-persistent HTTP: example (cont.)

User enters URL: `www.university.ac.uk/someDepartment/home.index`
(containing text, references to 10 jpeg images)



5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects

4. HTTP server closes TCP connection.



time

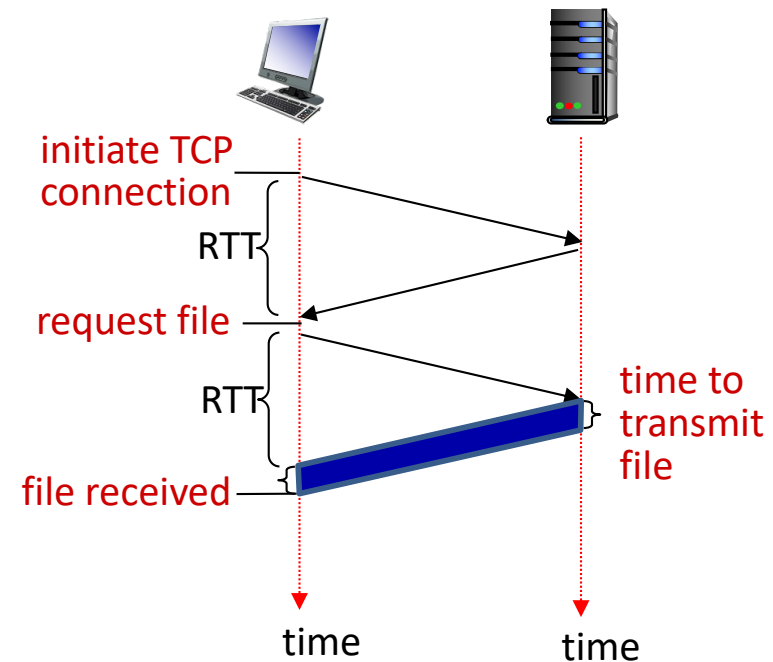


Non-persistent HTTP: response time

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

HTTP response time (per object):

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



Non-persistent HTTP response time = 2RTT + file transmission time

Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:


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

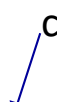
Persistent HTTP (HTTP1.1):


- 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 (cutting response time in half)

HTTP request message

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

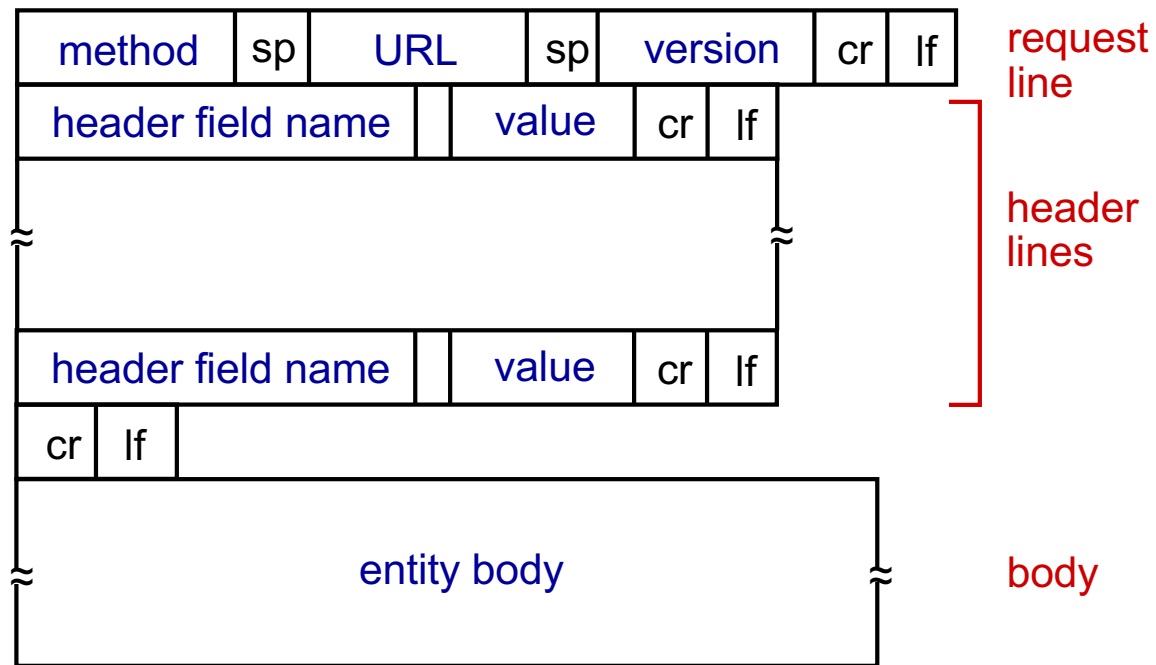
request line (GET, 
POST,
HEAD commands)

 carriage return character
line-feed character

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

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

HTTP request message: general format



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 POST HTTP request message

HTTP response message

status line (protocol  status code status phrase) HTTP/1.1 200 OK

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

Trying out HTTP (client side) for yourself

1. Netcat (telnet will also work) to your favorite Web server:

```
% nc -c -v www.cl.cam.ac.uk 80
```

- opens TCP connection to port 80 (default HTTP server port) at `www.cl.cam.ac.uk` anything typed in will be sent to port 80 at `www.cl.cam.ac.uk`

2. type in a GET HTTP request:

```
GET /~awm22/index.php HTTP/1.1  
Host: www.cl.cam.ac.uk
```

- by typing this in (hit carriage return twice), you send a 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)

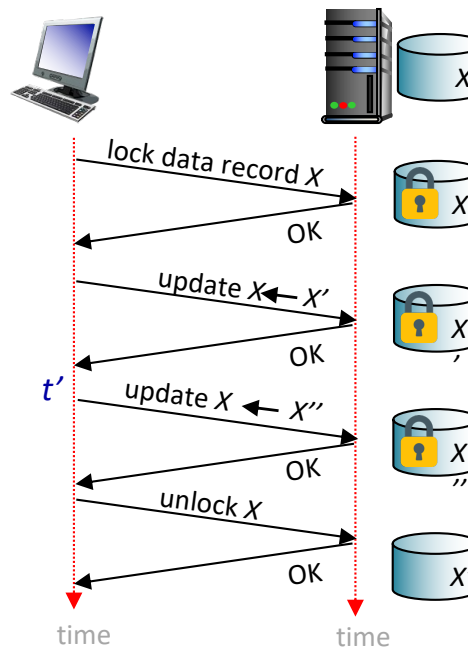
Although in readable ascii – you will notice this is not the webpage but a redirect
Automatically moving to an https secure connection

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 partial-but-never-entirely-completed transaction

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



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

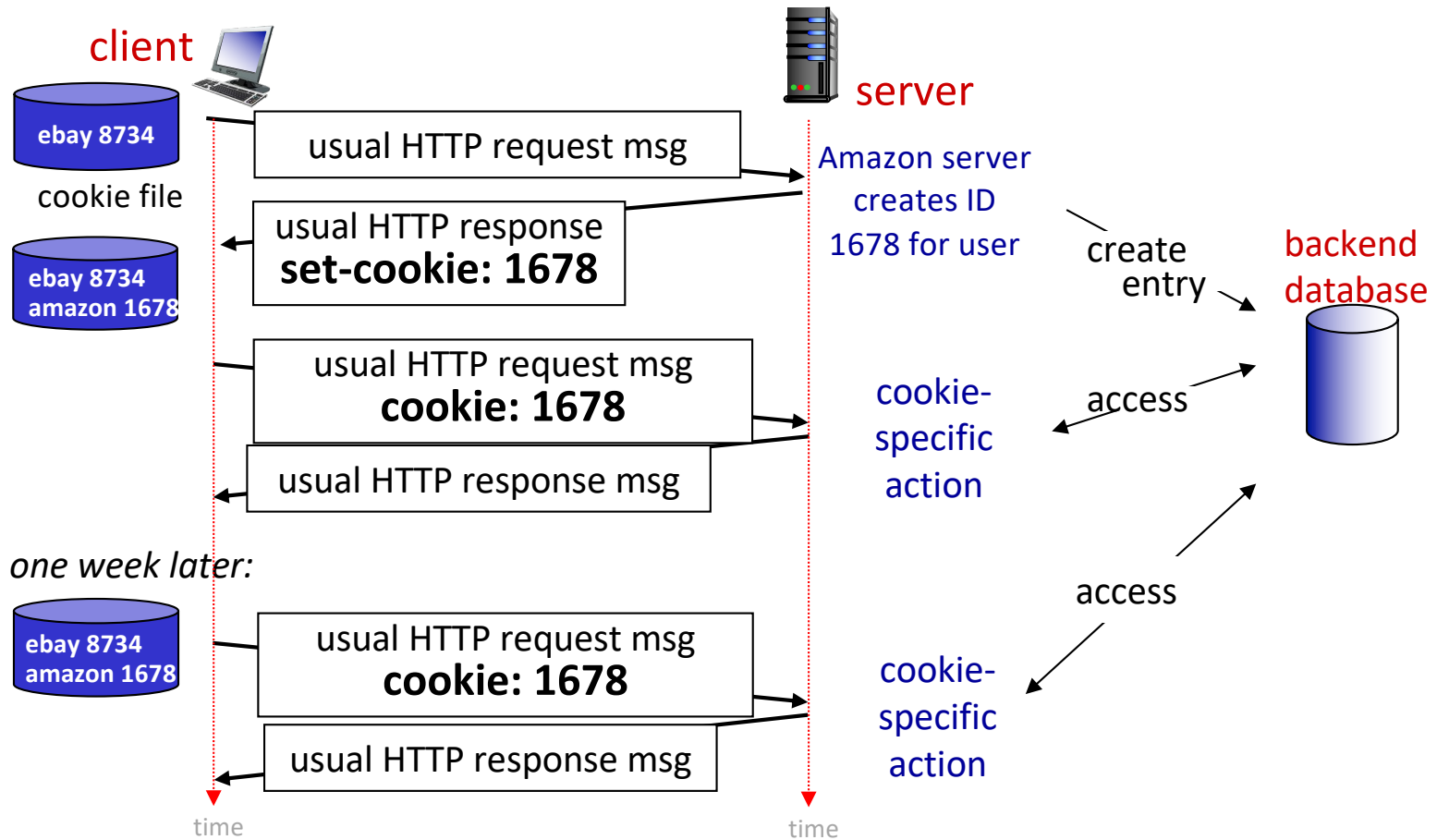
four components:

- 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



HTTP cookies: comments

What cookies can be used for:

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

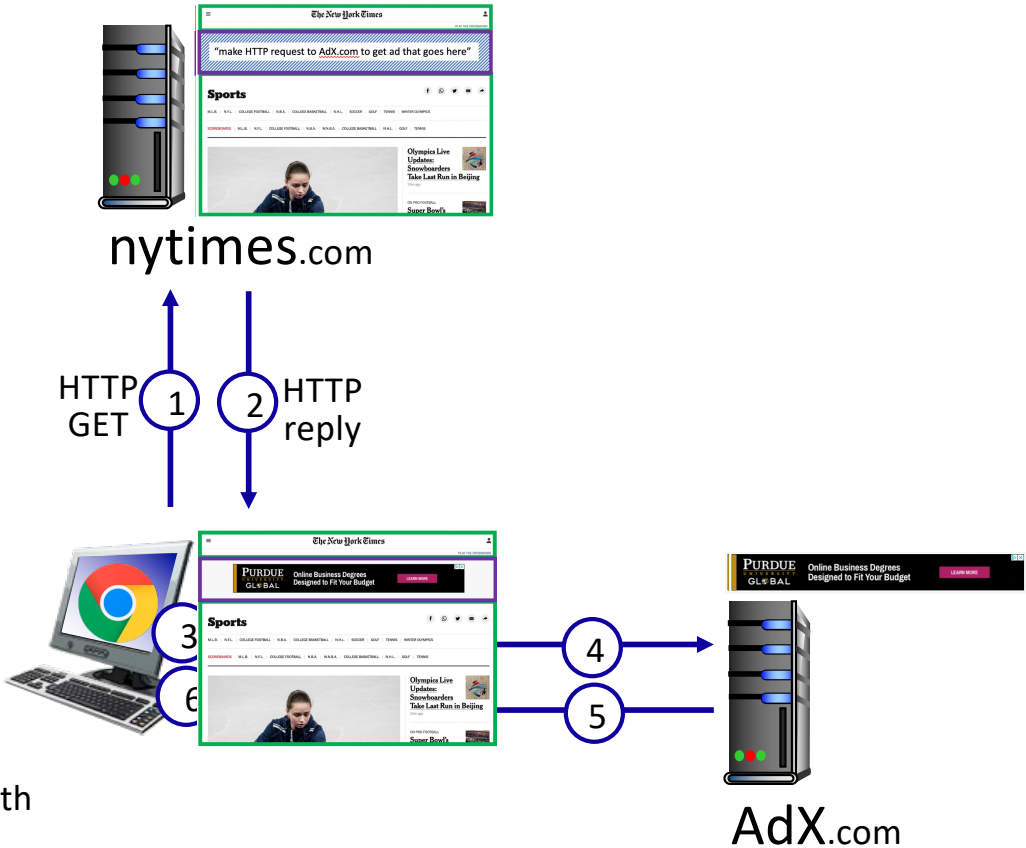
Challenge: How to keep state?

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

- aside
- 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

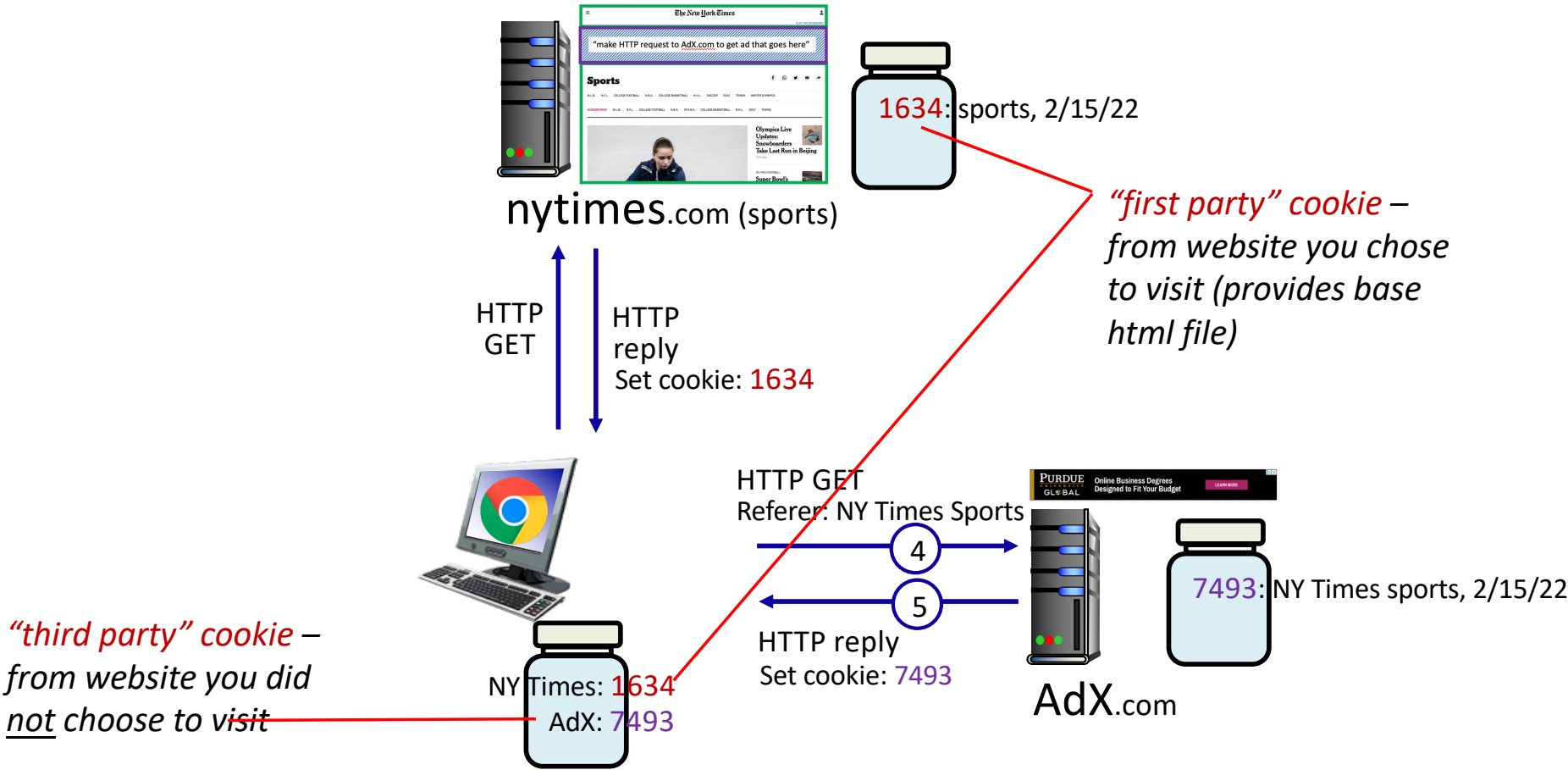
Example: displaying a NY Times web page

- 1 GET base html file from nytimes.com
- 2
- 4 fetch ad from AdX.com
- 5
- 7 display composed page

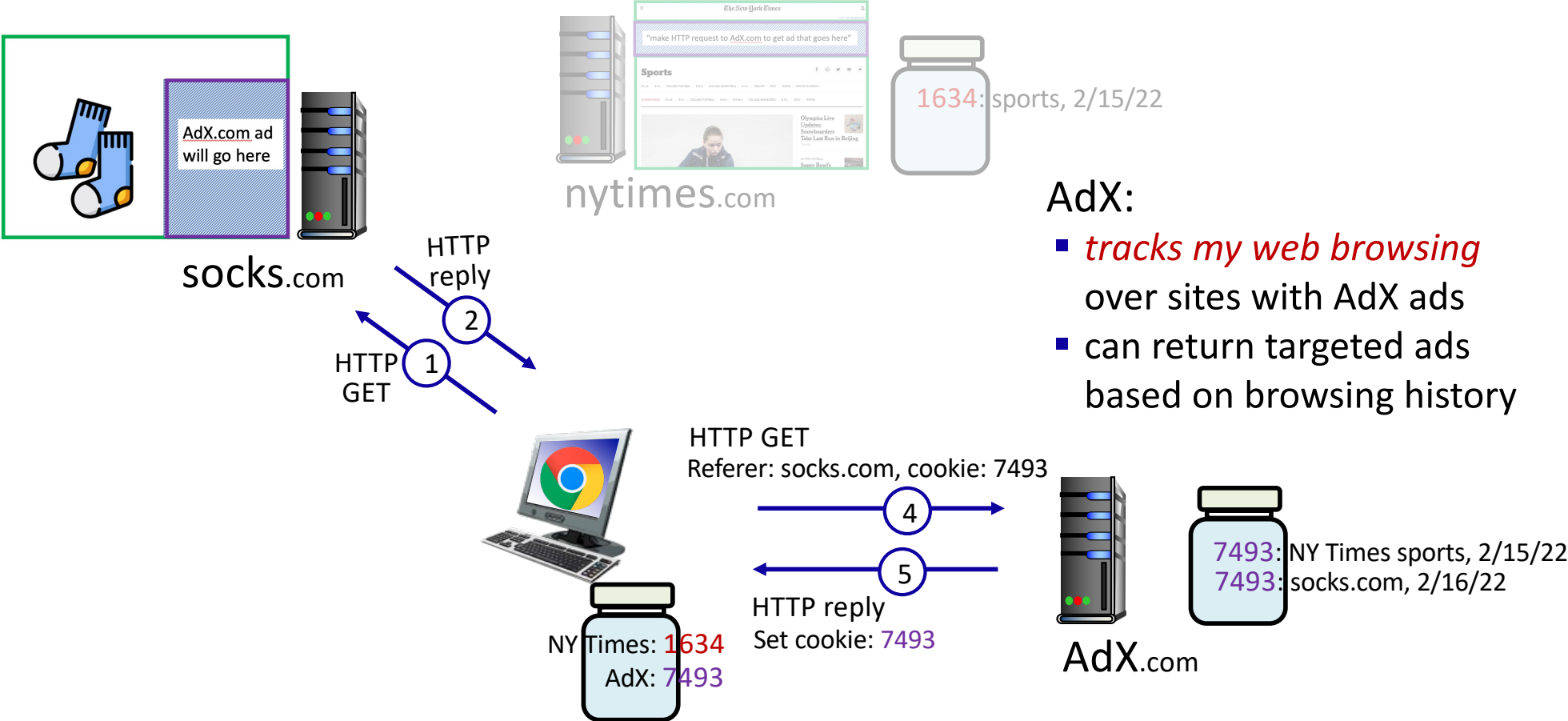


NY times page with embedded ad displayed

Cookies: tracking a user's browsing behavior

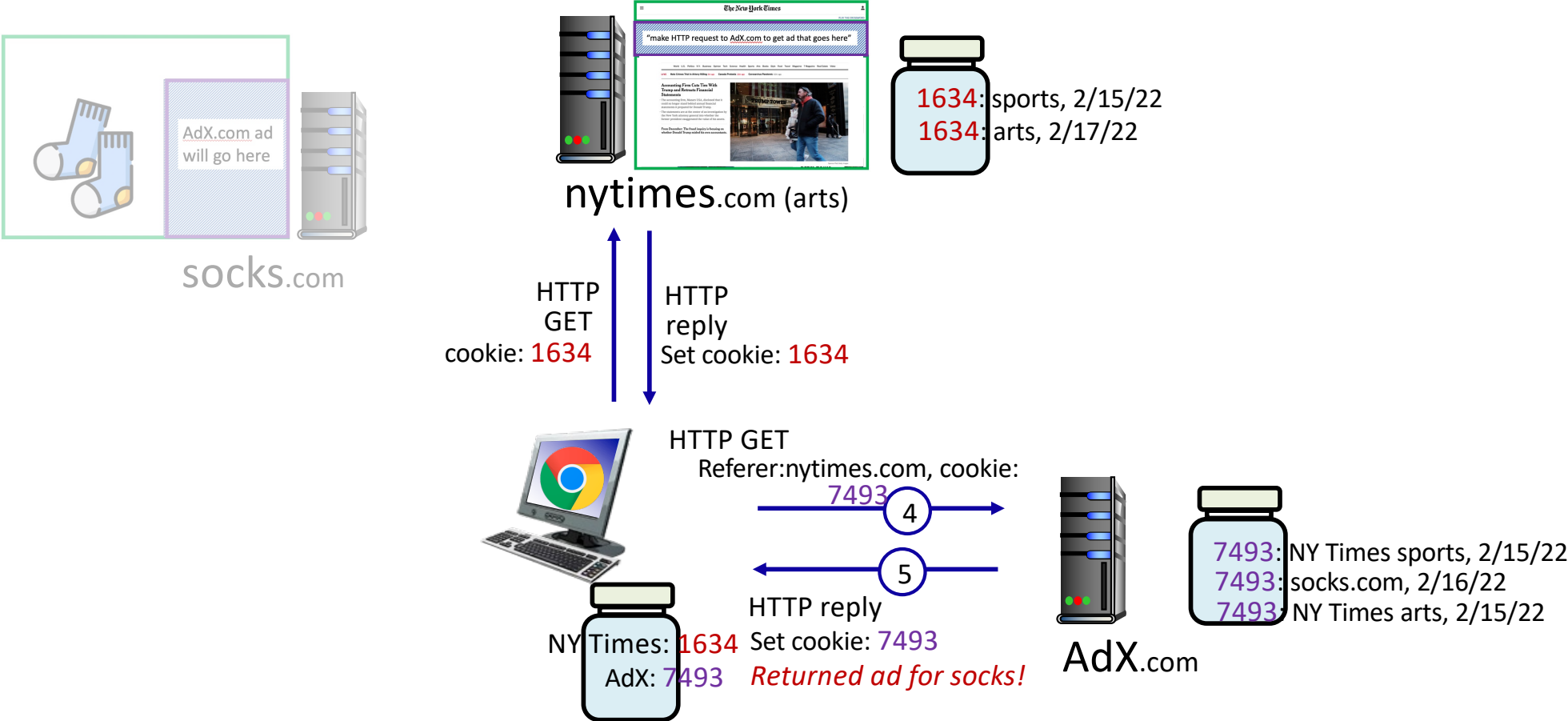


Cookies: tracking a user's browsing behavior



- AdX:
- *tracks my web browsing* over sites with AdX ads
 - can return targeted ads based on browsing history

Cookies: tracking a user's browsing behavior (one day later)



Cookies: tracking a user's browsing behavior

Cookies can be used to:

- track user behavior on a given website (**first party cookies**)
- track user behavior across multiple websites (**third party cookies**) without user ever choosing to visit tracker site (!)
- tracking may be *invisible* to user:
 - rather than displayed ad triggering HTTP GET to tracker, could be an invisible link

third party tracking via cookies:

- disabled by default in Firefox, Safari browsers
- to be disabled in Chrome browser in 2023

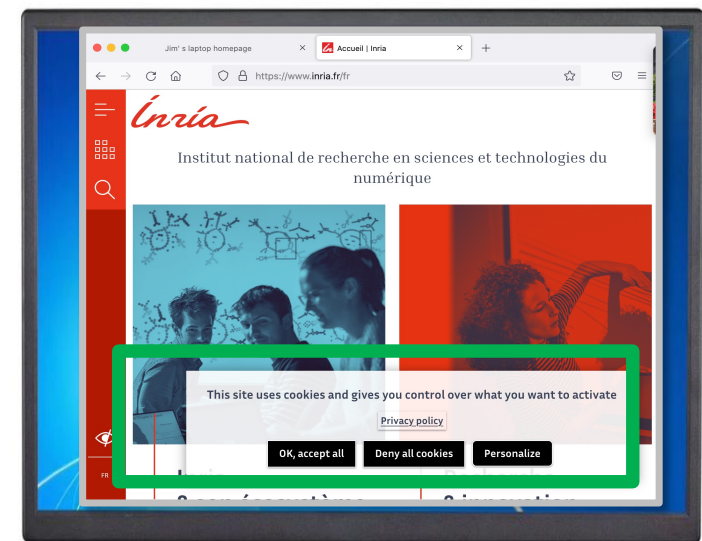
GDPR (EU General Data Protection Regulation) and cookies

“Natural persons may be associated with online identifiers [...] such as internet protocol addresses, cookie identifiers or other identifiers [...]. This may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them.”

GDPR, recital 30 (May 2018)



when cookies can identify an individual, cookies are considered personal data, subject to GDPR personal data regulations

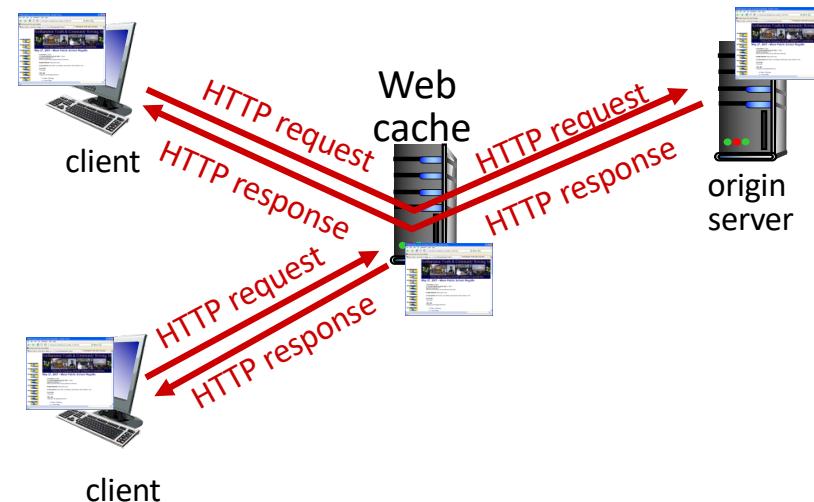


User has explicit control over whether or not cookies are allowed

Web caches

Goal: satisfy client requests without involving origin server

- user configures browser to point to a (local) *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 (aka proxy servers)

- Web cache acts as both client and server
 - server for original requesting client
 - client to origin server
- server tells cache about object's allowable caching in response header:

```
Cache-Control: max-age=<seconds>
```

```
Cache-Control: no-cache
```

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

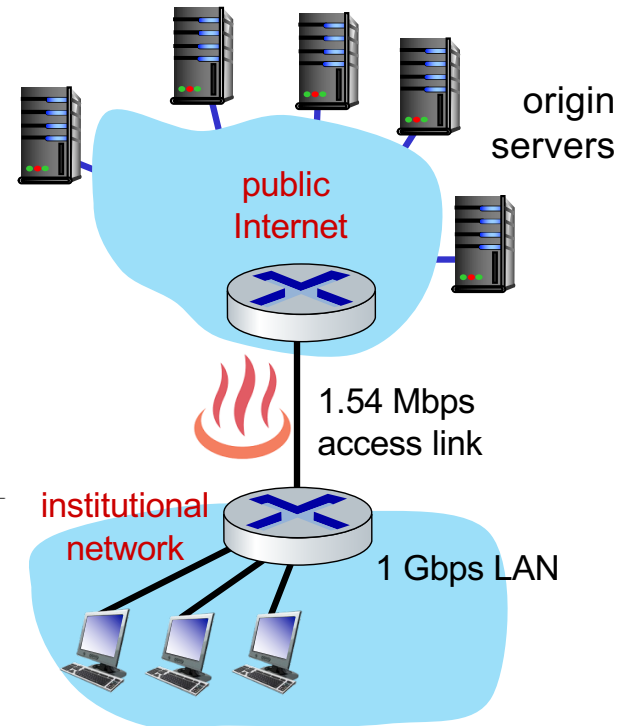
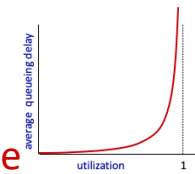
Caching example

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
 - avg data rate to browsers: 1.50 Mbps

Performance:

- access link utilization = .97 *problem: large queueing delays at high utilization!*
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay
= 2 sec + minutes + usecs



Option 1: buy a faster access link

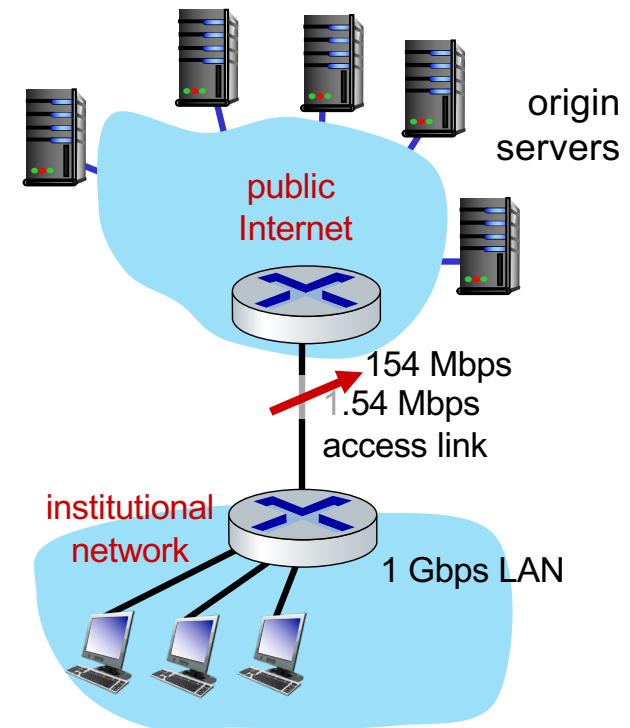
Scenario:

- access link rate: ~~1.54 Mbps~~ \rightarrow 154 Mbps
- RTT from institutional router to server: 2 sec
- web object size: 100K bits
- average request rate from browsers to origin servers: 15/sec
 - avg data rate to browsers: 1.50 Mbps

Performance:

- access link utilization = ~~.97~~ \rightarrow .0097
- LAN utilization: .0015
- end-end delay = Internet delay +
access link delay + LAN delay
= 2 sec + ~~minutes~~ + usecs

Cost: faster access link (expensive!) \rightarrow msecs



Option 2: install a web cache

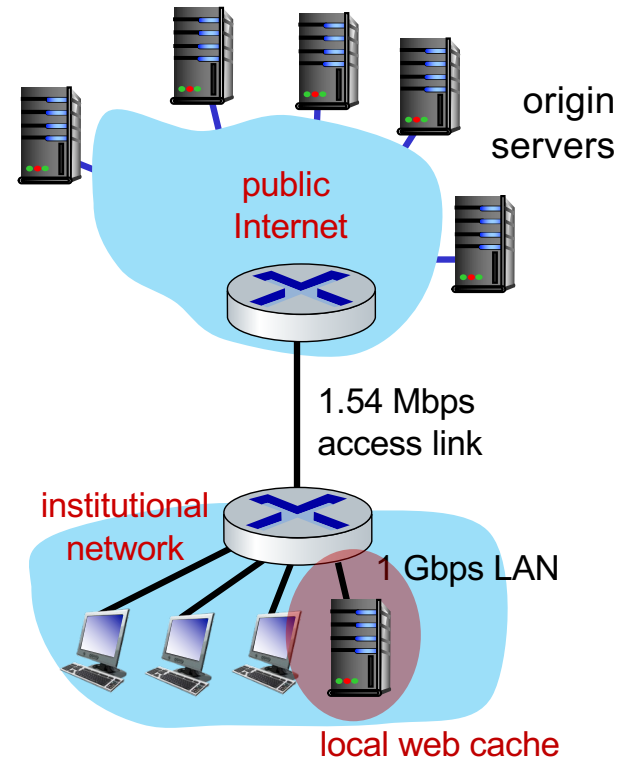
Scenario:

- 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
 - avg data rate to browsers: 1.50 Mbps

Cost: web cache (cheap!)

Performance:

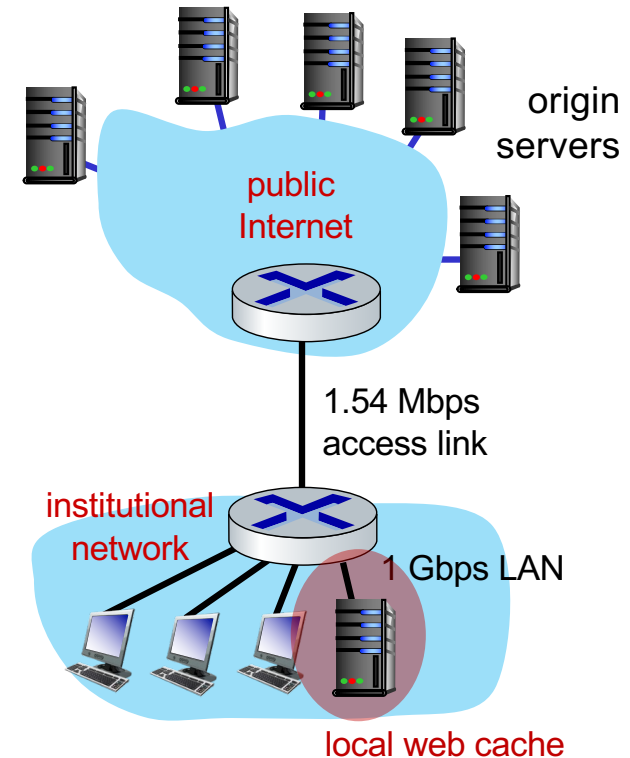
- LAN utilization: .?
 - access link utilization = ?
 - average end-end delay = ?
- How to compute link utilization, delay?*



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

suppose cache hit rate is 0.4:

- 40% requests served by cache, with low (msec) delay
- 60% requests satisfied at origin
 - rate to browsers over access link
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
 - access link utilization $= 0.9/1.54 = .58$
means low (msec) queueing delay at access link
- average end-end delay:
 - $= 0.6 * (\text{delay from origin servers})$
 - $+ 0.4 * (\text{delay when satisfied at cache})$
 - $= 0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs}$

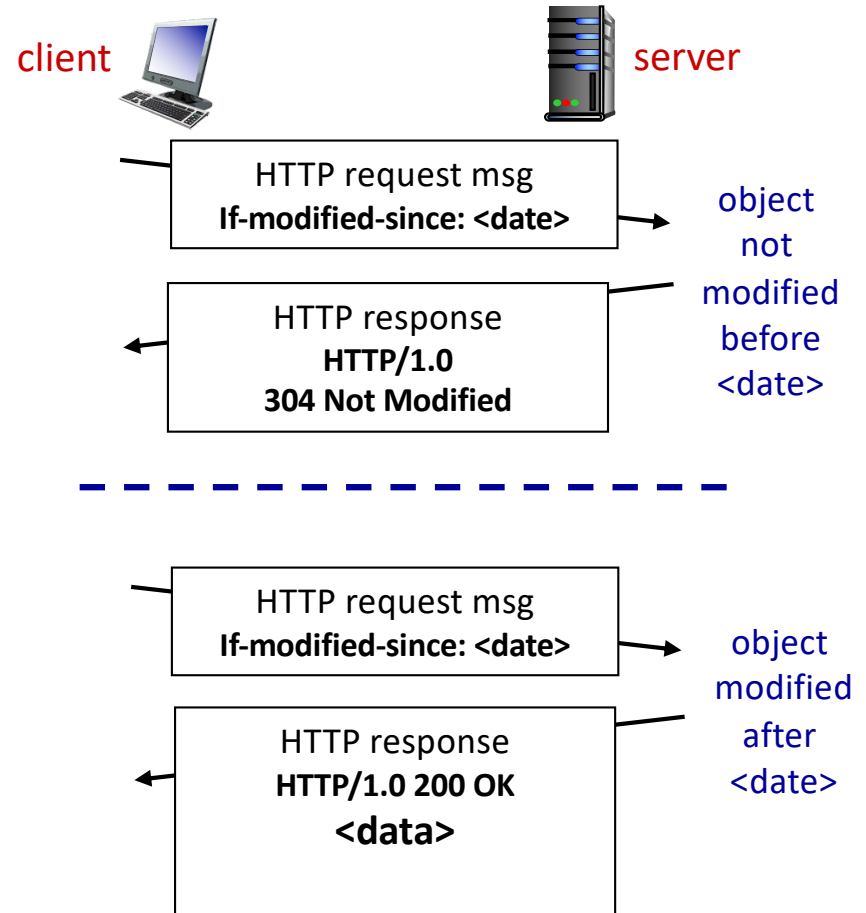


lower average end-end delay than with 154 Mbps link (and cheaper too!)

Browser caching: Conditional GET

Goal: don't send object if browser has up-to-date cached version

- no object transmission delay (or use of network resources)
- **client:** specify date of browser-cached copy in HTTP request
If-modified-since: <date>
- **server:** response contains no object if browser-cached copy is up-to-date:
HTTP/1.0 304 Not Modified

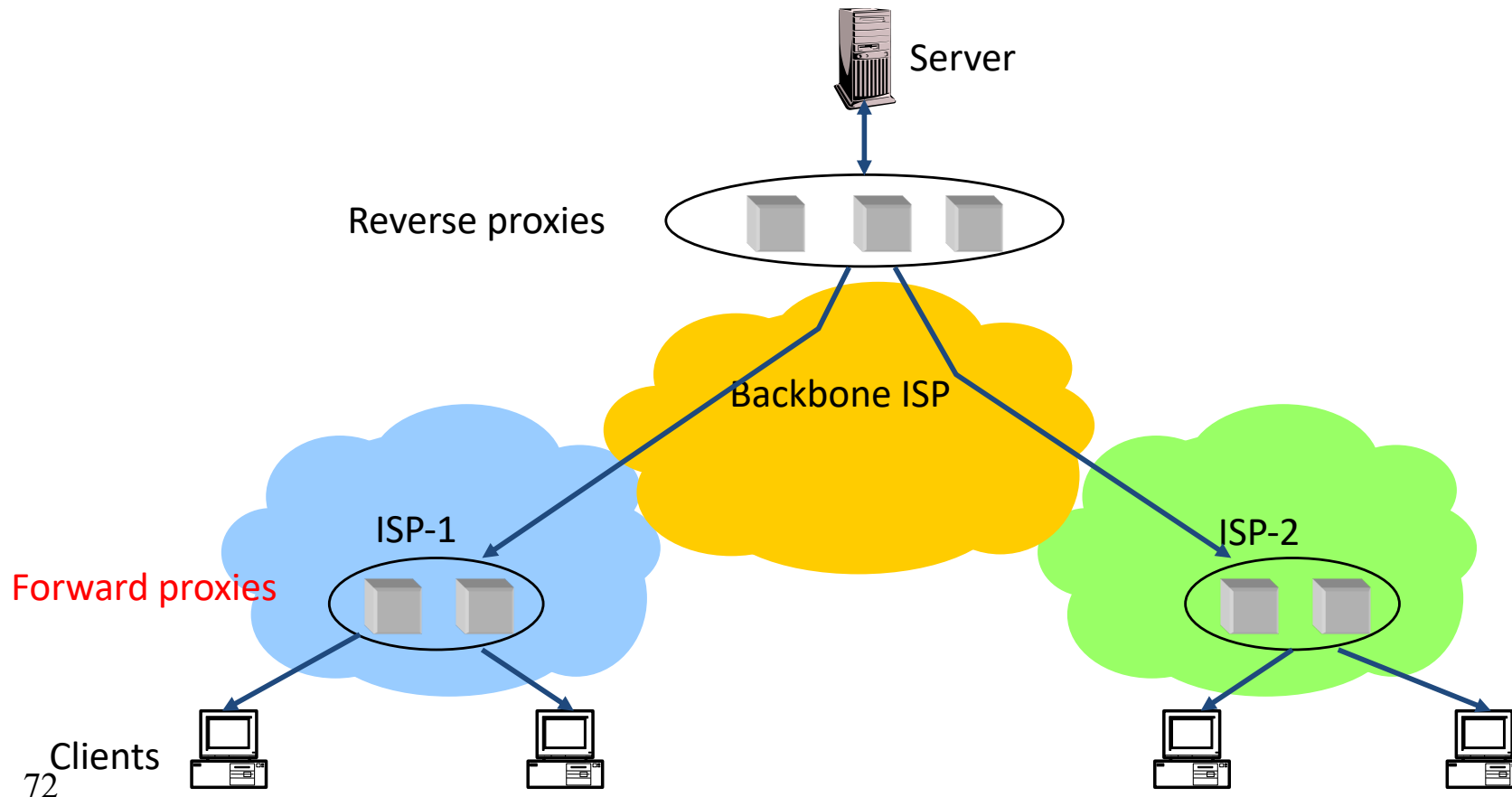


Improving HTTP Performance: Caching with Forward Proxies

Cache documents close to **clients**

→ reduce network traffic and decrease latency

- Typically done by ISPs or corporate LANs to reduce link usage



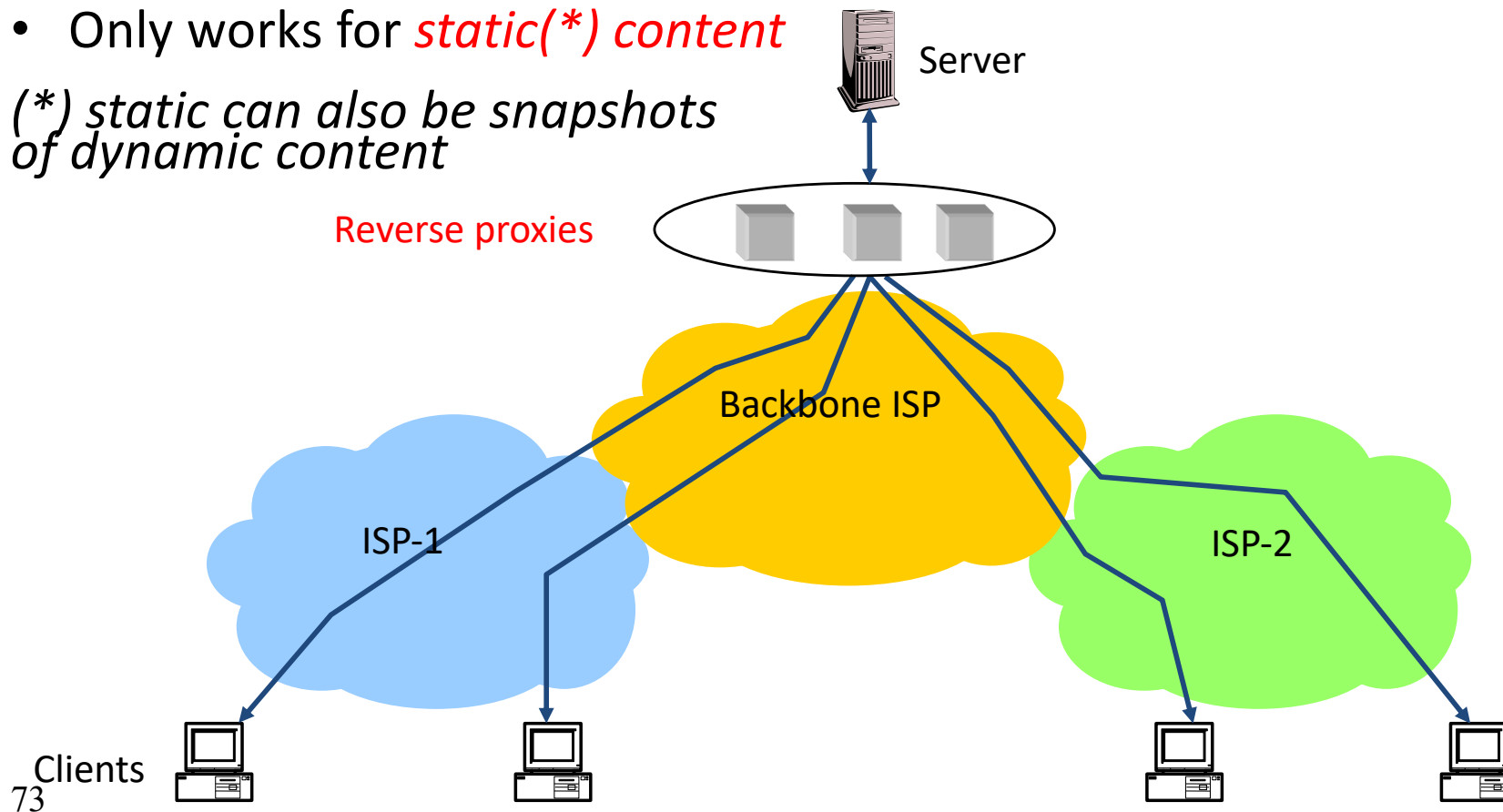
Improving HTTP Performance: Caching with Reverse Proxies

Cache documents close to **server**

→ decrease server load

- Typically done by content providers (e.g. scaling capacity for news site)
- Only works for *static(*) content*

() static can also be snapshots of dynamic content*

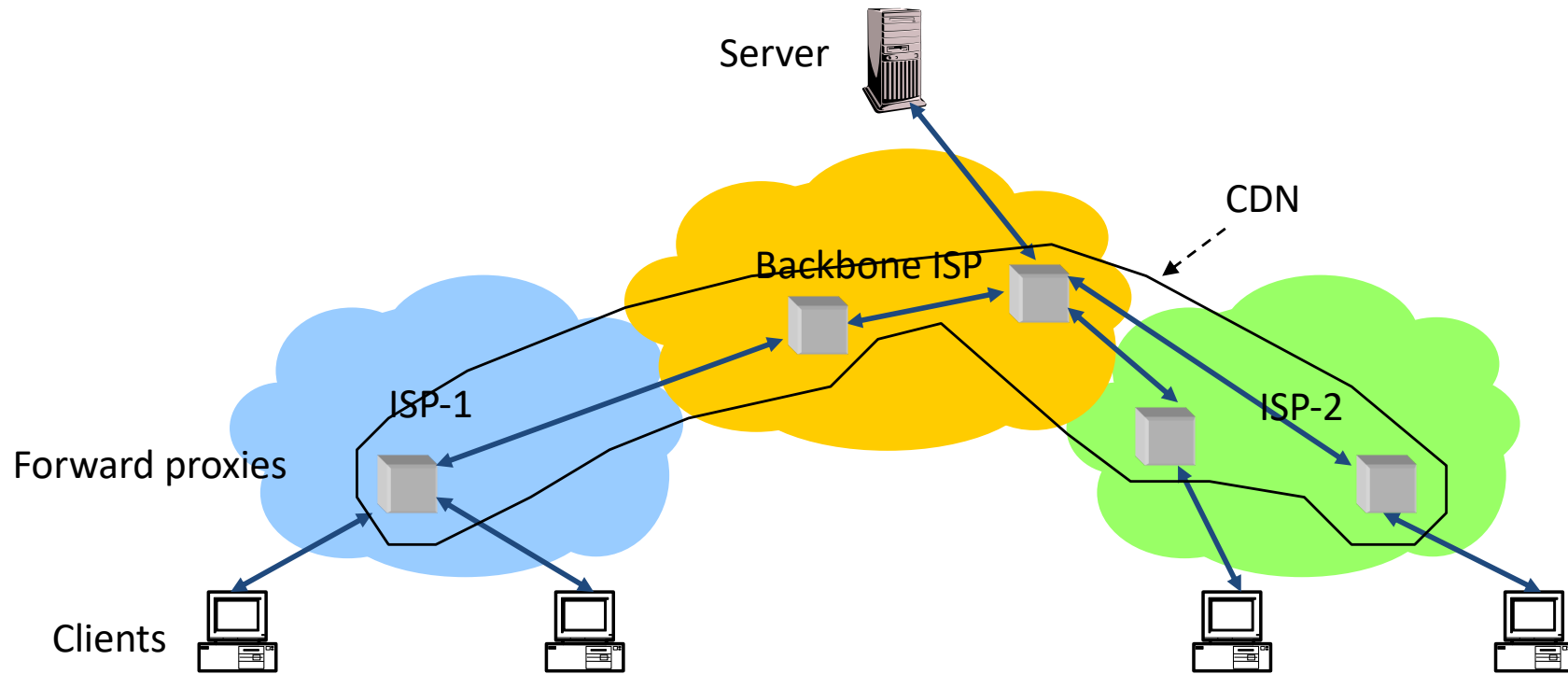


Improving HTTP Performance:

Caching w/ Content Distribution Networks

- Integrate forward and reverse caching functionality
 - One overlay network (usually) administered by one entity
 - *e.g.*, Akamai
- Provide document caching
 - **Pull**: Direct result of clients' requests
 - **Push**: Expectation of high access rate
- Also do some processing
 - Handle *dynamic* web pages
 - *Transcoding*
 - *Maybe do some security function – watermark IP*

Improving HTTP Performance: Caching with CDNs (cont.)



Improving HTTP Performance:
CDN Example – Akamai

- Akamai creates new domain names for each client content provider.
 - e.g., a128.g.akamai.net
- The CDN's DNS servers are authoritative for the new domains
- The client content provider modifies its content so that embedded URLs reference the new domains.
 - “Akamaize” content
 - e.g.: <http://www.bbc.co.uk/popular-image.jpg> becomes <http://a128.g.akamai.net/popular-image.jpg>
- *Requests now sent to CDN's infrastructure...*

Hosting: Multiple Sites Per Machine

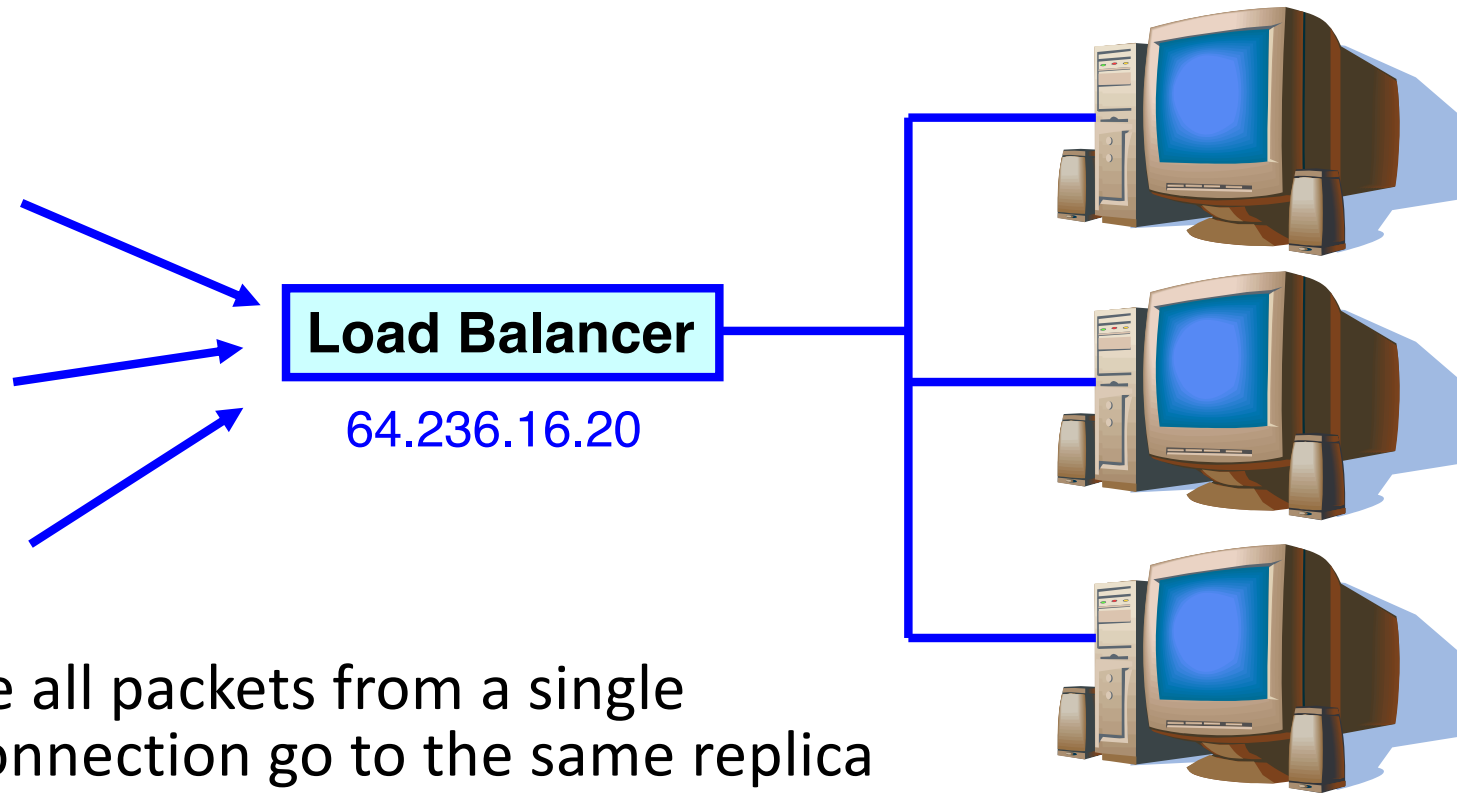
- Multiple Web sites on a single machine
 - Hosting company runs the Web server on behalf of multiple sites (*e.g.*, `www.foo.com` and `www.bar.com`)
- Problem: `GET /index.html`
 - `www.foo.com/index.html` **or** `www.bar.com/index.html`?
- Solutions:
 - Multiple server processes on the same machine
 - Have a separate IP address (or port) for each server
 - Include site name in HTTP request
 - Single Web server process with a single IP address
 - Client includes “Host” header (*e.g.*, `Host: www.foo.com`)
 - *Required header* with HTTP/1.1

Hosting: Multiple Machines Per Site

- Replicate popular Web site across many machines
 - Helps to handle the load
 - Places content closer to clients
- Helps when content isn't cacheable
- Problem: Want to direct client to particular replica
 - Balance load across server replicas
 - Pair clients with nearby servers

Multi-Hosting at Single Location

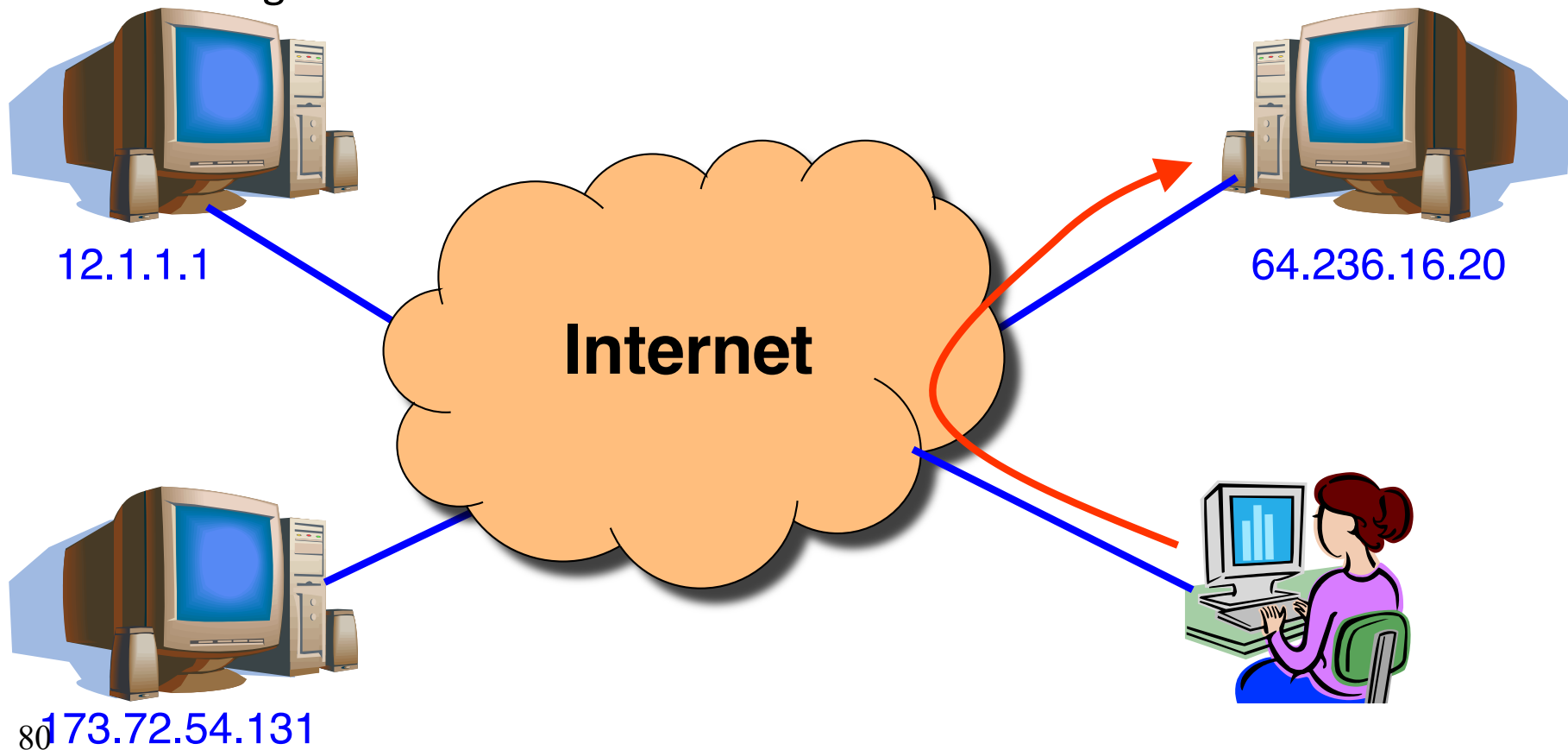
- Single IP address, multiple machines
 - Run multiple machines behind a single IP address



- Ensure all packets from a single TCP connection go to the same replica

Multi-Hosting at Several Locations

- Multiple addresses, multiple machines
 - Same name but different addresses for all of the replicas
 - Configure DNS server to return *closest* address



CDN examples round-up

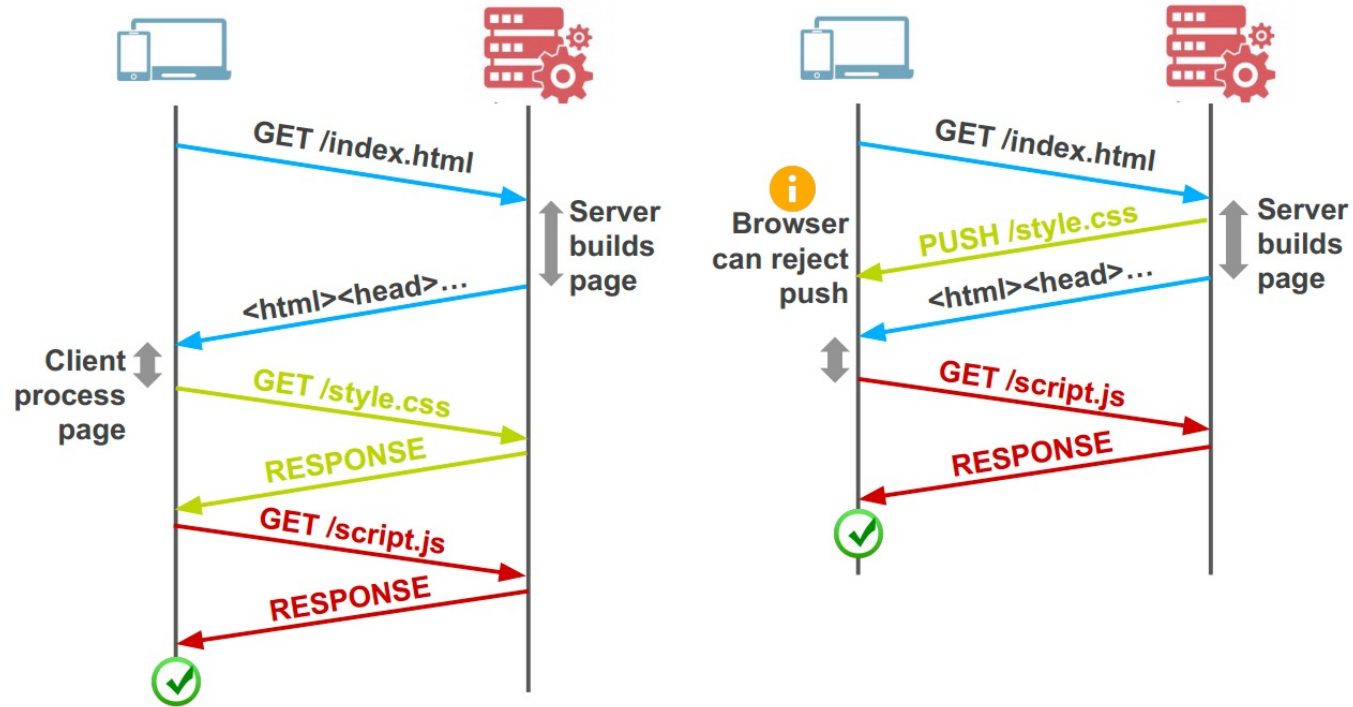
- **CDN using DNS**
DNS has information on loading/distribution/location
(akami uses this one)
- **CDN using anycast**
same address from DNS name but local routes
(ROOT DNS servers and 8.8.8.8 use this one)
- **CDN based on rewriting HTML URLs**
(akami example in previous slides)

After HTTP/1.1

SPDY (speedy) and its moral successor HTTP/2

- Binary protocol
- Multiplexing
- Priority control over Frames
- Header Compression
- Server Push

After HTTP/1.1



- Server Push
 - Proactively push stuff to client that it will need

After HTTP/1.1

SPDY (speedy) and its moral successor HTTP/2

- Binary protocol
 - More efficient to parse
 - More compact on the wire
 - Much less error prone as compared
 - to textual protocols

Wireshark decoders for the win

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

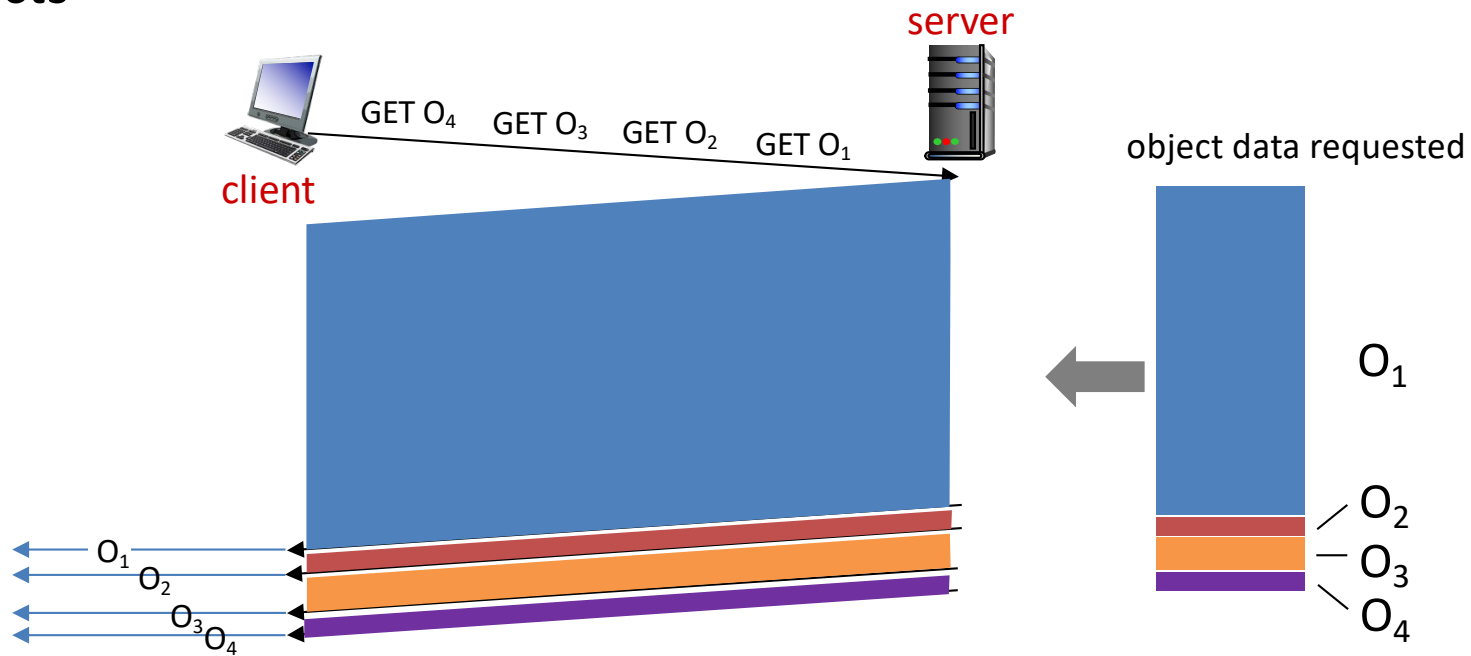
Key goal: decreased delay in multi-object HTTP requests

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

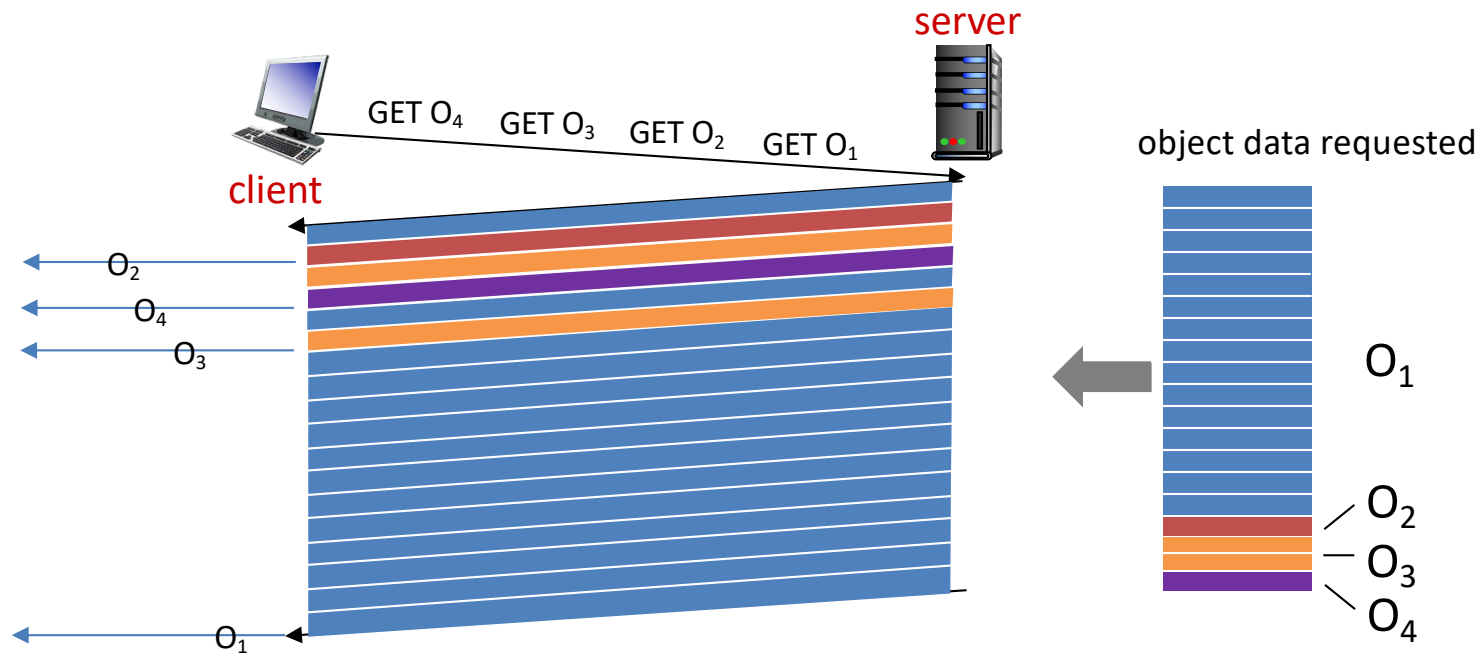
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



objects delivered in order requested: O_2 , O_3 , O_4 wait behind O_1

HTTP/2: mitigating HOL blocking

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



O₂, O₃, O₄ delivered quickly, O₁ slightly delayed

HTTP/2 to HTTP/3

HTTP/2 over single TCP connection means:

- recovery from packet loss still stalls all object transmissions
 - as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- no security over vanilla TCP connection
- **HTTP/3**: adds security, per object error- and congestion-control (more pipelining) over UDP

As at ~~2021~~ when I last looked



Other ongoing work includes QUIC for datagrams

Seriously! It adds QUIC crypto to “UDP” so isn’t totally silly.

Add QUIC and stir...

Quick UDP Internet Connections

Objective: Combine speed of UDP protocol with TCP's reliability

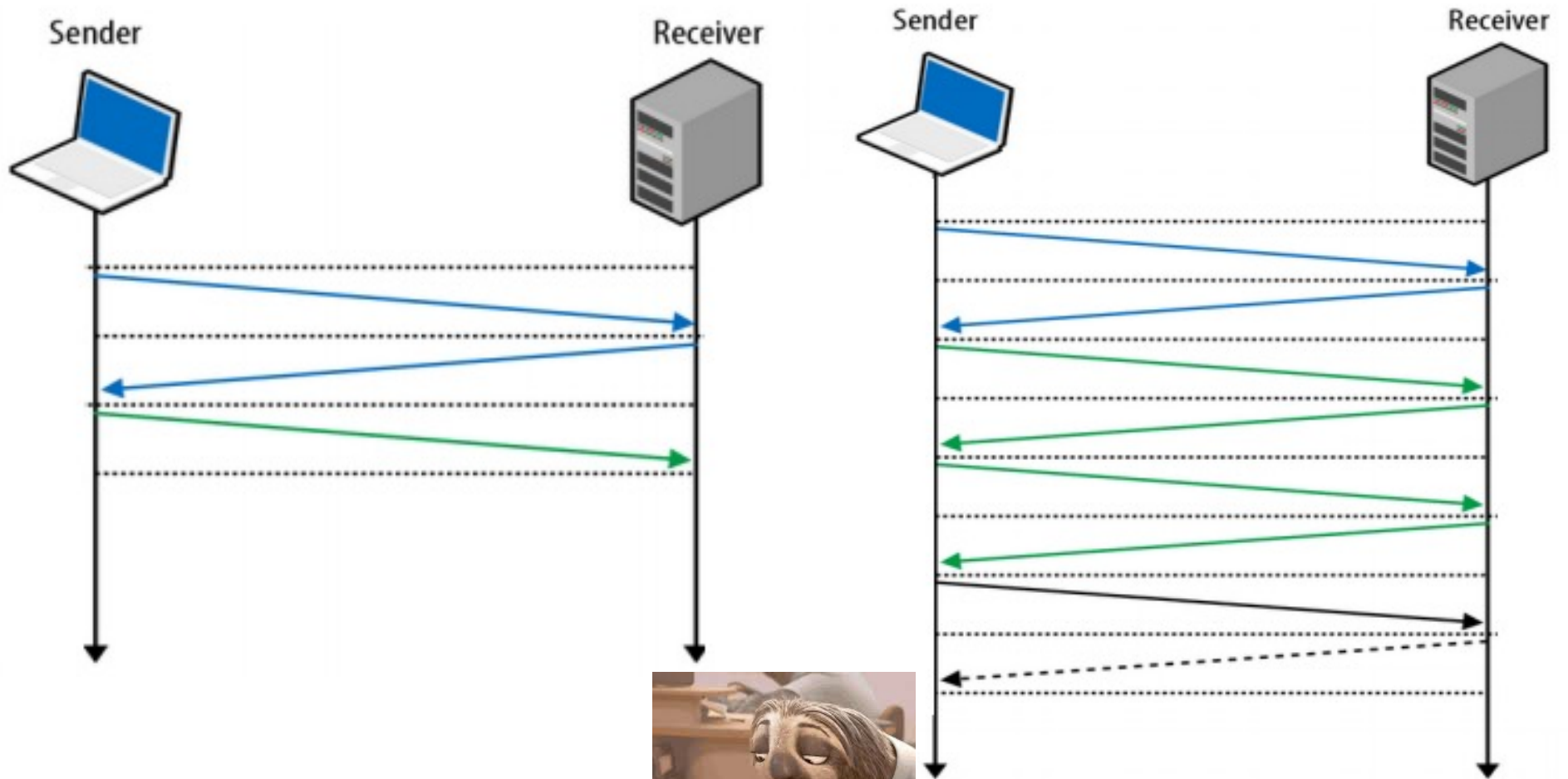
Problem: Very hard to make changes to TCP

- *Faster to implement new protocol on top of UDP*
- (Roll out features in TCP if they prove theory)

QUIC (First presented to IETF in ~2013):

- Reliable transport over UDP
- Uses FEC
- Default crypto
- Restartable connections

3-Way Handshake



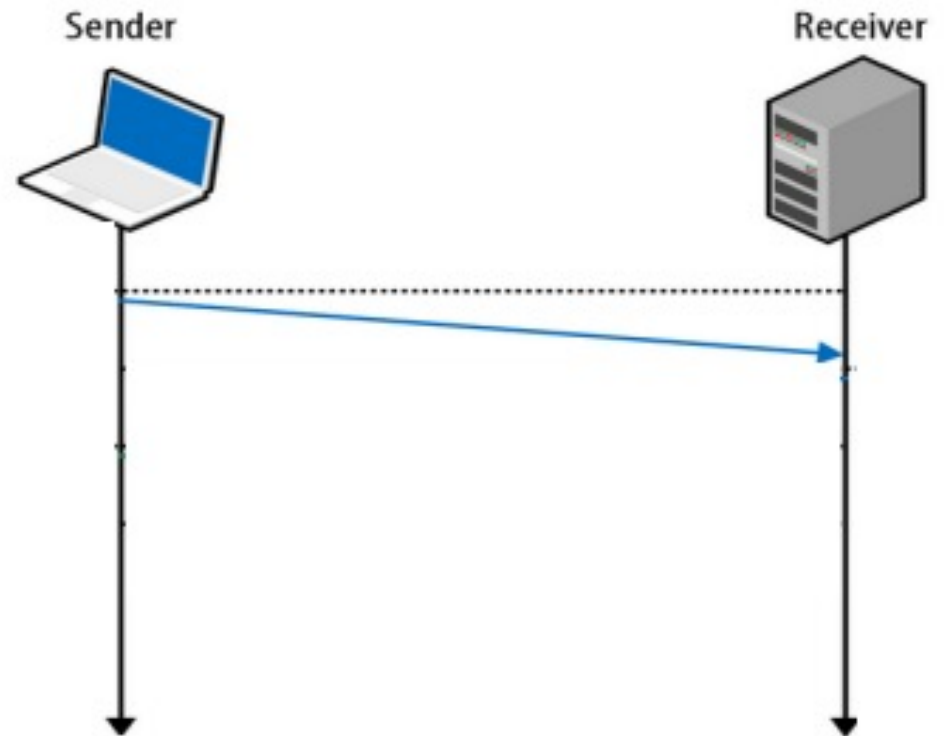
Without TLS



With TLS

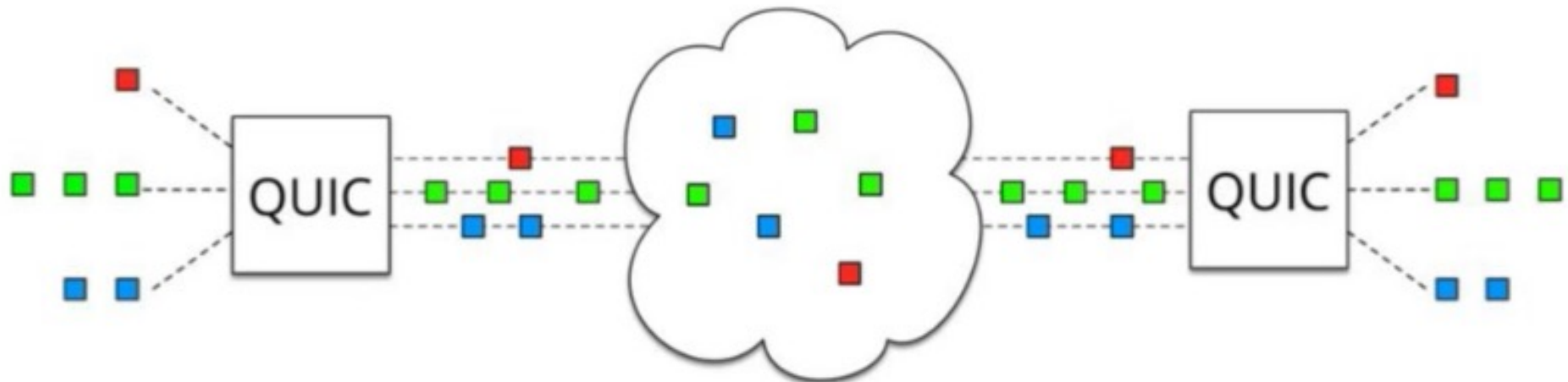
UDP

- Fire and forget
 - Less time spent to validate packets
 - Downside - no reliability, this has to be added on top of UDP



QUIC

- UDP does NOT depend on order of arriving packets
- Lost packets will only impact an individual resource, e.g., CSS or JS file.
- QUIC combined the best parts of HTTP/2 over UDP:
 - Multiplexing on top of non-blocking transport protocol



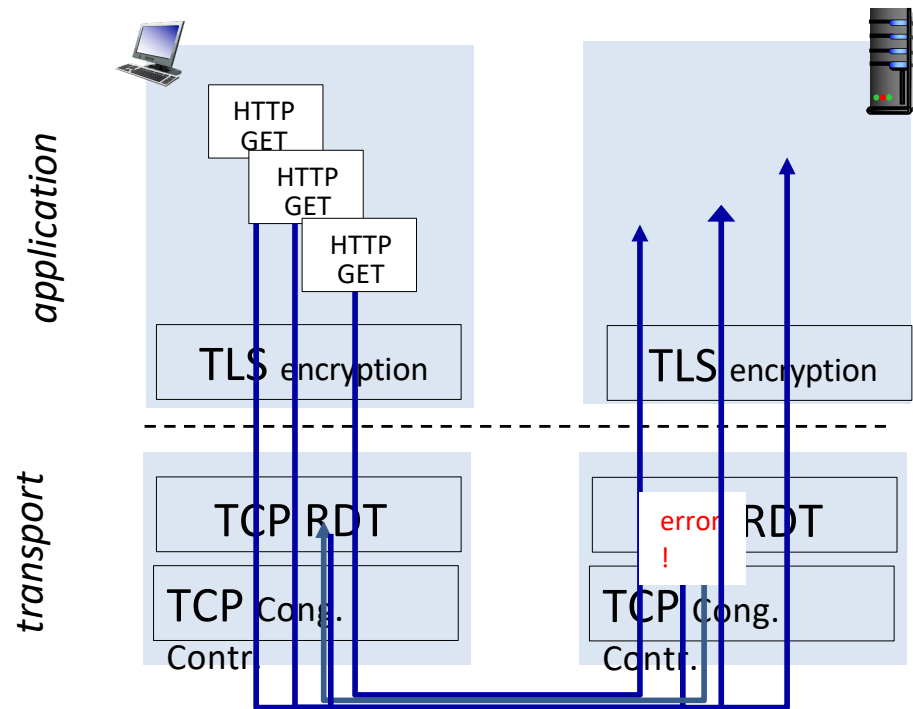
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this topic for connection establishment, error control, congestion control

- **error and congestion control:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT
- multiple application-level “streams” multiplexed over single QUIC connection
 - separate reliable data transfer, security
 - common congestion control

QUIC: streams – parallelism

no HOL blocking in transport or application



(a) HTTP 1.1

QUIC – more than just UDP

- QUIC outshines TCP under poor network conditions, shaving a full second off the Google Search page load time for the slowest 1% of connections.
- These benefits are even more apparent for video services like YouTube
 - Users report 30% fewer rebuffers with QUIC.

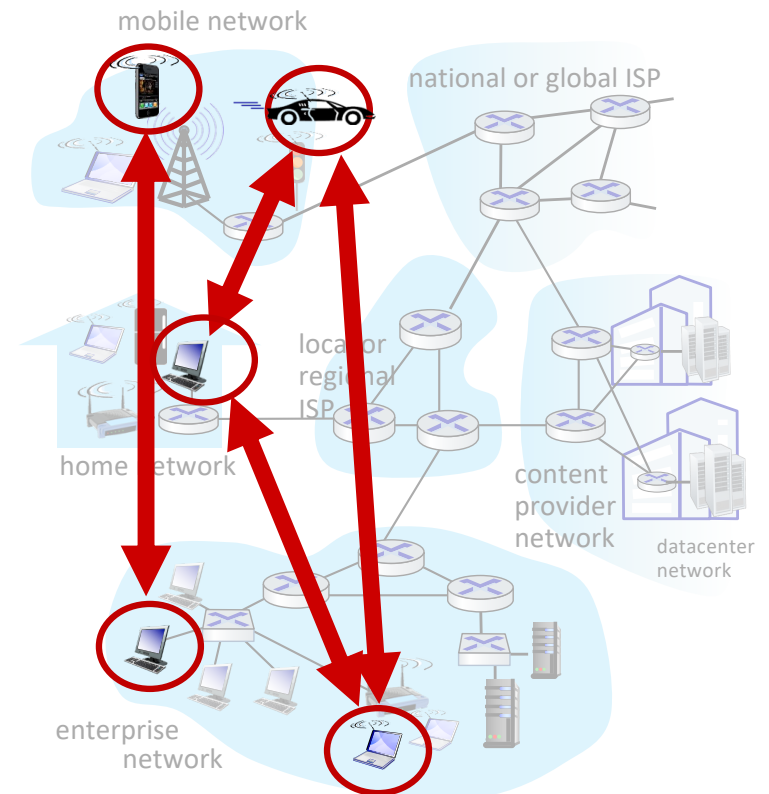
Why QUIC over UDP and not a new proto

- IP proto value for new transport layer
- Change the protocol – risk the wraith of
 - Legacy code
 - Firewalls
 - Load-balancer
 - NATs (the high-priest of middlebox)
- Same problem faces any significant TCP change

Every host is a server:
Peer-2-Peer

Peer-to-peer (P2P) architecture

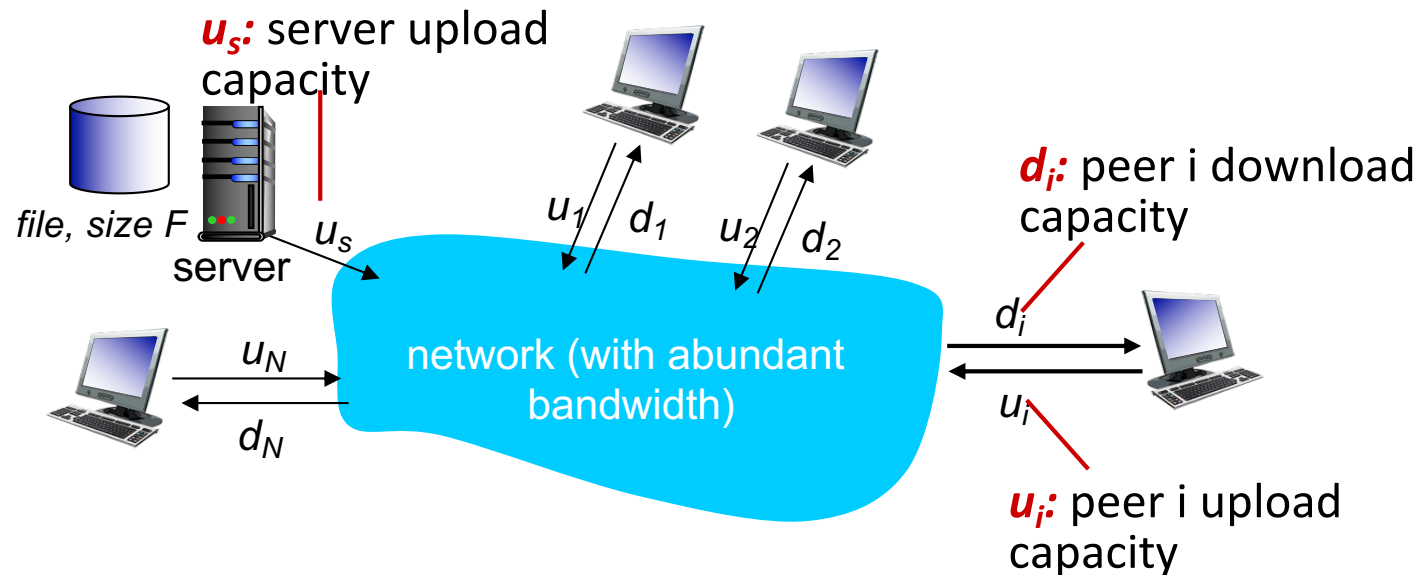
- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
 - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



File distribution: client-server vs P2P

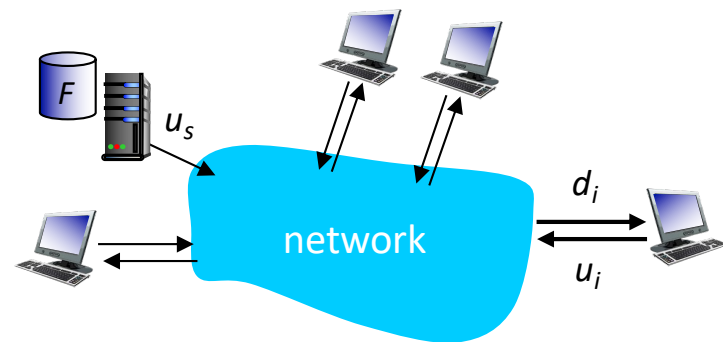
Q: how much time to distribute file (size F) from one server to N peers?

- peer upload/download capacity is limited resource



File distribution time: client-server

- **server transmission:** must sequentially send (upload) N file copies:
 - time to send one copy: F/u_s
 - time to send N copies: NF/u_s
- **client:** each client must download file copy
 - d_{min} = min client download rate
 - min client download time: F/d_{min}



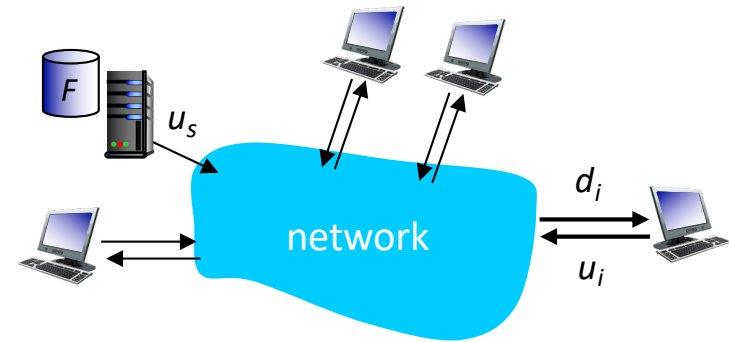
*time to distribute F
to N clients using
client-server approach*

$$D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- **server transmission:** must upload at least one copy:
 - time to send one copy: F/u_s
- **client:** each client must download file copy
 - min client download time: F/d_{min}
- **clients:** as aggregate must download NF bits
 - max upload rate (limiting max download rate) is $u_s + \sum u_i$



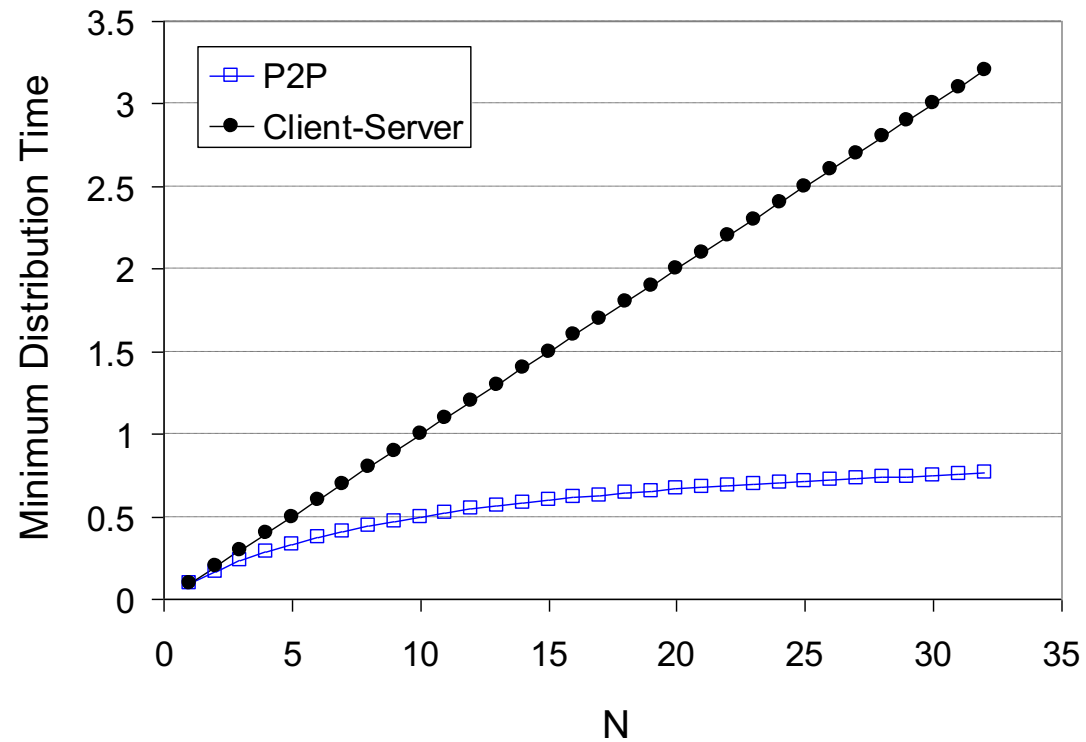
time to distribute F to N clients using P2P approach

$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

increases linearly in N ...
 ... but so does this, as each peer brings service capacity

Client-server vs. P2P: example

client upload rate = u , $F/u = 1$ hour, $u_s = 10u$, $d_{min} \geq u_s$

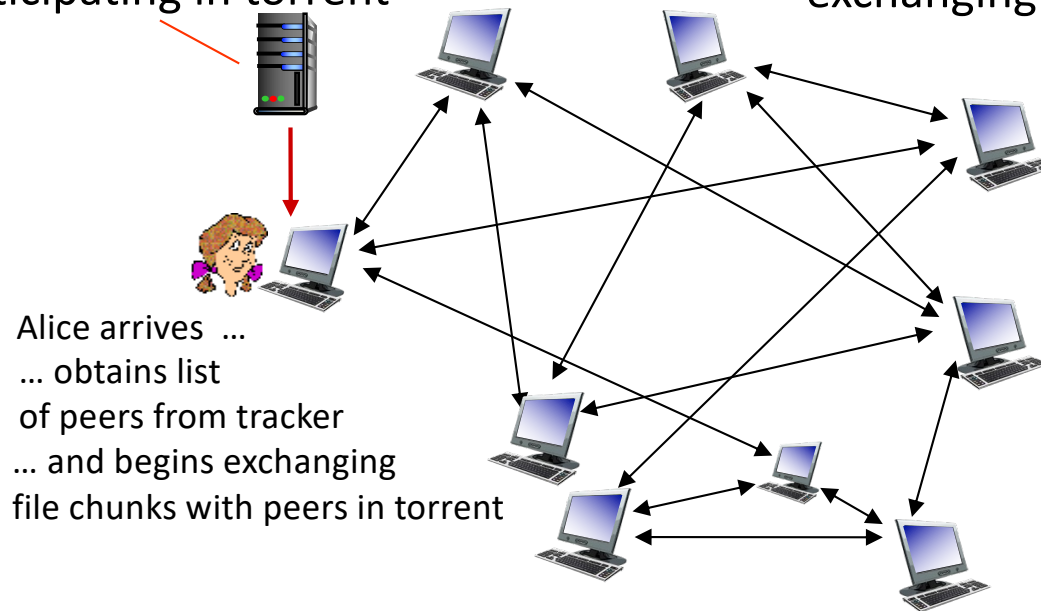


P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

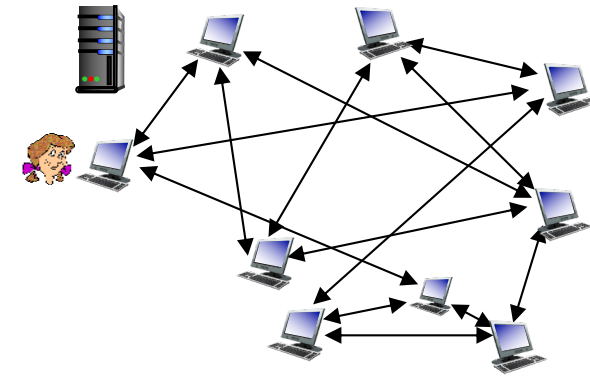
tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file



P2P file distribution: BitTorrent

- peer joining torrent:
 - has no chunks, but will accumulate them over time from other peers
 - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- peer exchanges prioritize rarer blocks
- *churn*: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent



BitTorrent: requesting, sending file chunks

Requesting chunks:

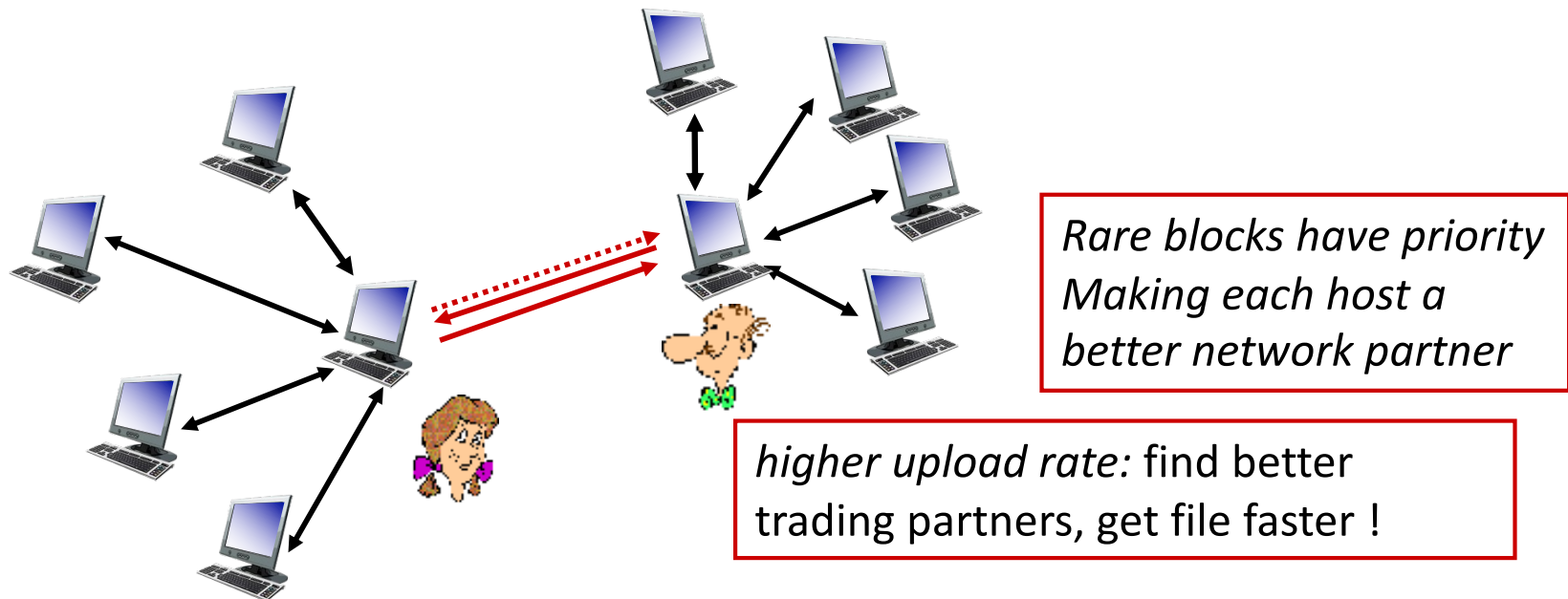
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

Sending chunks: tit-for-tat

- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
 - other peers are choked by Alice (do not receive chunks from her)
 - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
 - “optimistically unchoke” this peer
 - newly chosen peer may join top 4

BitTorrent: tit-for-tat

- (1) Alice “optimistically unchokes” Bob
- (2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
- (3) Bob becomes one of Alice’s top-four providers



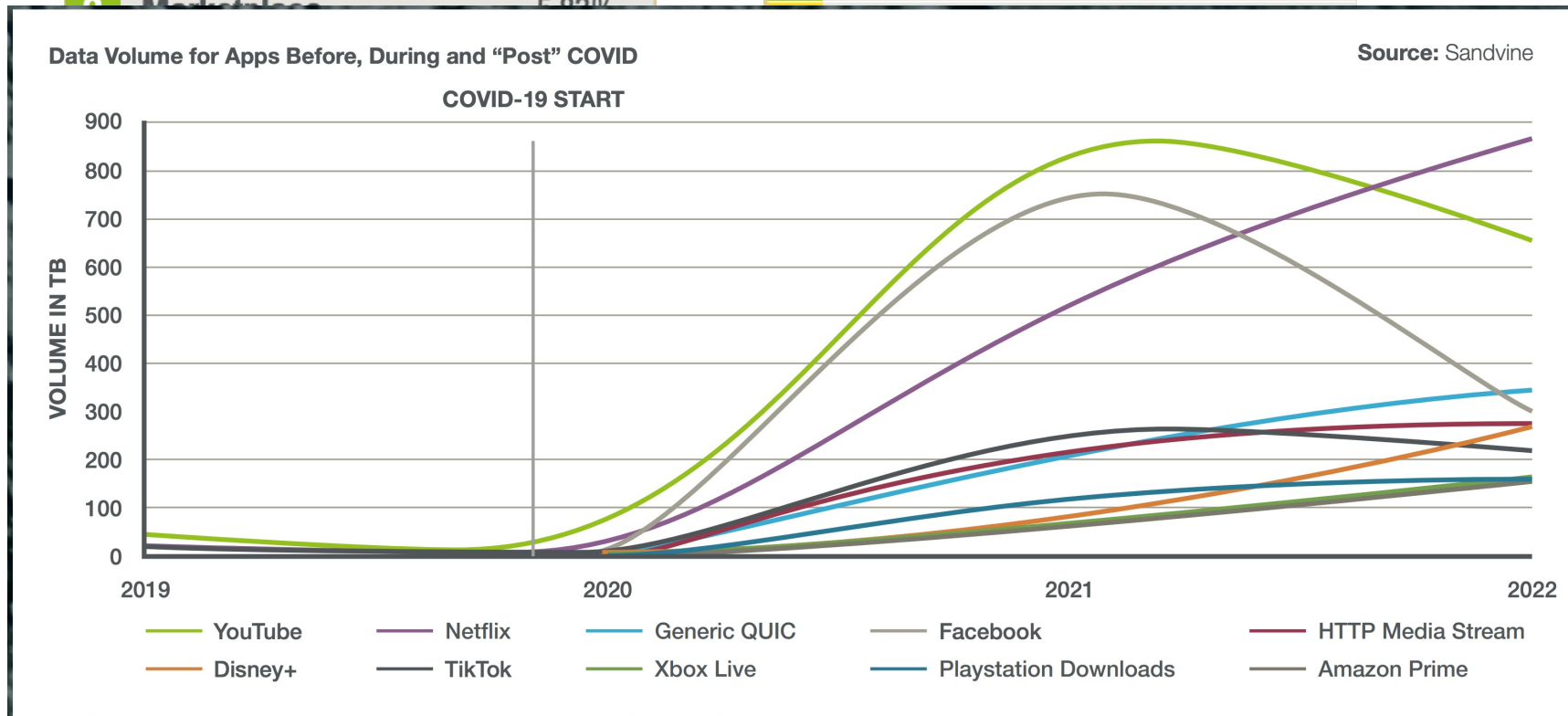
Internet

(current data is \$\$\$ or hard to get)

This info taken from an annual Sandvine report for 2022 <https://www.sandvine.com>

APP CATEGORY TOTAL VOLUME	
2022 Categories	Total Volume
1 Video	65.93%
2 Marketplace	5.92%

TOTAL TRAFFIC	
Application	Total Volume
1 Netflix	13.74%
2 YouTube	10.51%



Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
 - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)

- *challenge*: scale - how to reach ~1B users?

- *challenge*: heterogeneity

- different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)

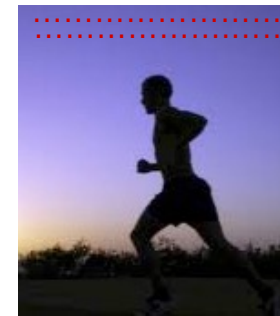
- *solution*: distributed, application-level infrastructure



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

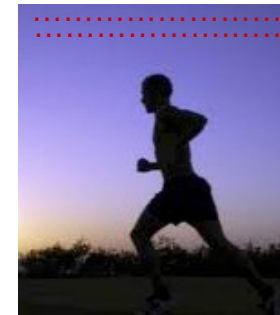


frame $i+1$

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 1 (CD-ROM) 1.5 Mbps
 - MPEG2 (DVD) 3-6 Mbps
 - MPEG4 (often used in Internet, 64Kbps – 12 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

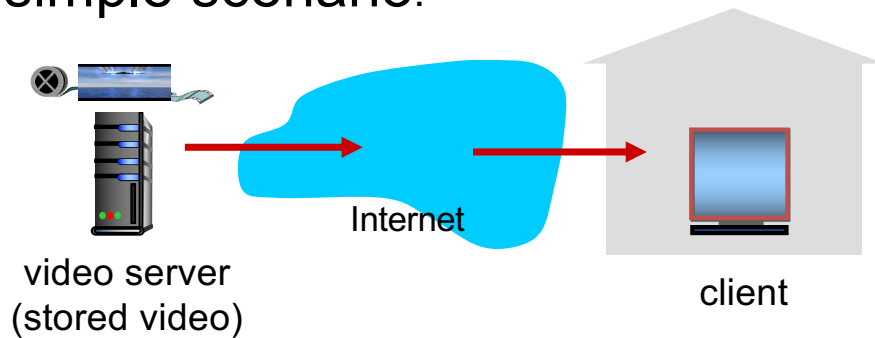
temporal coding example: instead of sending complete frame at $i+1$, send only differences from frame i



frame $i+1$

Streaming stored video

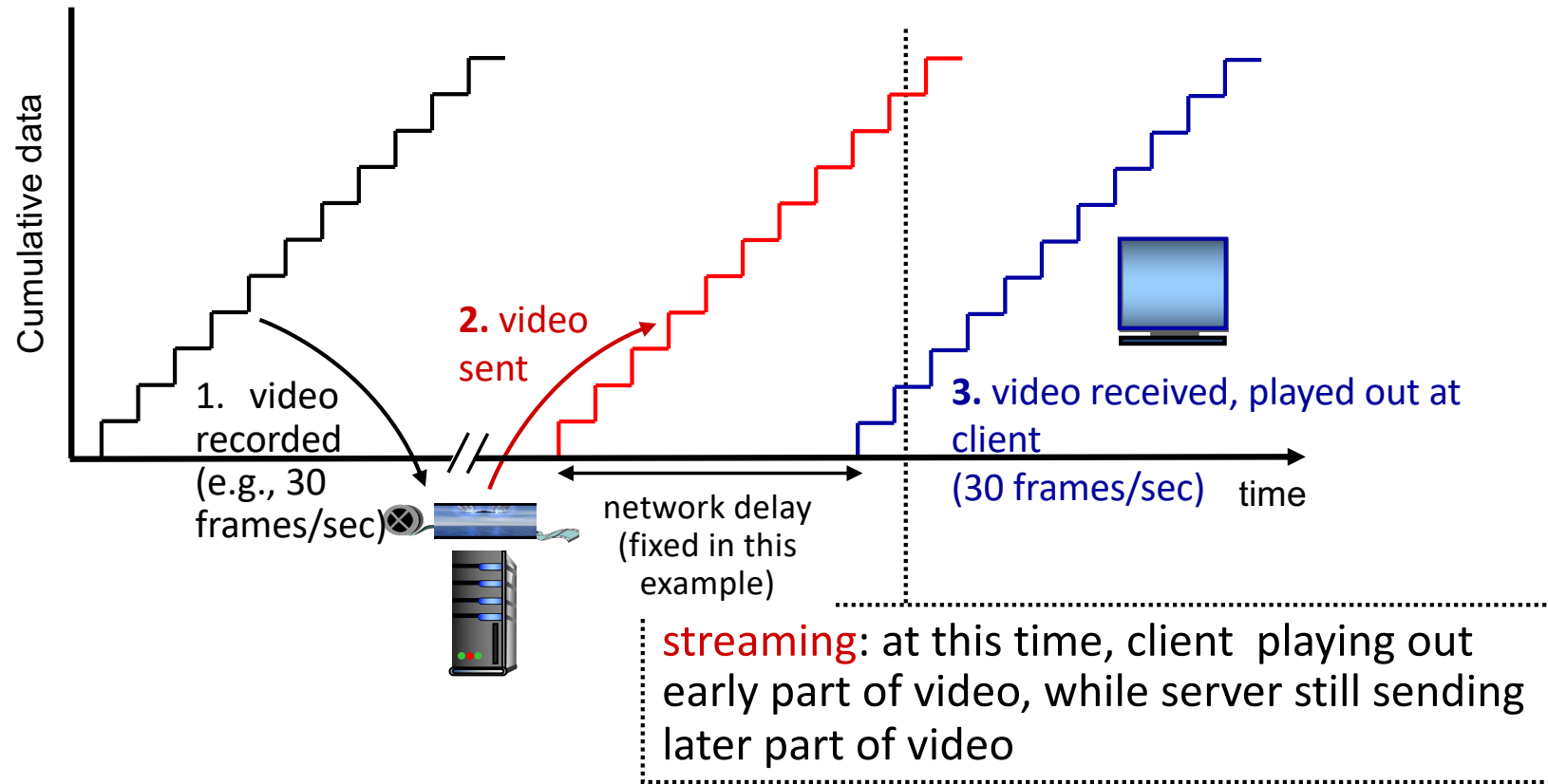
simple scenario:



Main challenges:

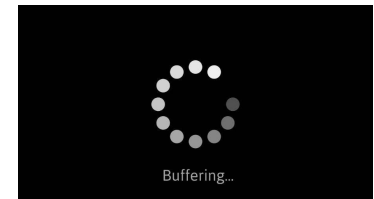
- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, access network, network core, video server)
- packet loss, delay due to congestion will delay playout, or result in poor video quality

Streaming stored video

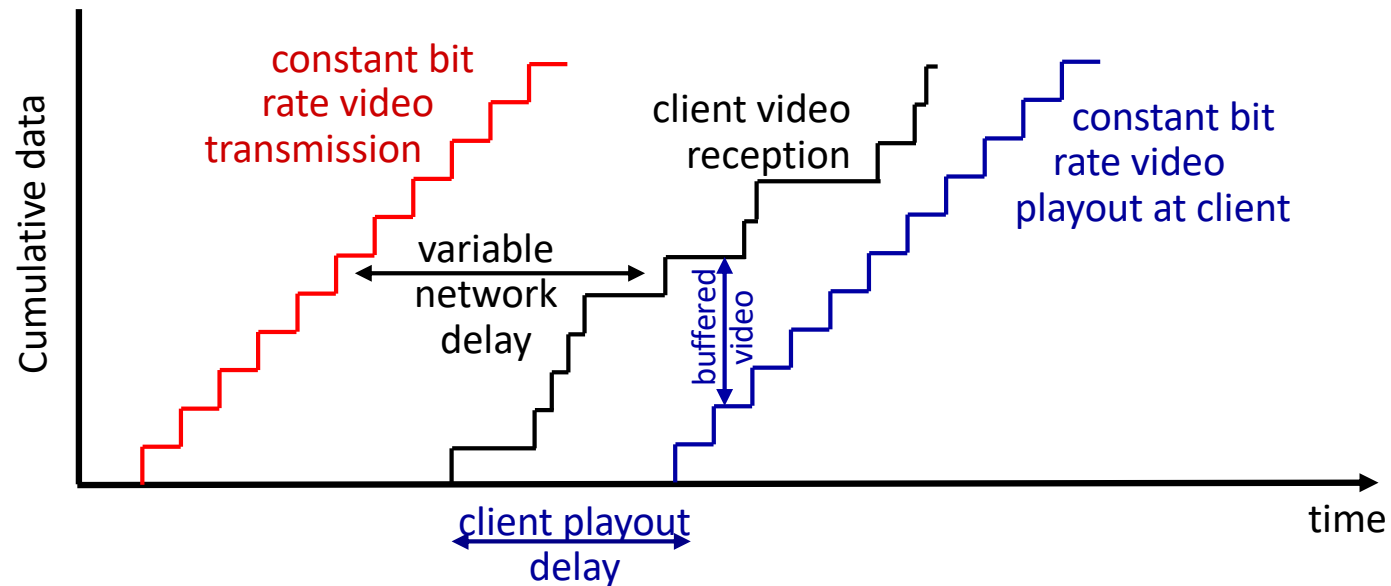


Streaming stored video: challenges

- **continuous playout constraint**: during client video playout, playout timing must match original timing
 - ... but **network delays are variable** (jitter), so will need **client-side buffer** to match continuous playout constraint
- other challenges:
 - client interactivity: pause, fast-forward, rewind, jump through video
 - video packets may be lost, retransmitted



Streaming stored video: playout buffering



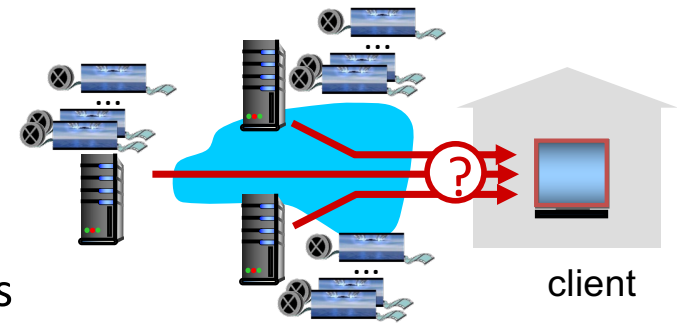
- *client-side buffering and playout delay*: compensate for network-added delay, delay jitter

Streaming multimedia: DASH

*D*ynamic, *A*daptive
*S*treaming over *H*TTP

server:

- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file*: provides URLs for different chunks

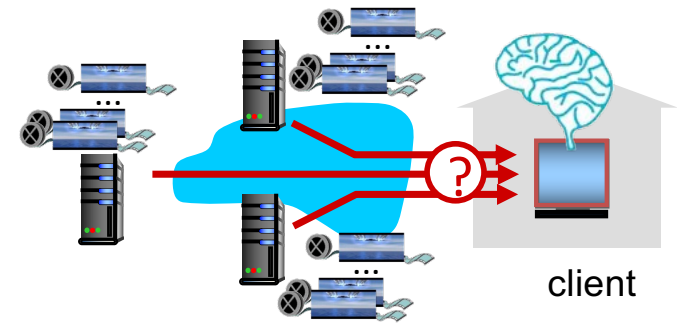


client:

- periodically estimates 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), and from different servers

Streaming multimedia: DASH

- “*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)



Streaming video = encoding + DASH + playout buffering

Content distribution networks (CDNs)

challenge: how to stream 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 (and possibly congested) path to distant clients

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

Content distribution networks (CDNs)

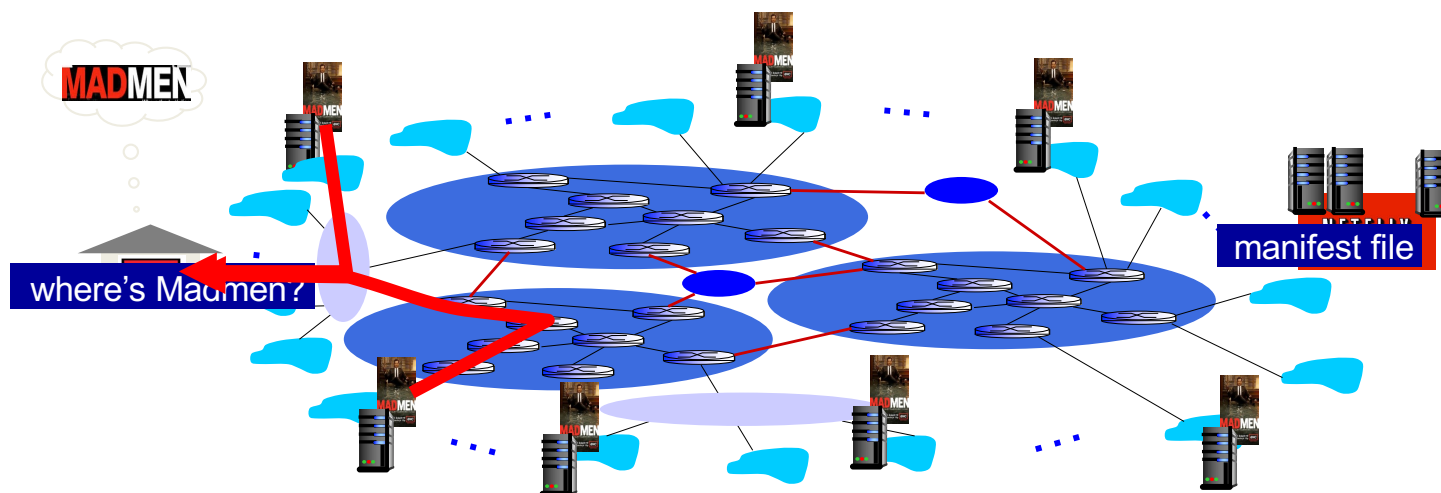
challenge: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *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
 - Akamai: 240,000 servers deployed in > 120 countries (2015)
 - *bring home*: smaller number (10's) of larger clusters in POPs near access nets
 - used by Limelight



Content distribution networks (CDNs)

- CDN: stores copies of content (e.g. MADMEN) at CDN nodes
- subscriber requests content, service provider returns manifest
 - using manifest, client retrieves content at highest supportable rate
 - may choose different rate or copy if network path congested



Content distribution networks (CDNs)



OTT challenges: coping with a congested Internet from the “edge”

- what content to place in which CDN node?
- from which CDN node to retrieve content? At which rate?

Summary

- Applications have protocols too
- We covered examples from
 - Traditional Applications (web)
 - Scaling and Speeding the web (CDN/Cache tricks)
- Infrastructure Services (DNS)
 - Cache and Hierarchy
- P2P Network examples
- Evolving standards (Email)
- Video CDN Stream challenges

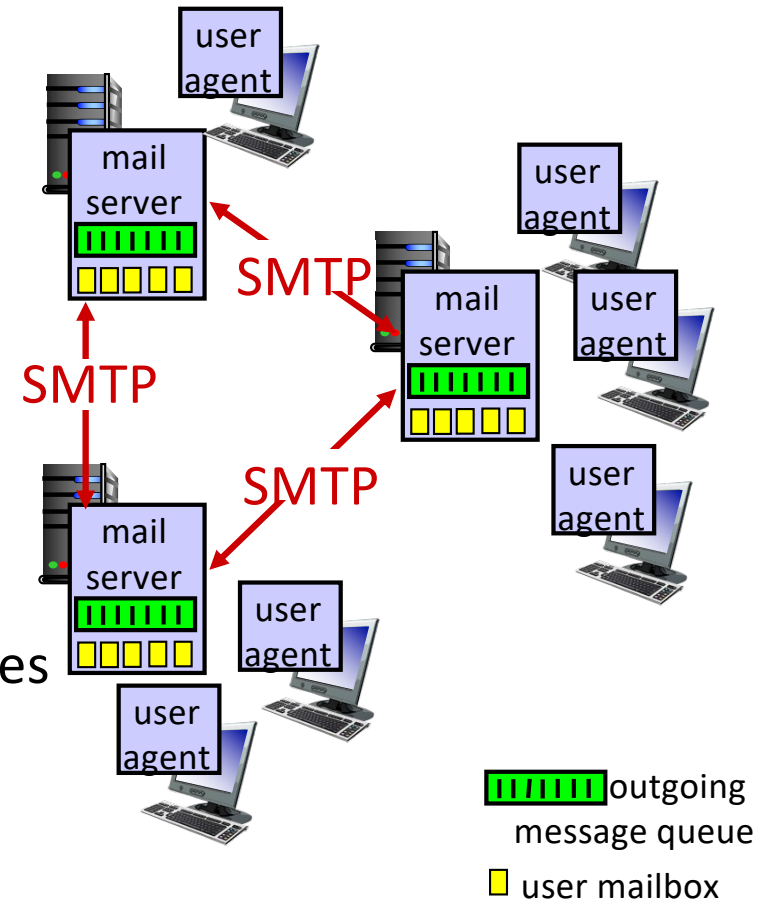
E-mail

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



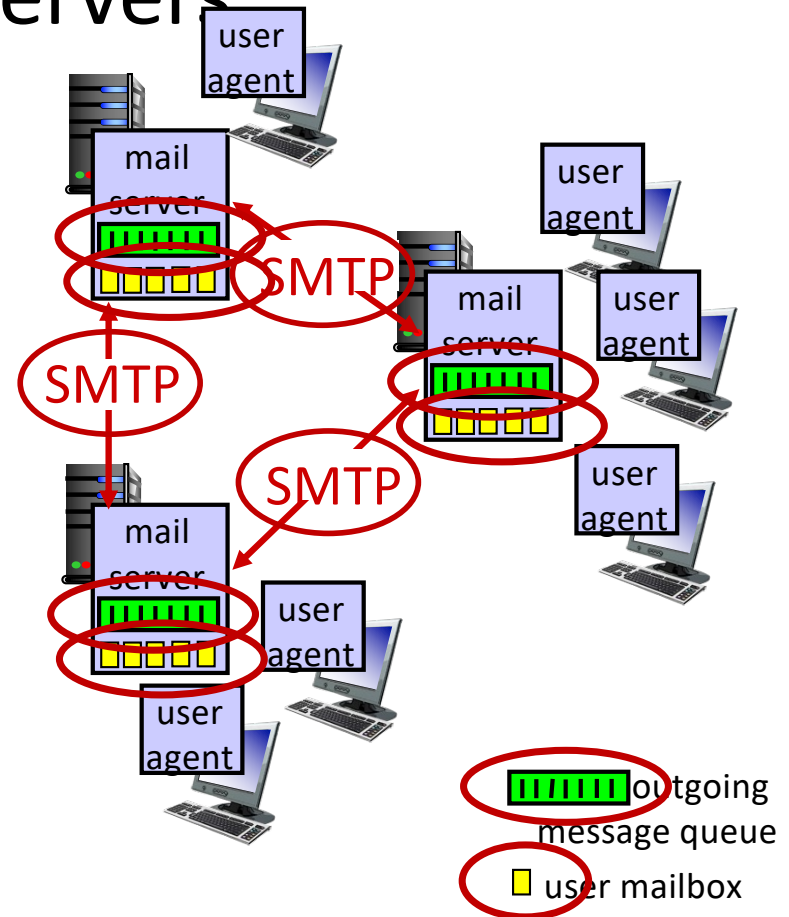
E-mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages

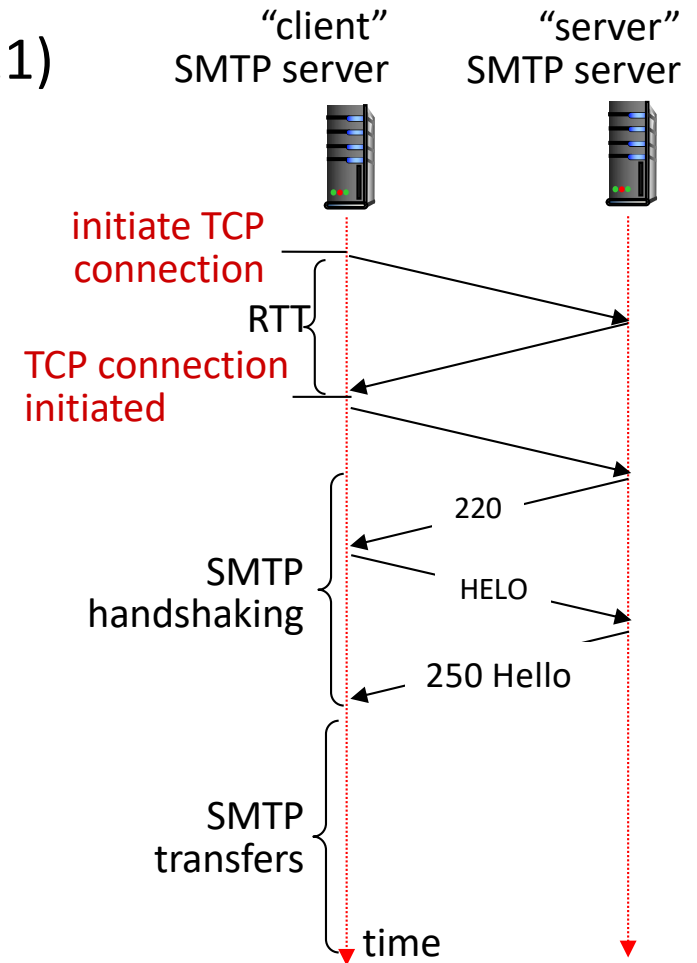
SMTP protocol between mail servers to send email messages

- **client**: sending mail server
- **“server”**: receiving mail server



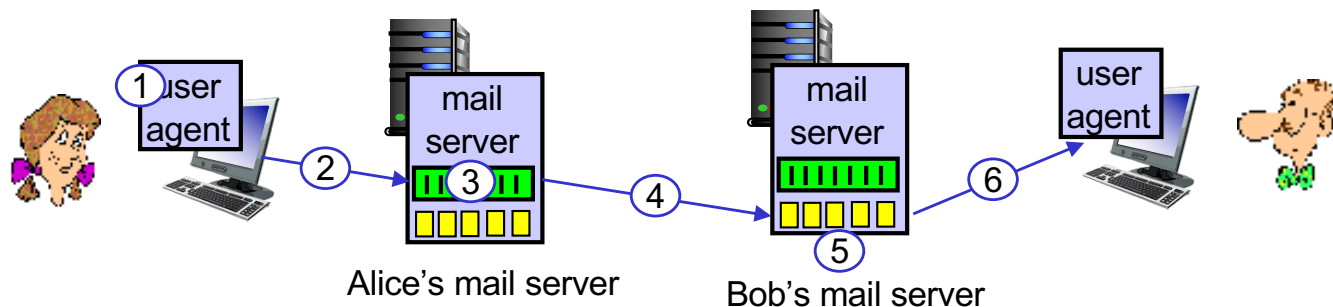
SMTP RFC (5321)

- 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
 - SMTP handshaking (greeting)
 - SMTP transfer of messages
 - SMTP closure
- command/response interaction (like HTTP)
 - **commands:** ASCII text
 - **response:** status code and phrase



Scenario: Alice sends e-mail to Bob

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



Sample SMTP interaction

S: 220 hamburger.edu

SMTP: observations

comparison with HTTP:

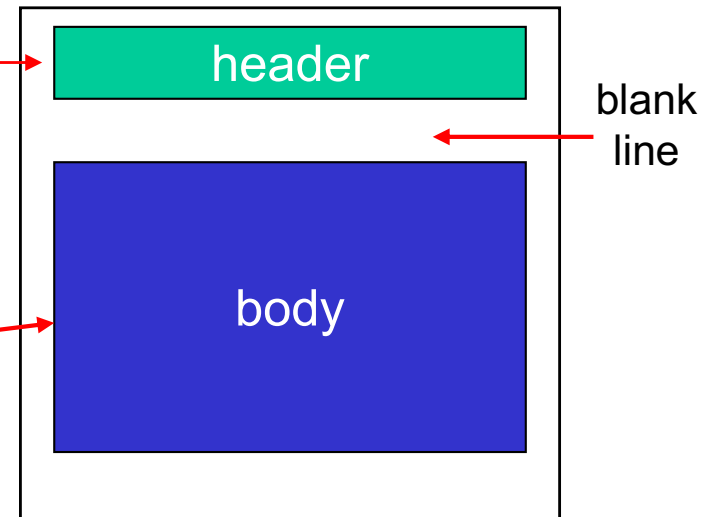
- HTTP: client pull
- SMTP: client push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message
- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

Mail message format

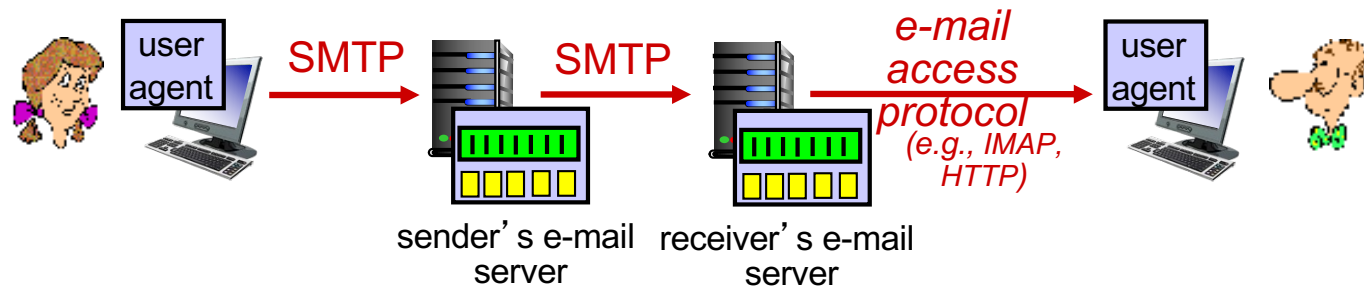
SMTP: protocol for exchanging e-mail messages, defined in RFC 5321 (like RFC 7231 defines HTTP)

RFC 2822 defines *syntax* for e-mail message itself (like HTML defines syntax for web documents)

- 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



Retrieving email: mail access protocols



- **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 SMTP (to send), IMAP (or POP) to retrieve e-mail messages