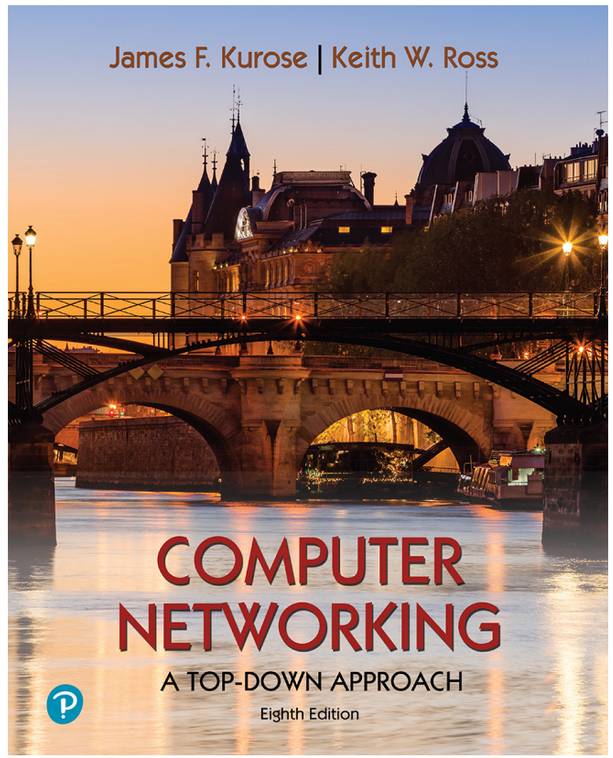
Application Layer

Thanks to:

JFK/KWR



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
- transport-layer service models
- network application architectures
  - client-server
  - peer-to-peer
- examine some popular application-layer protocols
  - HTTP
  - DNS
- programming network applications
  - socket API

Some network applications [1,2.1]

- social networking
- Web browsing
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Netflix)
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing (e.g., Zoom, Teams, Meet)
- Internet search
- remote login
- …

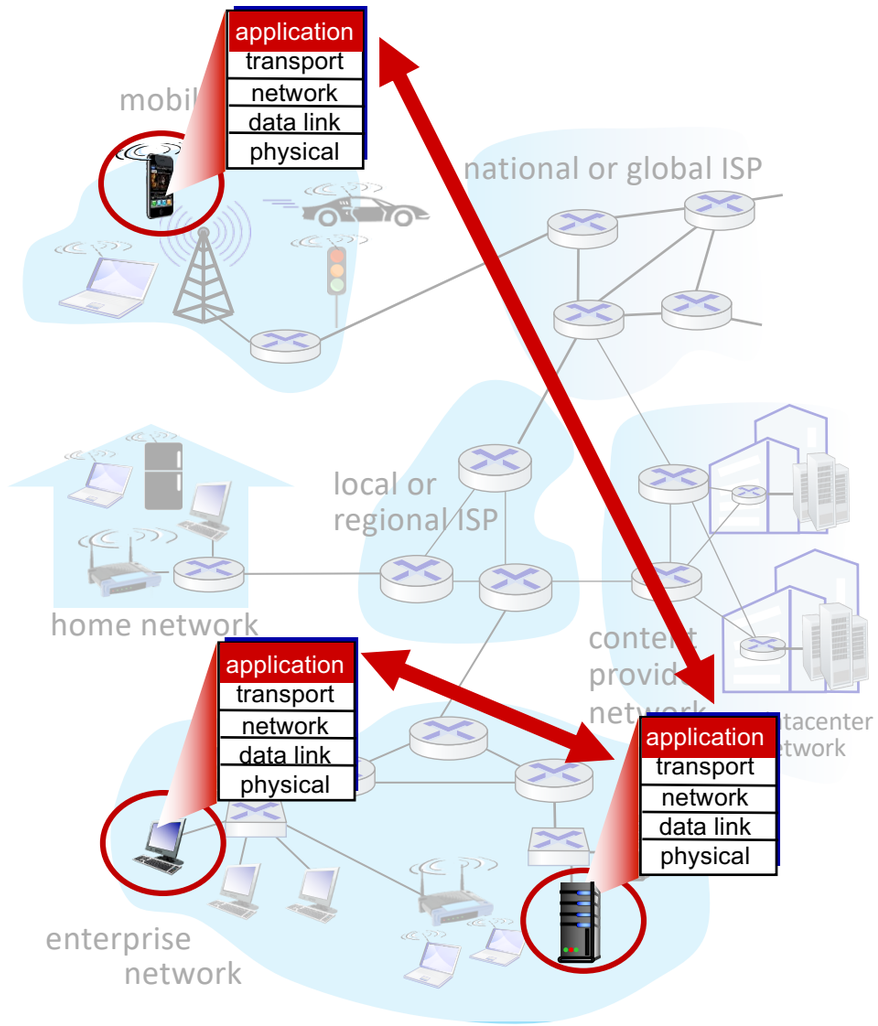
Creating a network application [1,2.1]

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



Network Architecture vs. Network Application Architecture [1,2.1]

Network Architecture:

- **example:** five-layer Internet architecture
- network architecture is fixed
- provides a specific set of services to applications

Network Application Architecture:

- network application architecture is to be designed to dictate **how the network application is structured over various end systems**
  - client-server
  - peer-to-peer

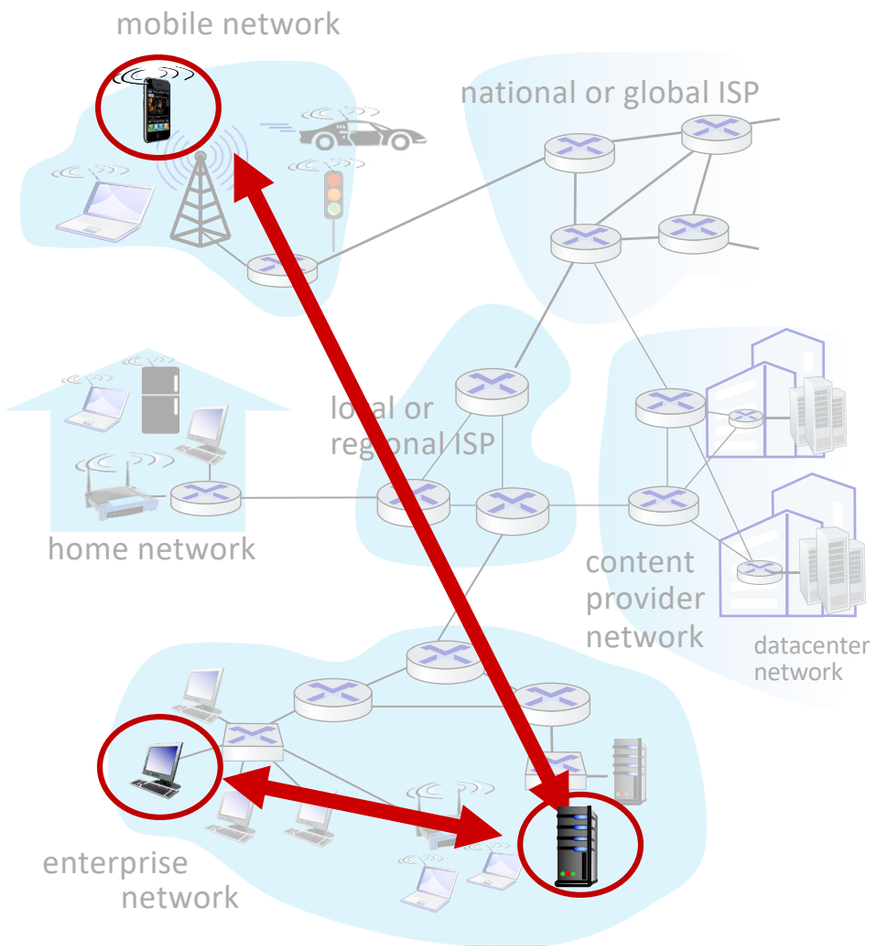
Client-Server paradigm (**examples: HTTP, FTP**) [1,2.1]

server:

- Host is **always-on**
- With fixed / well-known IP address
- often in data centers, for scaling

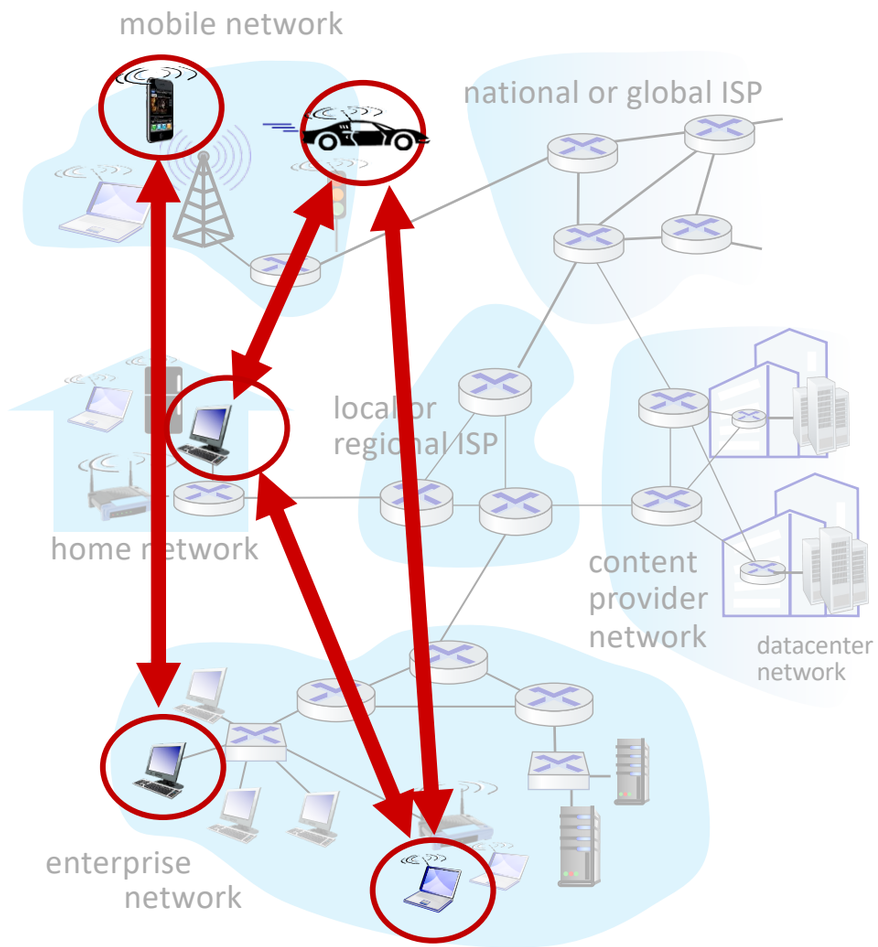
clients:

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



Peer-to-Peer architecture [1,2.1]

- there is *no* always-on server
- arbitrary end systems **"directly"** communicate
- peers request service from other peers, and provide service in return to other peers
  - ***self scalability*** – new peers bring new service demands, as well as new service capacity
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing



Processes communicating [1,2.1]

process: an application layer program running within a host

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

clients, servers

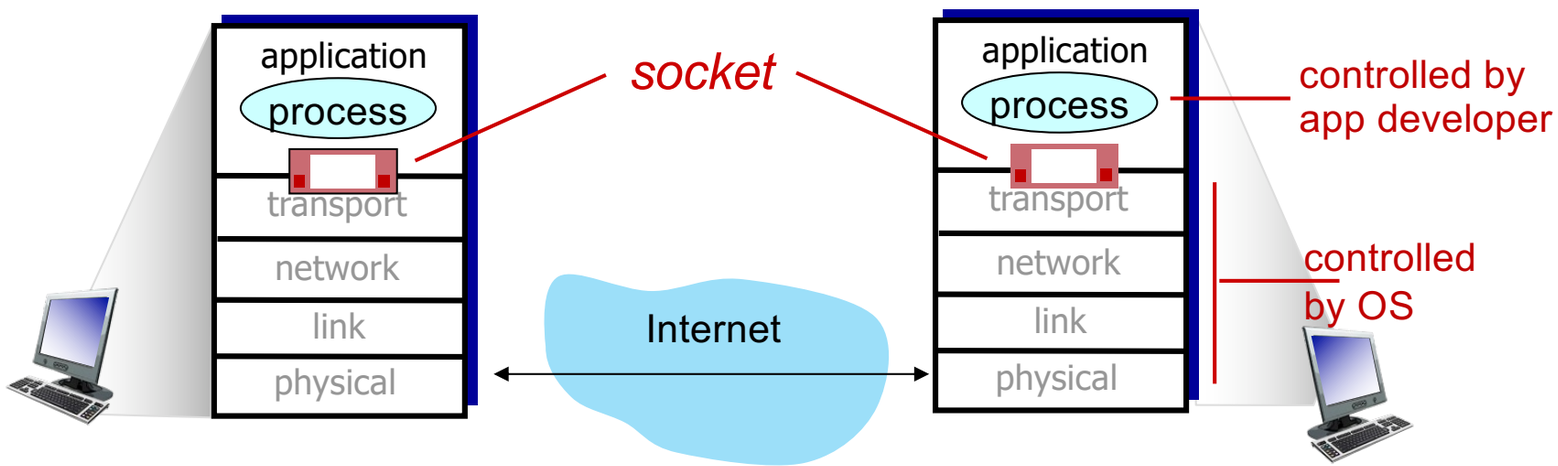
client process: process that **initiates** communication

server process: process that **waits** to be contacted

- **note:** in applications with P2P architectures the same process acts like the client as well as the server, taking turns

Sockets [1,2.1]

- a process sends/receives messages to/from a **software interface** called a **socket**
- also called **Application Programming Interface (API)** between a network application and the network
- analogous to sending parcels through a transport service
- the parcel becomes the responsibility of the transport service
- we have almost no control over how the transport service operates

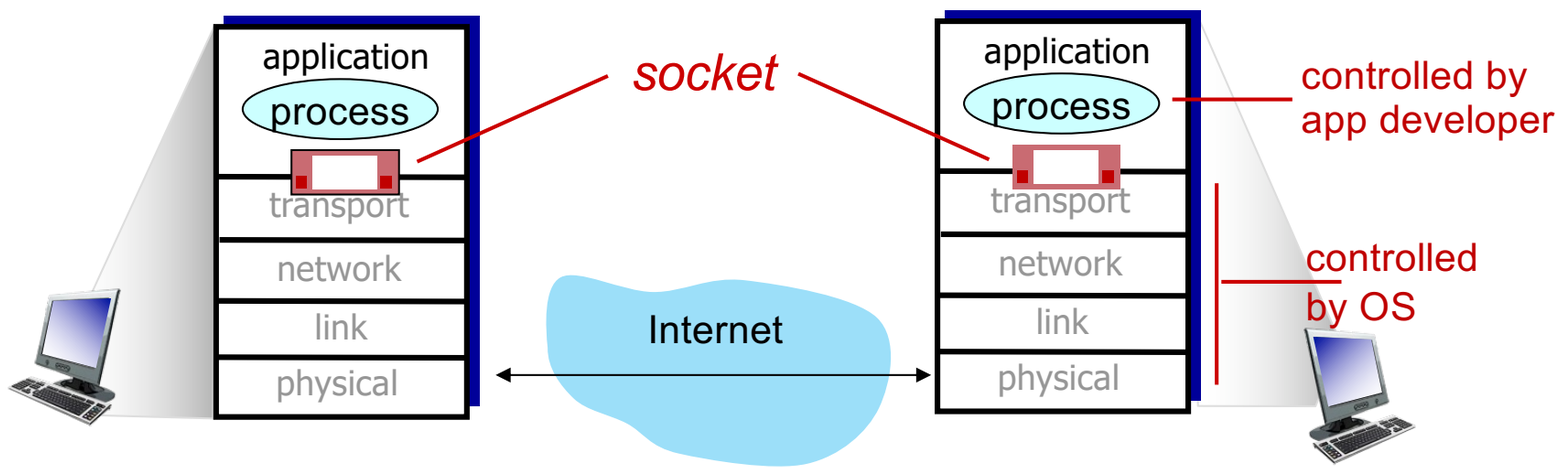


Sockets [1,2.1]

- the application developer has control of everything on the application-layer side of the socket but has little control of the transport-layer side
- the only control that the application developer has on the transport layer side are:

  (1) the choice of transport protocol, and

  (2) perhaps the ability to fix a few transport-layer parameters (e.g., **maximum buffer** and **maximum segment sizes**)



Addressing processes [1,2.1]

- to receive messages, a process must have an *identifier*
- host device has unique 32-bit IP address
- *Q:* Does IP address of host on which the process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host

- So, the process *identifier* includes both the IP address and the port number associated with the process on the host
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP messages to **`gaia.cs.umass.edu`** web server:
  - IP address: 128.119.245.12
  - port number: 80

More precisely, **socket address = IP address + port number**

What transport services an app might need? [1,2.1]

reliability

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- **loss-tolerant** apps (e.g., audio / video) can tolerate some loss

delay

- some apps (e.g., Internet telephony, interactive games) require **low delay** to be "effective"

throughput

- **bandwidth-sensitive** apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- **elastic apps** (e.g., email, FTP) make use of whatever throughput they get

security

- encryption, data integrity, end-point authentication, …

Transport service requirements: common apps [1,2.1]

application	data loss	throughput	delay sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video: 10kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

Internet transport protocols services [1,2.1]

TCP service:

- *reliable transport* between sending and receiving processes
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle / slow-down sender when network is overloaded
- *connection-oriented:* set-up required between client and server processes
- *does not provide:* guarantee of delay, minimum throughput, and security

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP at all?

Internet applications, and transport protocols [1,2.1]

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

An application-layer protocol defines: [1,2.1]

- types of messages exchanged
  - e.g., request, response
- message syntax
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in the fields
- rules for when and how the 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

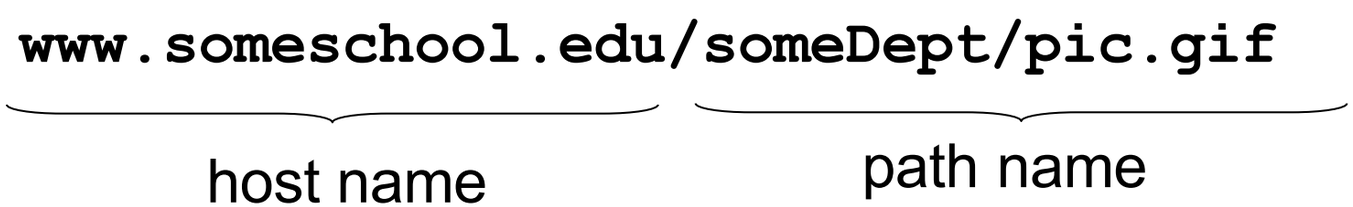
Network Application vs. Application Layer Protocol [1,2.1]

- A Network Application (e.g., Web Browsing) consists of the following:
  - A standard for document formats (e.g., HTML)
  - Web browsers on the client side (e.g., Chrome, Firefox, Internet Explorer, Safari)
  - Web servers (e.g., Apache, Microsoft)
  - Application Layer protocol (e.g., HTTP)

- So, an **Application Layer protocol is only one part of a Network Application**

Web and HTTP [1,2.2]

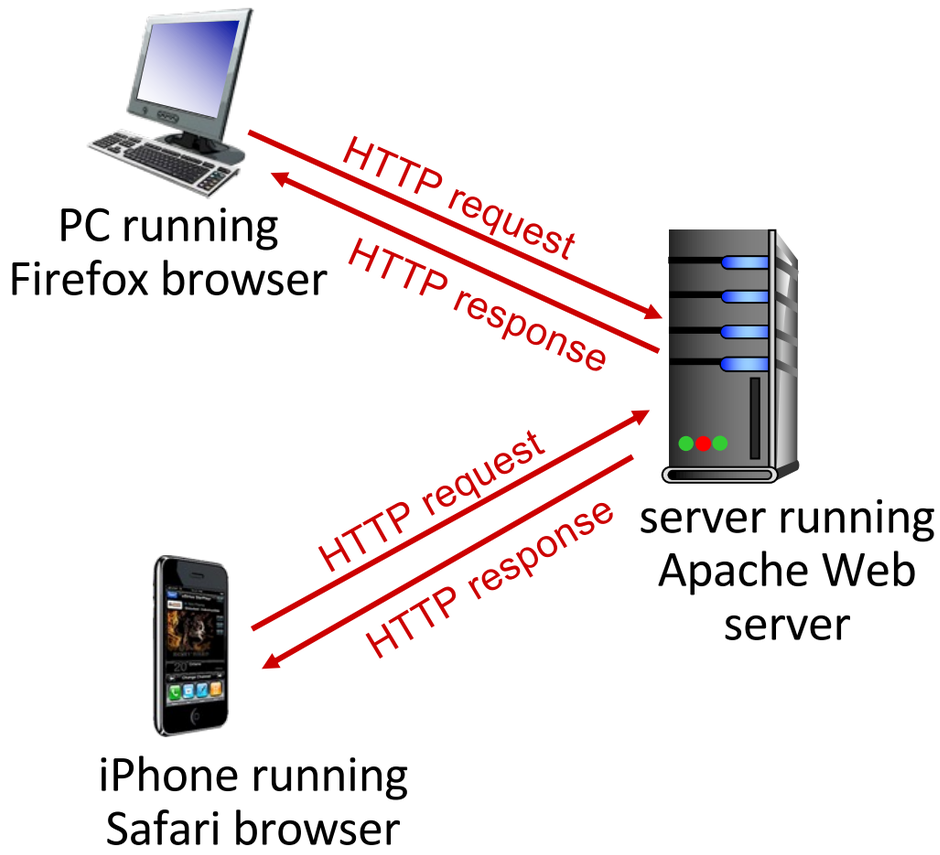
- a web page consists of *objects,* each of which can be stored on different Web servers
- an object can be an HTML file, a JPEG image, a Java applet, an audio file,…
- a web page consists of a ***base HTML-file*** which includes several ***referenced objects***, each addressable by a *URL,* e.g.,



HTTP overview [1,2.2]

HTTP: hypertext transfer protocol

- Web's application-layer protocol
- **client/server model:**
  - *client:* browser that requests for and receives (using HTTP protocol), and "displays" **Web objects**
  - *server:* Web server sends web objects (using HTTP protocol) in response to requests



HTTP overview (continued) [1,2.2]

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 is closed

*HTTP is "**stateless**"*

- server maintains *no* information about past client requests

aside

protocols that maintain "state" are complex!

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

HTTP connections: two types [1,2.2]

Non-persistent HTTP

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

downloading multiple objects required multiple TCP connections to be opened and closed

Persistent HTTP

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

Non-persistent HTTP: example [1,2.2]

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)

1a. HTTP client (process) initiates TCP connection to HTTP server (process) at **`www.someSchool.edu`** on port 80

1b. HTTP server at host **`www.someSchool.edu`** waiting for TCP connection request 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 the 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.) [1,2.2]

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)

4. HTTP server closes TCP connection.

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

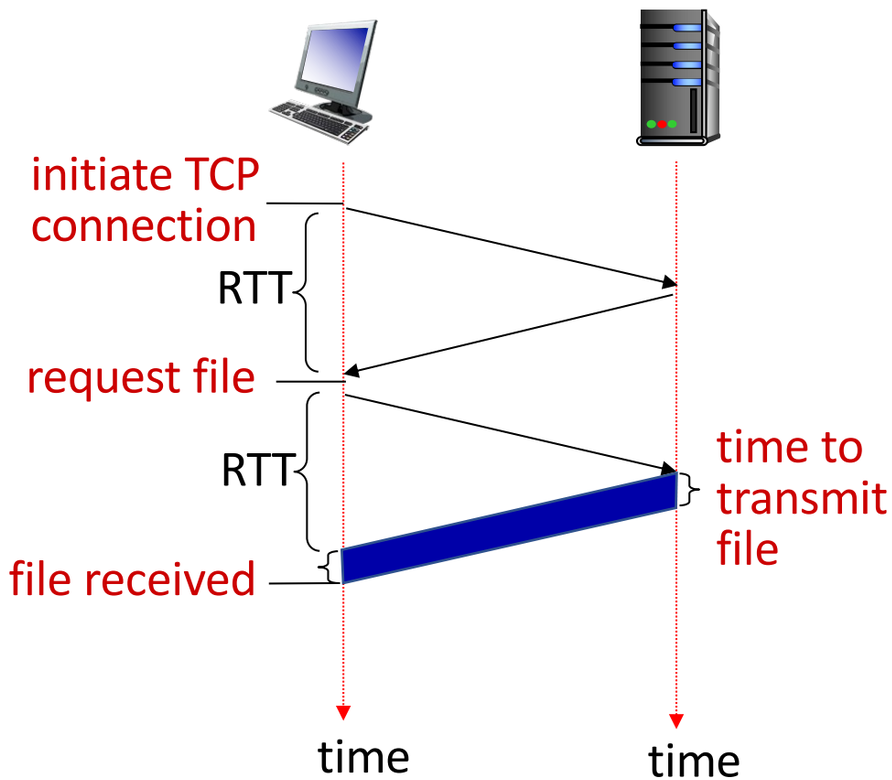
time

Non-persistent HTTP: response time [1,2.2]

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) [1,2.2]

Non-persistent HTTP issues:

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

Persistent HTTP (HTTP1.1):

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

HTTP/1.1 [1,2.2]

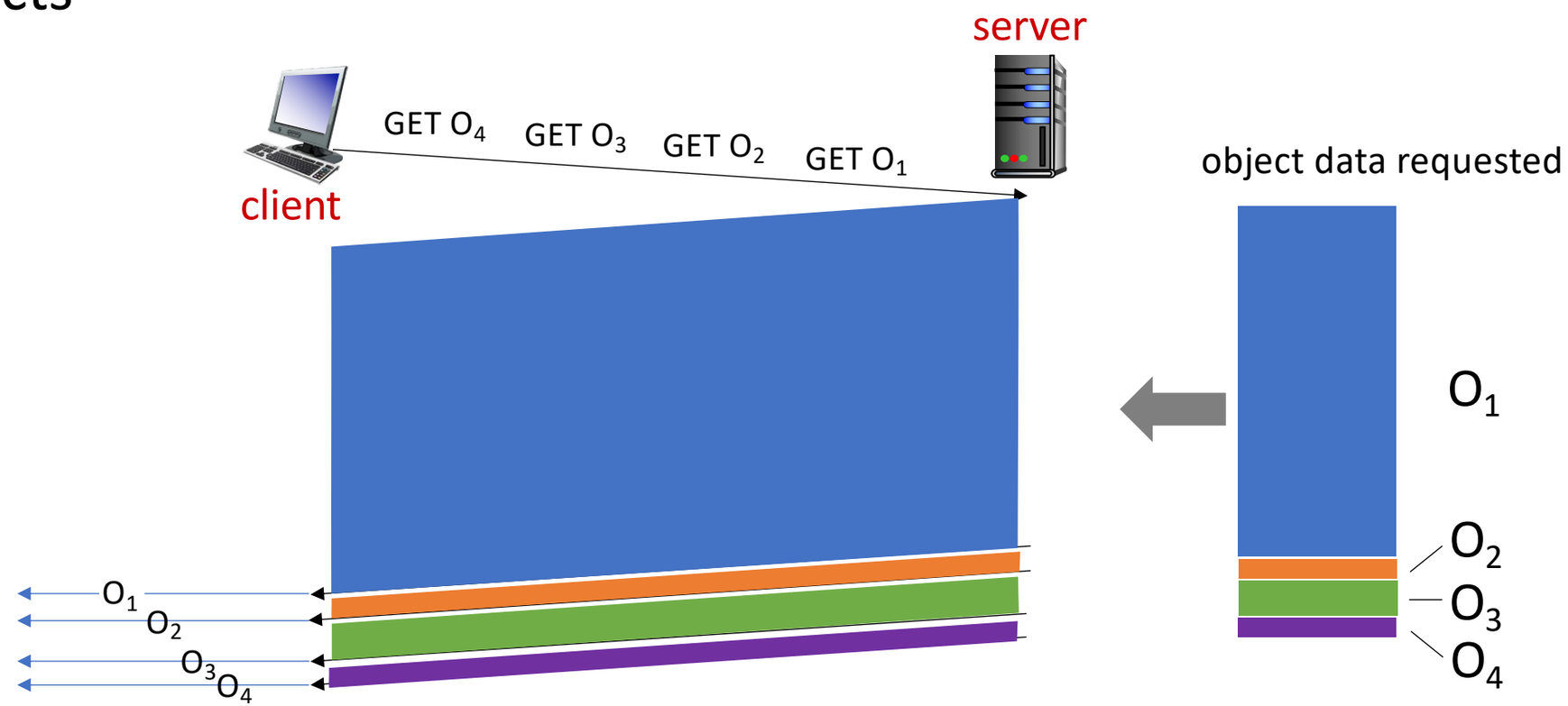
Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined object downloads over a 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 behind large object(s) resulting in head-of-line (HOL) blocking
- loss recovery (retransmitting lost TCP segments) stalls object transmission

HTTP/1.1: HOL blocking [1,2.2]

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 [1,2.2]

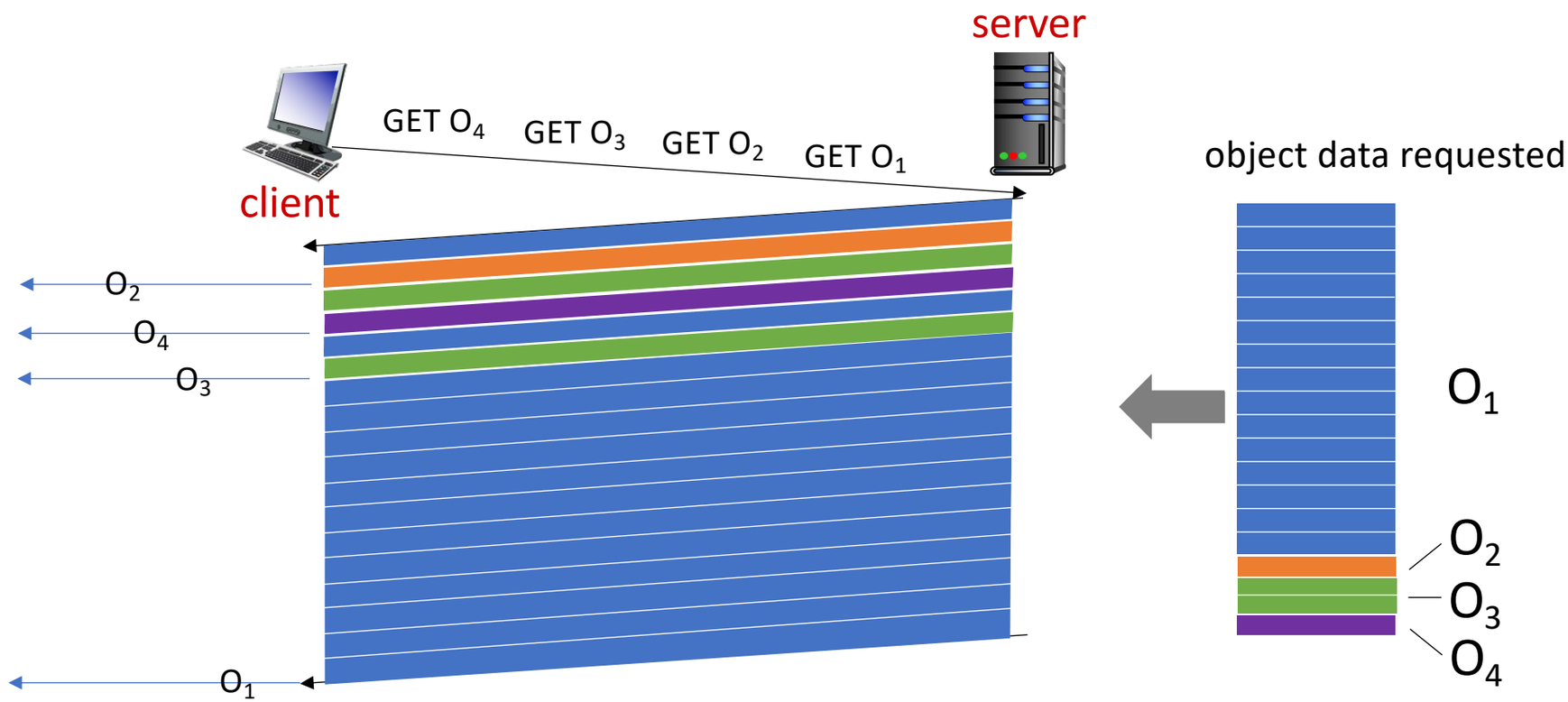
Key goal: decreased delay in multi-object HTTP requests

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

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

HTTP/2: mitigating HOL blocking [1,2.2]

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

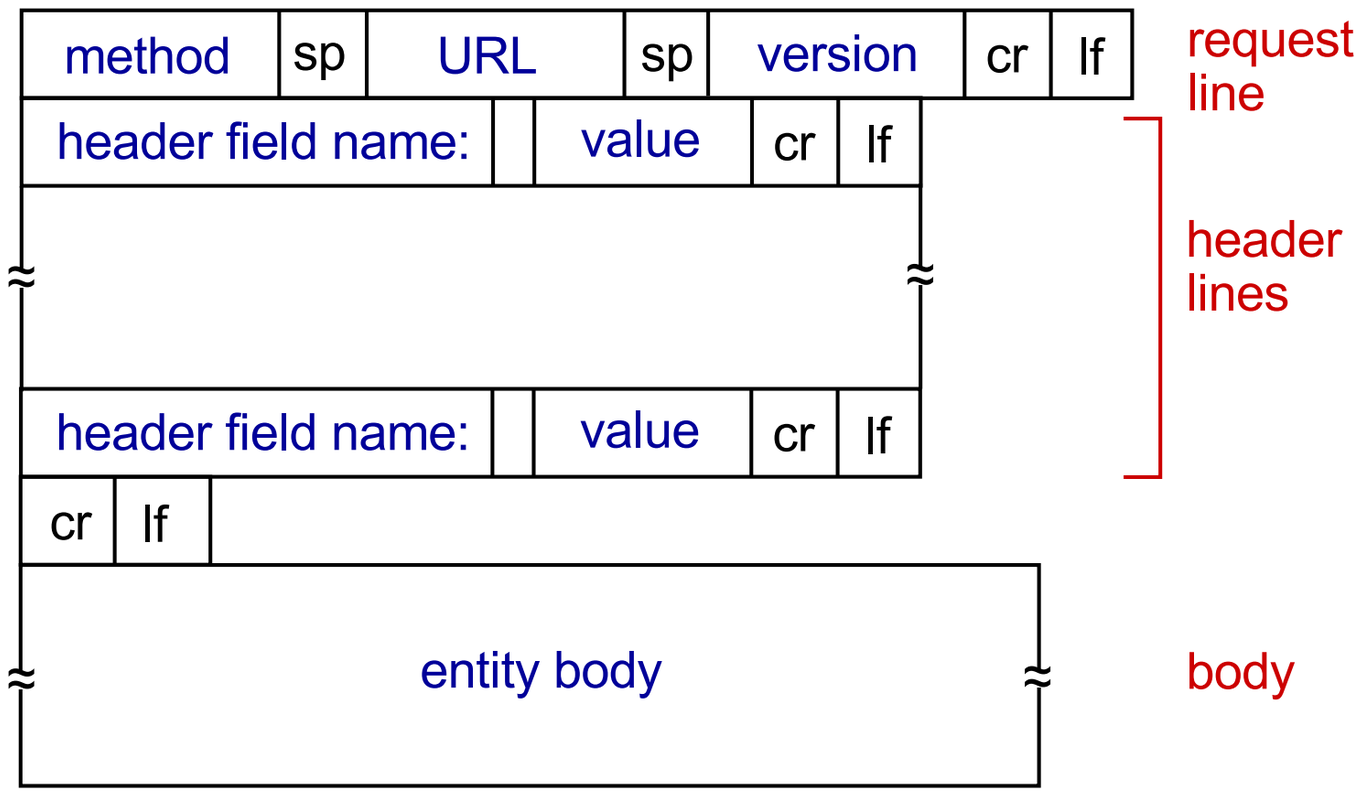


O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/1.0, HTTP/1.1, HTTP/2 and HTTP/3

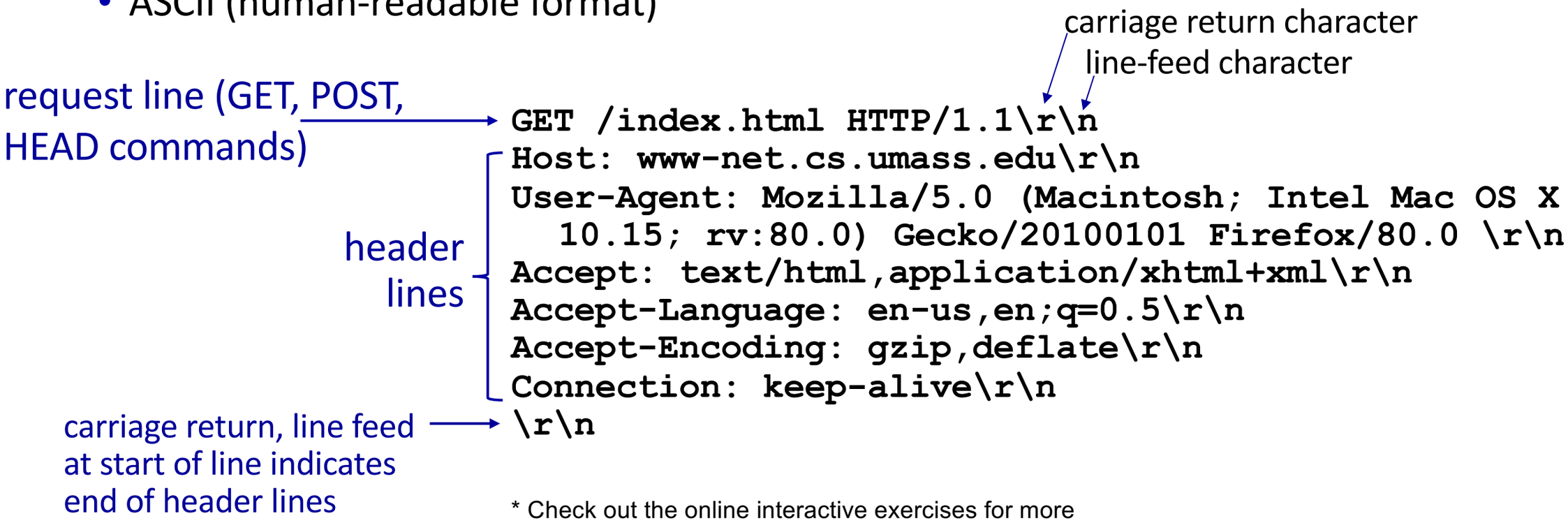
- **HTTP/1.0:** RFC 1945 (https://www.ietf.org/rfc/rfc1945.txt)
- **HTTP/1.1:** RFC 2616 (https://www.ietf.org/rfc/rfc2616.txt)
- **HTTP/2.0:** RFC 9113 (https://www.rfc-editor.org/rfc/rfc9113.html)
- **HTTP/3.0:** RFC 9114 (https://www.rfc-editor.org/info/rfc9114)

HTTP request message: general format [1,2.2]



HTTP request message [1,2.2]

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



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

Other HTTP request messages [1,2.2]

GET method (for fetching data from server):

- can include user data in URL field of HTTP GET request message (following a '?'):

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

POST method:

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

HEAD method:

- requests headers (only) that would be returned
- used for debugging

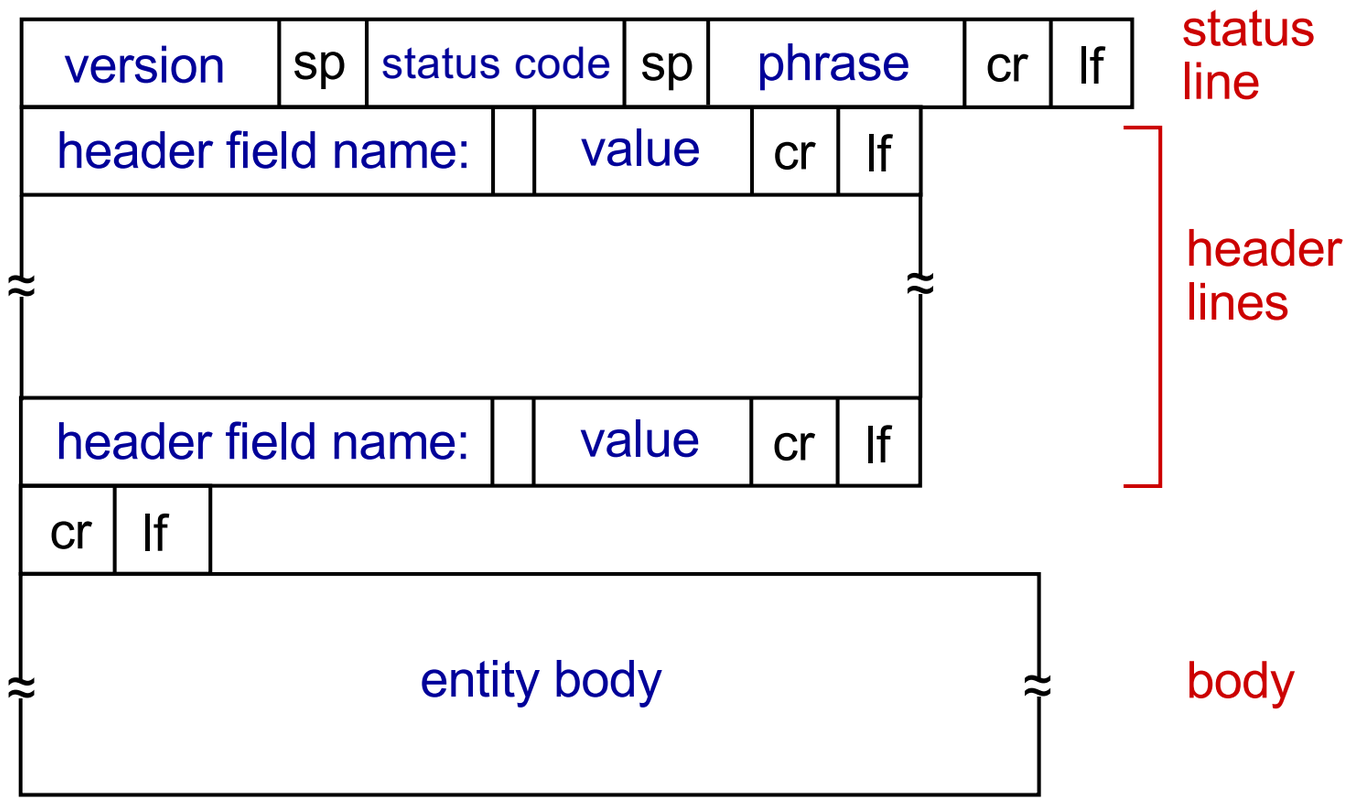
PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body

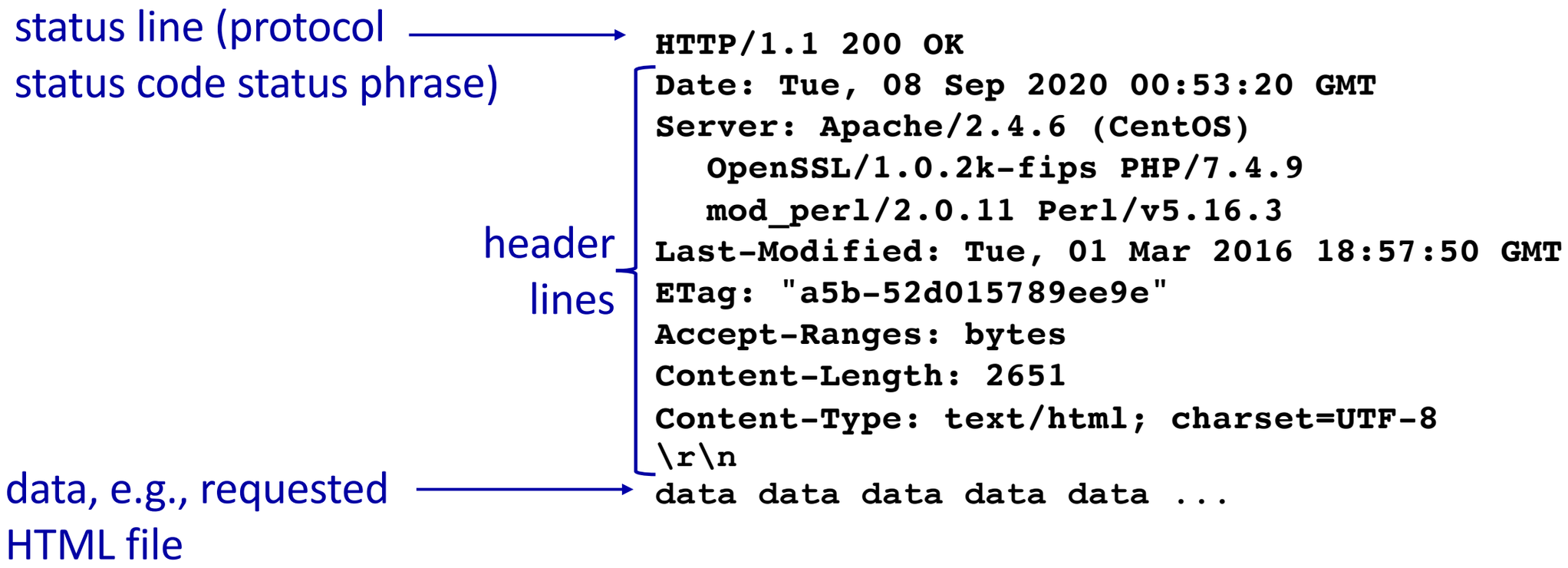
DELETE method:

- to delete a specific object on a Web server

HTTP response message: general format [1,2.2]



HTTP response message [1,2.2]



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

HTTP response status codes [1,2.2]

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

- requested object moved, new location specified later in this message (in **`Location: field`**)

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

HTTP cookies: comments [1,2.2]

HTTP was originally designed to be **stateless**, but …

*Cookies can be used to keep **state** information about clients for:*

- authorization
- shopping carts
- recommendations, etc.

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

Maintaining user/server state: cookies [1,2.2]

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

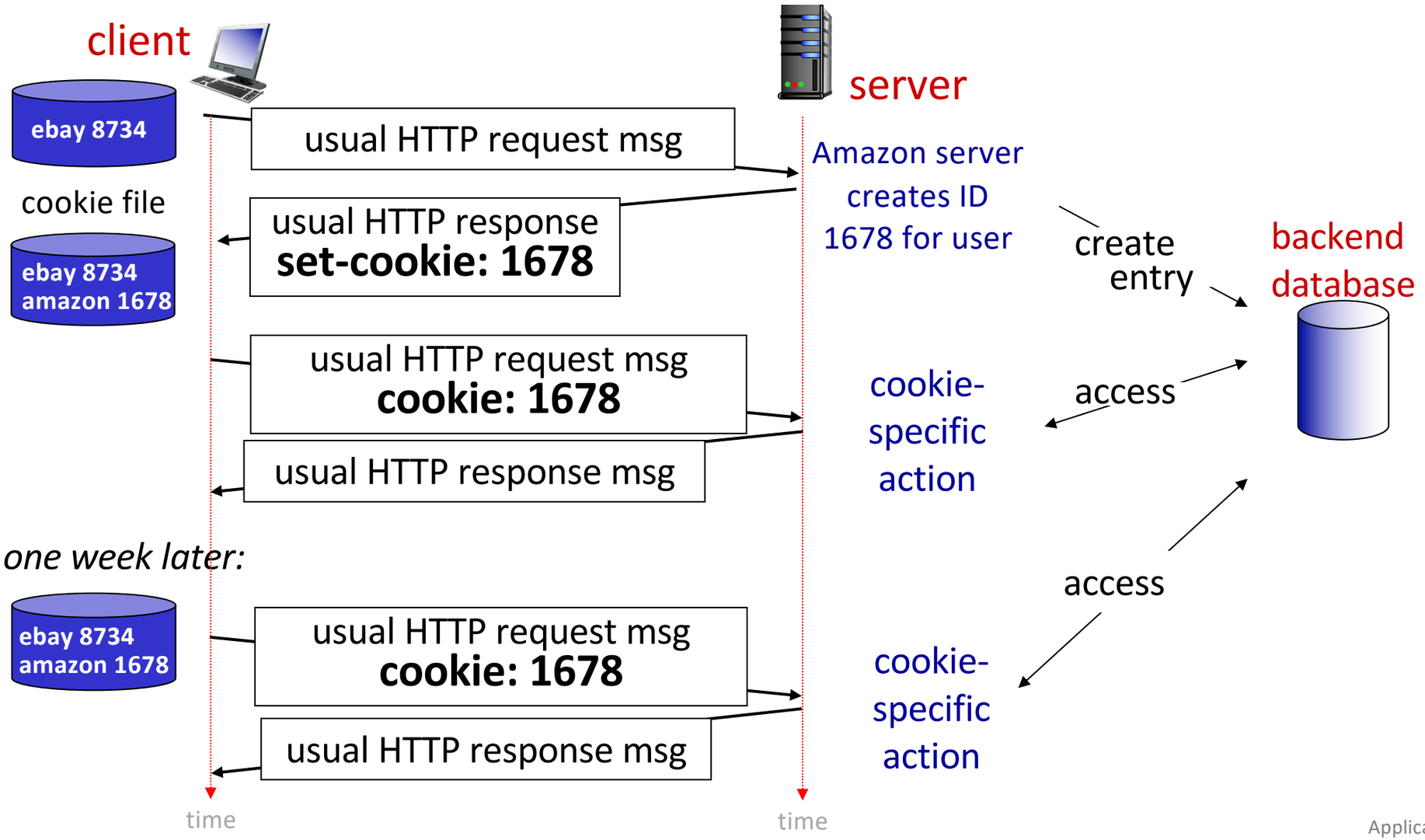
four components:

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 the browser on her laptop, visits specific e-commerce site for the 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

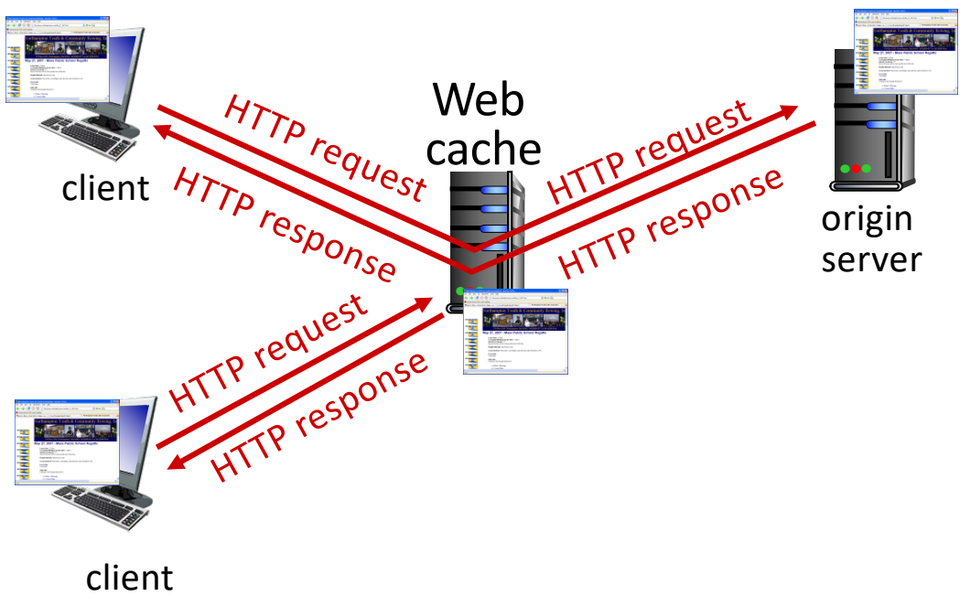
Maintaining user/server state: cookies [1,2.2]



Web caches [1,2.2]

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) [1,2.2]

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

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

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

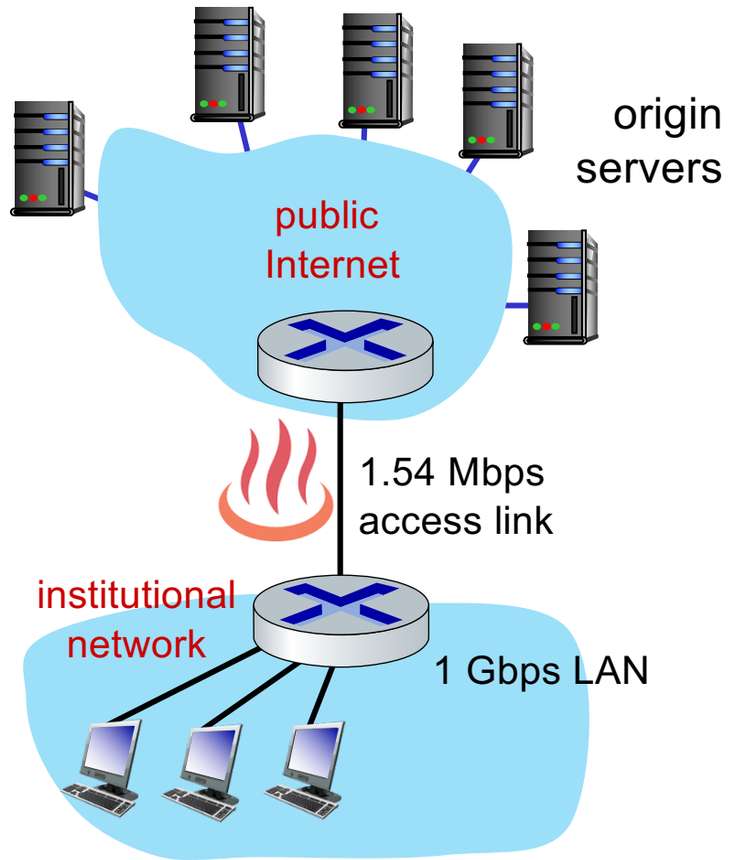
Why Web caching?

- 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 [1,2.2]

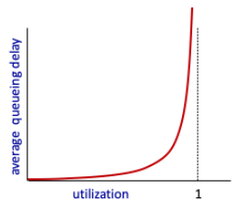
Scenario:

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



problem: large queueing delays at high utilization!

Option 1: buy a faster access link [1,2.2]

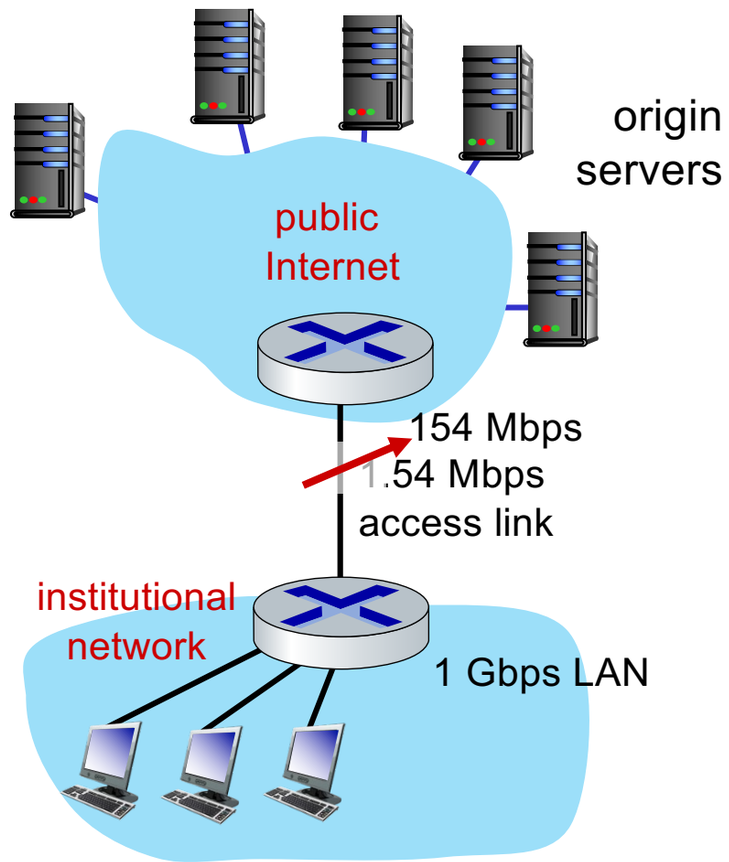
Scenario:

- access link rate: ~~1.54~~ 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~~ .0097
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + ~~minutes~~ + μsecs msecs

Cost: faster access link (expensive!)



Option 2: install a web cache [1,2.2]

Scenario:

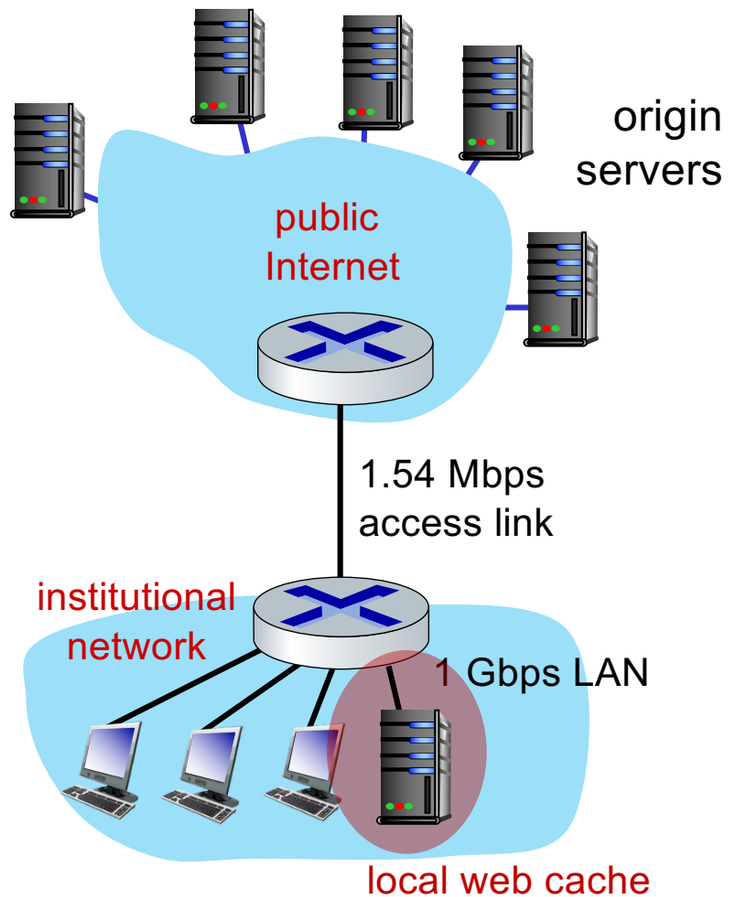
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- web object size: 100K bits
- average request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Cost: web cache (cheap!)

Performance:

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

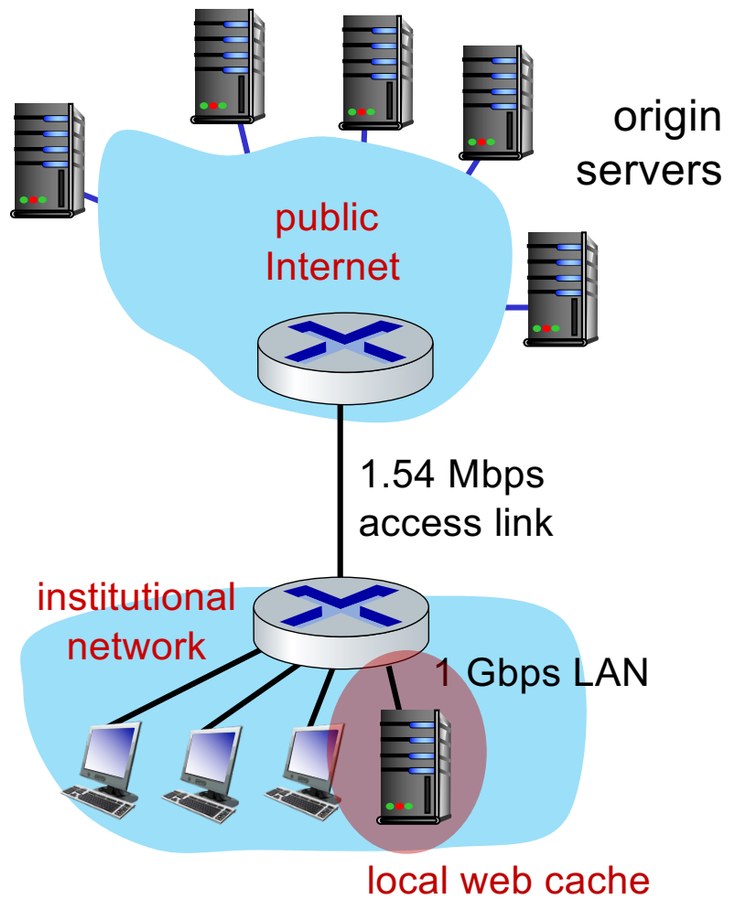
How to compute link utilization, delay?



Calculating access link utilization, end-end delay with cache: [1,2.2]

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 Mbps = .9 Mbps
  - access link utilization = 0.9/1.54 = .58 means low (msec) queueing delay at access link
- average end-end delay:
  = 0.6 * (delay from origin servers)
  + 0.4 * (delay when satisfied at cache)
  = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs



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

Conditional GET [1,2.2]

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

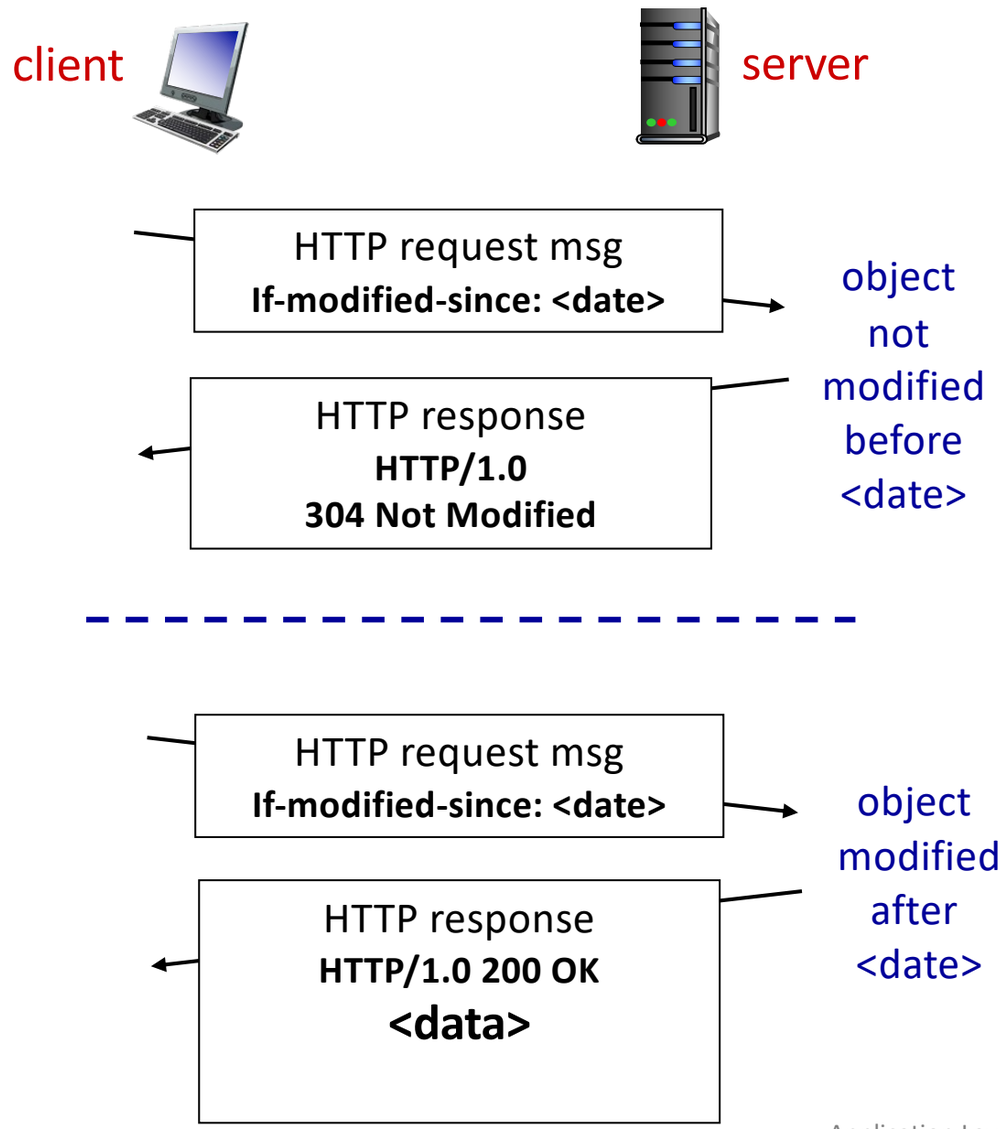
- no object transmission delay (or use of network resources)

- *client:* 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



DNS: Domain Name System [1,2.4]

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., cs.umass.edu - used by humans

For an invoking application, DNS is a black box providing a simple, straightforward translation service, but in reality …

Domain Name System (DNS):

- *A complex service*
- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, DNS servers communicate to *resolve* names (address/name translation)
- ***note:*** DNS is implemented as an application-layer (client-server) protocol using UDP and Port 53

DNS: services, structure [1,2.4]

DNS services:

- **hostname-to-IP-address translation**
- **host aliasing**
  - canonical hostname from alias hostnames that are more **mnemonic**
- **mail server aliasing**
  - allows the mail server and web server of a company to have the same alias hostname
- **load distribution**
  - replicated Web servers; many IP addresses correspond to one name

Q: Why not have a centralized DNS?

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

A: doesn‘t scale!

- Comcast DNS servers alone: 600 Billion DNS queries/day
- Akamai DNS servers alone: 2.2T DNS queries/day

Thinking about the DNS [1,2.4]

humongous distributed database:

- ~ billion records, each simple

handles many *trillions* of queries/day:

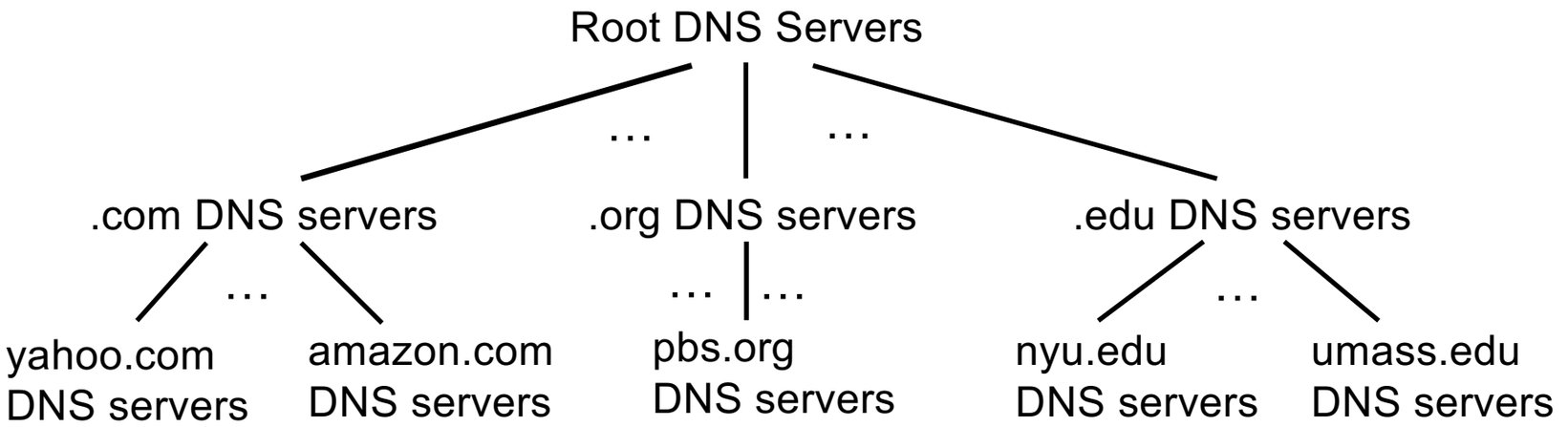
- *many* more reads than writes
- *performance matters:* almost every Internet transaction interacts with DNS - msecs count!

organizationally, physically decentralized:

- millions of different organizations responsible for their records

"bulletproof": reliability, security

DNS: a distributed, hierarchical database [1,2.4]



Root

Top Level Domain

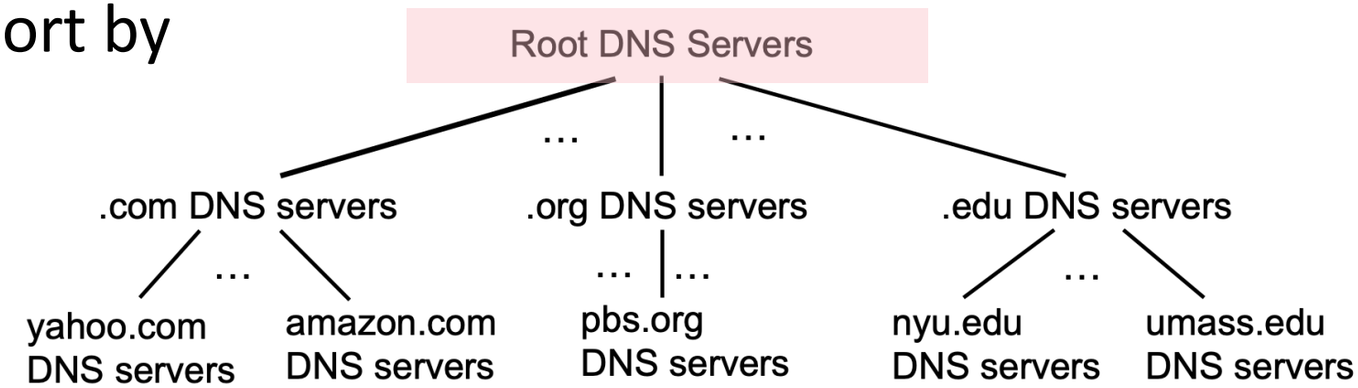
Authoritative

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

- **Local DNS server** queries root server to find **`.com`** DNS server
- Local DNS server queries **`.com`** DNS server to get **`amazon.com`** DNS server
- Local DNS server queries **`amazon.com`** DNS server to get IP address for **`www.amazon.com`**

DNS: root name servers [1,2.4]

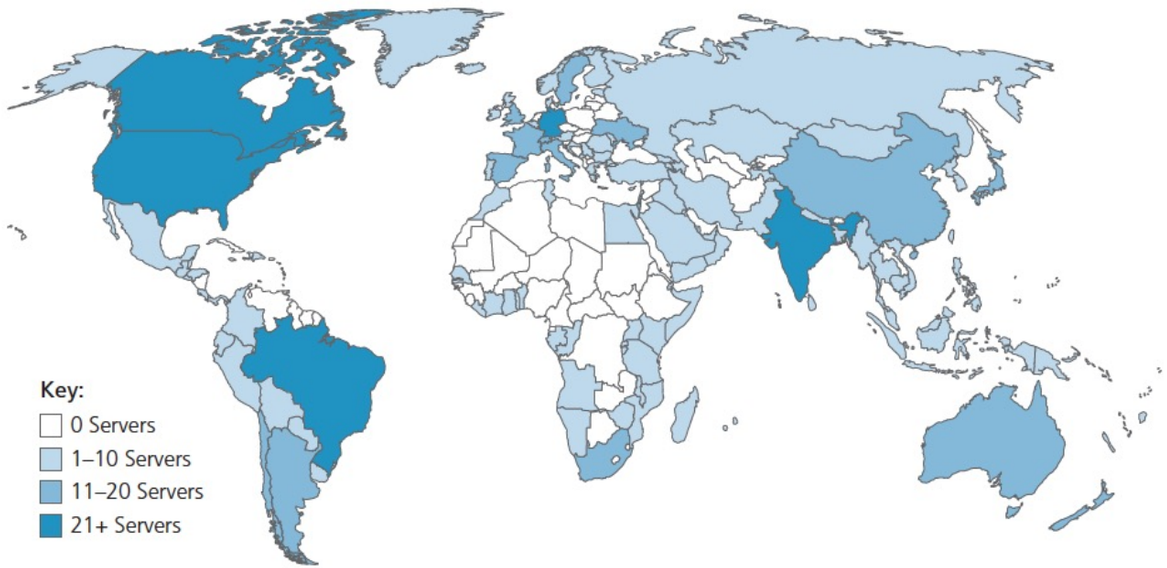
- official, contact-of-last-resort by name servers that cannot resolve name



DNS: root name servers [1,2.4]

- official, contact-of-last-resort by name servers that can not resolve name
- *DNS is an 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

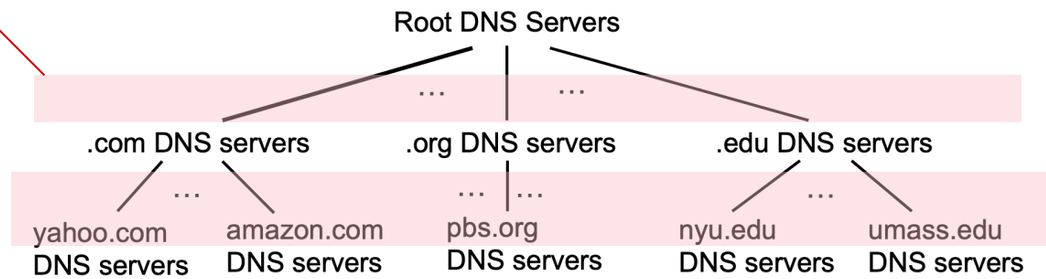
13 logical root name "servers" worldwide; each "server" replicated many times (~200 servers in US)



Top-Level Domain, and authoritative servers [1,2.4]

Top-Level Domain (TLD) servers:

- responsible for **`.com`**, **`.org`**, **`.net`**, **`.edu`**, **`.aero`**, **`.jobs`**, **`.museums`**, and all top-level **country domains**, e.g.: **`.in`**, **`.uk`**, **`.fr`**, **`.ca`**, **`.jp`**
- e.g., Verisign Global Registry Services: authoritative registry for **`.com`** TLD
- e.g., Educause: authoritative registry 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 servers [1,2.4]

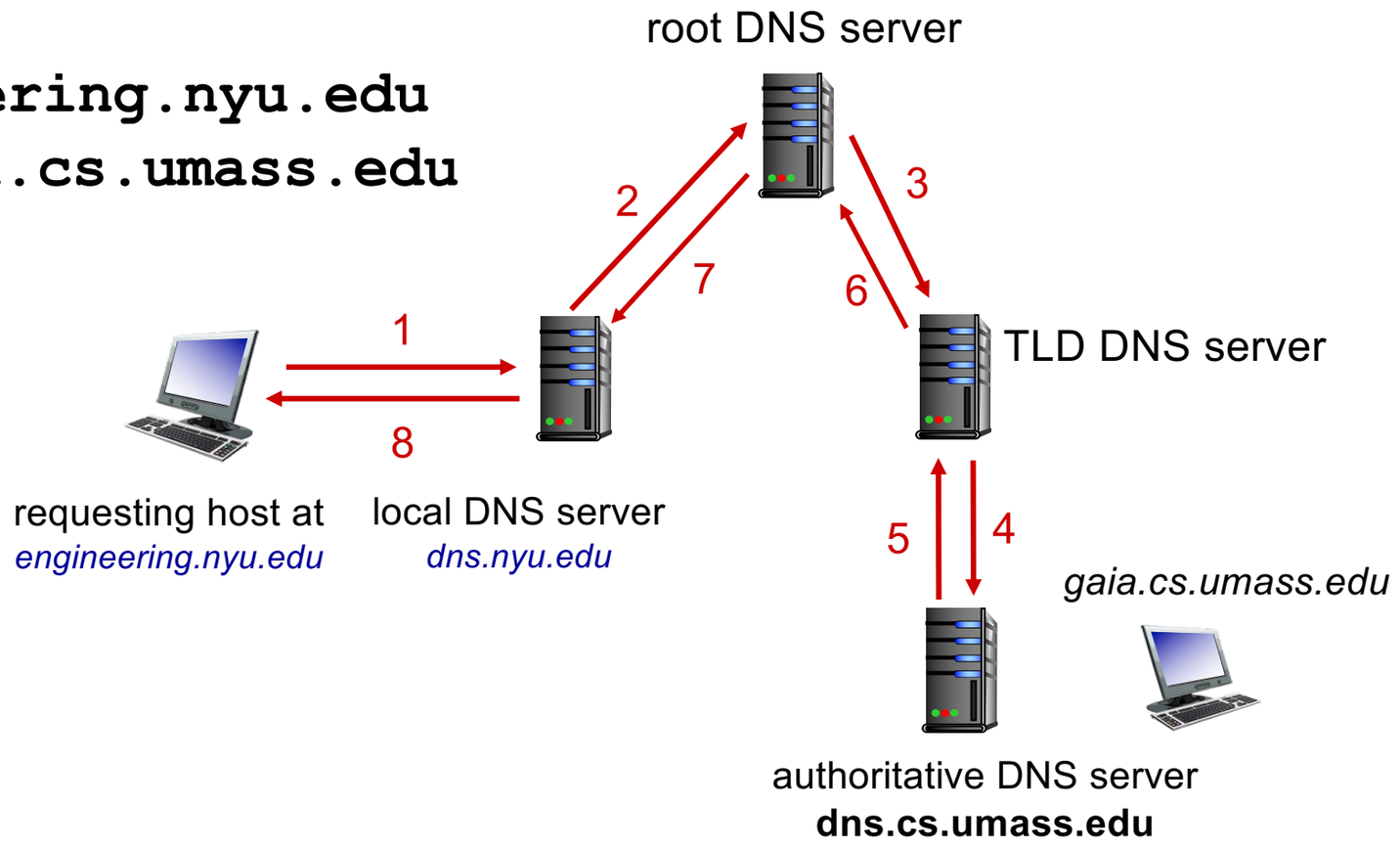
- when host makes DNS query, it is sent to its **local DNS server**
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (possibly out of date!)
    - Or, forwarding request into DNS hierarchy for resolution
  - each ISP has local DNS name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

DNS name resolution: recursive query [1,2.4]

Example: host at **`engineering.nyu.edu`** wants IP address for **`gaia.cs.umass.edu`**

Recursive query:

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

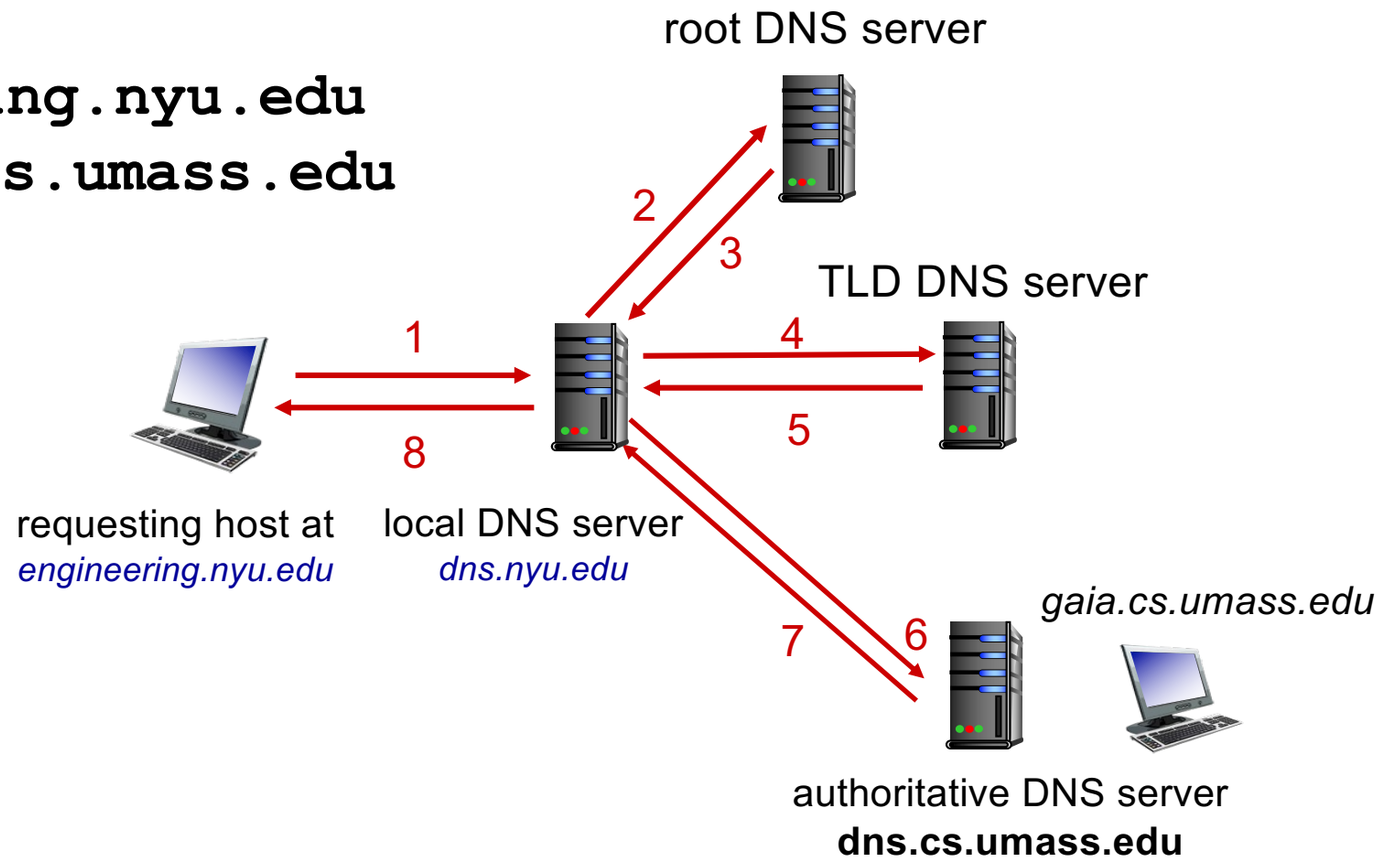


DNS name resolution: iterated query [1,2.4]

Example: host at **`engineering.nyu.edu`** wants IP address for **`gaia.cs.umass.edu`**

Iterated query:

- contacted server may reply with the name of server to contact
- "I don't know this name, but ask this server"



Caching DNS Information [1,2.4]

- once (any) name server learns a mapping, it *caches* the mapping, and *immediately* returns a cached mapping in response to a query
  - caching improves response time
  - cache entries timeout (disappear) after some time (TTL)
  - info on TLD servers typically cached in local name servers
- 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!*

DNS: services, structure [1,2.4]

DNS services:

- **hostname-to-IP-address translation**
- **host aliasing**
  - canonical hostname from alias hostnames that are more **mnemonic**
- **mail server aliasing**
  - allows the mail server and web server of a company to have the same alias hostname
- **load distribution**
  - replicated Web servers; many IP addresses correspond to one name

DNS records [1,2.4]

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 the authoritative name server for this domain
- used to route DNS queries further along in the query chain

type=CNAME

- **`name`** is alias name for some "canonical" (the real) name
- **`value`** is canonical name
- e.g., **`www.ibm.com`** is really **`servereast.backup2.ibm.com`**

type=MX

- **`value`** is the canonical name of the SMTP mail server associated with **`name`**

DNS records [1,2.4]

- If a DNS server is **authoritative** for a particular hostname, then the DNS server will surely contain a **Type A** record for the hostname

  - For example, the DNS server **`dns.umass.edu`** is authoritative for all hostnames ending with **`umass.edu`** and will maintain a Type A record for the hostname **`gaia.cs.umass.edu`**

`(gaia.cs.umass.edu, 128.119.245.12, A)`

- Even if the DNS server is **not authoritative**, it may contain a **Type A** record in its **cache** (maybe because it recently got that information during a recursive query)

DNS records [1,2.4]

- If a server is **not authoritative** for a hostname, then the server will contain a **Type NS** record for the domain that includes the hostname

- For example, an **`.edu`** TLD server is most likely not an authoritative for the host **`gaia.cs.umass.edu`** . This server will contain a **Type NS** record for a domain that includes the host **`gaia.cs.umass.edu`**, e.g.,

  `(umass.edu, dns.umass.edu, NS)`

- It will also contain a **Type A** record that provides the IP address of the DNS server in the Value field of the NS record.

- The **`.edu`** TLD server would also contain a **Type A** record, which maps the DNS server **`dns.umass.edu`** to an IP address, e.g.,

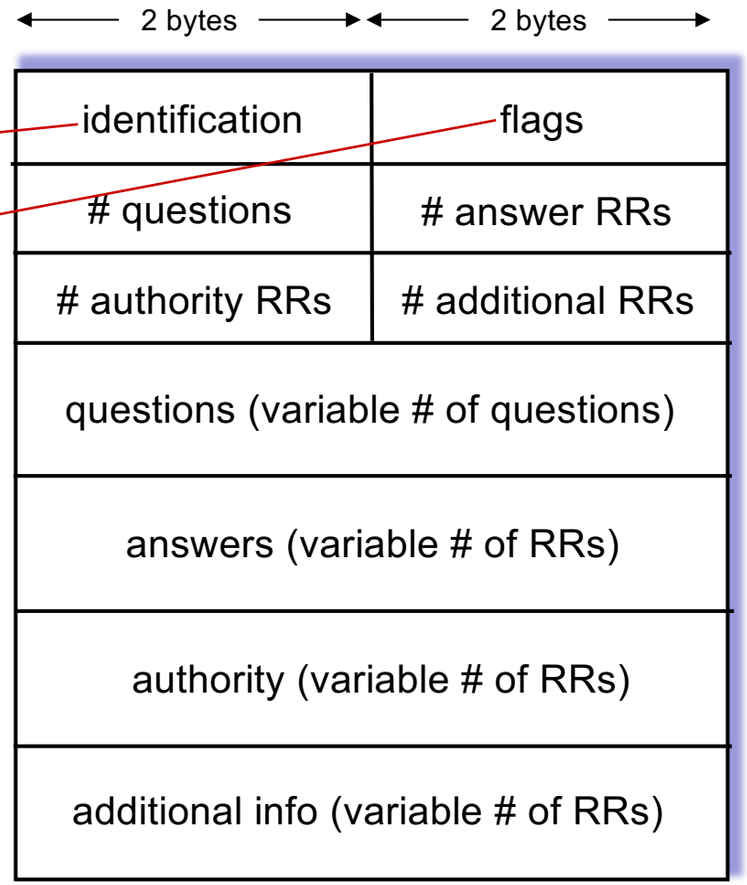
  `(dns.umass.edu, 128.119.40.111, A)`

DNS protocol messages [1,2.4]

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

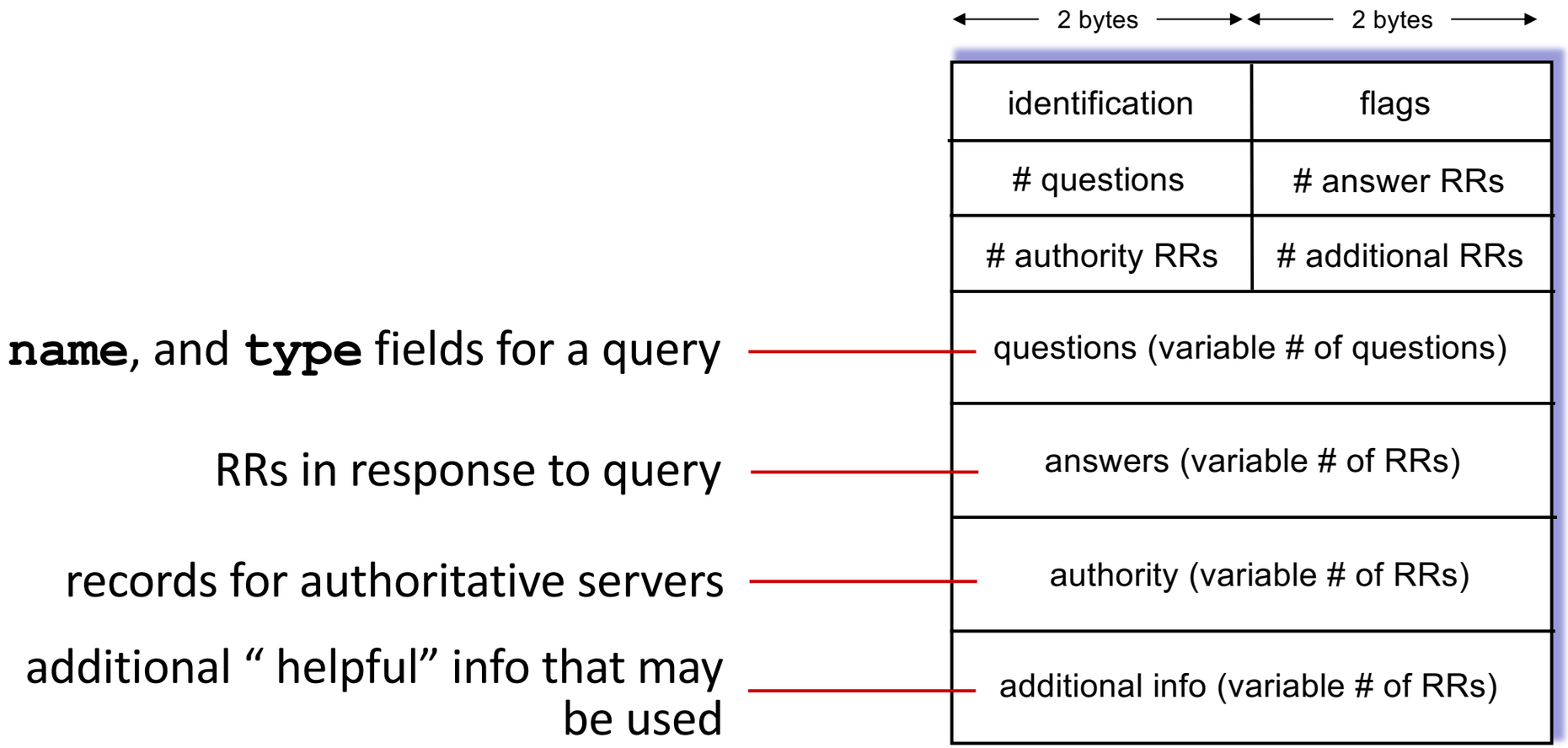
message header:

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



DNS protocol messages [1,2.4]

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



Getting your info into the DNS [1,2.4]

Example: A new startup "Network Utopia"

- register name **`networkuptopia.com`** at *DNS registrar* (e.g., Verisign Global Registry Services)
  - provide names, IP addresses of authoritative name servers (primary and secondary)
  - registrar inserts bith Type NS, and Type A records into a **`.com`** TLD server:

    `(networkutopia.com, dns1.networkutopia.com, NS)`

    `(dns1.networkutopia.com, 212.212.212.1, A)`
- create an authoritative server locally with IP address **`212.212.212.1`** and insert
  - type A record for **`www.networkuptopia.com`**
  - type MX record for **`networkutopia.com`**

References

[1] James Kurose and Keith Ross, “Computer Networking: A Top-down Approach” 8th edition, Addison Wesley 2021.

[2] Behrouz Forouzan, “Data Communication and Networking”, Tata McGraw Hill, 6th edition, 2022.