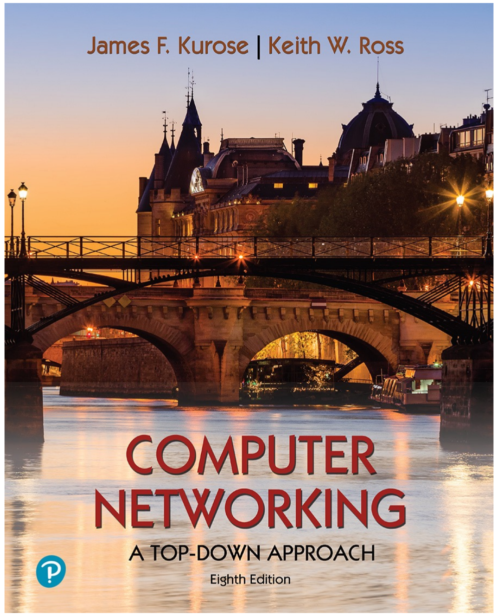
Introduction to Computer Networking

Guy Leduc

Chapter 2
Application Layer



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

Chapter 2: outline

2.1 Principles of network applications

2.2 Web and HTTP

2.3 The Domain Name System (DNS)

2.4 Socket programming with UDP and TCP

Application Layer: Overview

Our goals:

- Conceptual implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - DNS
- programming network applications
  - socket API

Some network apps

- social networking
- web
- 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
- Internet search
- remote login
- …

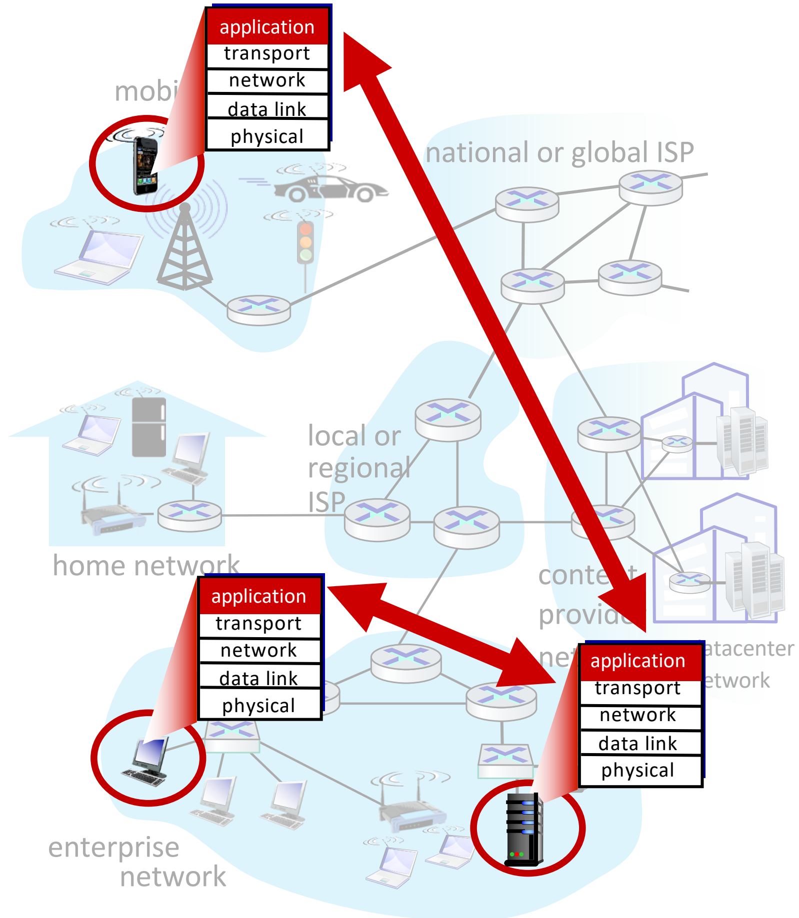
Creating a network app

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



Application architectures

Possible structure of applications:

- Client-server
- Peer-to-peer (P2P)

Client-server paradigm

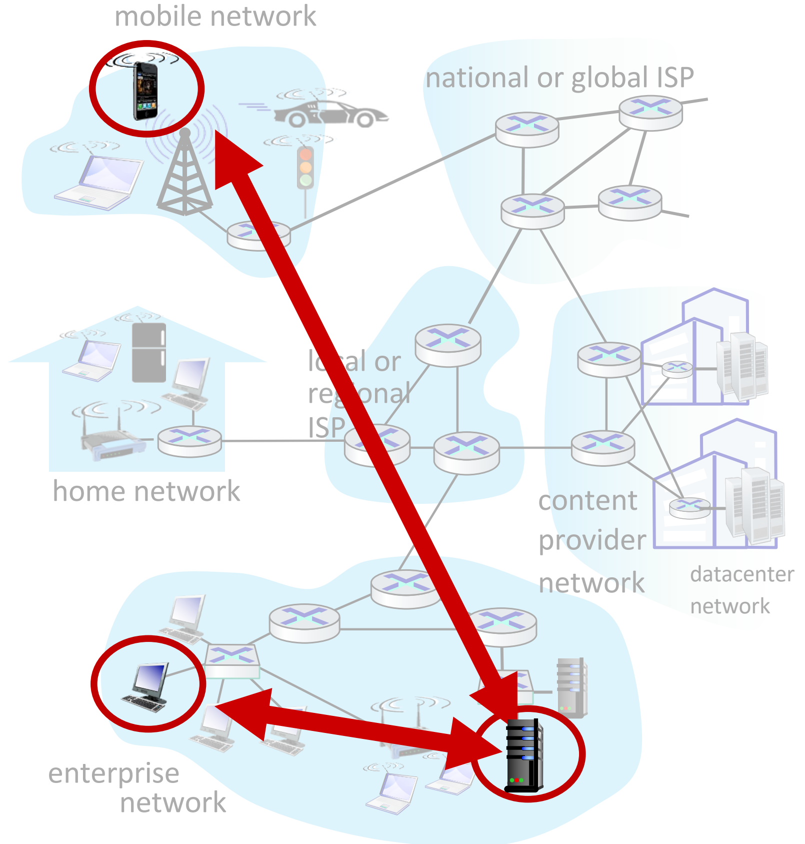
server:

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

clients:

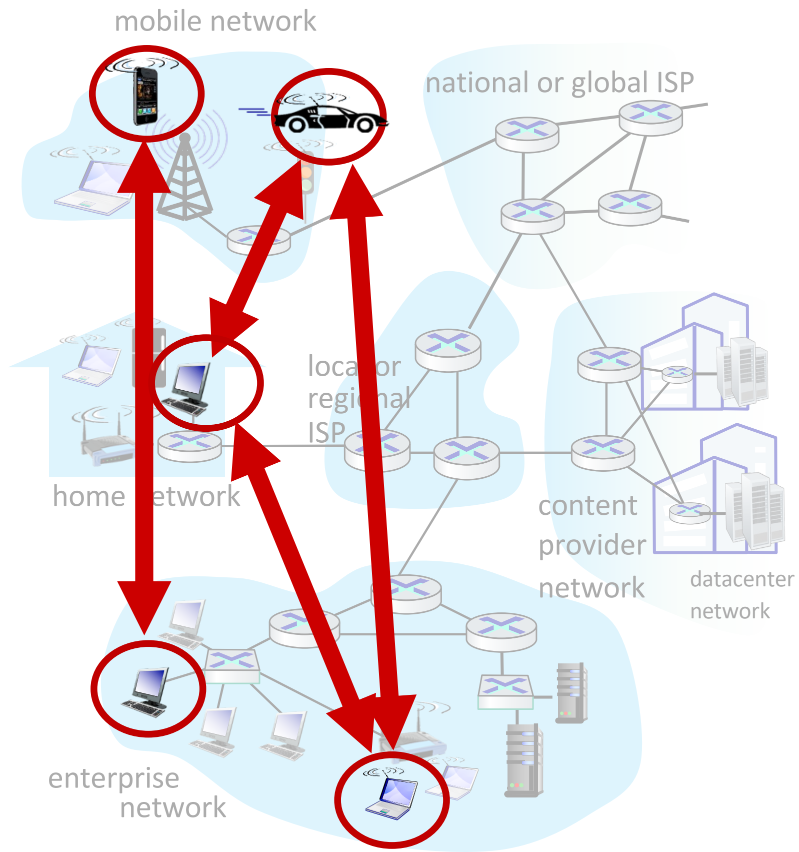
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-peer (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, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing



Hybrid of client-server and P2P

Skype
- voice-over-IP P2P application
- centralized server: finding address of remote party
- client-client connection: direct (not through server)

Text messaging
- chatting between two users is P2P
- centralized service: client presence detection/location

BitTorrent
- exchanging file chunks between users is P2P
- tracker: maintains list of peers participating in torrent

Processes communicating

process: program running within a host

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

clients, servers

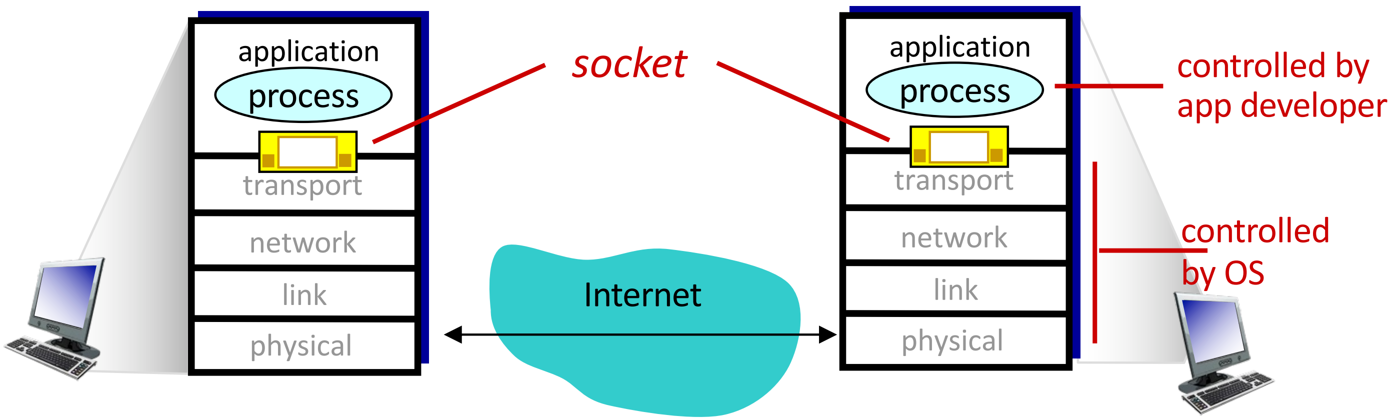
client process: process that initiates communication

server process: process that waits to be contacted

- aside: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



Addressing processes

- to receive messages, process must have *identifier*
- host device has (at least) one IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host
- *identifier* includes both IP address and port number associated with process on host
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

An application-layer protocol defines

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

Open protocols:

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

Proprietary protocols:

- e.g., Skype

What transport service does an application need?

Data integrity

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

Timing

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

Throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

Security

- encryption, data integrity, …

Transport service requirements of common applications

Application	Data loss	Throughput	Time Sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 10’s msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 10’s msec
text messaging	no loss	elastic	yes and no

Internet transport layer services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantees, security
- *connection-oriented:* setup required between client and server processes

useful for e.g. file transfer, email, web, streaming stored video

UDP service:

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

Q: why is there a UDP?

useful e.g. for internet telephony, interactive games, DNS

Securing TCP

Vanilla TCP & UDP sockets

- no encryption
- cleartext (e.g. passwords) sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

TLS implemented in app layer

- between apps and TCP
- apps use TLS libraries, that use TCP in turn

TLS socket API

- cleartext sent into socket traverse Internet *encrypted*

More on TLS in chapter 8 of book

Chapter 2: outline

2.1 Principles of network applications

2.2 Web and HTTP

2.3 The Domain Name System (DNS)

2.4 Socket programming with UDP and TCP

Web and HTTP

First, a quick review…

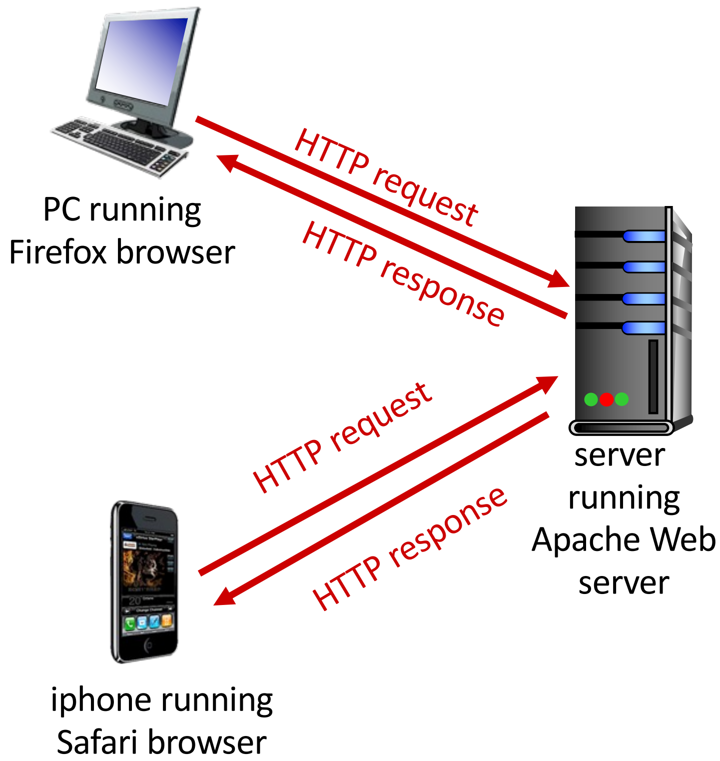
- Web page consists of objects, each of which can be stored on different Web servers
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of base HTML-file which includes several referenced objects, each addressable by a URL (Universal Resource Locator), e.g.:

$$\underbrace{\texttt{http:}}_{\text{protocol name}}\texttt{//}\underbrace{\texttt{www.someschool.edu}}_{\text{host name or IP addr}}\texttt{:}\underbrace{\texttt{8080}}_{\text{port (if non standard)}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

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 (why?):

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

Protocols that maintain "state" are complex!

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

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

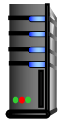
Persistent HTTP

- 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.someSchool.edu/someDepartment/home.index`
(containing text, references to 10 jpeg images)

1a. HTTP client 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 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

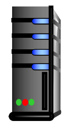
3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Non-persistent HTTP: example (cont.)

User enters URL: `www.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

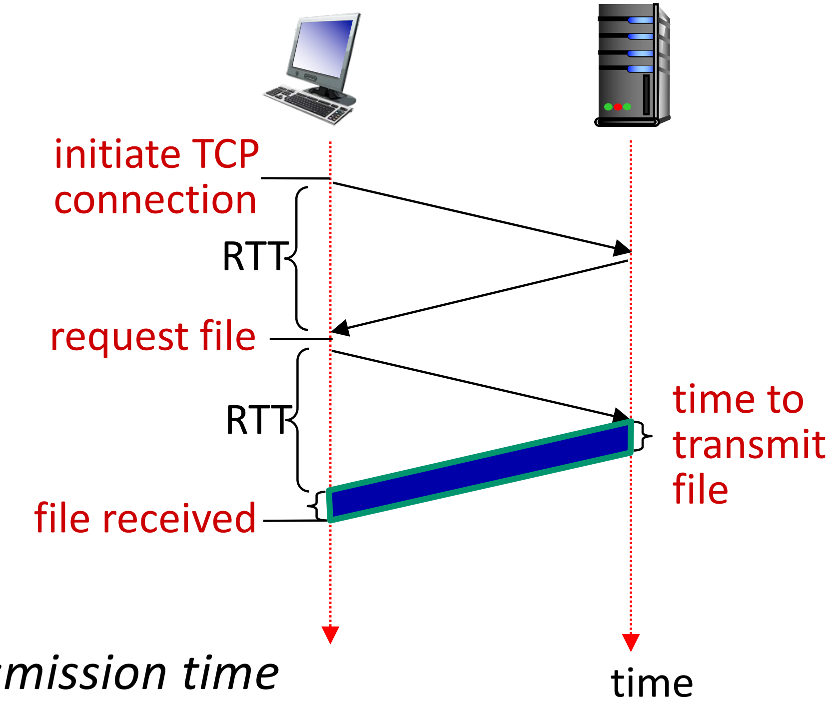
6. Steps 1-5 repeated for each of 10 jpeg objects

time

Non-persistent HTTP: response time

RTT (definition): Round Trip Time : 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

File transmission time = file size divided by the average throughput of underlying TCP connection

Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open 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 when requests are pipelined (cutting response time in half)

HTTP request message

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

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

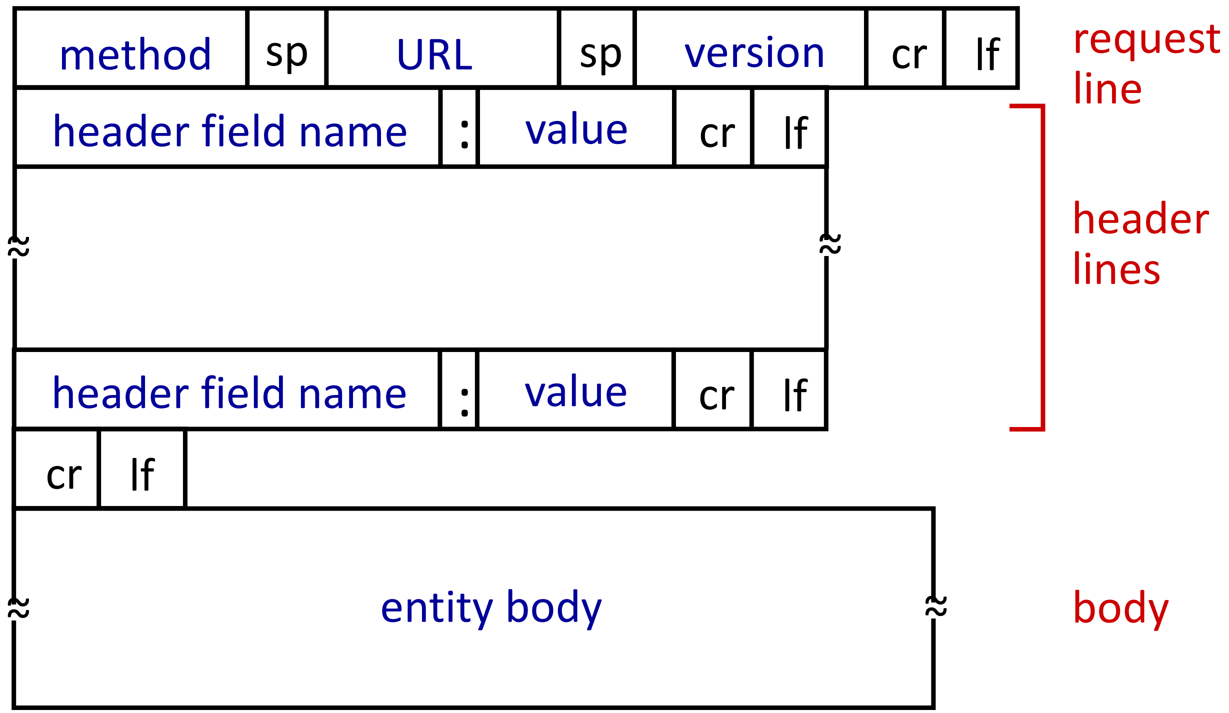
Why needed?

header lines

Persistent mode

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

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
- more generally POST sends data to a remote object specified in URL field

GET method (another technique 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/object that exists at specified URL with content in entity body of PUT HTTP request message

HTTP response message

status line (protocol
status code,
status phrase)

header
lines

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

data,
e.g., requested HTML file

HTTP response status codes

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

  200 OK
  - request succeeded, requested object later in this msg

  301 Moved Permanently
  - requested object moved, new location specified later in this msg (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. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

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

2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

- by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

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

Maintaining user/server state: cookies

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

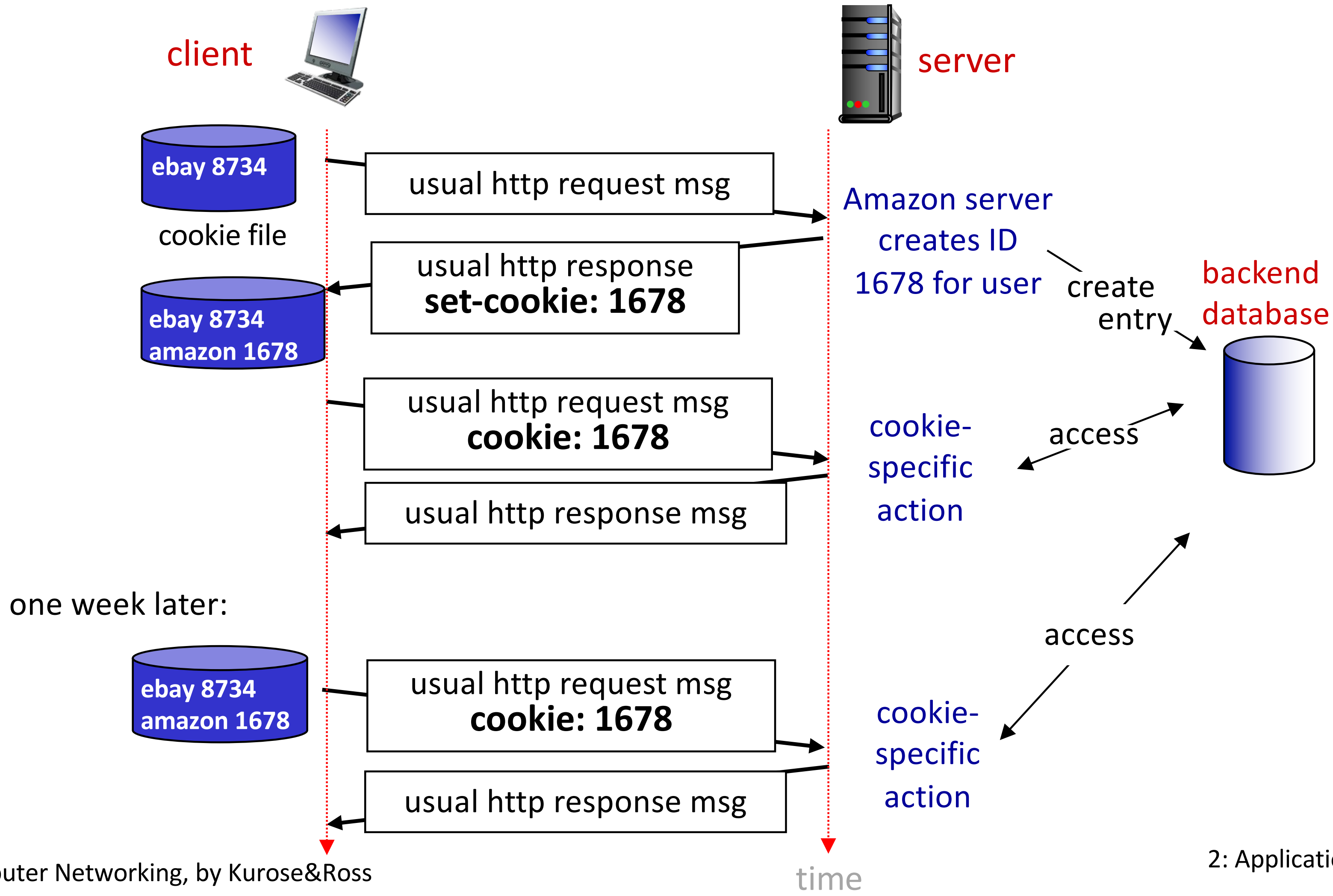
four components:

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:

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

aside

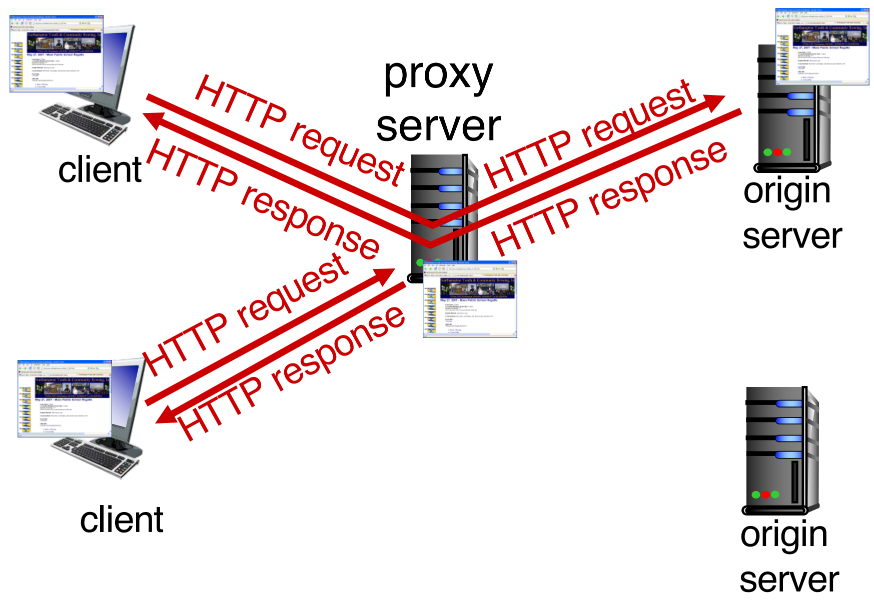
cookies and privacy:

- 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

Web caches (proxy server)

goal: satisfy client request without involving origin server

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



Web caches (proxy servers)

- cache acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

1. reduce response time for client request: cache is closer to client
2. reduce traffic on an institution's access link
3. reduce load on servers

- Internet is dense with caches: enables "poor" content providers to more effectively deliver content

Caching example

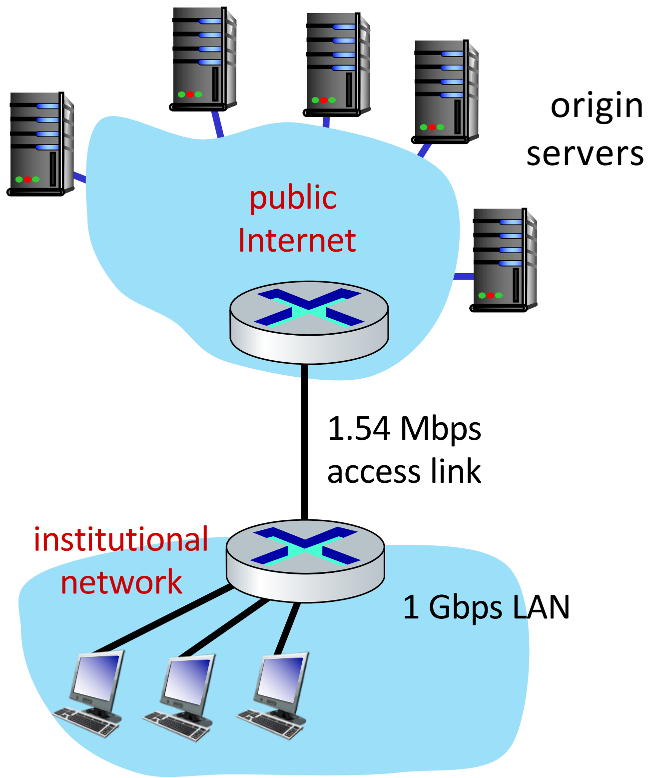
Scenario:

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

Performance:

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

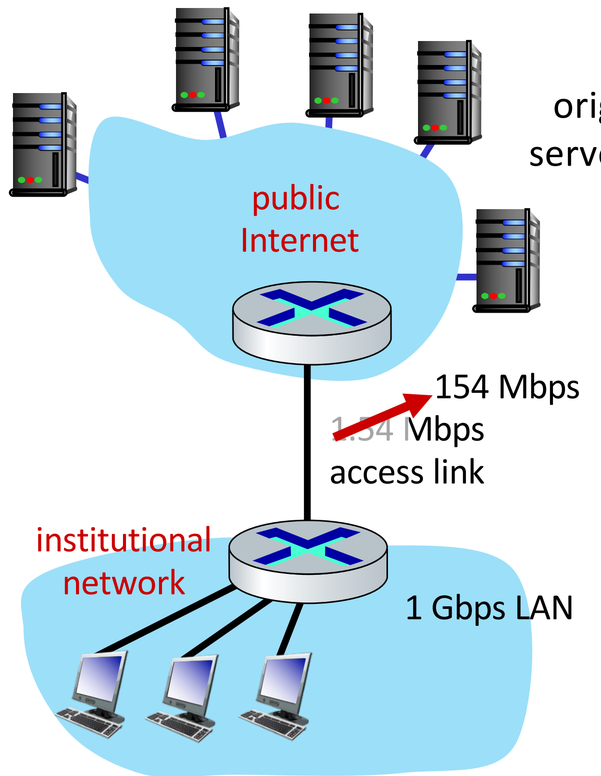
problem: large delays at high utilization!



Caching example: buy a faster access link

Scenario:

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



Performance:

- LAN utilization: .0015
- access link utilization = ~~.97~~ → .0097
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + ~~minutes~~ msecs + μsecs

Cost: faster access link (expensive!)

Caching example: install a web cache

Scenario:

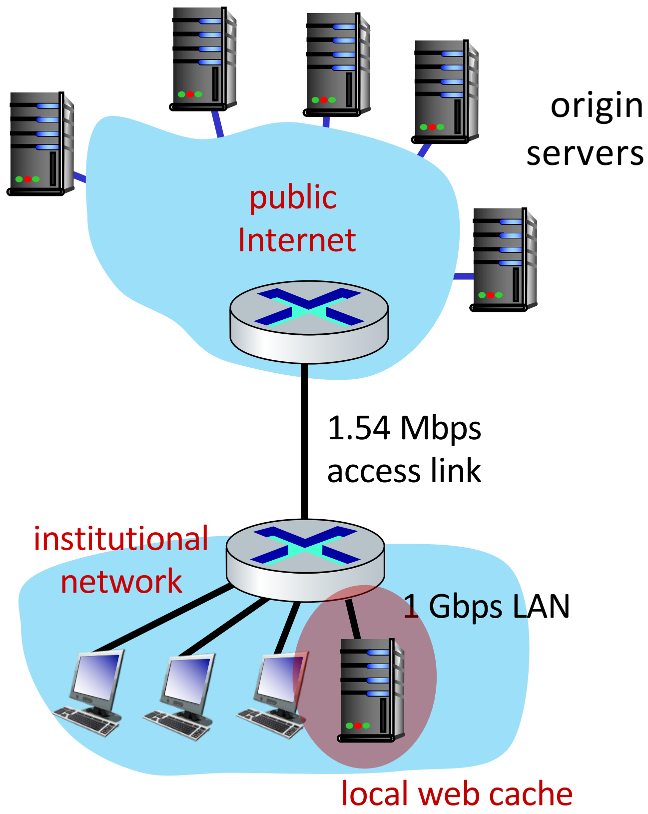
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:

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

How to compute link utilization, delay?

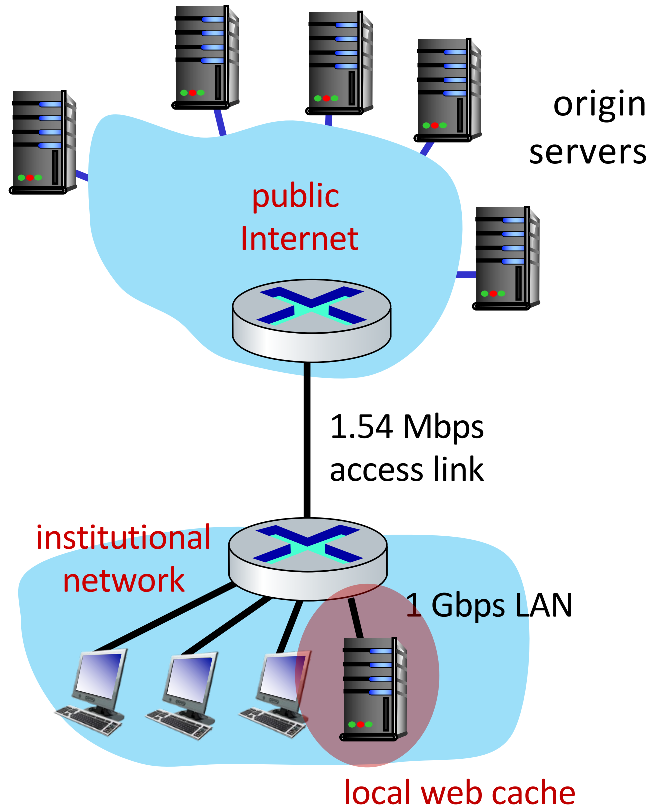
Cost: web cache (cheap!)



Caching example: install a web cache

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

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link
  = 0.6 * 1.50 Mbps = 0.9 Mbps
- utilization = 0.9/1.54 = 0.58
- average end-end delay
  = 0.6 * (delay from origin servers)
  + 0.4 * (delay when satisfied at cache)
  = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs



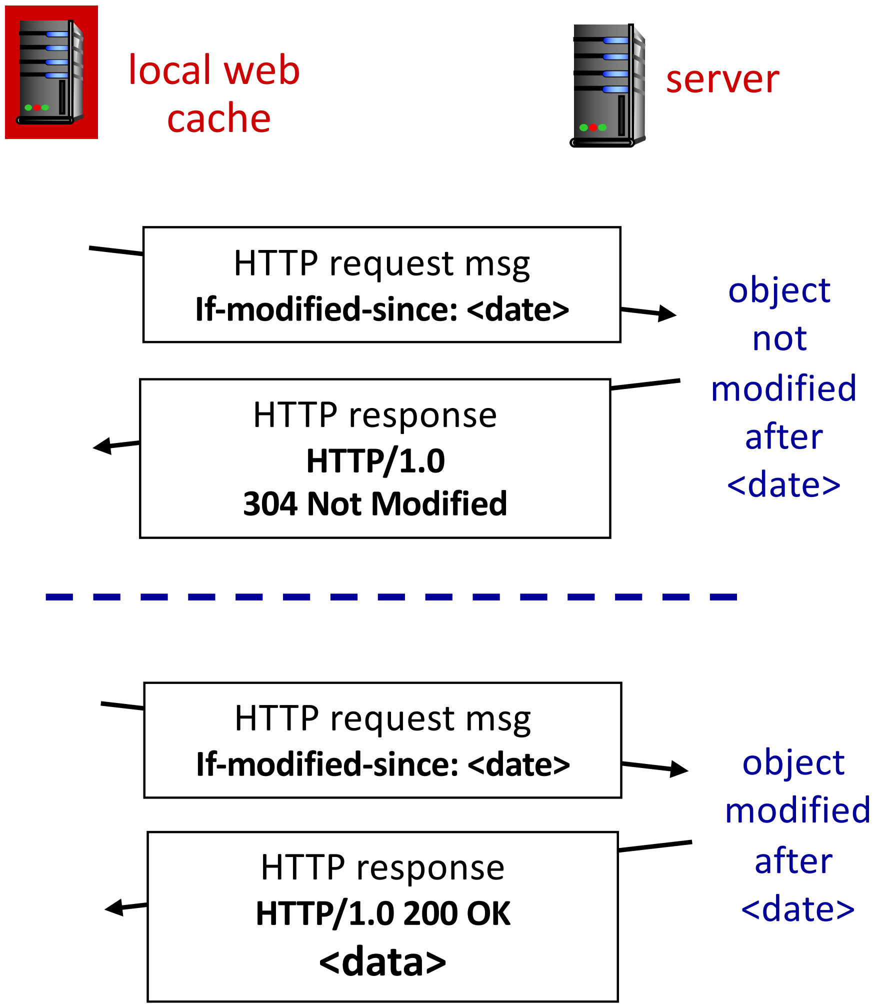
lower average end-end delay than with 154 Mbps link (and cheaper too!)

Conditional GET

- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *cache:* specify date of cached copy in HTTP request

  If-modified-since: <date>
- *server:* response contains no object if cached copy is up-to-date:

  HTTP/1.0 304 Not Modified



HTTP/2

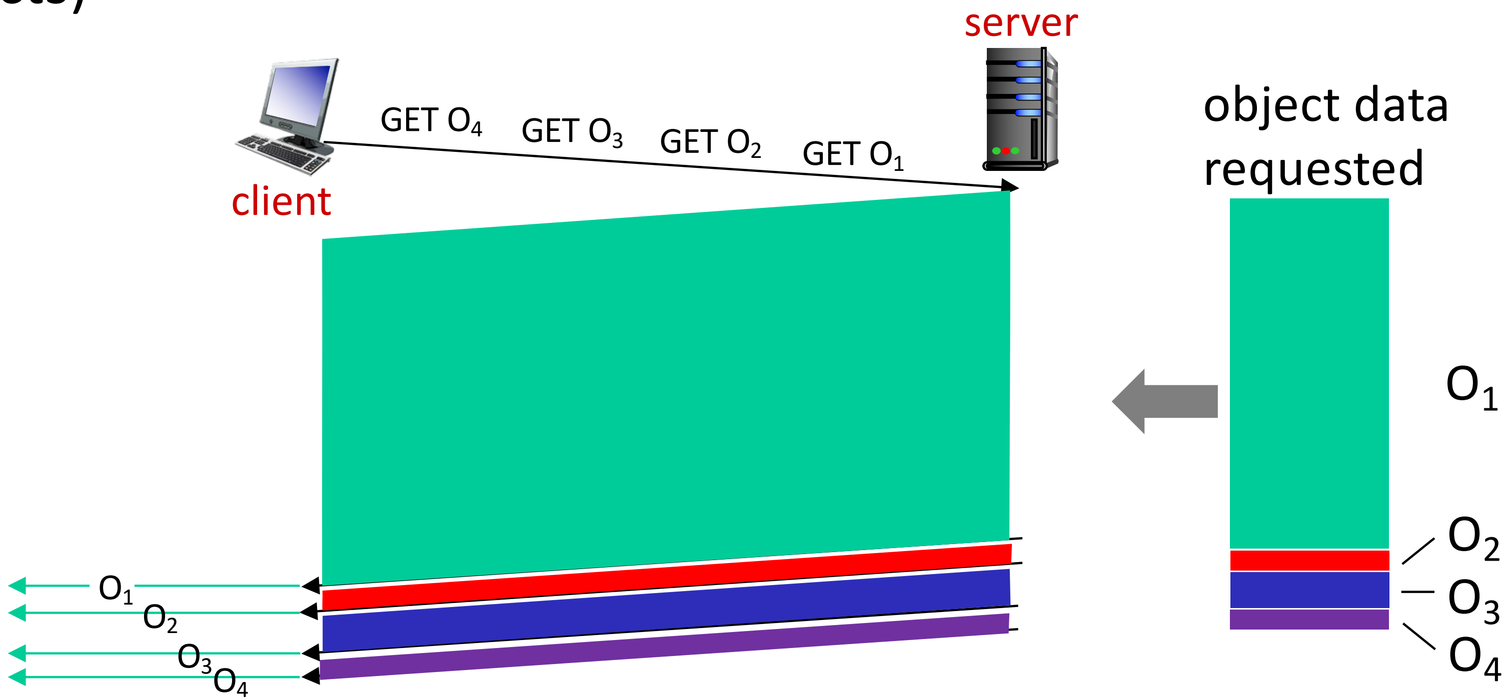
Key goal: decreased delay in multi-object HTTP requests

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

HTTP1.1: HOL blocking

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



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

HTTP/2

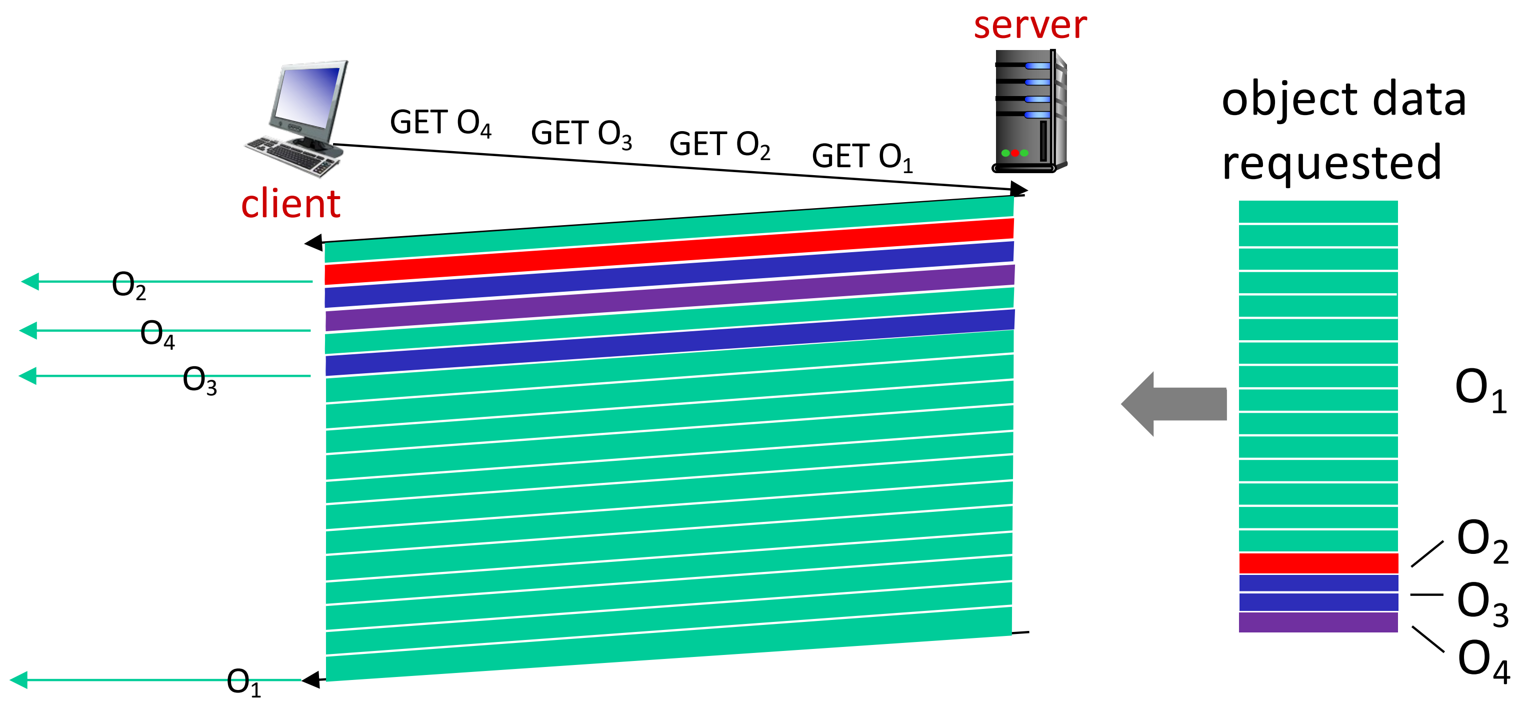
Key goal: decreased delay in multi-object HTTP requests

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

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

HTTP/2: mitigating HOL blocking

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



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/2 to HTTP/3

Key goal: decreased delay in multi-object HTTP requests

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
  - more on HTTP/3 in transport layer

Chapter 2: outline

2.1 Principles of network applications

2.2 Web and HTTP

2.3 The Domain Name System (DNS)

2.4 Socket programming with UDP and TCP

DNS: Domain Name System

People: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

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

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

Domain Name System:

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

DNS: services, structure

DNS services

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

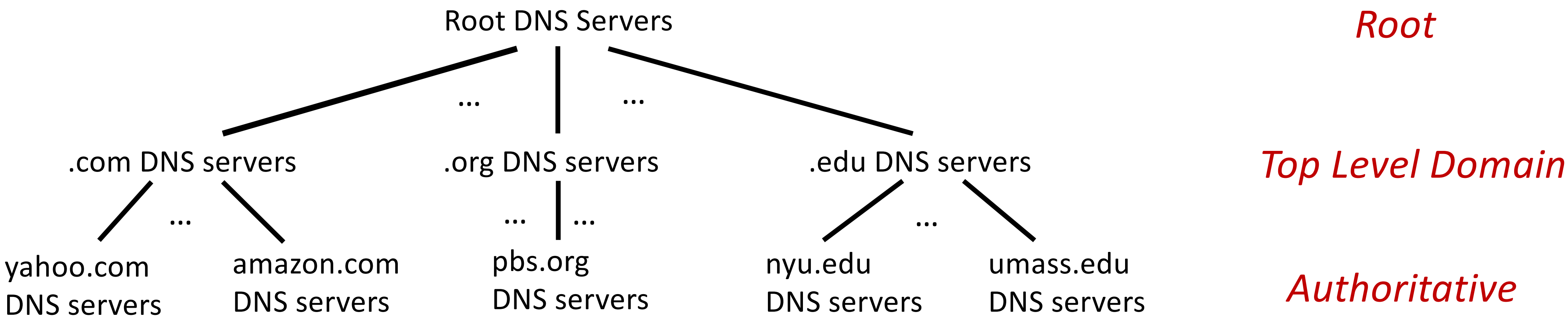
Q: Why not centralize DNS?

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

A: doesn‘t scale!

- Comcast DNS servers alone: 600 billion DNS queries per day

DNS: a distributed, hierarchical database



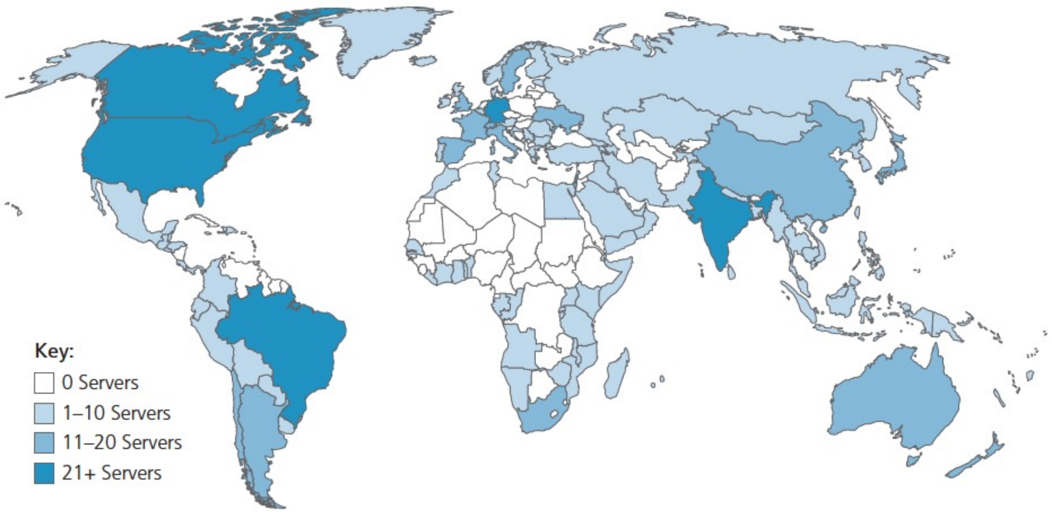
Client wants IP address for www.amazon.com; 1st approximation:

- 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 cannot resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

Root DNS managed by 13 organizations worldwide, each deploying one logical "server" replicated many times (~200 servers in US, more than 400 worldwide)



TLD and authoritative servers

- Top-level domain (TLD) servers:
  - responsible for .com, .org, .net, .edu, .jobs, …
    and all top-level country domains, e.g.: .uk, .fr, .be
  - Network Solutions maintains servers for .com and .net TLD
  - DNS Belgium for the .be TLD

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

Local DNS name server

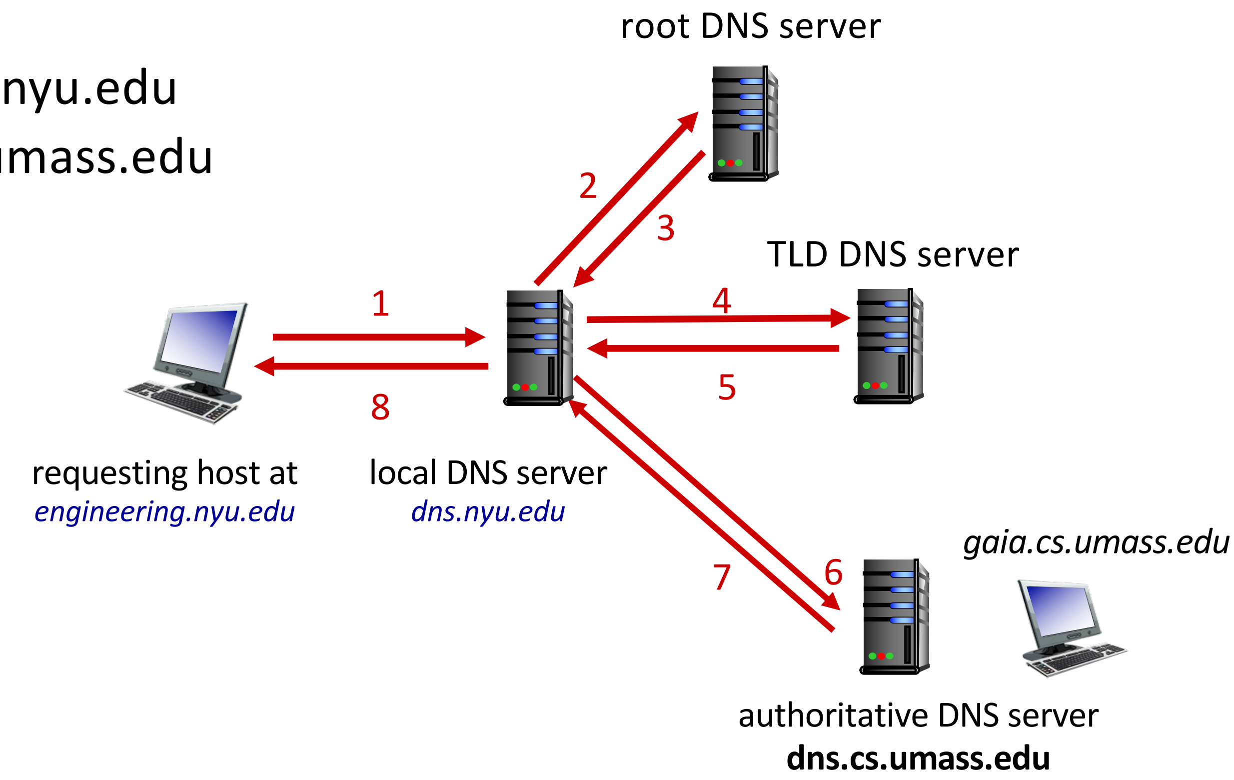
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

DNS name resolution: iterated query

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

Iterated query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"
- caching in local DNS
  - can skip steps

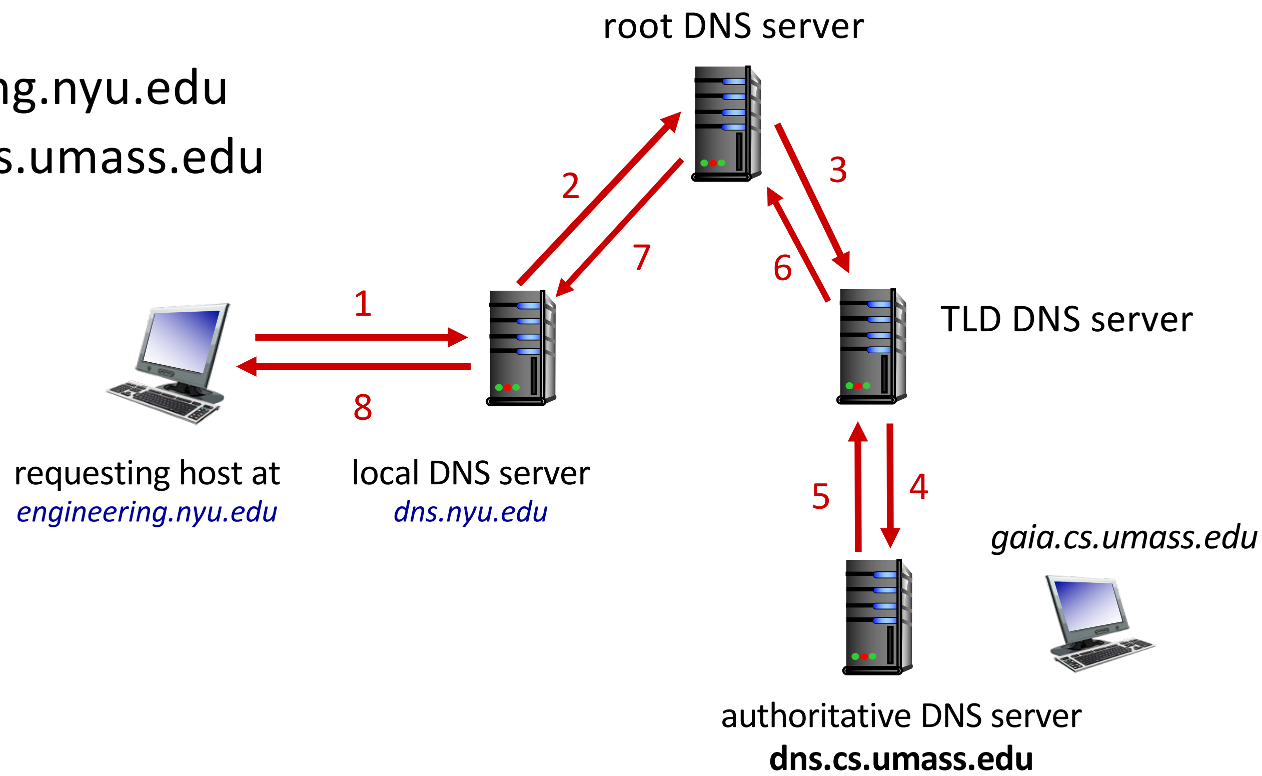


DNS name resolution: recursive query

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?
- caching, where? what?



Caching and updating DNS records

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

DNS records

DNS: distributed database storing resource records (RR)

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

- Type=A (or AAAA)
  - **`name`** is hostname (aka FQDN)
  - **`value`** is IPv4 (or IPv6) address
- Type=NS
  - **`name`** is domain, DNS zone (e.g. amazon.com)
  - **`value`** is hostname of authoritative name server for this domain
  - domain name ≠ hostname
- Type=CNAME
  - **`name`** is alias name for some "canonical" (the real) name

  `www.ibm.com` is really

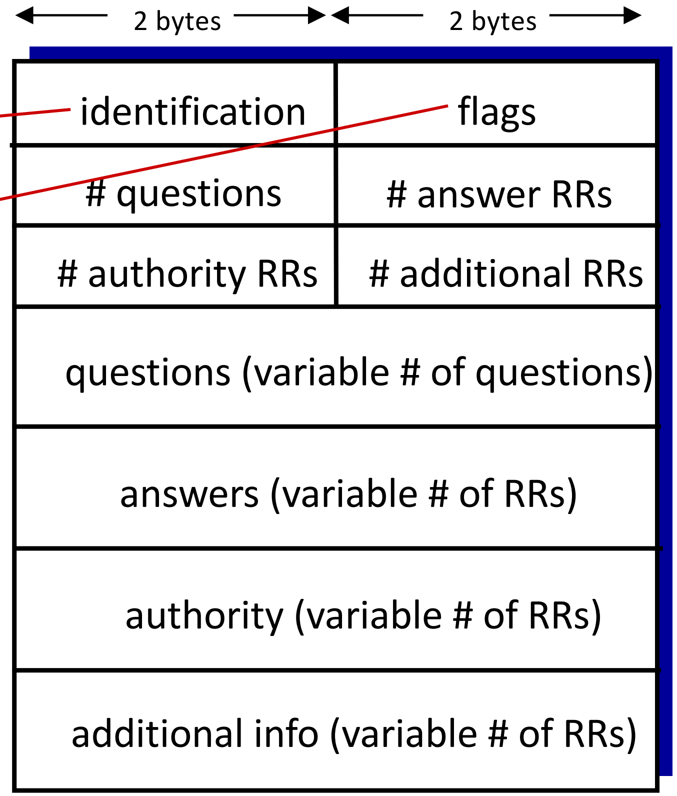
  `servereast.backup2.ibm.com`
  - **`value`** is canonical name
- Type=MX
  - **`value`** is name of mail server associated with the domain **`name`**

DNS protocol messages

DNS *query* and *reply* messages, both with same *message format*
(usually sent over UDP, why?)

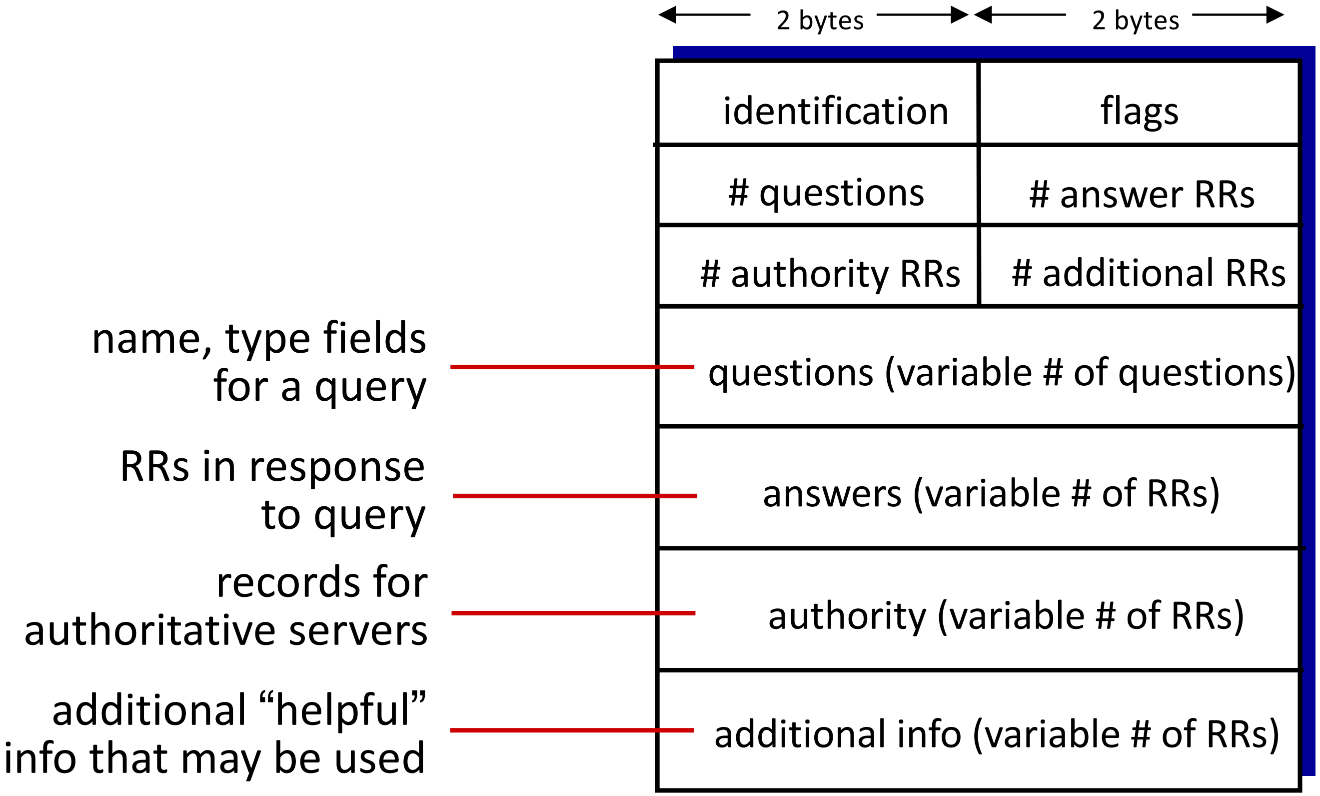
msg header

- 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:*



Inserting records into DNS

Example: new startup "Network Utopia"

- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two 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.networkuptopia.com
  - type MX record for networkutopia.com

Chapter 2: outline

2.1 Principles of network applications

2.2 Web and HTTP

2.3 The Domain Name System (DNS)

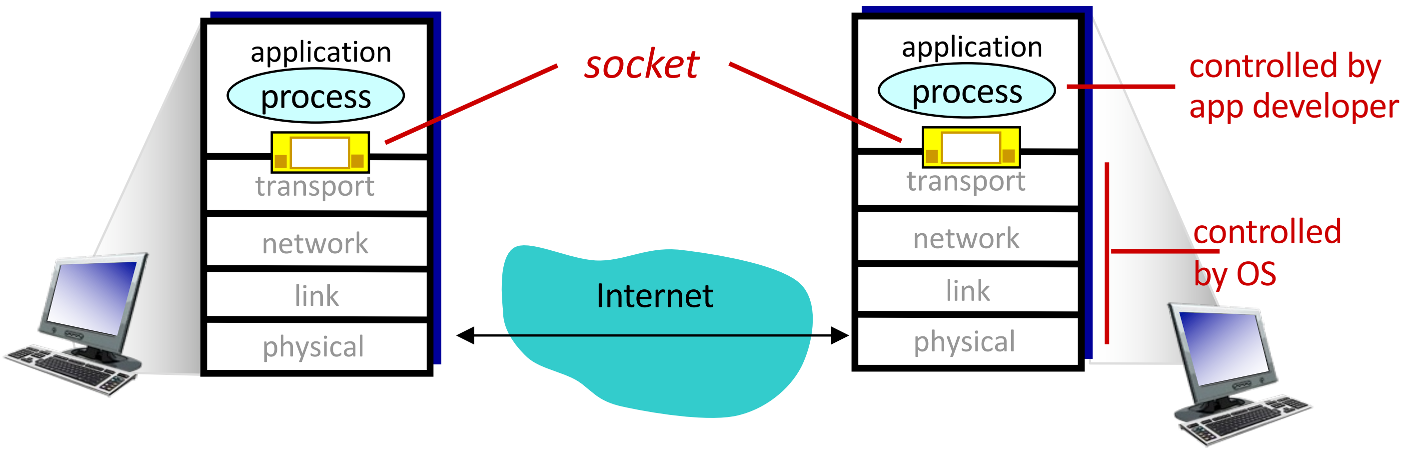
2.4 Socket programming with UDP and TCP

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol

Socket API introduced in BSD4.1 UNIX, 1981



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Socket programming *with UDP*

UDP: no "connection" between client & server processes

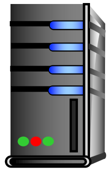
- no handshaking before sending data
- sender process explicitly attaches IP destination address and port # to each application data unit (aka datagram)
- receiver process extracts sender IP address and port# from received datagram

UDP: transmitted datagram may be lost or received out-of-order

Application viewpoint:

UDP provides *unreliable* transfer of datagrams between client and server

Client/server socket interaction: UDP



Server (running on serverIP)

Client

socket call
+ bind call

create socket, port=**x**,
for incoming requests:
serverSocket =
DatagramSocket(x)

read datagram from
serverSocket

discovers
client host
address,
port #

write reply to
serverSocket
specifying client
address,
port number

create socket,
clientSocket =
DatagramSocket()

port# unspecified,
dynamically assigned
by OS,
no binding necessary

Create datagram with server IP and
port=x; send datagram request
via clientSocket

read datagram from
clientSocket

close
clientSocket

UDP observations & questions

- Both client and server processes use same DatagramSocket
- Destination IP address and port # are explicitly attached to datagram by client and server processes when sent through socket
- Source IP address and port # are discovered by client and server processes when they receive a datagram
- Can multiple clients use the server?
  - How many sockets needed by server?
  - How many port numbers used at server side?

Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact (welcoming socket)

Client contacts server by:

- creating TCP socket specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

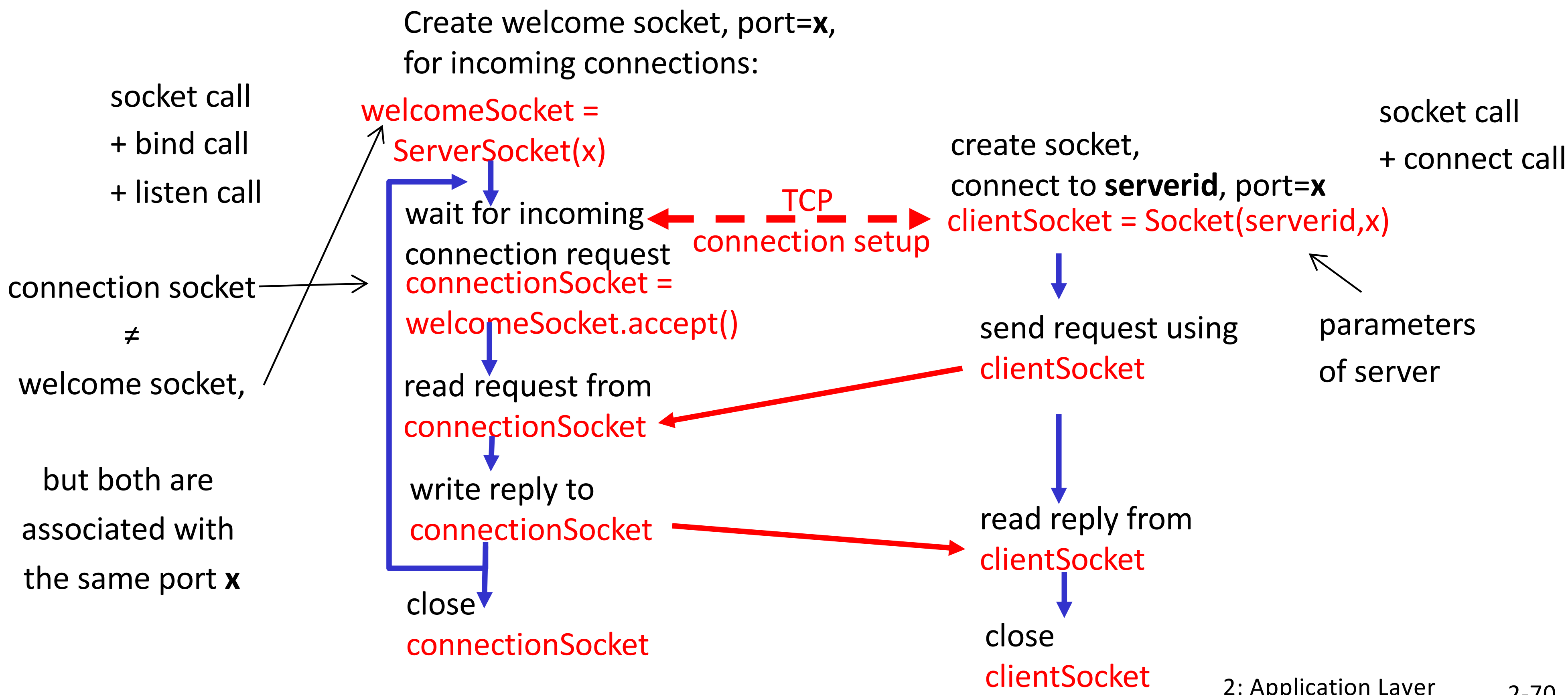
- When contacted by client, server TCP creates new socket for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source IP and source port numbers used to distinguish clients (more in Chap 3)

application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP

Server (running on **serverid**) Client



TCP observations & questions

- Two types of socket primitives: ServerSocket and Socket
- When client knocks on server's welcome socket, server creates a dedicated connectionSocket and completes TCP connection
- Destination IP address and port # are not explicitly attached to application data unit sent and received by client and server processes
  - The application data unit is just a block of application data bytes
- Can multiple clients use the server?
  - How many sockets needed by server?
  - How many port numbers used at server side?

Chapter 2: Summary

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, throughput, delay
- Internet transport service model
  - connection-oriented, reliable, byte-stream : TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - DNS
- socket programming:
  - TCP, UDP sockets

Chapter 2: Summary

Most importantly: learned about *protocols*

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data* (payload): info being communicated

Important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”