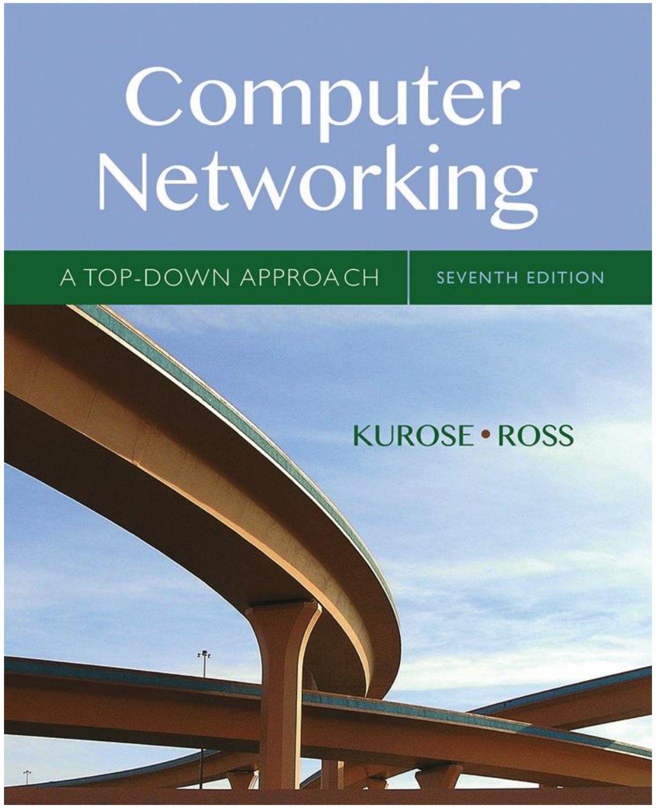
Chapter 2
Application Layer



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

Some network apps

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

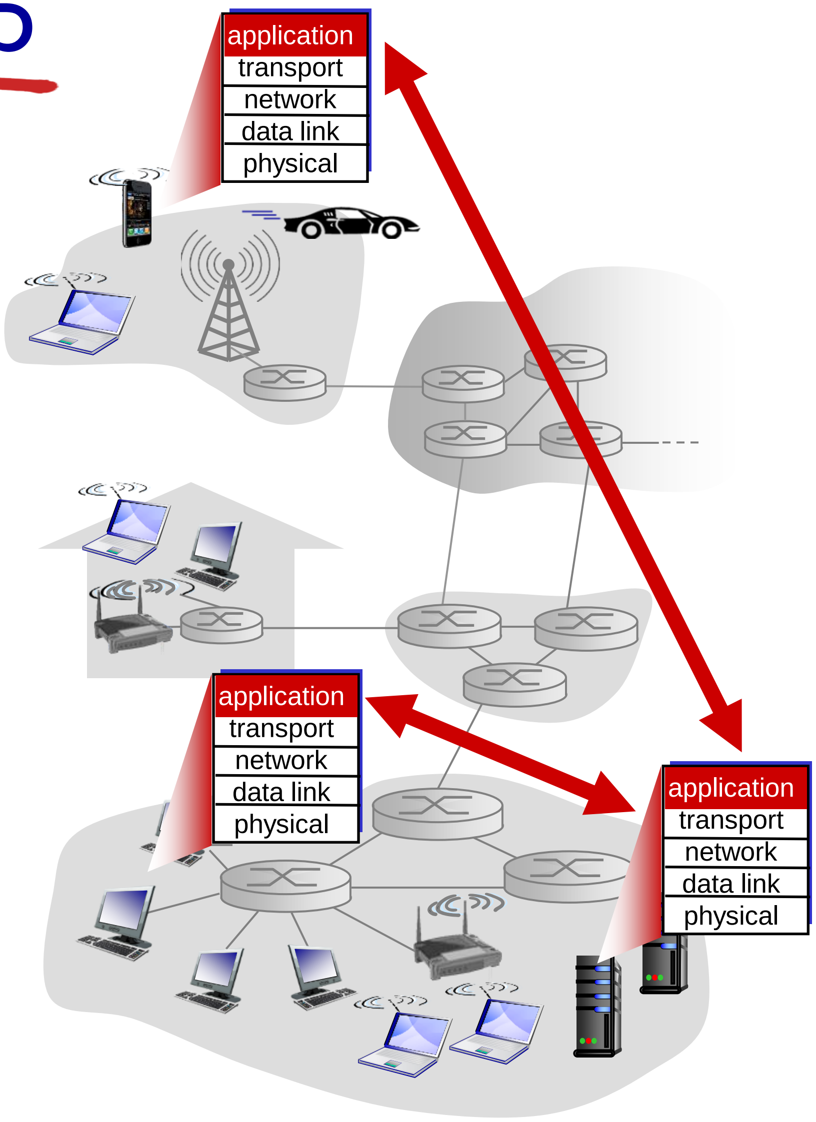
Creating a network app

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

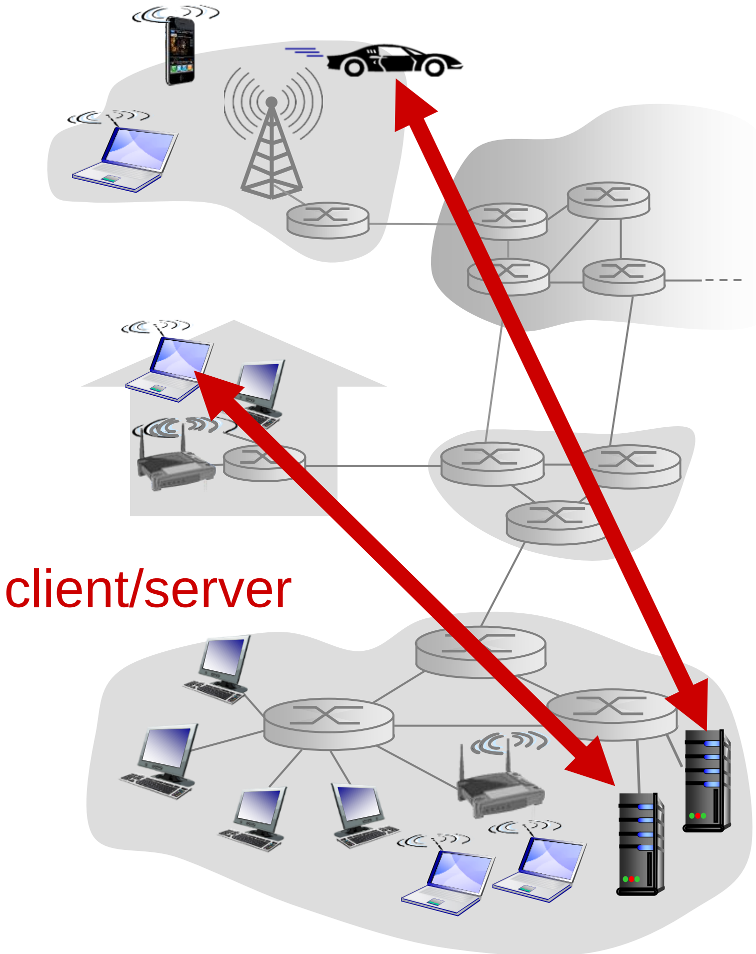


Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)

Client-server architecture



server:

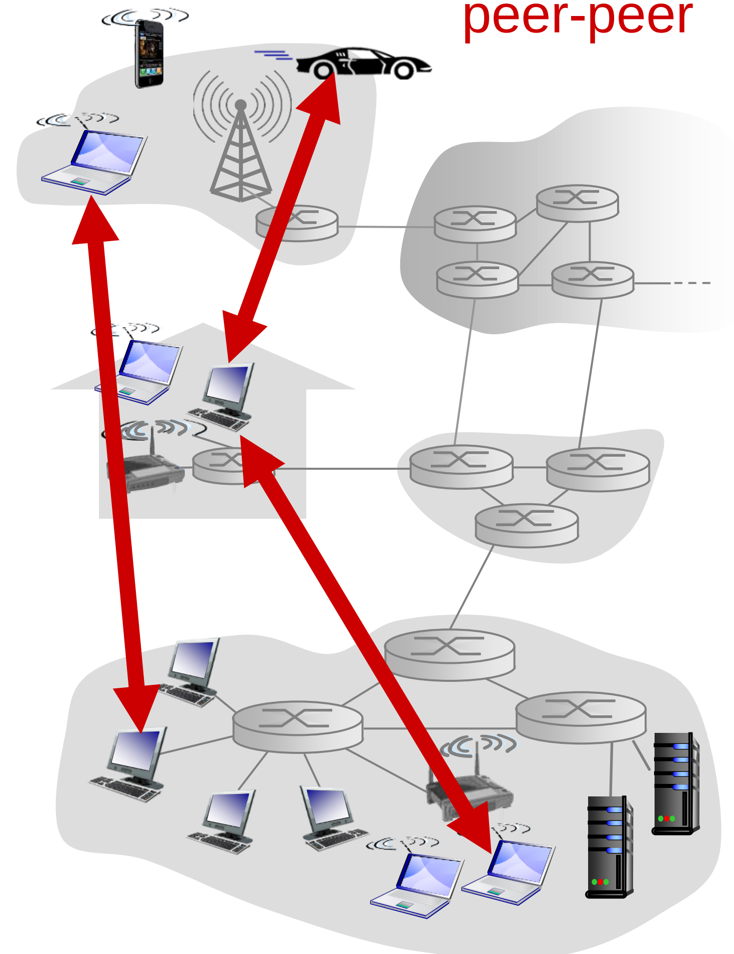
- always-on host
- permanent IP address
- data centers for scaling

clients:

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

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



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

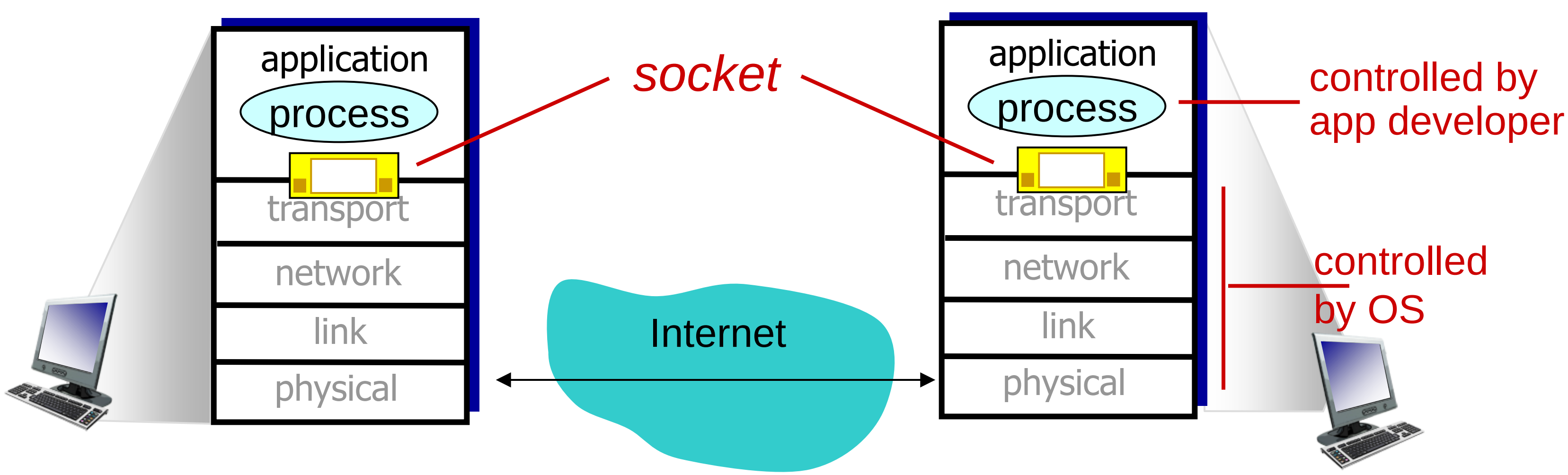
client process: process that initiates communication

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



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit 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 numbers 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…

App-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
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:
- e.g., Skype

What transport service does an app need?

data loss

- 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

Transport service requirements: common apps

application	data loss	throughput	time sensitive
file transfer	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, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
smartphone messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process (in order delivery, recovery of lost data)
- *flow control:* sender won' t overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

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?

More about transport protocols in chapter#3

Internet apps: application, transport protocols

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

More about transport layer protocols in chapter 3….

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

Web and HTTP

First, a review…

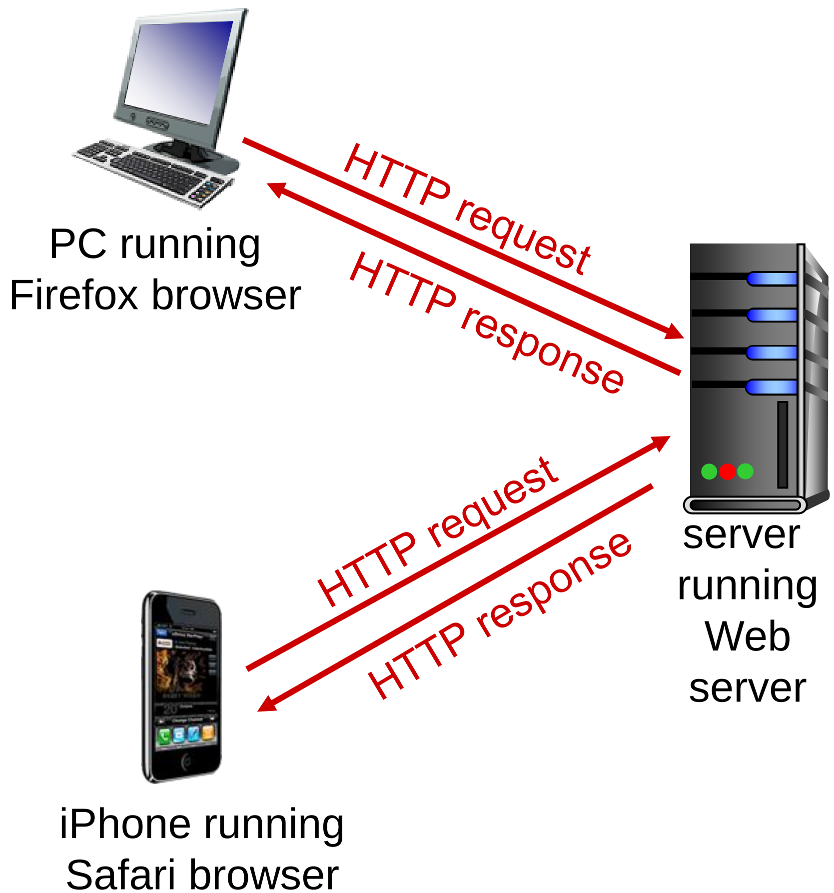
- *web page* consists of *objects*
- object can be HTML file, JPEG image, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL,* e.g.,

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

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)

uses TCP:

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

HTTP connections

non-persistent HTTP

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

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server, serially.
- reduces TCP overhead

Non-persistent HTTP (Serial)

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`

(contains 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 (cont.)

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**

time

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

Non-persistent HTTP: response time

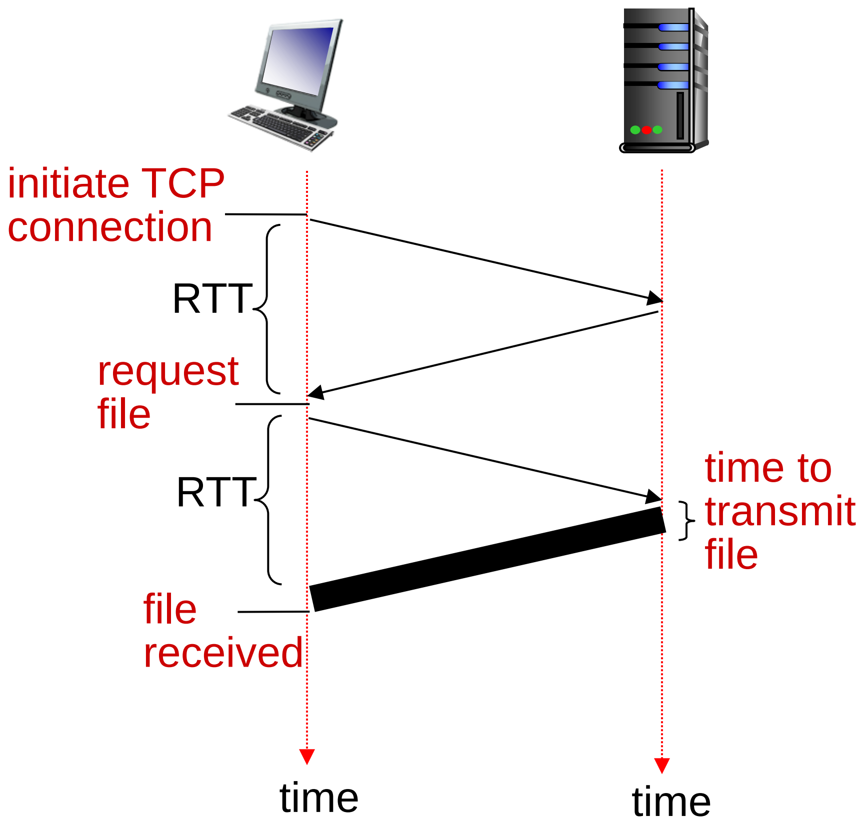
RTT (definition): time for a *small packet* to travel from client to server and back

HTTP response time:

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

  2RTT+ file transmission time

File transmission time = ?



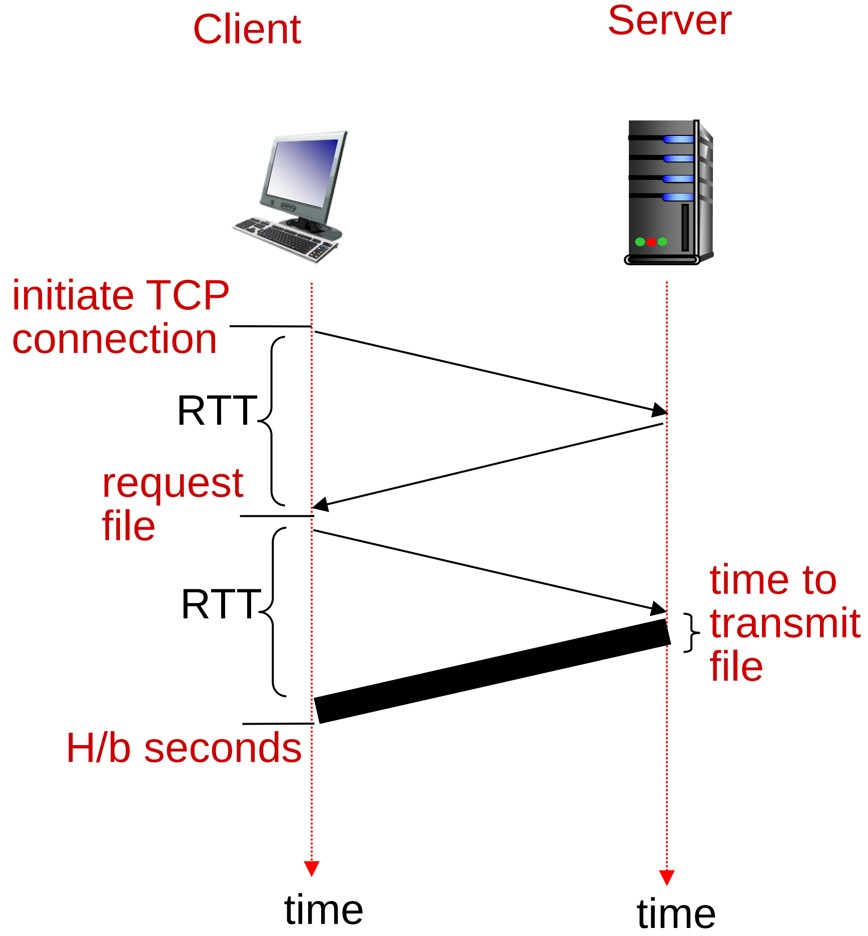
Non-persistent HTTP: response time

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

HTTP response time:

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

  2RTT+ file transmission time



Persistent HTTP

non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection

persistent HTTP:

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- For example, it will take total of 6 RTTs to download a webpage containing 4 objects.

Class Exercise

- *Consider a client fetching 5 objects from a server. Assume that all objects are of same sizes (B bits) and the client is using persistent HTTP connection. The transmission rate is R bits/second. Draw the timeline diagram for the request response events between client and server and find out the total delay.*

HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:

carriage return character

line-feed character

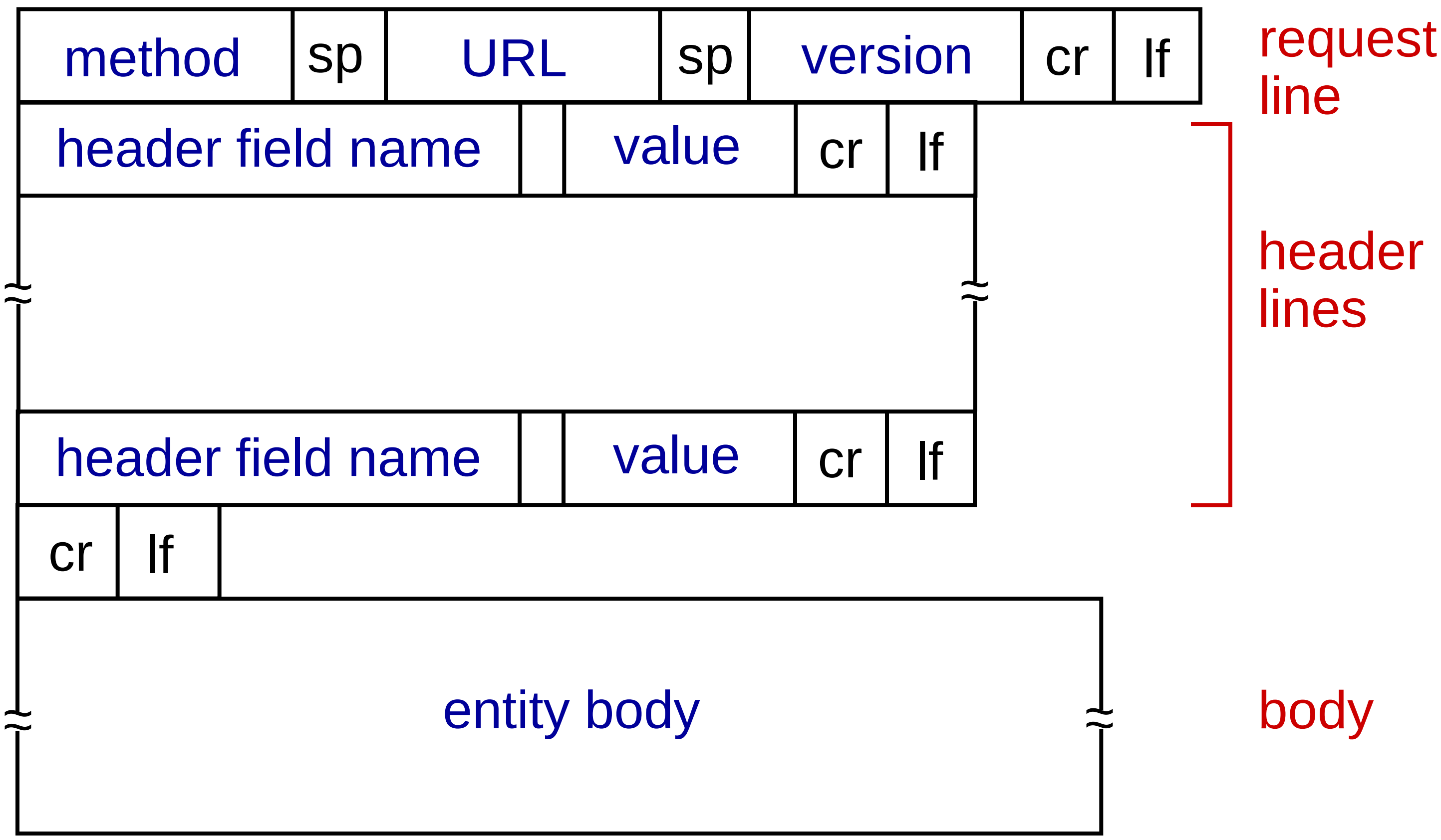
request line
(GET, POST,
HEAD commands)

header
lines

```
GET /index.html HTTP/1.1\r\n
Host: www.someschool.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept-Language: en-us,en;q=0.5\r\n
Connection: close\r\n
\r\n
```

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

HTTP request message: general format



Method types

- GET: retrieve a file
- POST: submitting a form to a server
- HEAD: asks server to leave requested object out of response, and just get meta-data
- Put: uploads file in entity body to path specified in URL
- Delete: remove named source

HTTP response message

status line (protocol status code status phrase)

```
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
Content-Length: 2652\r\n
Connection: close\r\n
Content-Type: text/html\r\n
\r\n
```

header lines

data, e.g., requested HTML file

```
data data data data data ...
```

HTTP response status codes

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

`200 OK`

  - request succeeded, requested object later in this msg

`301 Moved Permanently`

  - requested object moved, new location specified later in this msg (Location:)

`400 Bad Request`

  - request msg not understood by server

`404 Not Found`

  - requested document not found on this server

`505 HTTP Version Not Supported`

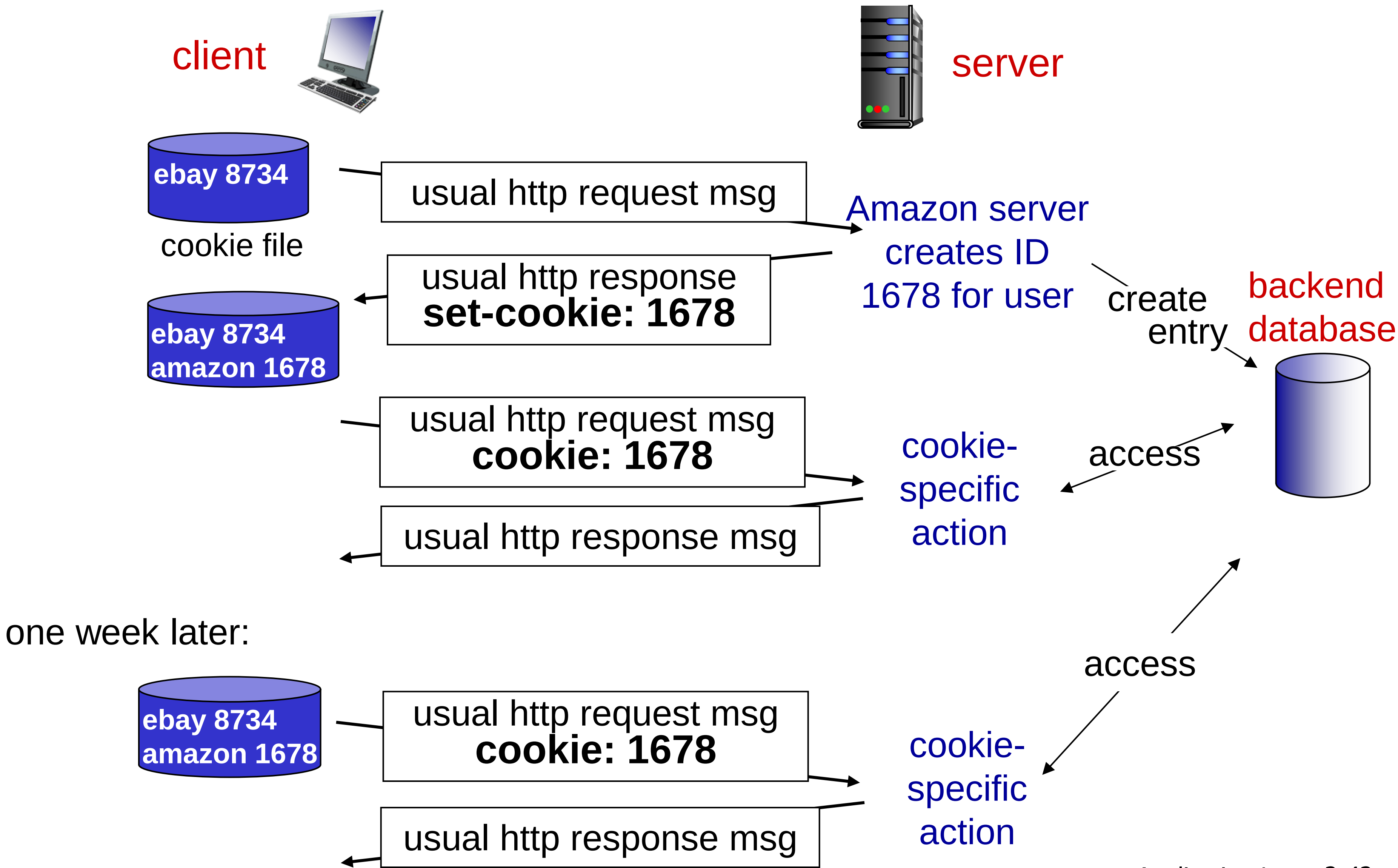
User-server state: cookies

many Web sites use cookies

example:

- Susan always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)



User-server state: cookies

many Web sites use cookies

four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

Cookies (continued)

what cookies can be used for:

- authorization
- shopping carts
- recommendations

aside

cookies and privacy:

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

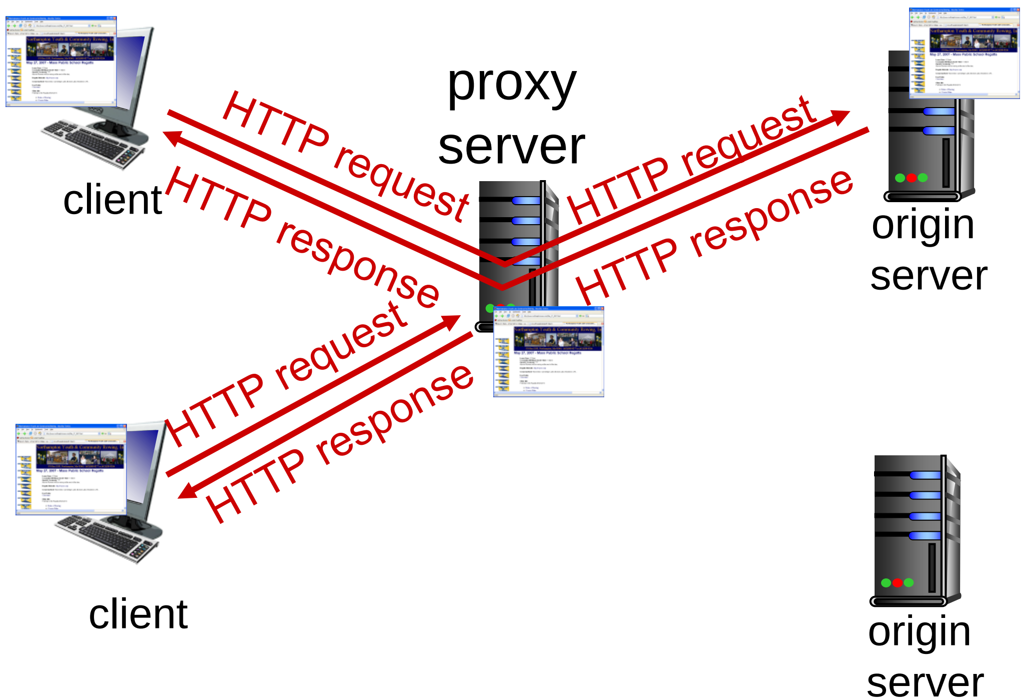
how to keep "state":

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

Web caches (proxy server)

goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - If object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

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

why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables "poor" content providers to effectively deliver content

Caching example:

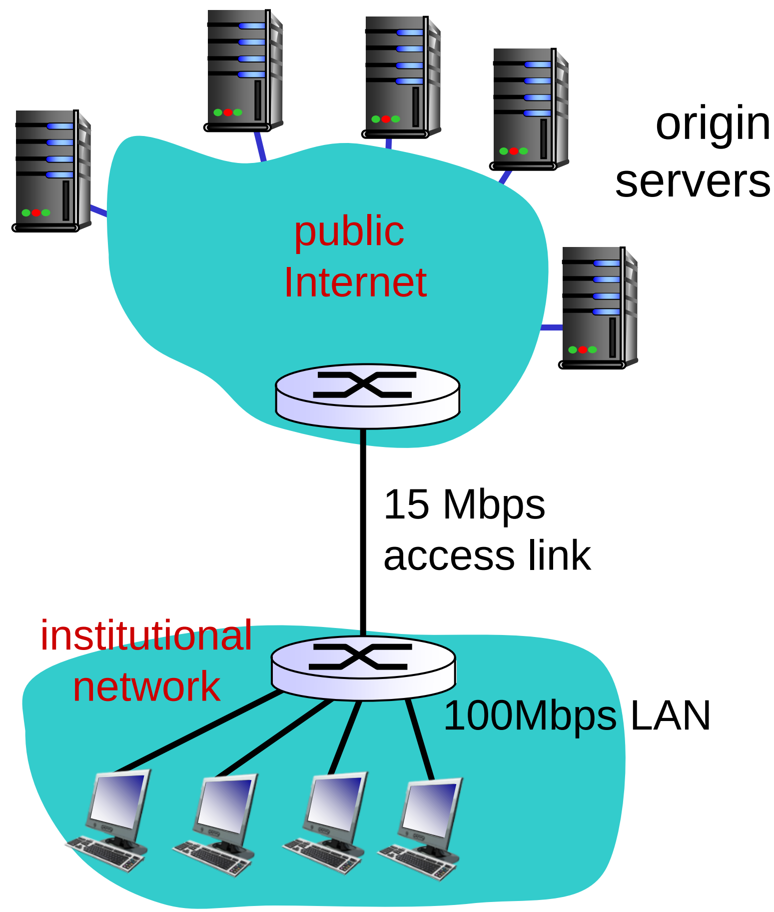
assumptions:

- avg object size: 1Mbits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 100 Mbps
- RTT from *internet router* to any origin server: 2 sec
- access link rate: 15 Mbps

consequences:

- LAN utilization: 1%
- access link utilization = 100%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + seconds + msecs

≥ 3 seconds



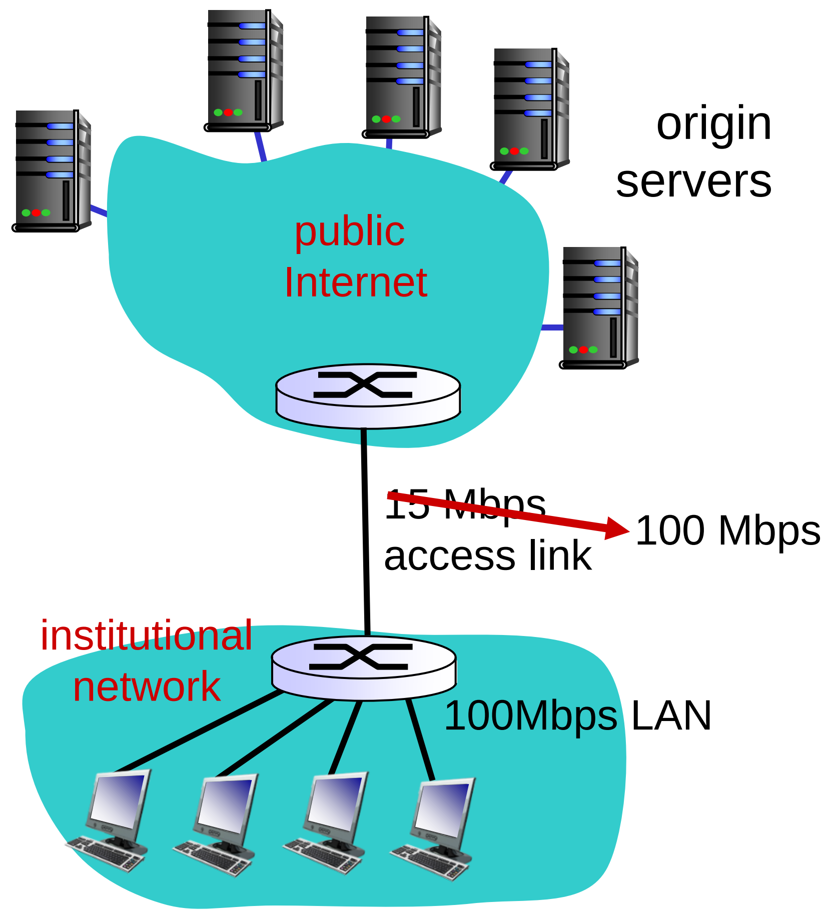
Caching example: fatter access link

assumptions:

- avg object size: 1Mbits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 100 Mbps
- RTT from internet router to any origin server: 2 sec
- access link rate: ~~15 Mbps~~ 100 Mbps

consequences:

- LAN utilization: 1%
- access link utilization = ~~100%~~ 15%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + ~~seconds~~ msecs + msecs



Cost: increased access link speed (not cheap!)

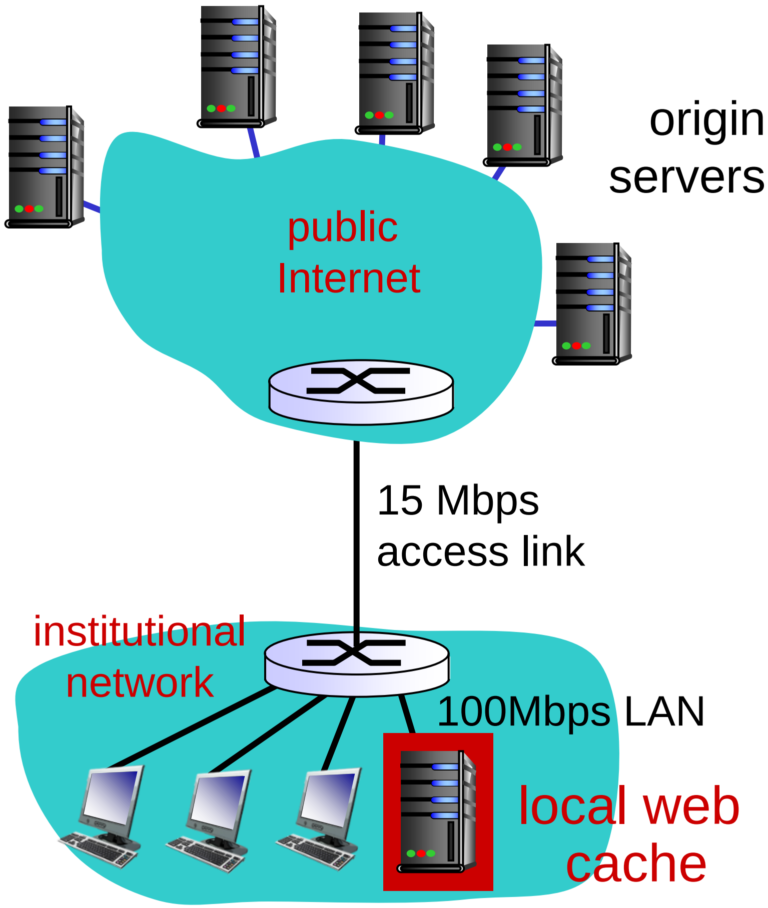
Caching example: install local cache

assumptions:

- avg object size: 1M bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 100 Mbps
- RTT from internet router to any origin server: 2 sec
- access link rate: 15 Mbps

consequences:

- LAN utilization: 1%
- access link utilization = 100%
- total delay = Internet delay + access delay + LAN delay
  = 0 + 0 + 15 msecs

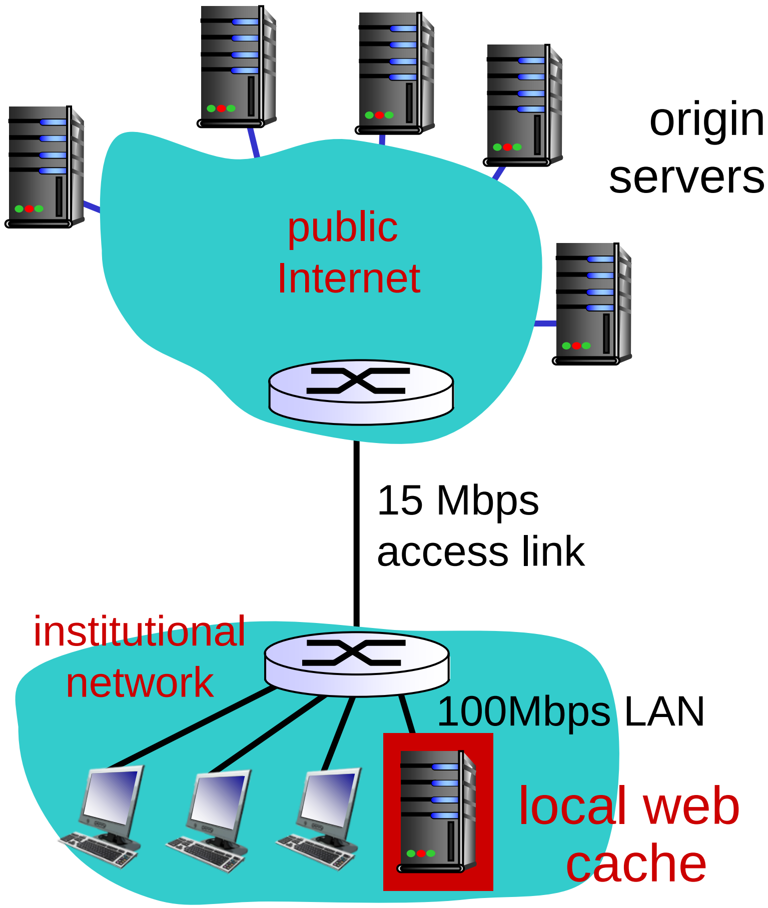


Cost: web cache (cheap!)

Caching example: install local cache

Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin



- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (3.15) + 0.4 (0.15) = ~ 1.95 secs
  - less than with 100 Mbps access 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