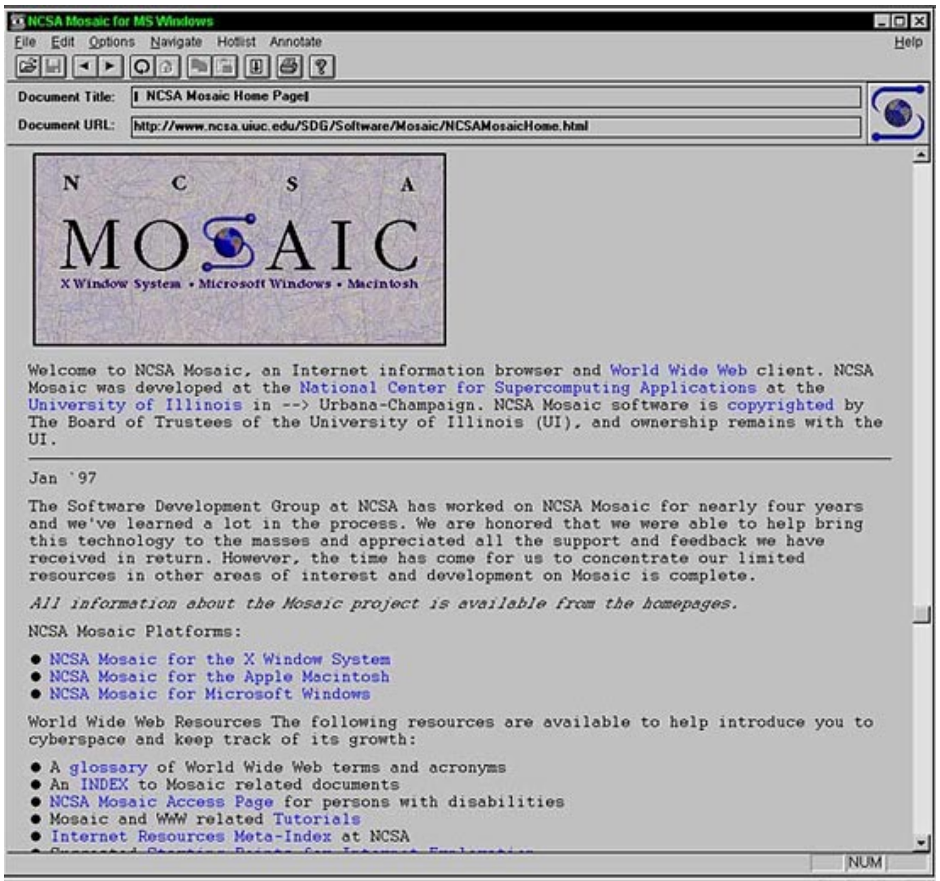
Application Layer
- Web and HTTP -

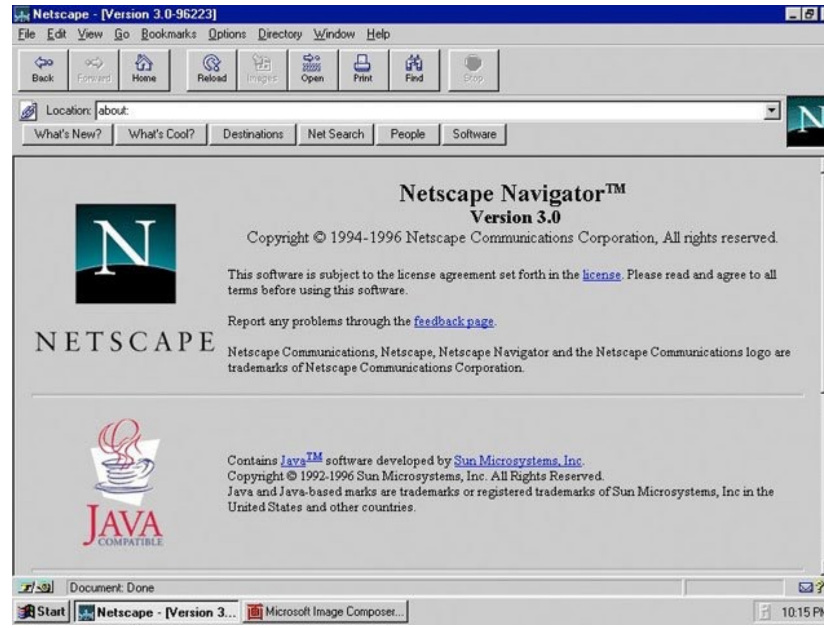
Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Web History

- Before the 1970s-1980s
  - Internet used mainly by researchers and academics
  - Log in remote machines, transfer files, exchange e-mail
- Late 1980s and early 1990s
  - Initial proposal for the Web by Berners-Lee in 1989
  - Competing systems for searching/accessing documents
    - Gopher, Archie, WAIS (Wide Area Information Servers), …
    - All eventually subsumed by the World Wide Web
- Growth of the Web in the 1990s
  - 1991: first Web browser and server
  - 1993: first version of Mosaic browser

Early Web Browsers





Web and HTTP (HyperText Transport Protocol)

First, a review…

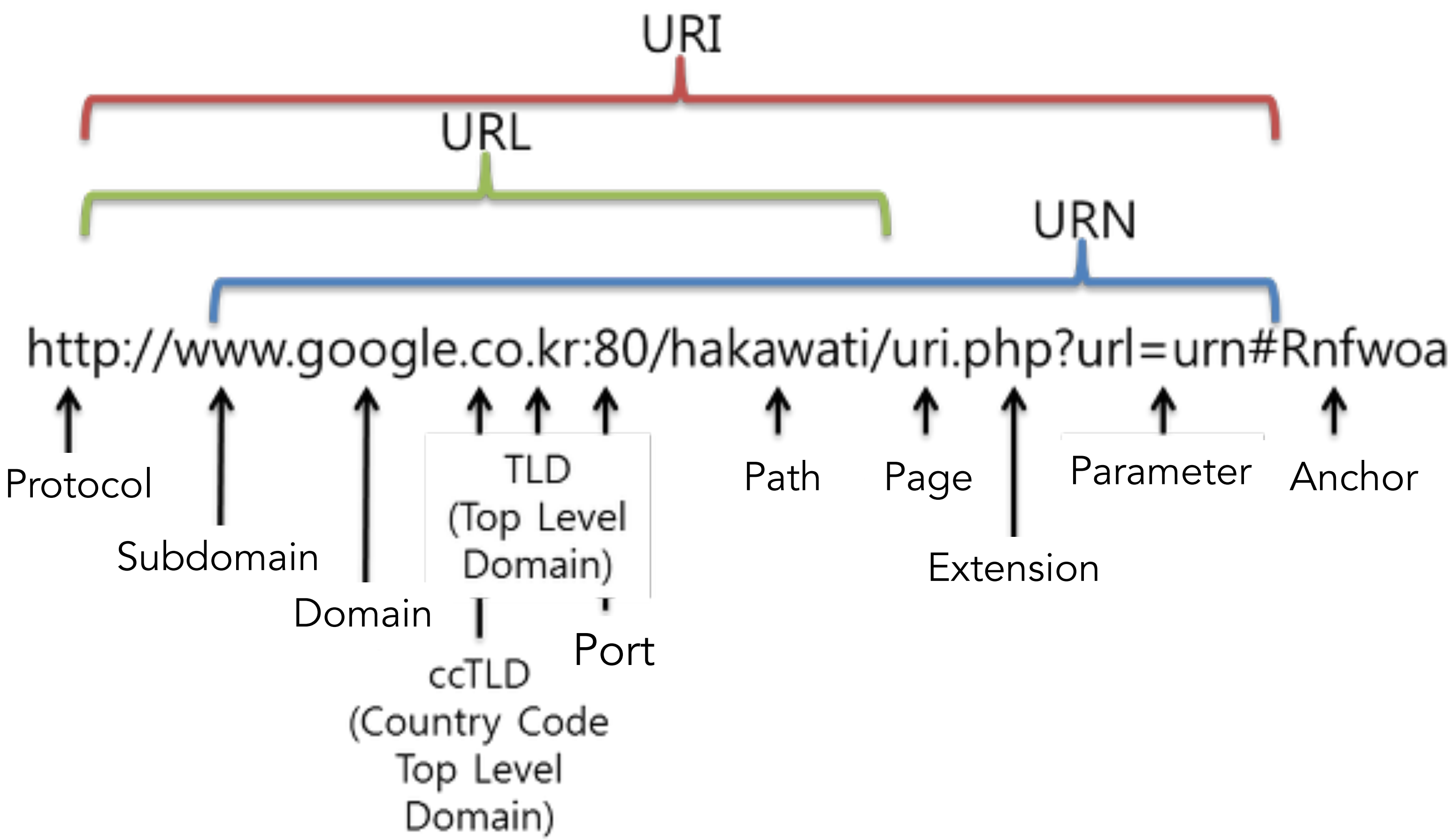
- *web page* consists of *objects*
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL,* e.g.,

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

URI, URN, URL

- Uniform Resource Identifier (URI)
  - Denotes a resource independent of its location or value
  - A pointer to a "black box" that accepts request methods
- Formatted string
  - Protocol for communicating with server (e.g., http)
  - Name of the server (e.g., www.foo.com)
  - Name of the resource (e.g., coolpic.gif)
- Name (URN), Locator (URL), and Identifier (URI)
  - URN: globally unique name, like an ISBN # for a book
  - URI: identifier representing the contents of the book
  - URL: location of the book

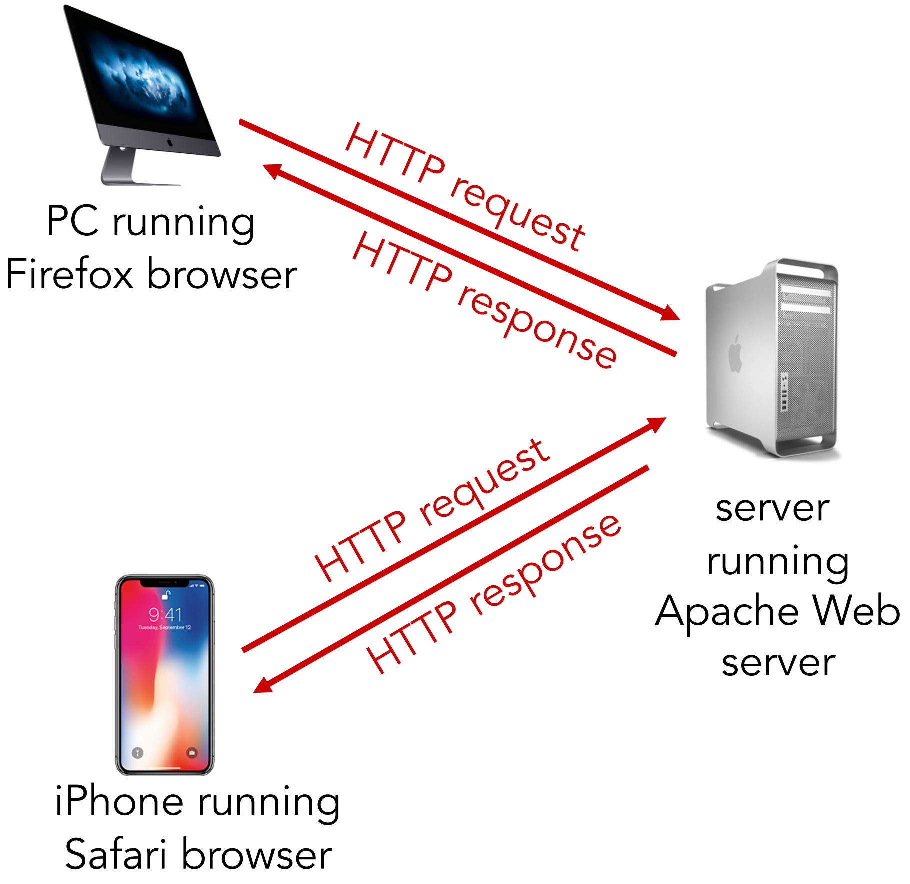
URI, URL, URN


URI
URL
URN
http://www.google.co.kr:80/hakawati/uri.php?url=urn#Rnfwoa
Protocol
Subdomain
Domain
TLD
(Top Level
Domain)
ccTLD
(Country Code
Top Level
Domain)
Port
Path
Page
Extension
Parameter
Anchor

HTTP Overview

HTTP (HyperText Transfer Protocol)

- Web is 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

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

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

non-persistent HTTP

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

persistent HTTP

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

Non-Persistent HTTP

Suppose user enters URL:

`www.someSchool.edu/someDepartment/home`

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 objects in someDepartment/home

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

time

Non-Persistent HTTP

Suppose user is connected to URL:

`www.someSchool.edu/someDepartment/home`

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

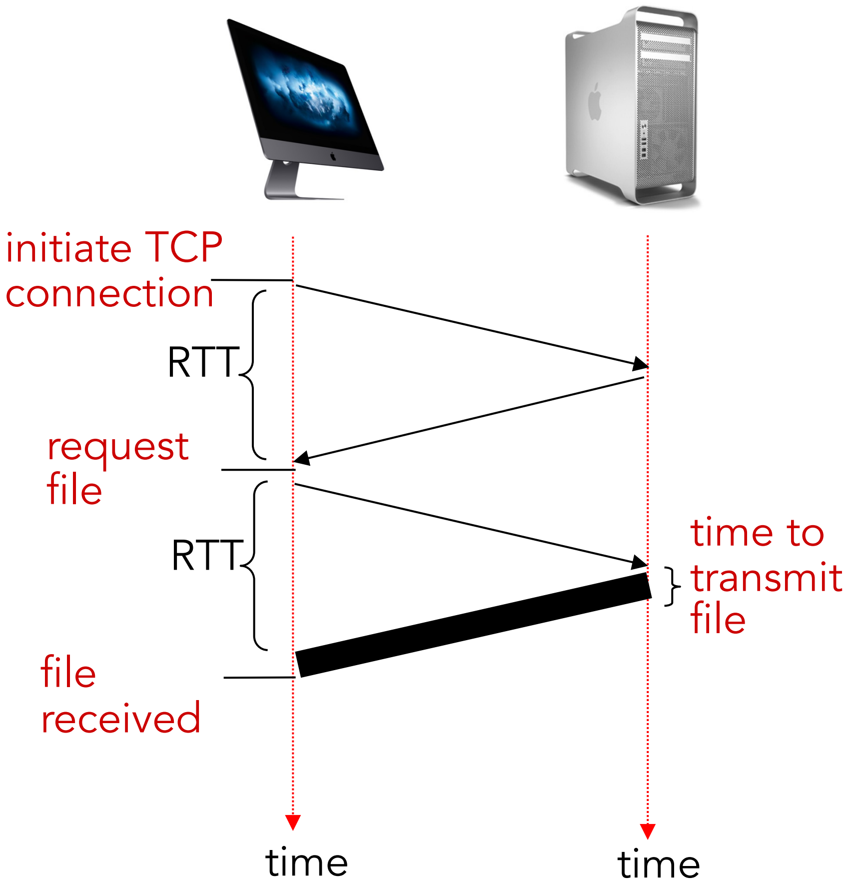
time

Non-Persistent HTTP: Response Time

RTT (definition): 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 = 2·RTT+ file transmission time



Persistent HTTP

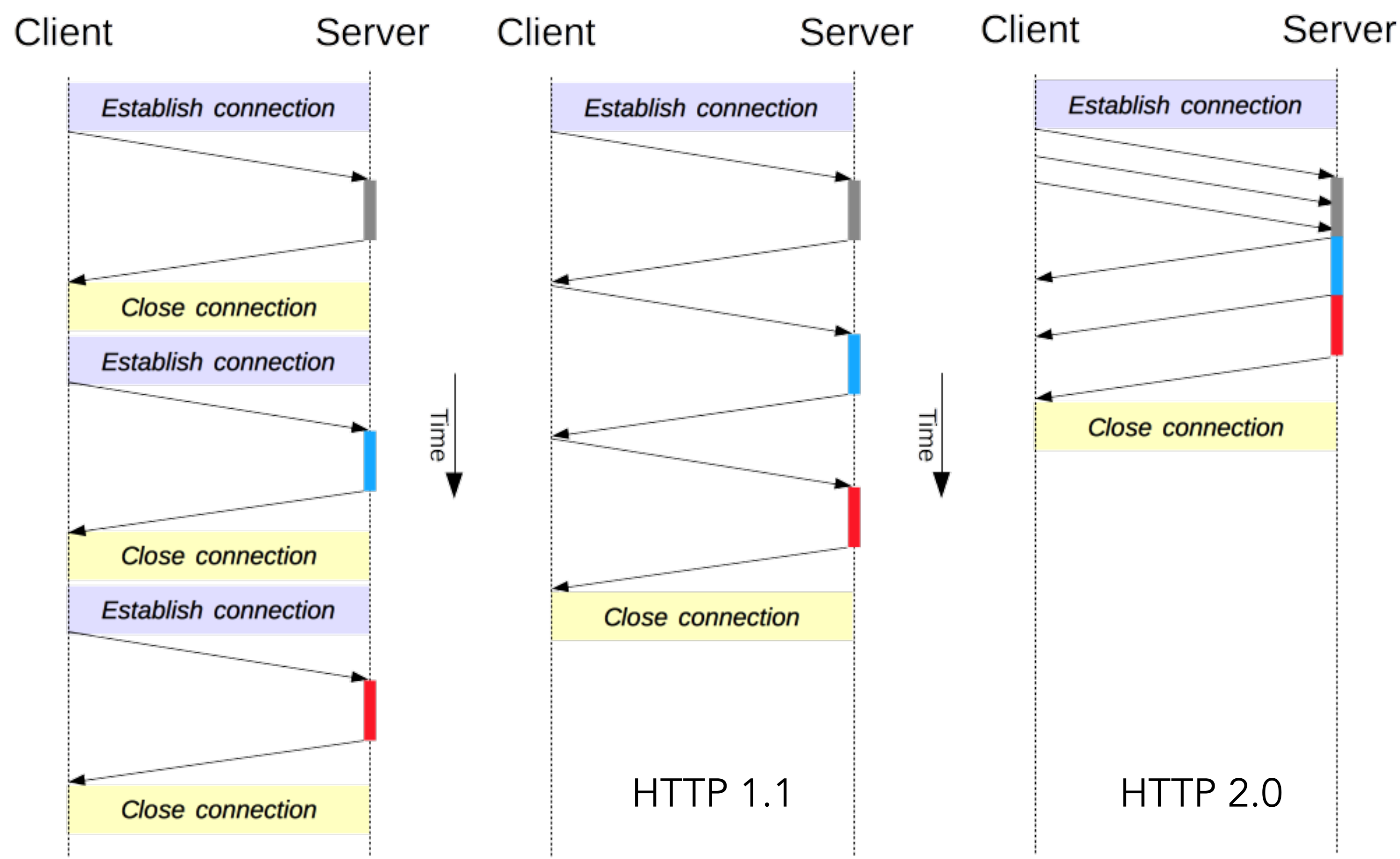
non-persistent HTTP issues:

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

persistent HTTP:

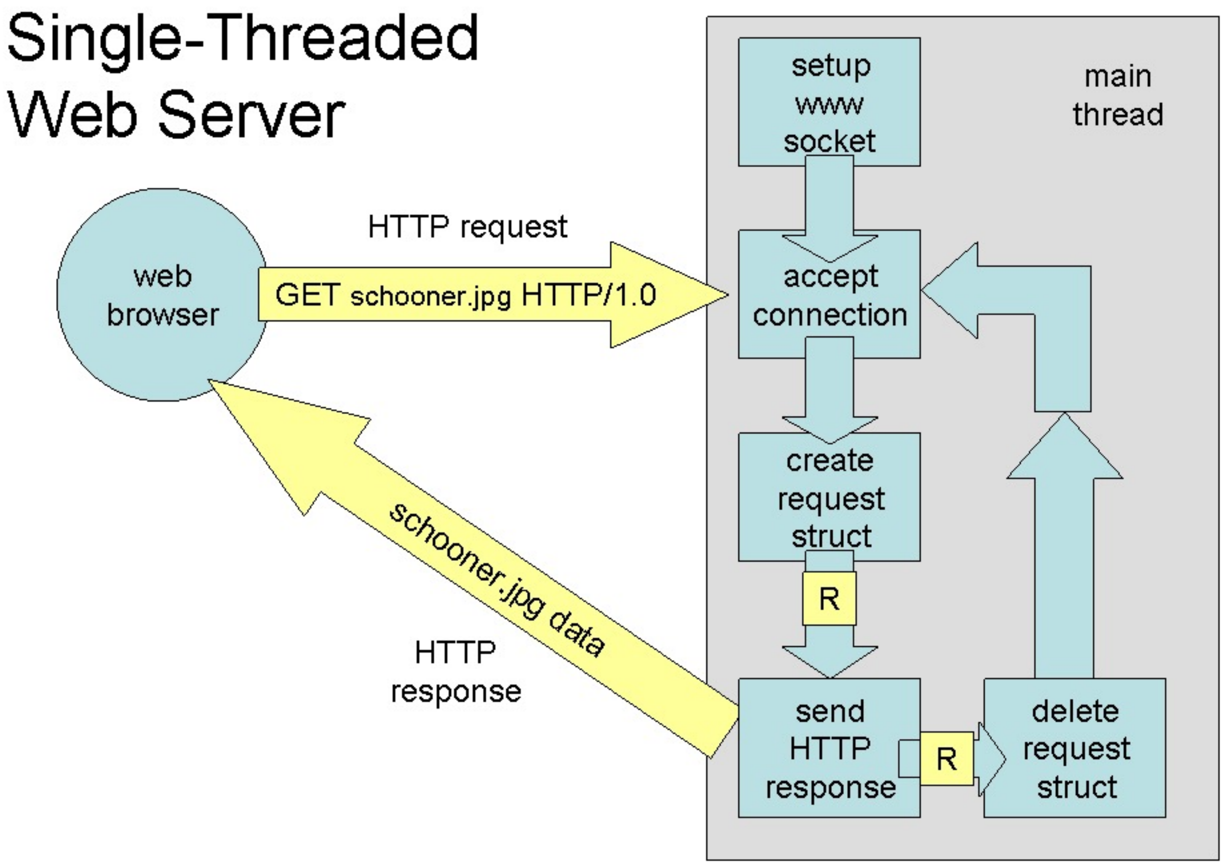
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP 1.1 and 2.0



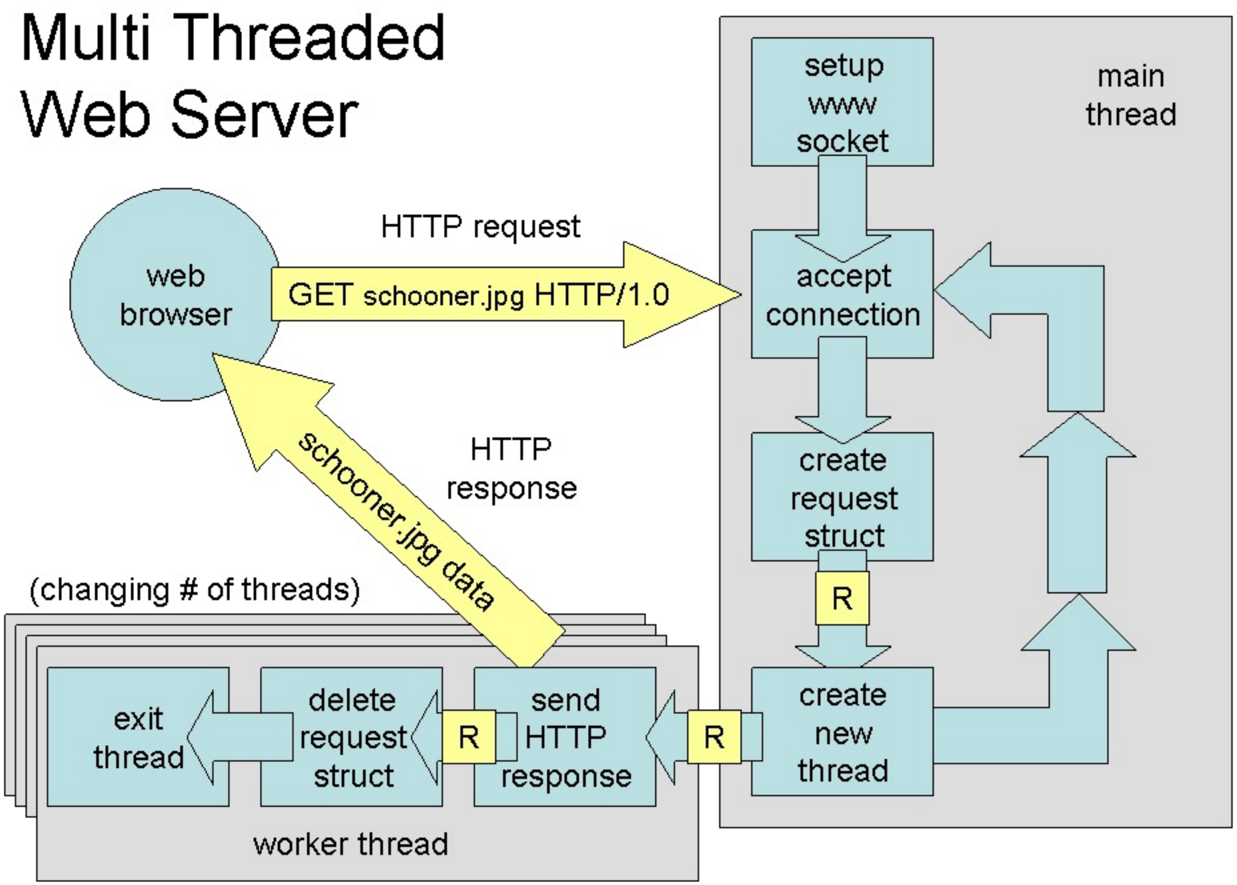
Non-Persistent connection　Persistent connection　HTTP Multiplexing

HTTP Server Architecture

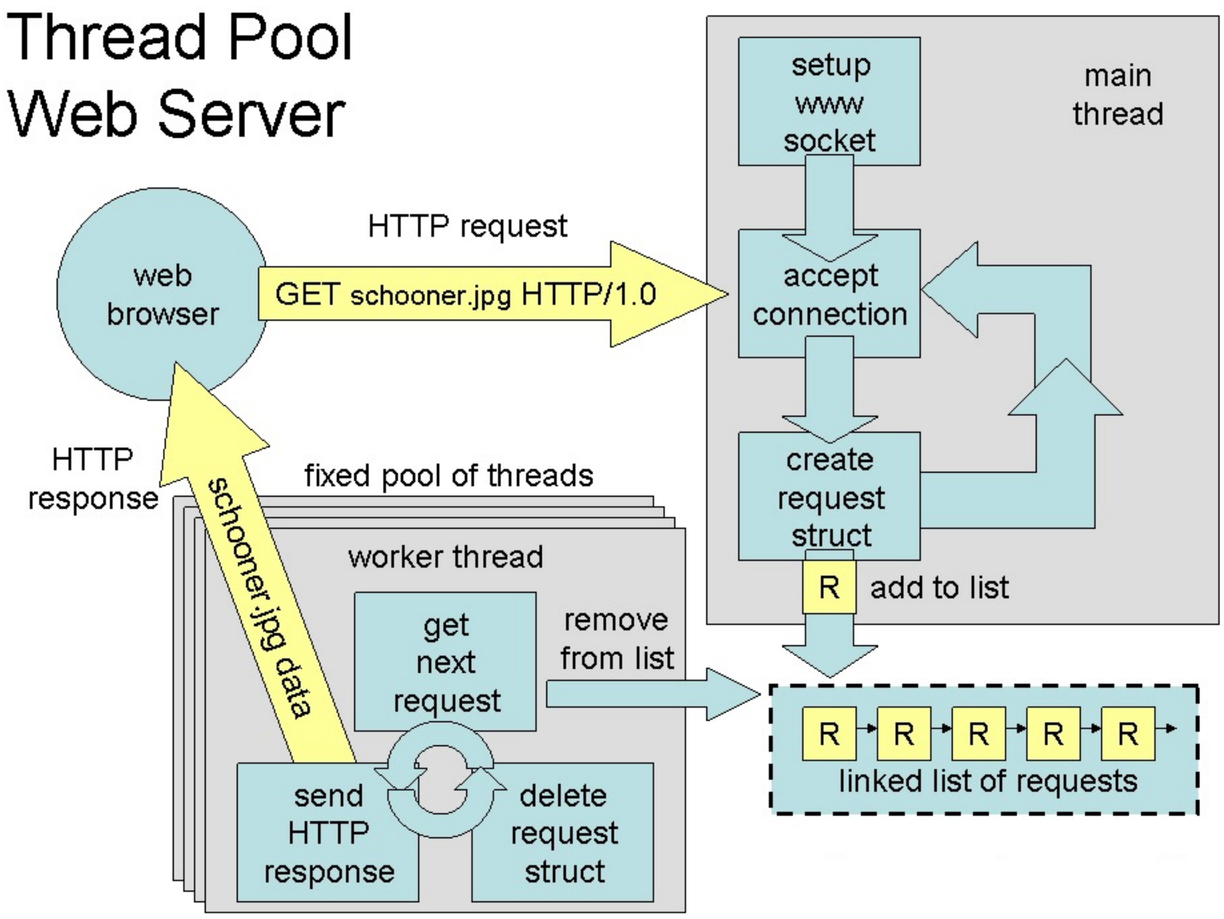


Thanks to Prof. Douglas Thain at U of Notre Dame

HTTP Server Architecture



HTTP Server Architecture



HTTP Request Message

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

carriage return character

line-feed character

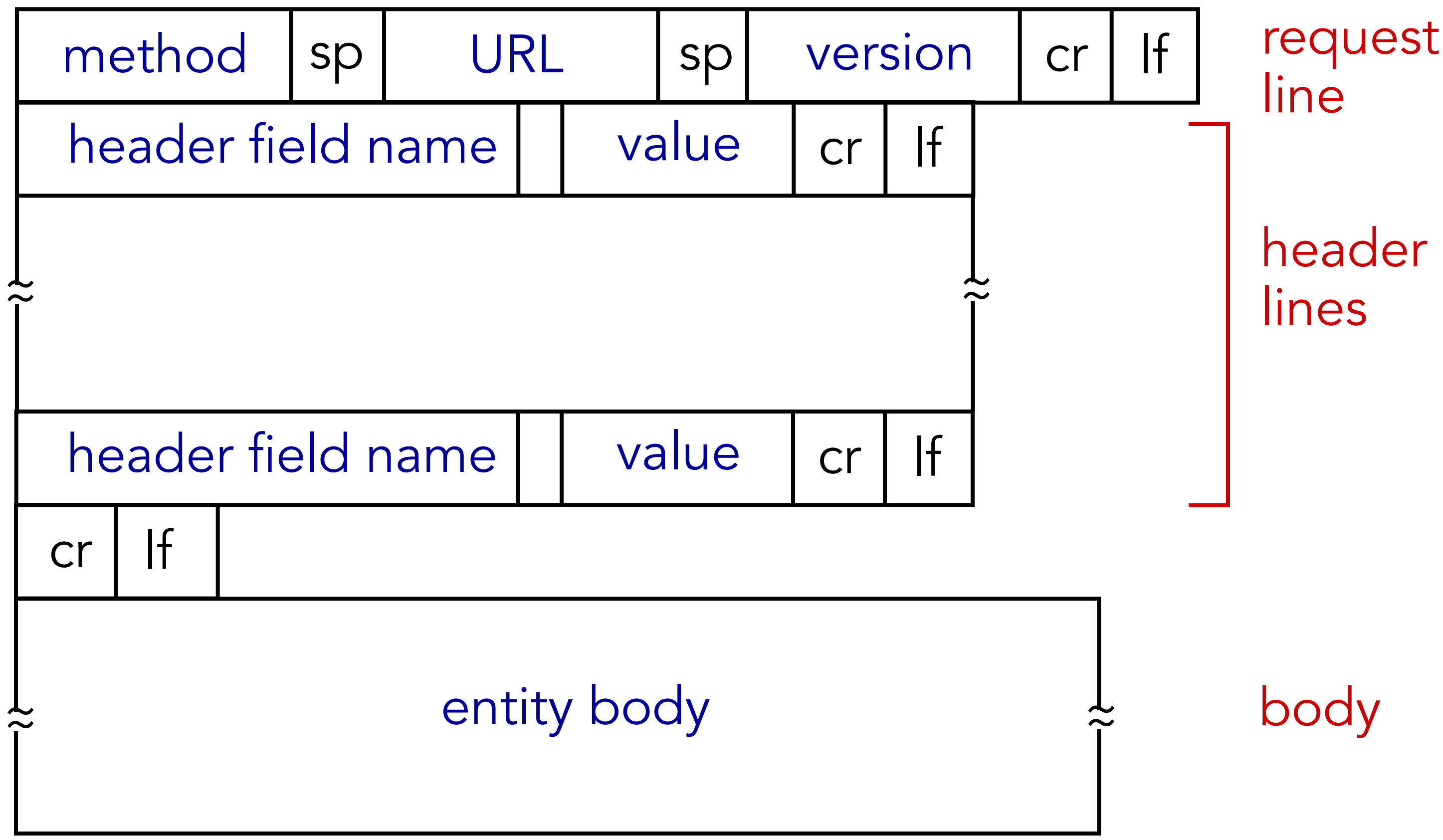
request line (GET, POST, HEAD commands)

header lines

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

```
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
```

HTTP Request Message: General Format



Uploading Form Input

POST method:

- web page often includes form input
- input is uploaded to server in entity body

URL method:

- uses GET method
- input is uploaded in URL field of request line:

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

Method Types

HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP Response Message

status line
(protocol
status code
status phrase)

header
lines

data, e.g.,
requested
HTML file

```
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 ...
```

HTTP Response Status Code

- status code appears in the 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-side States: Cookies

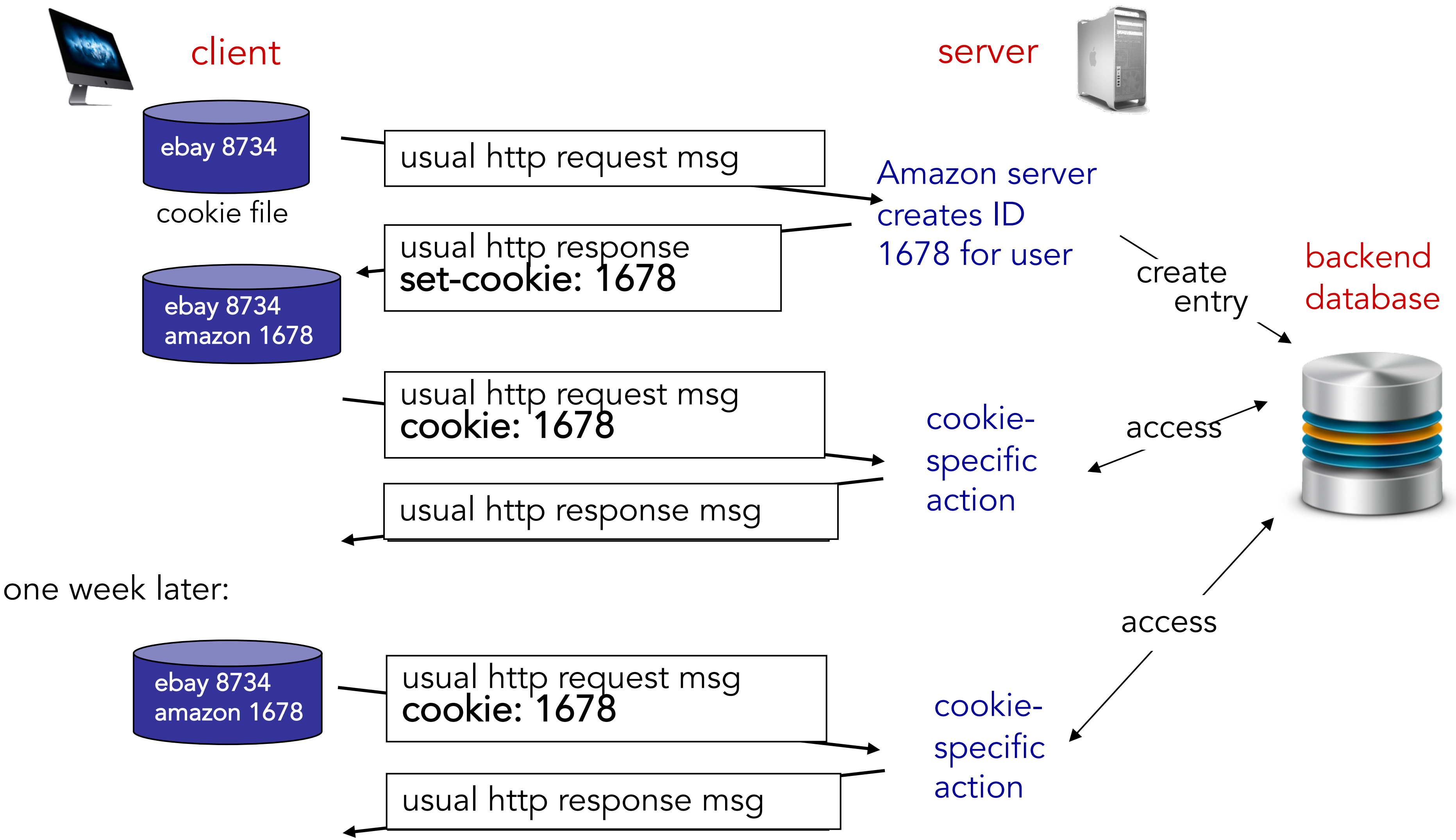
Many Web sites use cookies by *four components:*

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

Example:

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

Cookies: keeping "States"



Cookies

What cookies can be used for:
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

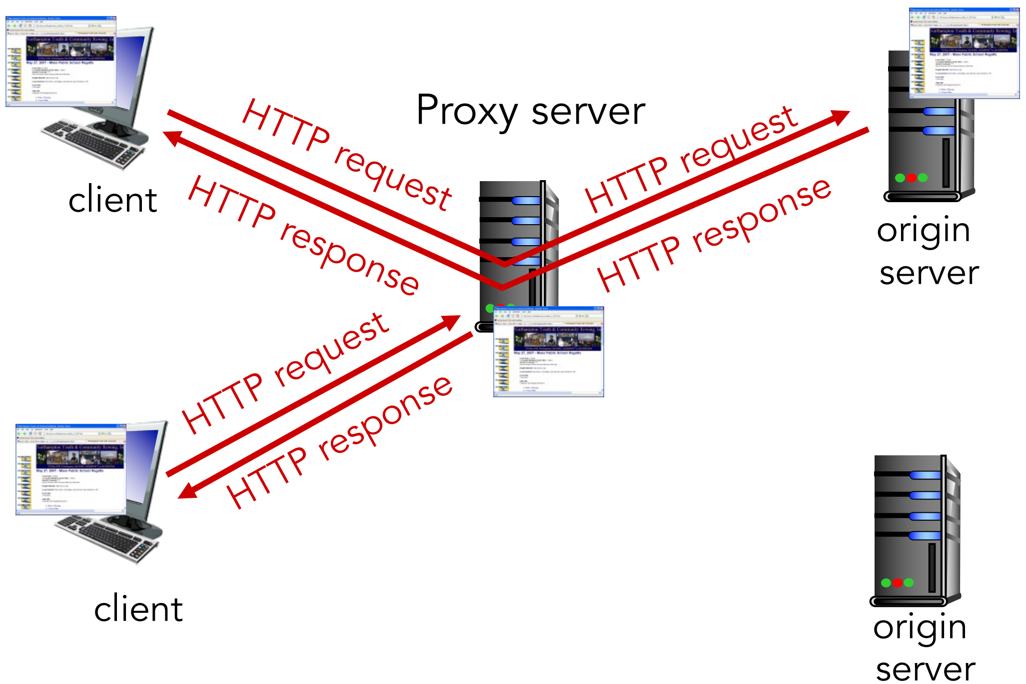
Cookies and privacy:
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

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
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



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 (so too does P2P file sharing)

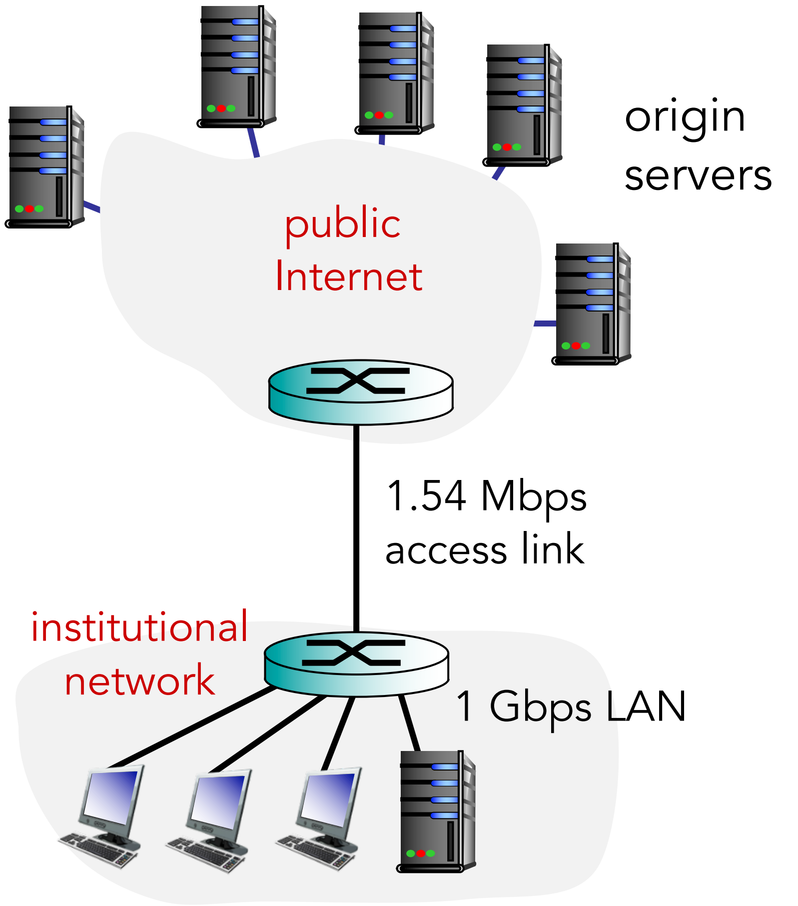
Caching Example

assumptions:

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

consequences:

- LAN utilization: 0.15%
- access link utilization = 99% *problem!*
- total delay = Internet delay + access delay + LAN delay = 2 sec + minutes + usecs



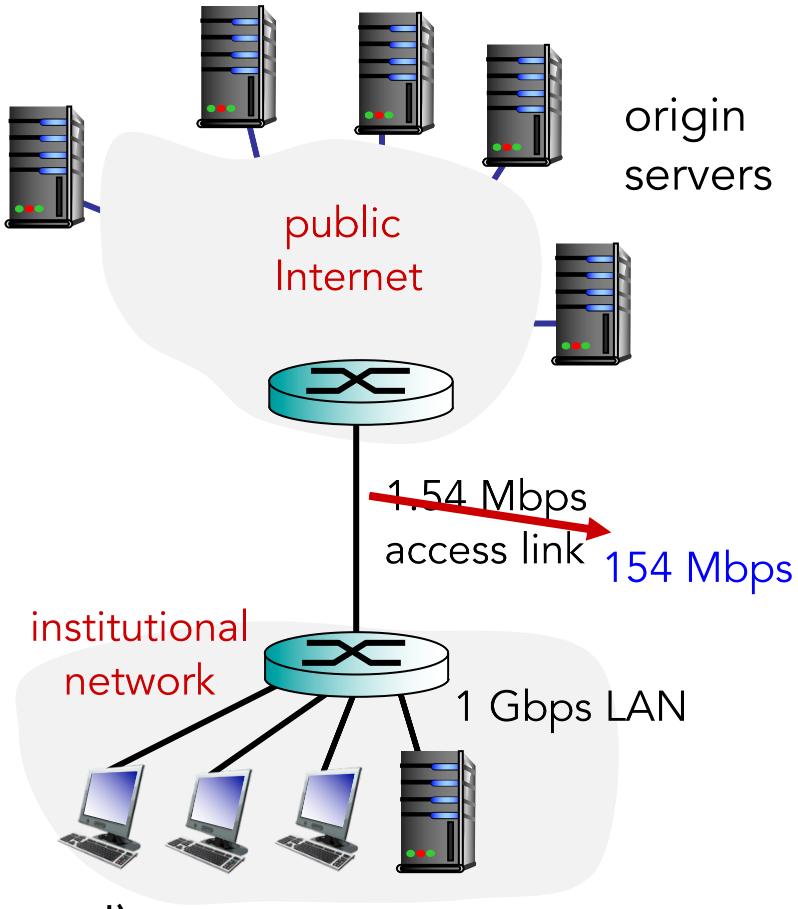
Caching Example: fatter access link

assumptions:

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

consequences:

- LAN utilization: 0.15%
- access link utilization = ~~99%~~ 0.99%
- total delay = Internet delay + access delay + LAN delay = 2 sec + ~~minutes~~ msecs + usecs



Cost: increased access link speed (not cheap!)

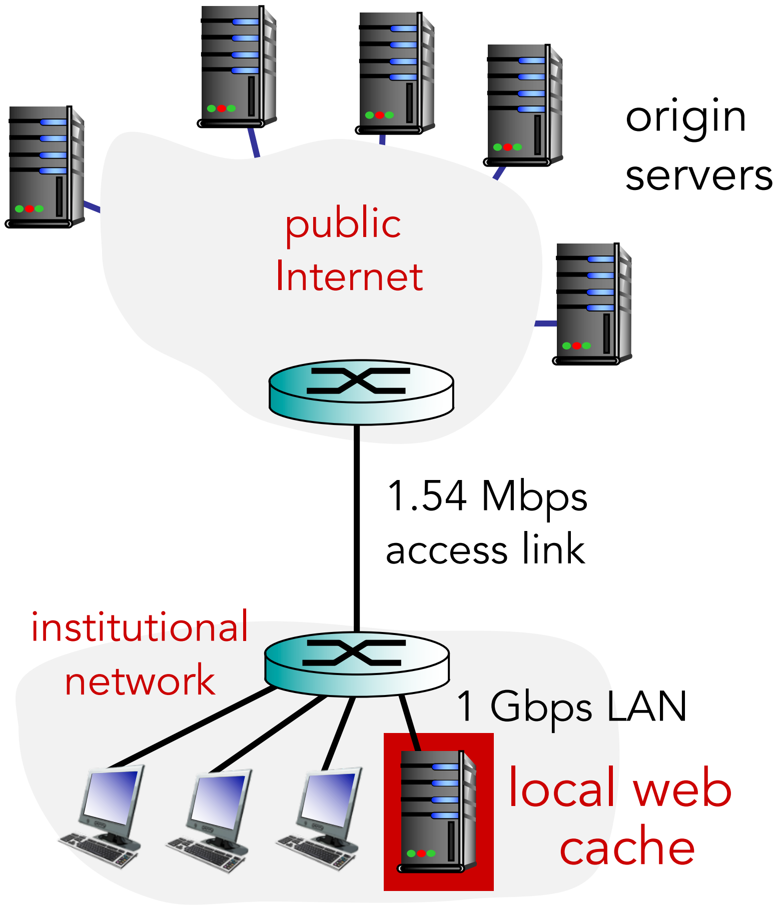
Caching Example: install local cache

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

consequences:
- LAN utilization: 0.15%
- access link utilization = ??
- total delay = ??

Cost: web cache (relatively cheap!)

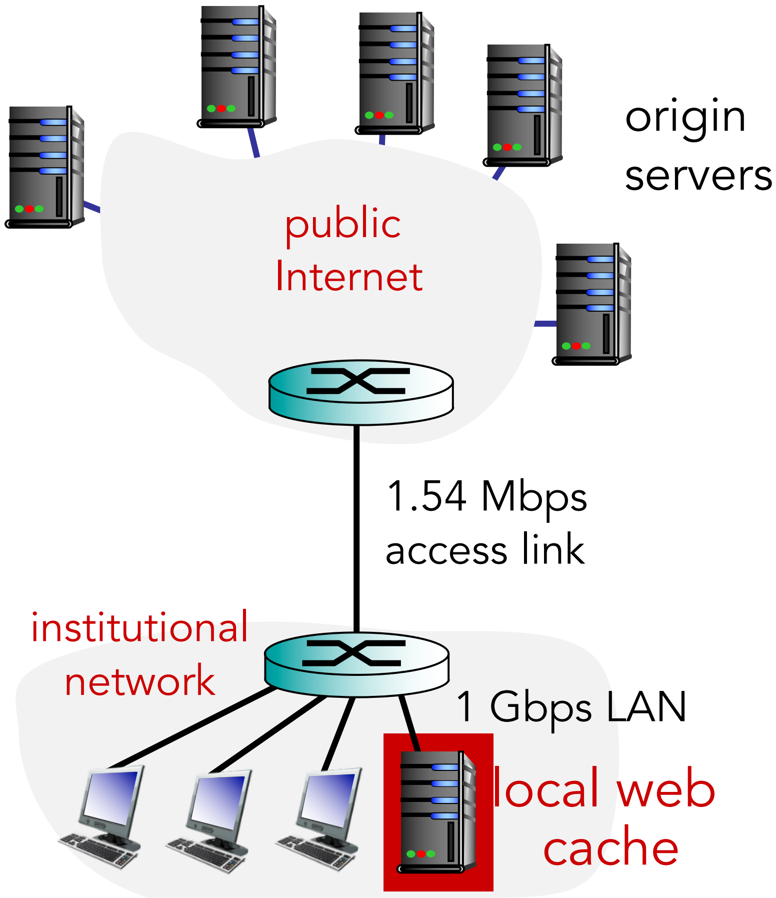
How to compute link utilization, delay?



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



- access link utilization:
  - 60% of requests use access link
- data rate to browsers over access link = 0.6*1.50 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58 (58%)

- total delay
  - = 0.6 * (delay from origin servers) + 0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)

Conditional GET (with Web Cache)

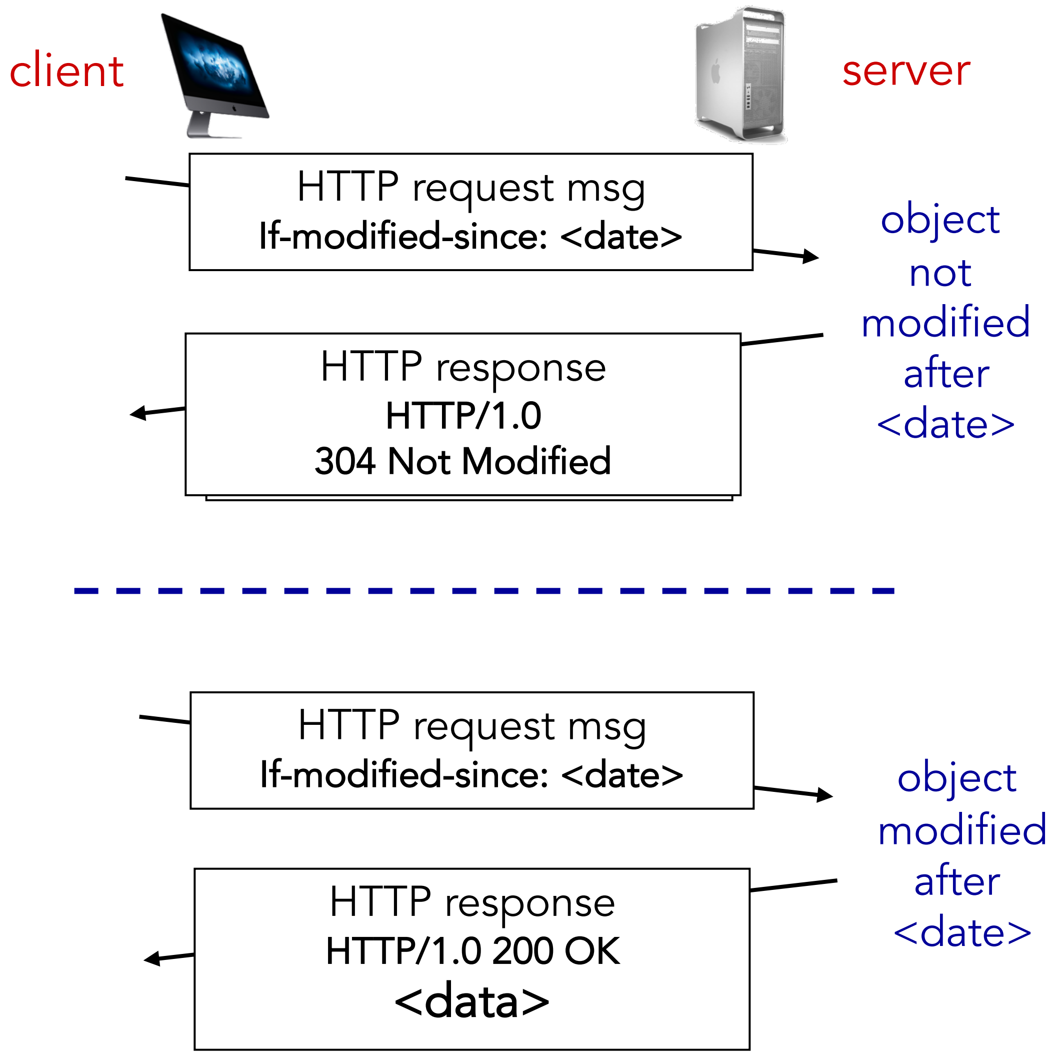
- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization

- *cache (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



Application Layer

- P2P -

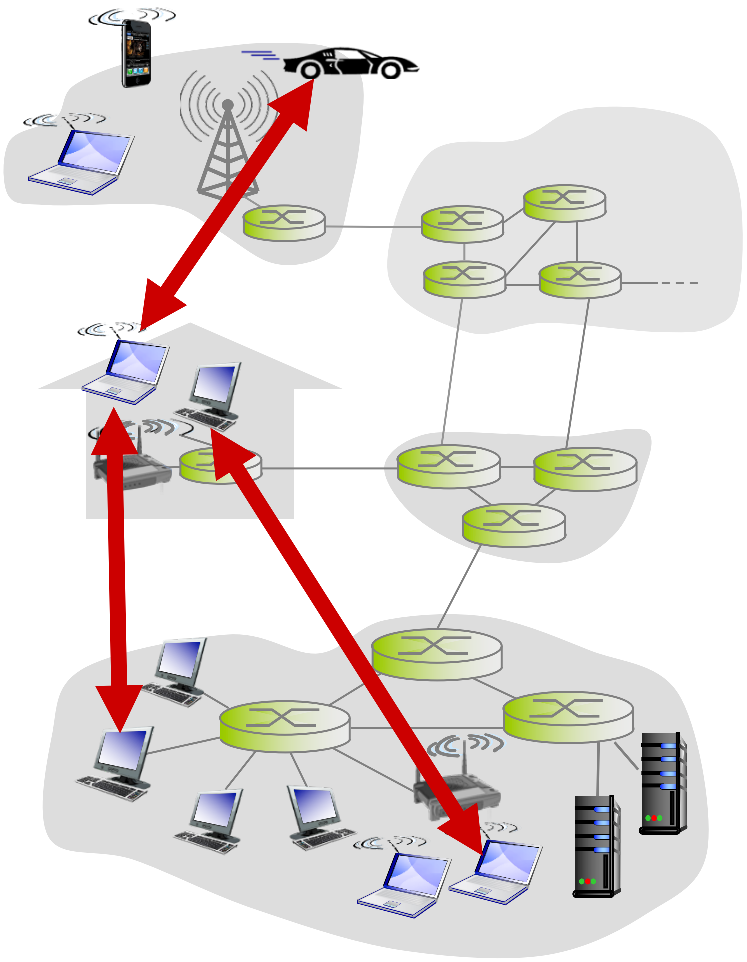
Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Pure P2P Architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

examples:

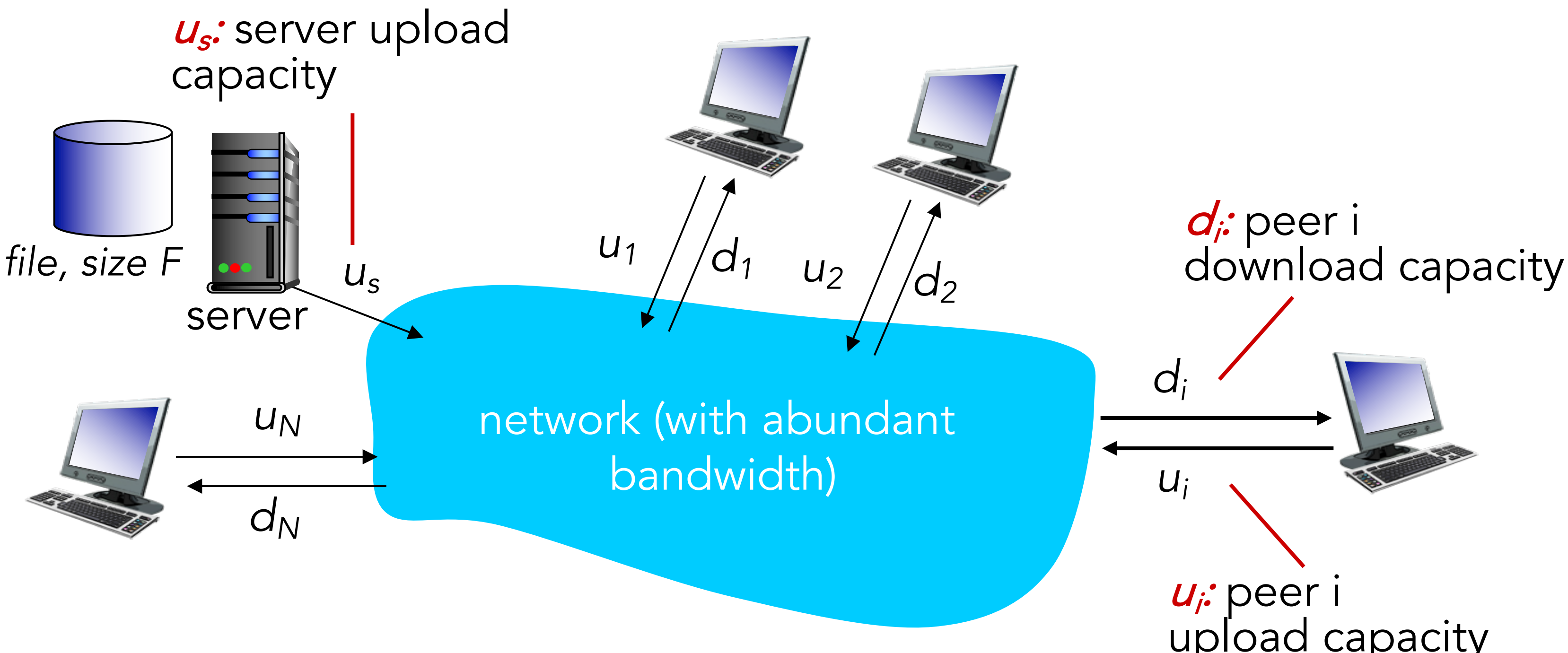
  - file distribution (BitTorrent)
  - Streaming (Streamroot)
  - VoIP (Skype)



File distribution: client-server vs P2P

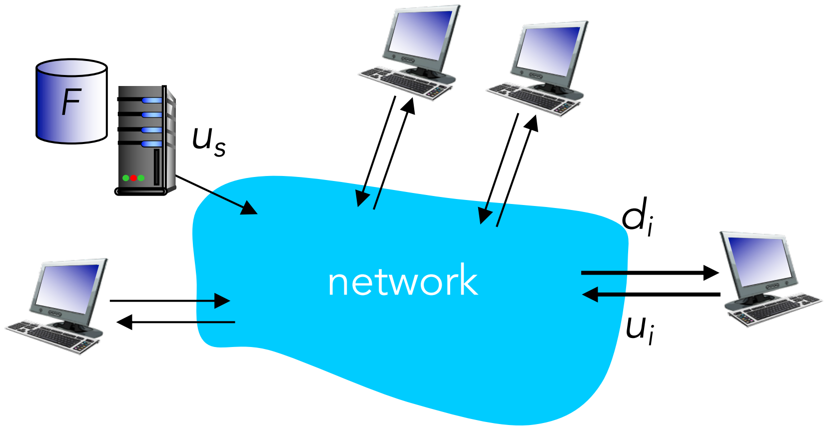
Question: how much time to distribute file (size *F*) from one server to *N* *peers*?

- 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}

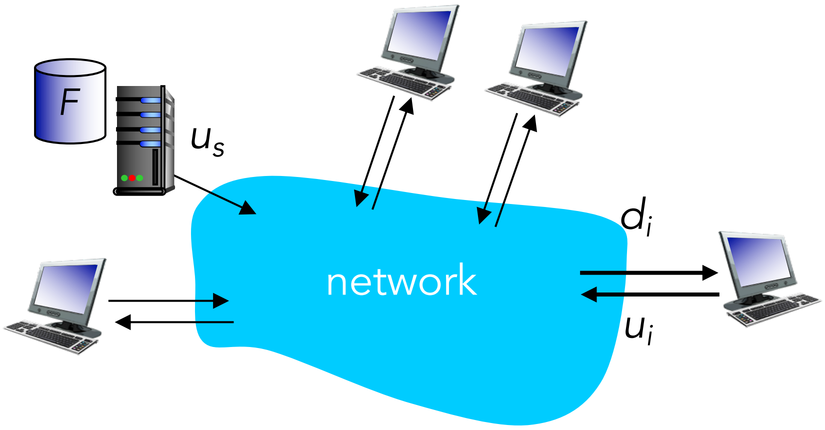
Time to distribute F to N clients using client-server approach

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

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

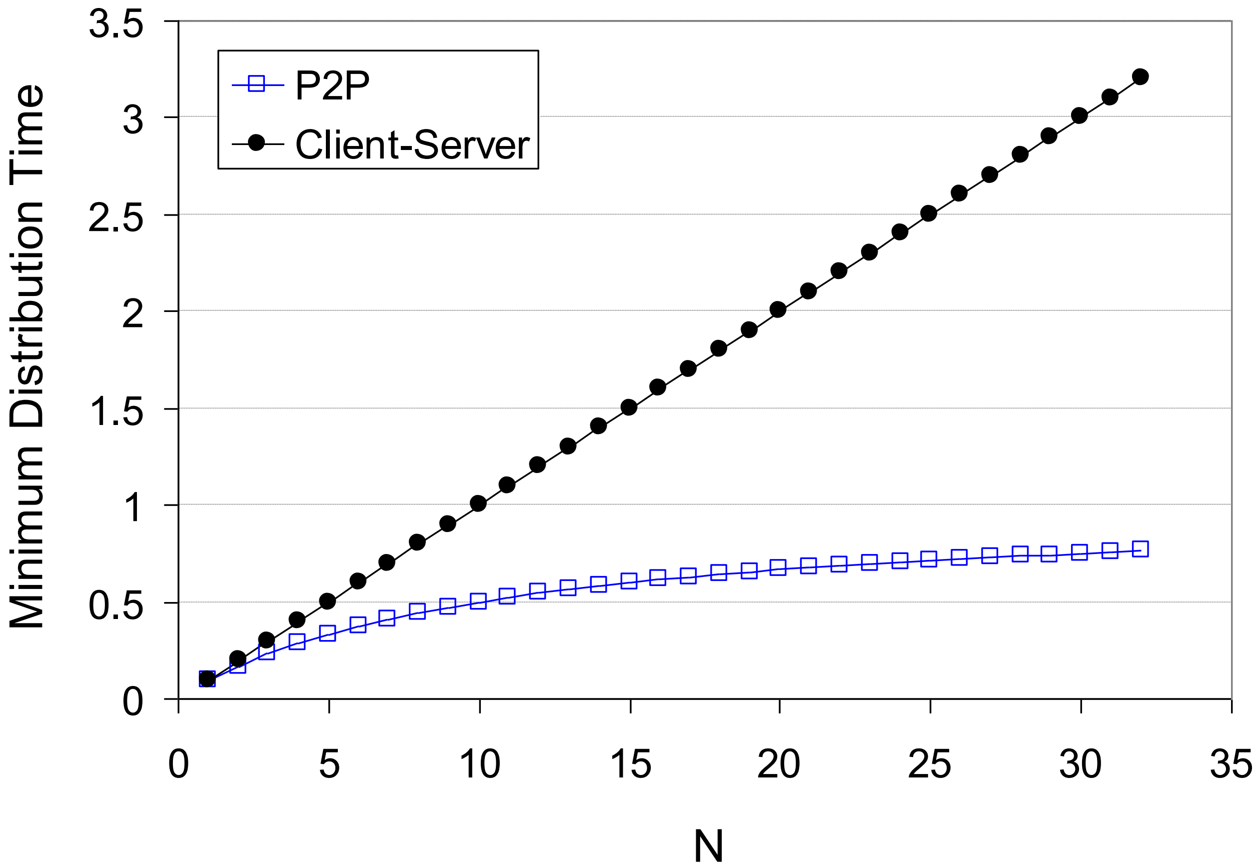
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

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, $u_s = 10u$, $d_{min} \geq u_s$

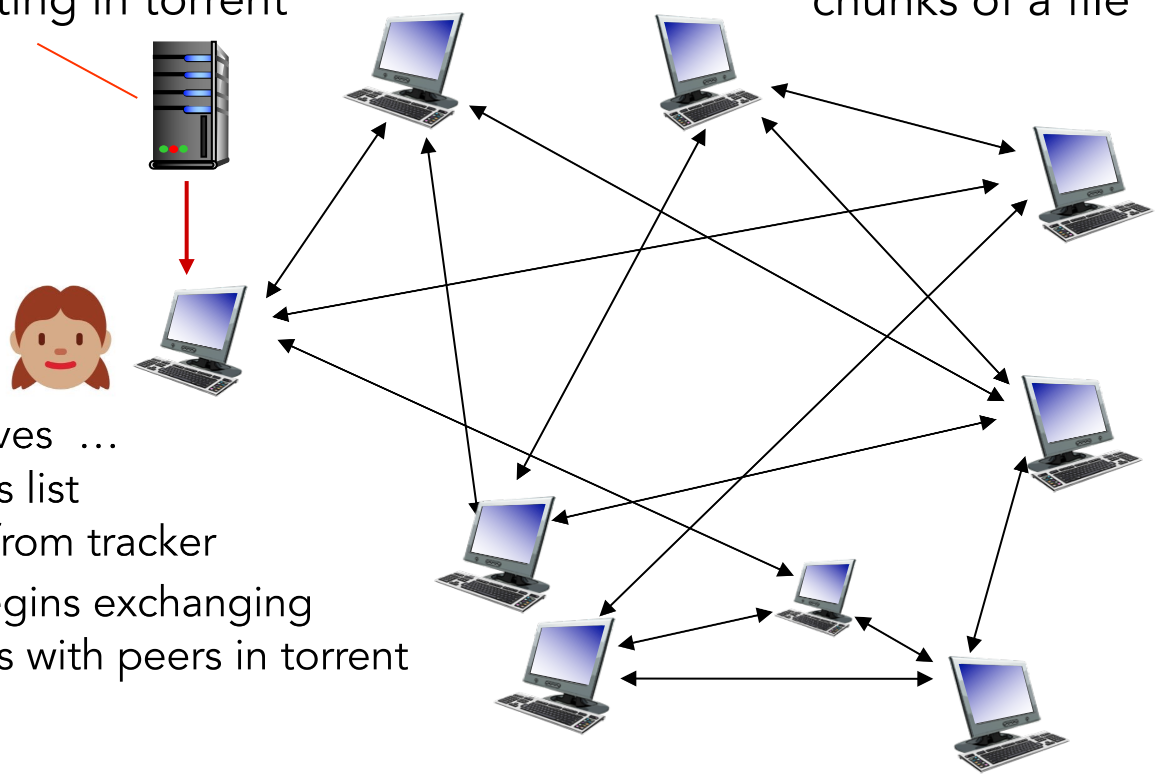


P2P file distribution: BitTorrent

- File divided into 256kb chunks
- Peers in torrent send/receive file chunks

tracker: tracks peers participating in torrent

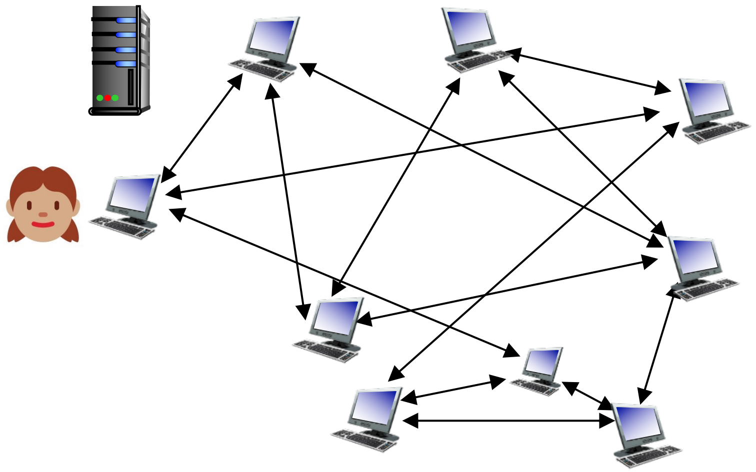
torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

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
- *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:

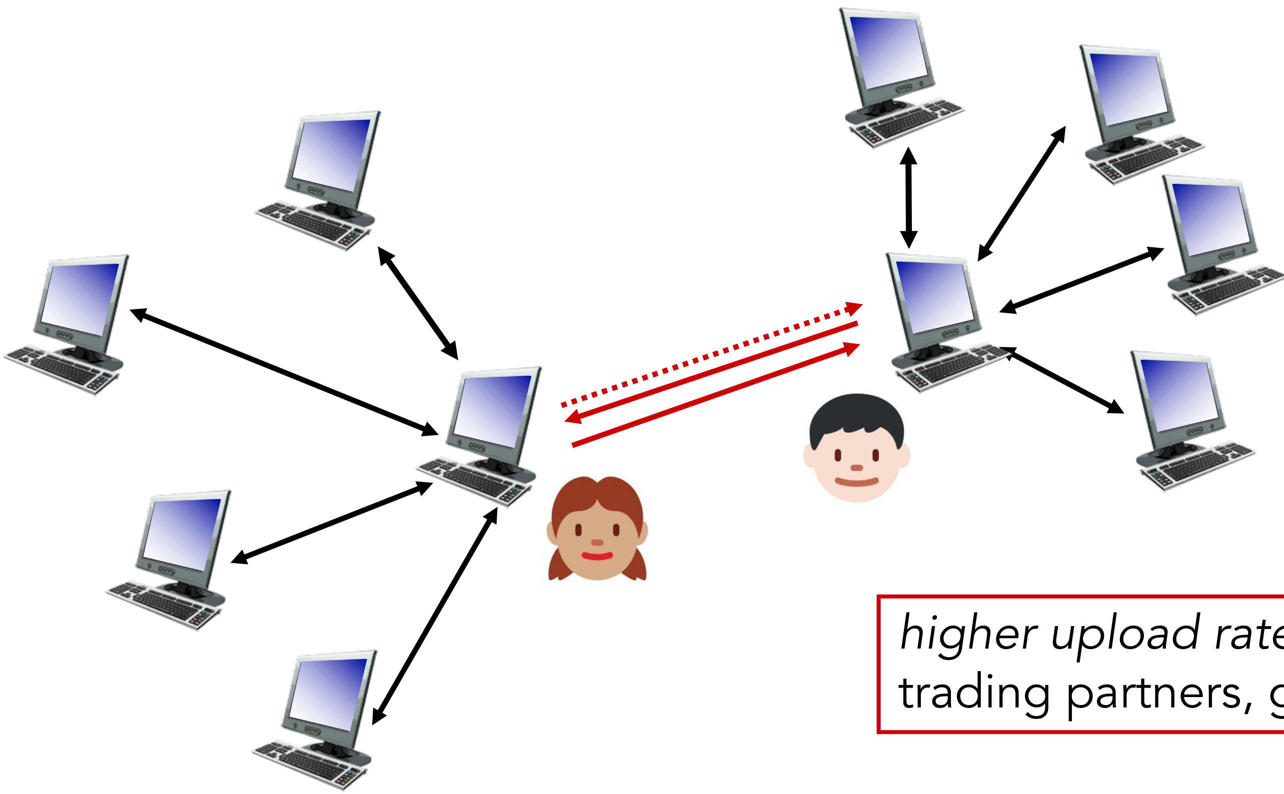
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

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 (cannot 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



higher upload rate: find better trading partners, get file faster !