eecs489 COMPUTER NETWORKS

Lecture 9: HTTP

Content Delivery Infrastructure

Peer-to-peer (p2p):
- hybrid p2p with a centralized server
- pure p2p
- hierarchical p2p
- end-host (p2p) multicast

Content-Distribution Network (CDN)
- HTTP Overview
- HTTP Performance
- HTTP Caching
- Content Distribution Network

A Web Page

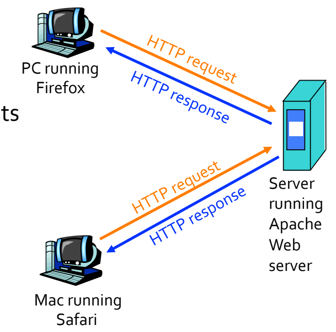
A web page consists of a base HTML-file which may include references to one or more objects

- an object can be another HTML file, a JPEG image, a Java applet, an audio file, a flash video, etc.
- each object is addressable by a URL
- example URL:

$$\underbrace{\texttt{http}}_{\text{protocol}}\texttt{://}\underbrace{\texttt{www.mgoblue.com}}_{\text{host name}}\texttt{/}\underbrace{\texttt{images/pic.gif}}_{\text{path name}}$$

HTTP Overview

HTTP: HyperText Transfer Protocol
- Web's application-layer protocol
- client/server model
  - client: browser that requests, receives, and "displays" Web objects
  - server: sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068
- HTTP/2: RFC 7540 (May 2015)



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!

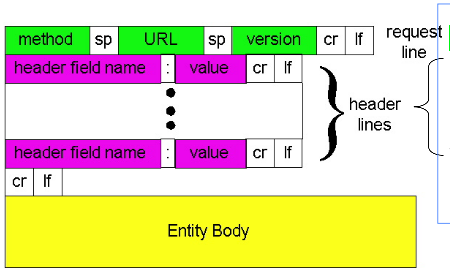
- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, and must be reconciled

HTTP 1.x Request Message

Two types of HTTP messages: request, response

HTTP request message:

- in ASCII (human-readable format)
- general format:



example

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
```

(extra carriage return, line feed)

Carriage return, line feed indicates end of message

Method Types (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

Uploading form, input alternatives:

1. POST method:
   - web pages often include form input
   - input is uploaded to server in entity body
2. as parameter to GET URL method:
   - input is uploaded in URL field of request line:
     www.somesite.com/animalsearch?monkeys&banana

input parameters

HTTP 1.x Response Message Example

first line: status line (protocol status code, status phrase)

header lines

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

blank line

data, e.g., requested HTML file

HTTP $1.x$ Response: Status Line

HTTP-version 3-digit-response-code Reason-phrase

- `1XX` – informational
- `2XX` – success
  - `200 OK`: request succeeded, requested object later in this message
- `3XX` – redirection
  - `301 Moved Permanently`: requested object moved, new location specified later in this message (“Location:” in header)
  - `303 Moved Temporarily`
  - `304 Not Modified`
- `4XX` – client error
  - `400 Bad Request`: request message not understood by server
  - `404 Not Found`: requested document not found on this server
- `5XX` – server error
  - `505 HTTP Version Not Supported`

Client-side States: Cookies

HTTP is “stateless”

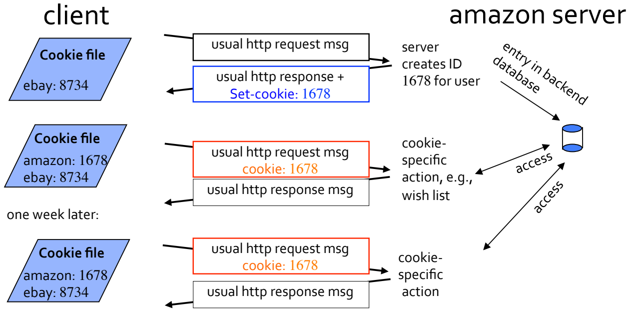
- server maintains no information about past client requests
- but sometimes it may be useful to keep per-client states, for example for:
  - authorization
  - shopping carts
  - wish list
  - recommendations
  - user session state (Web e-mail)

States or user ID (to look up server-side states) kept at client side using cookies

Client-side States: Cookies

Four components:

1. cookie header line in the HTTP response message
2. cookie header line in HTTP request message
3. cookie file kept on client host and managed by client browser
4. back-end database at Web server



“Abuse” of Cookies

Excellent marketing opportunities and concerns for privacy:

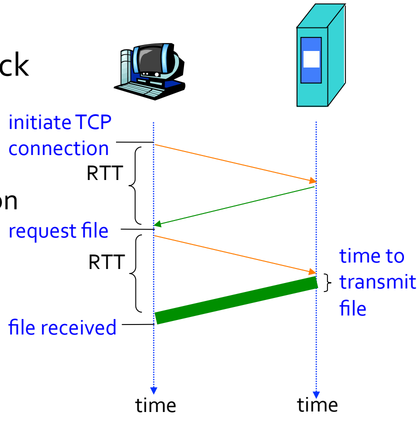
- cookies permit sites to learn a lot about you
- you may unknowingly supply personal info to sites
- advertising companies tracks your preferences and viewing history across sites, example scenario:
  - ad company contracted with (1) a social networking site, (2) a book store, and (3) a clothing store
  - you view your friend’s travel photos to Hawaii at the social networking site
  - when you visit the bookstore, a travel book about Hawaii is pushed to you
  - when you visit the clothing store, a swimming goggle is pushed to you
  - at all three places a travel agency’s extra-low price, expiring in 30 seconds, Hawaii vacation package is pushed to you

Object Request Response Time

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

Response time:

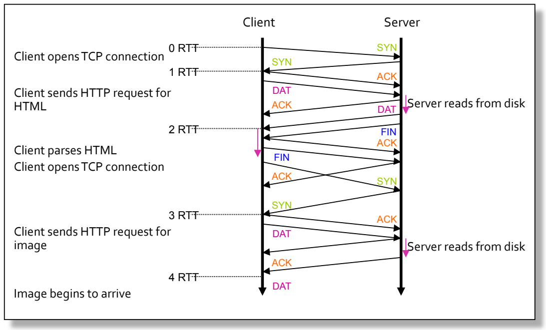
- 1 RTT to initiate TCP connection
- 1 RTT for HTTP request and the first few bytes of HTTP response to return
- file transmission time
- total = 2RTT+transmit time



HTTP 1.0

HTTP 1.0 uses non-persistent connections:

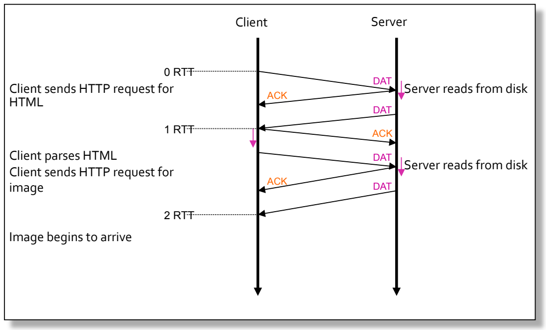
- at most one object is sent over a TCP connection
- object transmission completion detected by `recv()` returning 0 (connection closed)
- why is this not a good design?



HTTP 1.1

HTTP 1.1 uses persistent connections:

- server leaves connection open after sending responses
- subsequent HTTP messages between the same client/server to fetch multiple objects are sent over the same connection



How to Mark End of Message?

Three options:

Content-Length in header:

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …....
Content-Length: 6821
Content-Type: text/html
```

How to Mark End of Message?

Implied length, e.g., `304` (cache fresh) never has content

Transfer-Encoding: chunked (HTTP 1.1)

- after headers, each chunk comprises content length in hex, CRLF, then body; length 0 indicates end-of-chunk

```
HTTP/1.1 200 OK<CRLF>
Transfer-Encoding: chunked<CRLF>
<CRLF>
25<CRLF>
This is the data in the first chunk<CRLF>
1A<CRLF>
and this is the second one<CRLF>
0<CRLF>
```

Pipelined and Parallel Connections

Persistent without pipelining:

- client issues new request only when previous response has been received
- one RTT for each referenced object

Persistent with pipelining:

- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all referenced objects
- default in HTTP 1.1

Browsers can open parallel TCP connections to fetch referenced objects (even in HTTP 1.0)

HTTP Modeling

Assume Web page consists of:

- 1 base HTML page (of size L bits)
- M images (each also of size L bits)

Non-persistent HTTP:

- $M+1$ TCP connections in series
- response time = $(M+1)*2*\text{RTT} + (M+1)*L/\mu$, μ: path speed

Persistent HTTP (with pipelining):

- 2 RTTs to request and receive base HTML file
- 1 RTT to request and receive M images
- response time = $3*\text{RTT} + (M+1)*L/\mu$

HTTP Modeling

Assume Web page consists of:

- 1 base HTML page (of size L bits)
- M images (each also of size L bits)

Non-persistent HTTP with n parallel connections

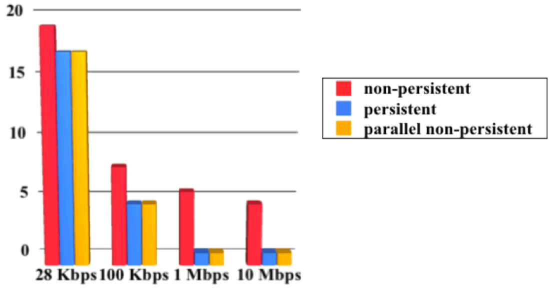
- suppose M/n evenly
- 1 TCP connection for base file
- M/n parallel connections for images
- n-parallel response time = $(M/n + 1)*2*\text{RTT} + (M/n+1)*L/\mu$

compare:

- non-persistent response time = $(M+1)*2*\text{RTT} + (M+1)*L/\mu$
- persistent response time = $3*\text{RTT} + (M+1)*L/\mu$

HTTP Response time (in seconds)

RTT = 100 msec, $L = 5$ Kbytes, $M = 10$, and $n = 5$

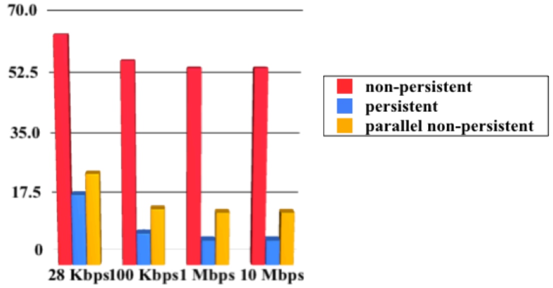


For low bandwidth, transmission time dominates over connection and response time

⇒ performance of persistent connections comparable to that of parallel connections

HTTP Response time (in seconds)

RTT = 1 sec, $L = 5$ Kbytes, $M = 10$, and $n = 5$



For larger RTT, TCP establishment and slow start delays dominate over response time

⇒ persistent connections now give significant improvement: particularly in high bandwidth × delay networks

HTTP/2

Based on Google's SPDY (2009)

RFC 7540 came out in May 2015 (written by the two authors of SPDY)

Chrome browser already has SPDY built-in

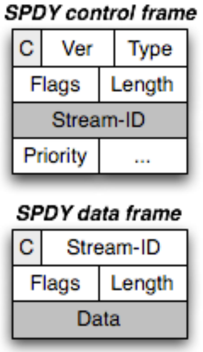
Problems with HTTP 1.x:

- pipelining still suffers from head-of-line blocking (if first item is large, the rest has to wait)
- parallel streams solves HoL blocking, but on bandwidth-limited channel, too many streams clog up the channel

HTTP/2

Some changes from 1.1:

- headers no longer in text format
- separate control and data headers
- stream multiplexing over a single TCP connection:
  - each stream has an ID, data is tagged with stream ID
  - each stream can also have different priority
- server push: don't have to wait for client to parse page before initiating download
- header compression



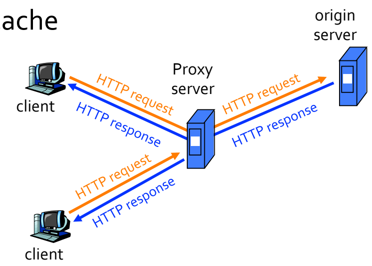
[Grigorik]

Performance improvement: up to 64% reduction in page load time

Web Caches (Proxy Server)

Goal: satisfy client request without involving origin server

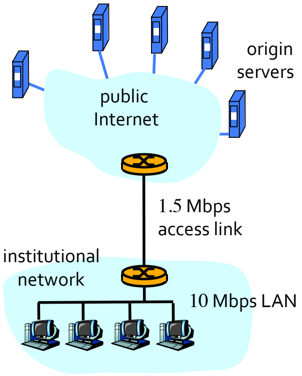
- user sets browser to direct all web accesses via cache
- browser sends all HTTP requests to cache
  - if object is not cached, cache requests object from origin server, then returns object to client
  - else cache returns object
- cache acts as both client and server
  - typically cache is installed by ISP (university, company, residential ISP)
  - must be transparent, allow for plug-n-play



Web Caching Example: No Caching

Parameters:

- average object size = 100,000 bits
- avg. # of requests to servers = 15/sec
- Internet latency between a router on the public Internet and any server = 2 secs



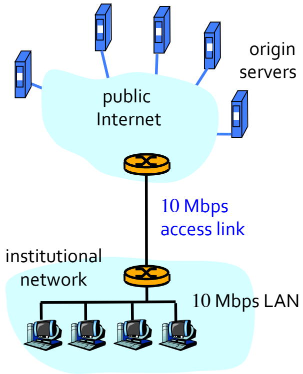
Resulting performance:

- utilization on LAN = 15%
- utilization on access link = 100%
  over-utilized link causes long queue (delay of minutes)
- total delay = Internet delay + access delay + LAN delay
  = 2 secs + minutes + milliseconds

Web Caching Example: No Caching

Possible solution

- increase access link bandwidth to, say, 10 Mbps (often a costly upgrade)



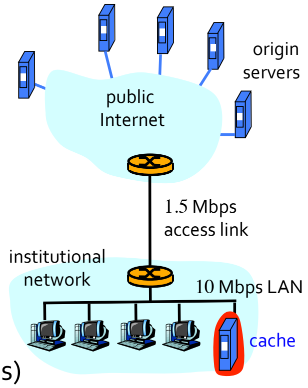
Performance:

- utilization on LAN = 15%
- utilization on access link = 15%
- total delay = Internet delay + access delay + LAN delay
  = 2 secs + msecs + msecs

Web Caching Example: With Caching

Another solution: install cache

- assume hit rate of 0.4



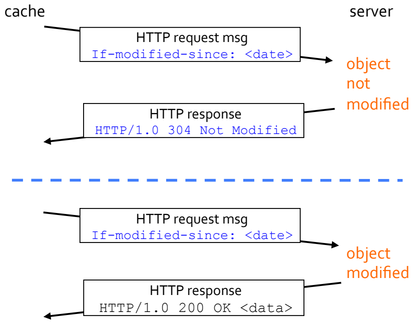
Performance:

- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msecs)
- avg. total delay = Internet delay + access delay + LAN delay
  = .6*(2.01) secs + msecs < 1.4 secs

Conditional `GET`

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

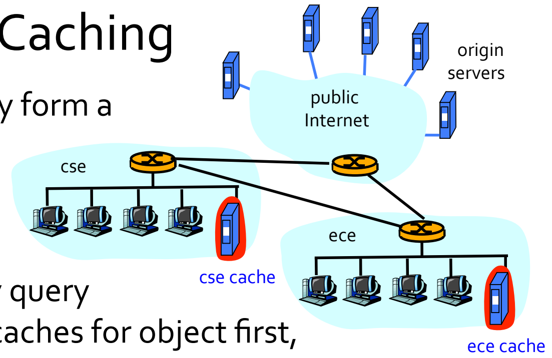
- cache: specifies date of cached copy in HTTP request `If-modified-since: <date>`
- server: response contains no object if cached copy is up-to-date: `HTTP/1.0 304 Not Modified`



May be used with or without TTL,
TTL hard to set, depends on site content

Cooperative Caching

Multiple caches may form a distributed cache



Instead of going directly to origin server, a cache may query one or more other caches for object first, e.g., cse cache queries ece cache first

To eliminate frequent inter-cache query-reply, each cache may push an index of its contents to other caches, i.e., ece cache tells cse cache all the objects it is holding

Frequently, this "index" is in the form of a Bloom Filter

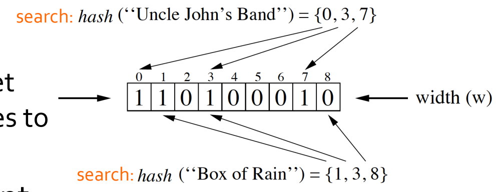
Bloom Filter

An efficient, lossy way of describing a set, comprising:

- a bit vector of length w
- a family of independent hash functions
  - each maps an element of the set to an integer in $[0, w)$

To insert an element:

- for each hash function, set the bit the element hashes to



To search for an element:

- for each hash function, examine the bit the element hashes to
- if any bit is not set, the element is definitely not in the set
- if all the bits are set, the element may be in the set (potential for false positive)

Bloom Filter

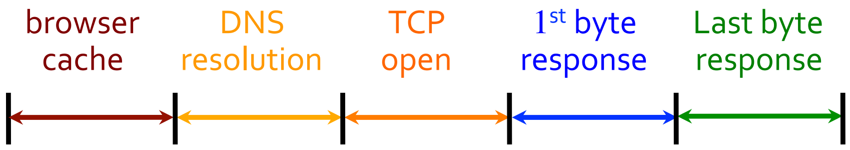
The false positive rate is a well-defined, linear function of:

1. width(w),
2. the number of hash functions, and
3. the number of elements in the set

- wider filters are always more accurate
- optimal tradeoff between filter storage and accuracy is when about half of the bits are set

Bloom Filters also useful in maintaining p2p supernode backbone and distributed storage in data center network

Variable Delay



browser cache

DNS resolution

TCP open

1st byte response

Last byte response

Sources of variability of delay

- browser cache hit/miss, need for cache revalidation
- DNS cache hit/miss, multiple DNS servers, errors
- TCP handshake, packet loss, high RTT, server accept queue
- RTT, busy server, CPU overhead (e.g., CGI script)
- response size, receive buffer size, congestion

Limitations of Web Caching

Significant fraction ($>50\%$) of HTTP objects are not cacheable

Why not?

- dynamic data: stock prices, scores, web cams
- scripts: results based on passed parameters
- use of cookies: results may be based on passed data
- advertising / analytics: owner wants to measure #hits
  - random strings in content ensure unique counting
- HTTPS: encrypted data is not cacheable
- multimedia: object larger than cache or not allowed to be cached due to intellectual property rights

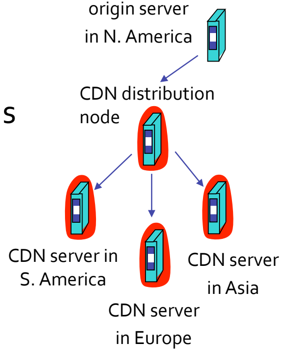
How to ensure scalability of web server when content is not cacheable?

Content Distribution Networks (CDNs)

Streaming large files (e.g., video) from a single origin server in real time requires large amount of bandwidth

Solution: replicate content to hundreds of servers throughout the Internet

- place servers in edge/access network
- content pre-downloaded to servers
- when user downloads content, direct user to the server closest to it
- placing content "close to" user avoids network delay and loss of long paths



CDNs vs. Content Owners

Maintaining your own network of such servers is expensive (both CAPEX and OPEX)

CDN providers maintain a network of servers and sell content replication service to multiple content owners

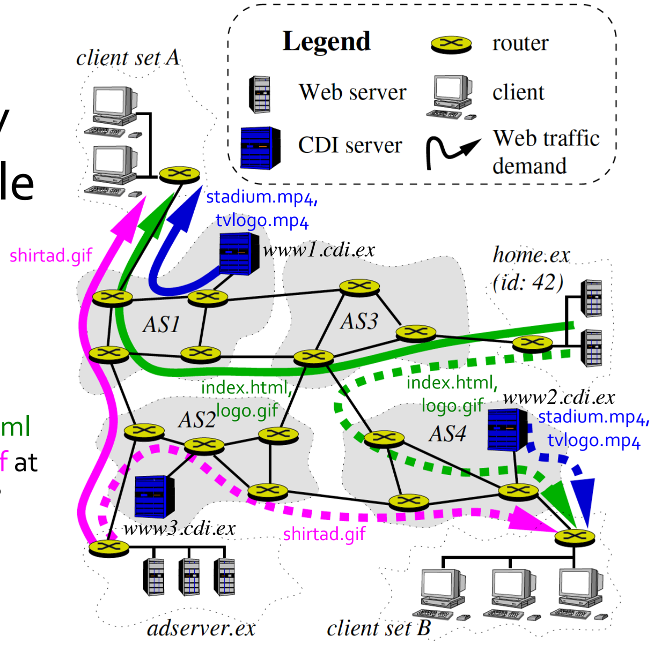
- example of content owners: ABC, HBO, Netflix
- example of CDN providers: Akamai, Limelight
  - Akamai has ~25K servers spread over ~1K clusters world-wide

CDN replicates owners' content in CDN servers

When owner updates content, CDN updates servers

Some large content owners operate their own CDNs: Amazon, Google/YouTube, Netflix (virtual)

Sample Delivery (Example Only)



Why don't we store index.html and shirtad.gif at the CDN also?

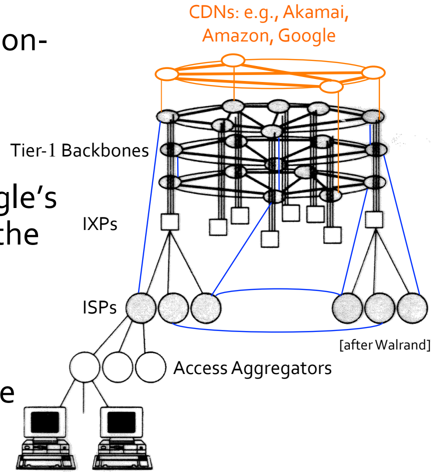
[Frank13]

Content Distribution Network

CDN nodes create application-layer overlay network

Larger CDNs may have their own WANs, e.g., Google's B4, that interconnect with the rest of the Internet like any other ISP's network

CDN directs a request to the server closest to the client (how?)



[after Walrand]

Client Redirection

How to direct clients to a particular server?

As part of application: HTTP redirect

- pros: application-level, fine-grained control
- cons: additional load and RTTs, hard to cache

As part of naming: DNS

- pros: well-suited to caching, reduce RTTs
- cons: relies on proxies and estimations, not accurate

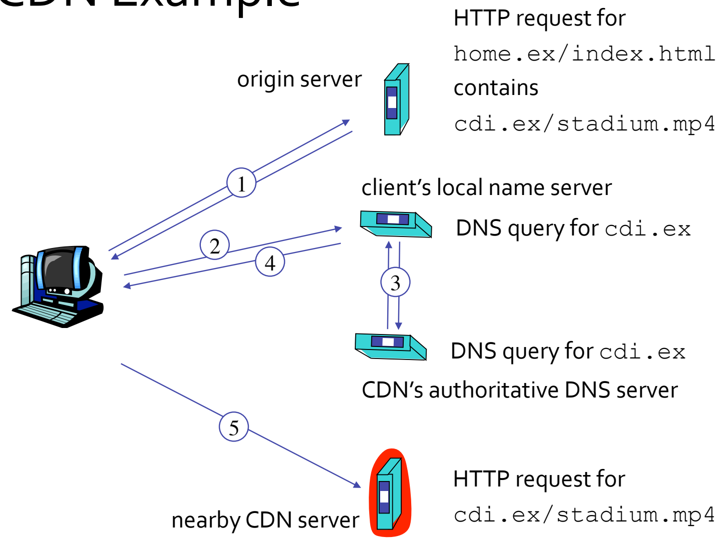
Pros and cons of each?

DNS-based Redirection

Clients are directed to the closest server as part of the DNS name resolution process:

1. client asks its local DNS resolver to resolve CDN's server's name
2. the local DNS resolver is directed to CDN's authoritative name server by DNS
3. CDN's name server either returns the address of server closest to the DNS resolver or an ordered list of addresses, ranked by distance to local DNS resolver

CDN Example



origin server

HTTP request for `home.ex/index.html` contains `cdi.ex/stadium.mp4`

client’s local name server

DNS query for `cdi.ex`

DNS query for `cdi.ex`

CDN’s authoritative DNS server

nearby CDN server

HTTP request for `cdi.ex/stadium.mp4`

Server Selection

How to choose which server to direct a client?

- server load
- client-server distance
  - CDN maintains a “map”, estimating distances between access ISPs and CDN nodes
  - CDN’s name server uses “map” to determine server closest to the local DNS resolver
  - DNS resolver used as proxy for client → inaccurate location
    - CDN doesn’t know client’s address at name resolution time
  - distance can be measured using different metrics, e.g., latency, loss rate → only estimated
- delivery cost (ISP pricing)