CS168

Lecture 19

Today in Networking

- 22nd anniversary of Mozilla’s official launch (1998)

- The first web browser to really take off was *Mosaic*
  - Developed at National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana–Champaign
  - Funding from “Gore Bill”
- One of its developers (Marc Andreessen) went on to found Netscape
  - Internally, Netscape’s browser (Netscape Navigator) was called “Mozilla”
  - This browser *totally dominated* the web for a crucial period

- In 1998, Mozilla the organization released the browser code under an open license
- .. this eventually evolved into Firefox
- .. and all the other things the Mozilla Foundation does for the Internet!

The Web

Where are we?

- Before the break, I said we were starting to look at *user-facing* things

- Started with DNS, which (at least initially) provided a user-facing system for interacting with the network: *names* instead of addresses

- Today:

  - The web — a game changing user-facing killer app

The Web

- Abbreviated history and motivation

- The basics
  - HTML, clients, servers, URLs
  - Basic HTTP

- Availability, scalability, and performance
  - Caching
  - Content Delivery Networks
  - TCP and HTTP

- Back to basics
  - Statelessness

The Web: Abbr. Hist.

The Web: Very abbreviated history

- In 1989, Tim Berners-Lee (then a software engineer at CERN) saw a problem
  - Lots of information
  - Information being added to and *changed* all the time
  - People come and go

  - Information gets lost
    - It's often recorded — somewhere!

  - CERN had a documentation system — CERNDOC
    - Hierarchical
    - Frustrating — information is not always hierarchical!

  - Pitched a solution — “Information Management: A Proposal”

The Web: Very abbreviated history

- In 1989, Tim Berners-Lee (then a software engineer at CERN) saw a problem
  - Lots of information
  - Information being added to and *changed* all
  - People come and go

  - Information gets lost
    - It's often recorded — somewhere!

  - CERN had a documentation system — CERNDOC
    - Hierarchical
    - Frustrating — information is not always hierarchical!

  - Pitched a solution — “Information Management: A Proposal”

> The actual observed working structure of the organisation is a multiply connected "web" whose interconnections evolve with time.
>
> — From "Information Management: A Proposal”

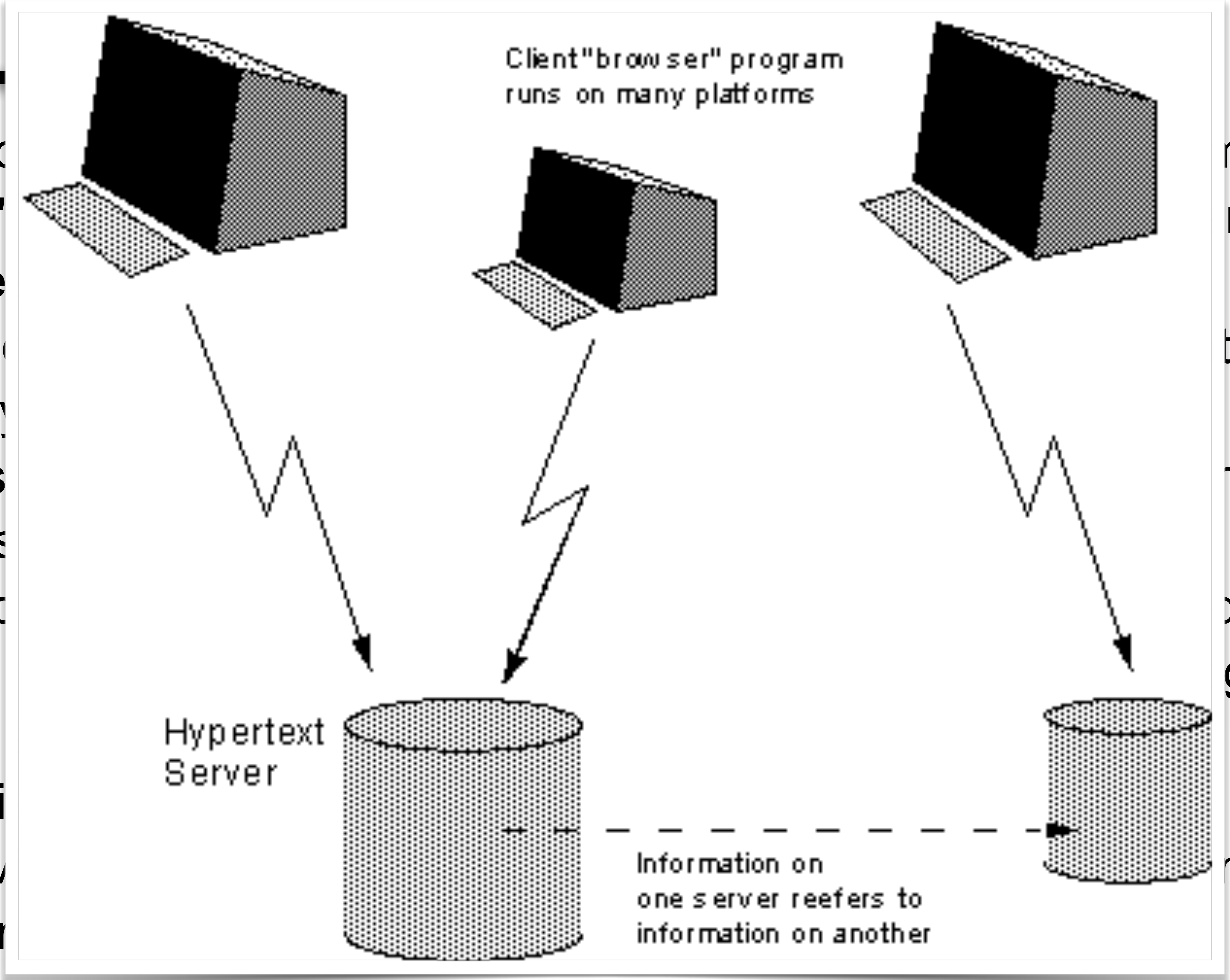
The Web: Very abbreviated history

- The method of storage must not place its own restraints on the information
  - .. a "web" of notes with links … is far more useful than a fixed hierarchical system.
- Remote access across networks
  - CERN is distributed, and access from remote machines is essential.
- Heterogeneity
  - Access is required to the same data from different types of system
- Non-Centralisation
  - Information systems start small and grow. They also start isolated and then merge. A new system must allow existing systems to be linked together without requiring any central control or coordination.
- Access to existing data
  - If we provide access to existing databases as though they were in hypertext form, the system will get off the ground quicker.

From “Information Management: A Proposal”, Tim Berners-Lee, CERN, 1989, 1990

The Web: Very abbreviated history

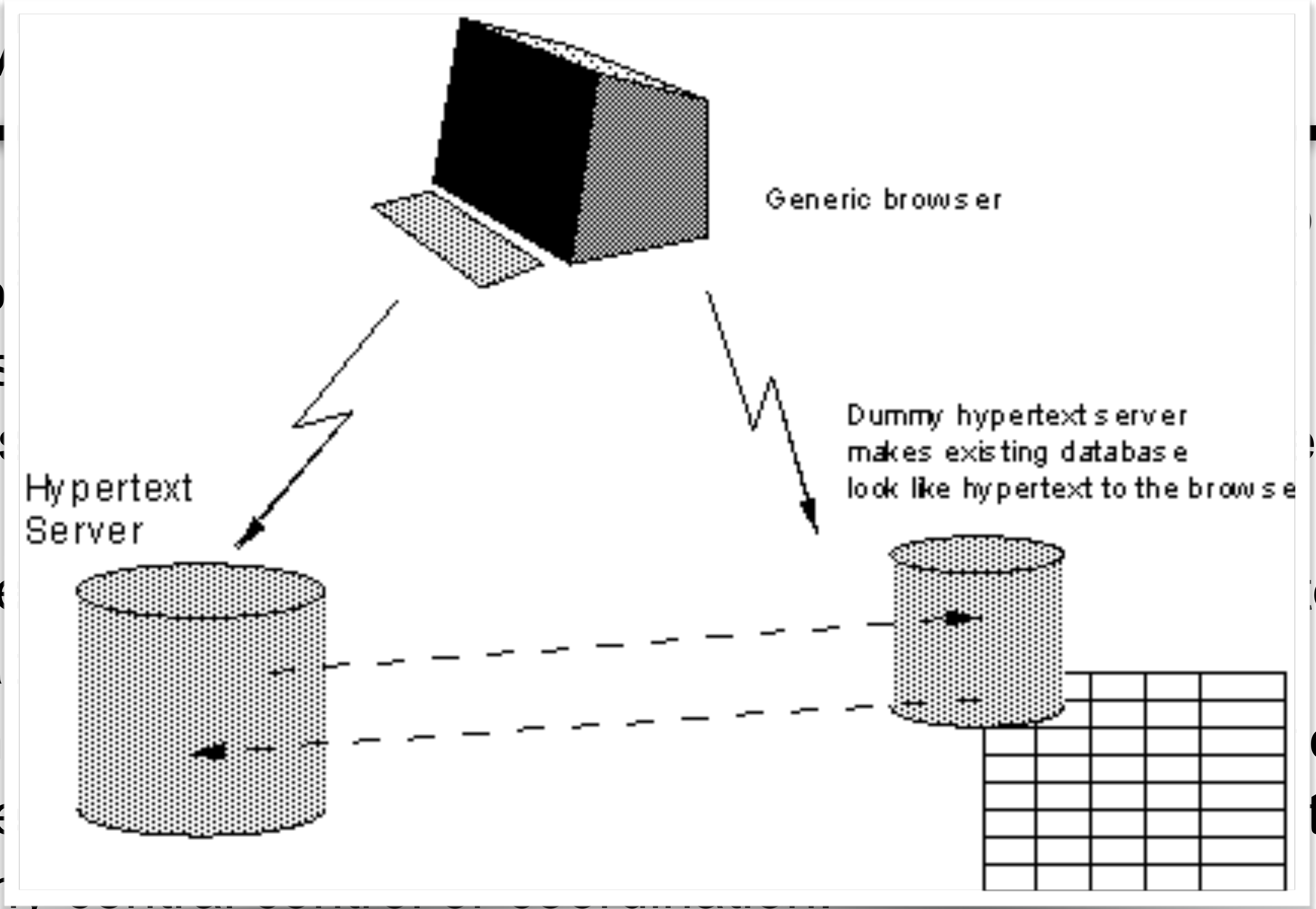
- The method c nation
  - .. a "web" rarchical system.
- Remote acce
  - CERN is tial.
- Heterogeneity
  - Access is n
- Non-Centralis
  - Informatic and then merge. A gether without requiring
- Access to exi
  - If we prov n hypertext form, the syster



Client"browser" program runs on many platforms
Hypertext Server
Information on one server reefers to information on another

From “Information Management: A Proposal”, Tim Berners-Lee, CERN, 1989, 1990

The Web: V


Generic browser
Hypertext Server
Dummy hypertext server makes existing database look like hypertext to the browse

- The method of rmation
  - .. a "web" o ierarchical system.
- Remote access
  - CERN is di ntial.
- Heterogeneity
  - Access is re em
- Non-Centralisa
  - Information ed and then merge. A ne together without requiring an
- Access to existing data
  - If we provide access to existing databases as though they were in hypertext form, the system will get off the ground quicker.

From “Information Management: A Proposal”, Tim Berners-Lee, CERN, 1989, 1990

The Web: Why was it so successful?

- It wasn’t trying to force anything
  - Didn’t need to structure data in a particular way
  - Didn’t need to store data in a particular format
  - Didn't need to use a particular computer/database system
  - Didn’t need to abandon existing (working) systems
- Had networks in mind from the beginning!
- Provided *integrated* interface to *scattered* information
- Was designed to be a *practical solution* to a *specific problem*
- They didn't try to charge for the technology
- *.. not all of this was new, but this was where they first all came together*

> Every good work of software starts by scratching a developer's personal itch.
>
> — Eric Raymond

The Web: Why was it so successful?

- What made it successful in the beginning is what makes it successful now!
  - It gives a lot of leeway for how websites work (didn’t over-specify)
  - Not tied to any one underlying system
  - No central authority — you can just add your own server/content
  - The ability to quickly navigate information from different sources

The Web: Why study it?

- The early web was mind-numbingly simple *technically*
- And was not cutting edge *intellectually* in terms of information representation
- But it was/is a successful and practical system that changed the world!

> - No professor could design something so simple
>   - Enough functionality to be effective
>   - Not enough to prove her cleverness
>
> — Professor Scott Shenker

- We couldn't possibly have a class about the Internet that didn't look at it!

The Web: Basics

The Web: Basic requirements

- Something to represent content with links: **HTML**

- Client program to access/navigate/display content (e.g. HTML): **Web browser**

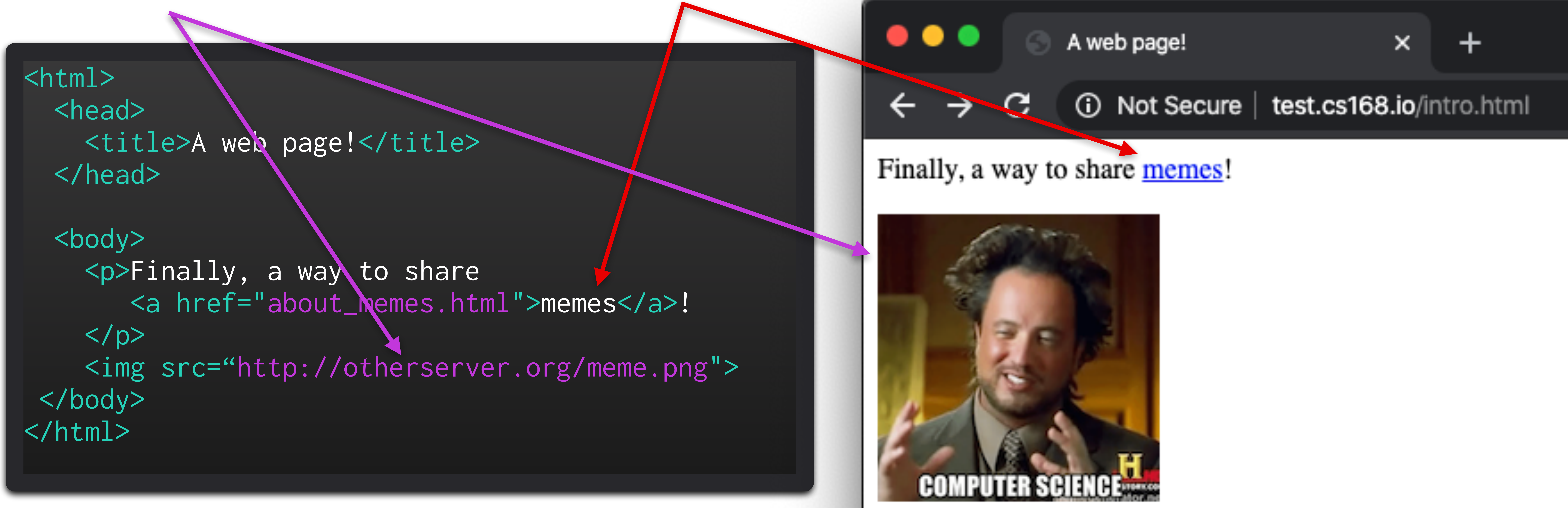
- A way to reference content: **URLs**
  - It's how you link/embed content to/in other content across a network
  - First general "handle" for arbitrary Internet content
  - Not just naming a host/processes (address/port)

- Something to host content: **Web servers**

- A protocol to get content from server to client: **HTTP**
  - Turns web URLs into TCP connections

Web basics

- HTML: HyperText Markup Language - Represent content with links
- Browser: Access/navigate/display content
- Provide *integrated* interface to *scattered* information



Embed another resource

Link to another resource

```
<html>
  <head>
    <title>A web page!</title>
  </head>

  <body>
    <p>Finally, a way to share
       <a href="about_memes.html">memes</a>!
    </p>
    <img src=“http://otherserver.org/meme.png">
 </body>
</html>
```

Finally, a way to share memes!

Web basics: URL Syntax

scheme : //host[:port] /path /resource

scheme	Typically a protocol: http, ftp, https, smtp, rtsp, *etc*.
host	DNS hostname or IP address
port	Defaults to protocol's standard port *e.g.* http: 80 https: 443
path	Traditionally reflecting file system
resource	Identifies the desired resource (traditionally a file) Can also extend to program executions: `http://us.f413.mail.yahoo.com/ym/ShowLetter?box=%40B%40Bulk&MsgId=2604_1744106_29699_1123_1261_0_28917_3552_1289957100&Search=&Nhead=f&YY=31454&order=down&sort=date&pos=0&view=a&head=b`

Web basics: URL Syntax

scheme : //host[:port]/path/resource[?query][#fragment]

scheme	Typically a protocol: http, ftp, https, smtp, rtsp, *etc*.
host	DNS hostname or IP address
port	Defaults to protocol’s standard port *e.g.* http: 80 https: 443
path	Traditionally reflecting file system
resource	Identifies the desired resource (traditionally a file)
query	e.g., search terms if resource is search program
fragment	Sub-part of resource (e.g., paragraph on web page)

Questions?

Flashback: Do we name the right things?

- URLs basically are hostname plus filename

- Is it ideal?
  - What if you move the file to another machine?
  - What if you want to replicate the file on many hosts so it's always available? Do you even care *which* host it's stored at?

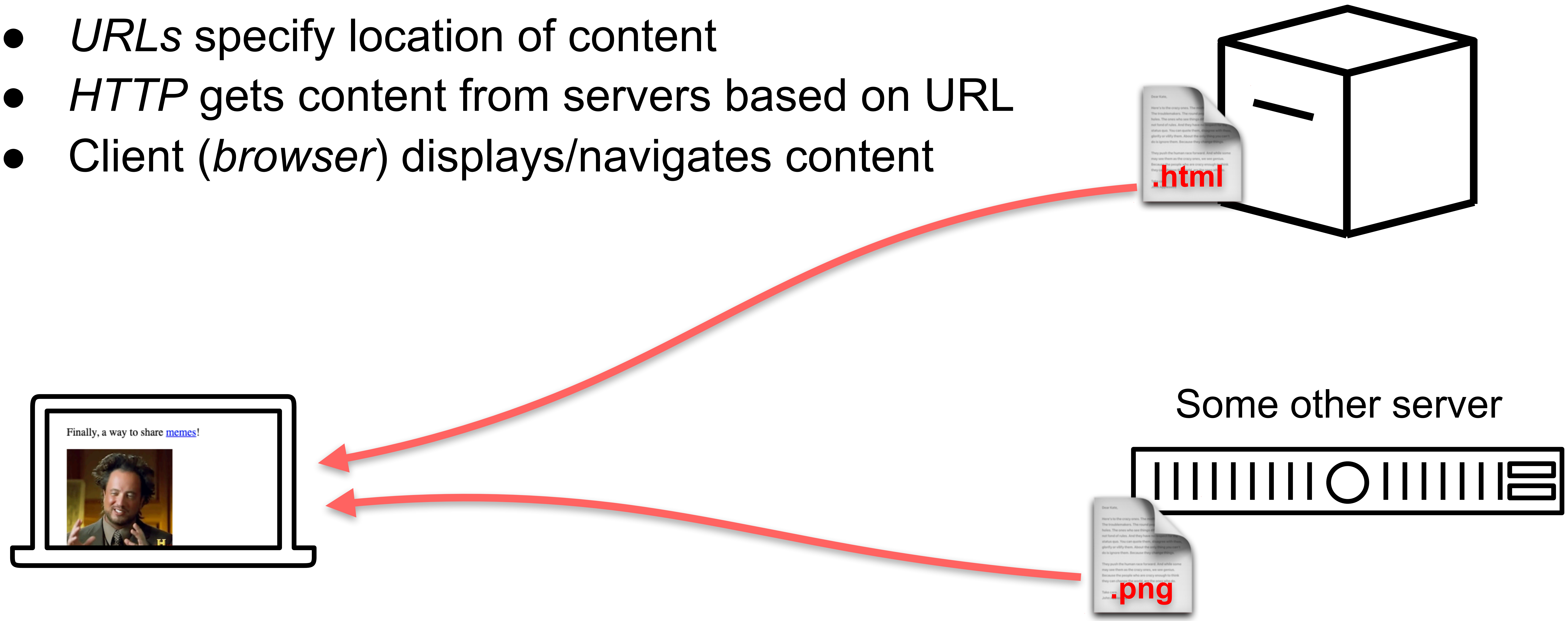
  - Should we be naming the *content* directly, rather than server+filename?
    - See: Information-Centric Networking, Content-Centric Networking, and **Named Data Networking**

  - Is the web more about accessing services? (your banking, Facebook, …)
    - Modern services certainly aren't tied to a specific host!
    - And a lot is *dynamic* — ***you're not fetching a file, you're running a program***
    - Should we be naming services directly?

Web basics: putting it all together

- *HTML* represents content with links/embeddings
- *Web servers* host the content
- *URLs* specify location of content
- *HTTP* gets content from servers based on URL
- Client (*browser*) displays/navigates content

cs168 server



Finally, a way to share memes!

Questions?

The Web: Basic requirements

- Something to represent content with links: **HTML**

- Client program to access/navigate/display content (e.g. HTML): **Web browser**

- A way to reference content: **URLs**
  - It’s how you link/embed content to/in other content across a network
  - First general “handle” for arbitrary Internet content
  - Not just naming a host/processes (address/port)

- Something to host content: **Web servers**

- A protocol to get content from server to client: **HTTP**
  - Turns web URLs into TCP connections

Basic HTTP

HyperText Transfer Protocol (HTTP)

- Focusing our discussion on common/current versions of HTTP:
  - HTTP 1.0 (1996) and HTTP 1.1 (1997)
  - These are (significant) outgrowth of original “HTTP 0.9”

- HTTP 2 published in 2015
  - Largely based on work by Google
  - As of 2020, 44% of websites use it
  - Significant departure; largely performance optimizations

- HTTP 3 forthcoming standard
  - Largely based on work by Google
  - As of 2020, 5% of websites use it (more or less Google and Facebook?)
  - Significant departure; largely performance optimizations

https://w3techs.com/technologies/details/ce-http2

HyperText Transfer Protocol (HTTP)

- The basics of HTTP:

- Client-server architecture

- Client connects to server on well-known TCP port 80
- Client issues request
- Server issues reply

You should basically understand what this is saying.

(We'll go into details, though.)

- Protocol is "stateless"

We'll come back to this.

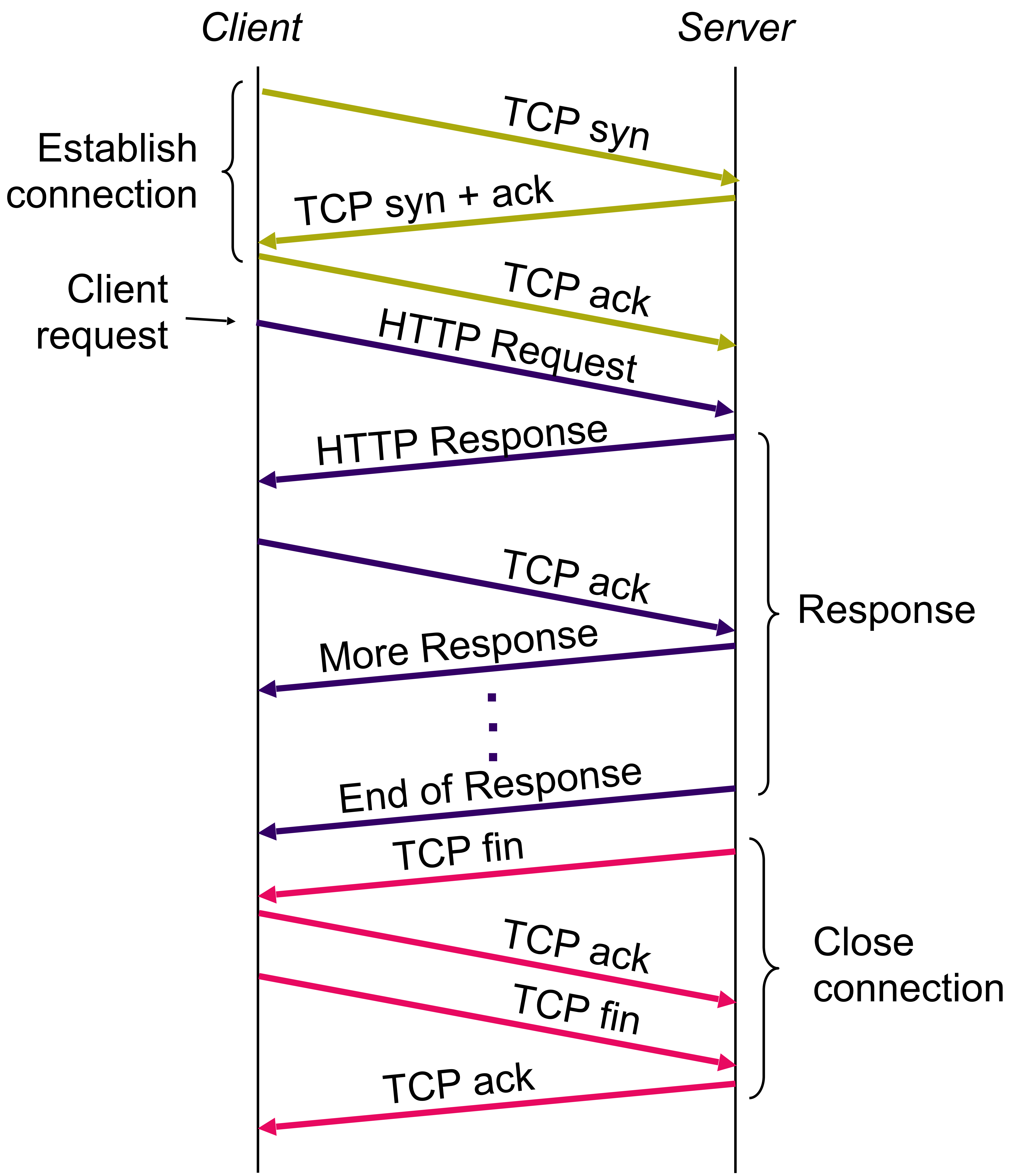
Inside an HTTP exchange

- (Simple HTTP 1.0 “GET” request)

- Client creates TCP connection (port 80)
- Client sends ***request***

- Server sends ***response*** packets
- Client ACKs them
  - Note: There may be unshown ACKs

- Server closes connection



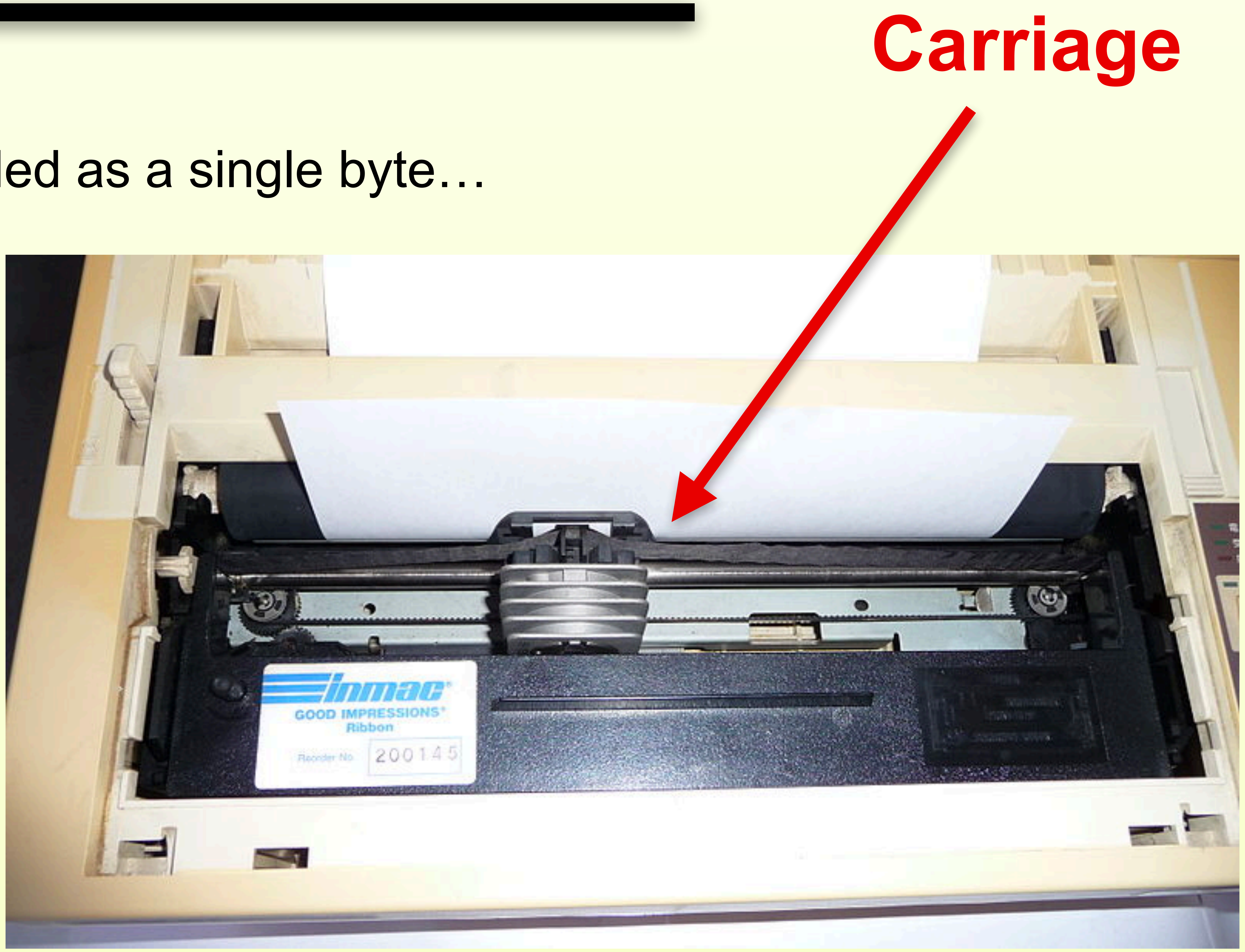
Inside an HTTP 1.0/1.1 request

- “Plain text” (“Latin 1” encoding)
  - Lines separated with CR LF

Request for
http://www.someschool.edu/main/about.html

Sidenote: CR and LF

- In common text encodings (ASCII, Latin 1, UTF-8)…
  - Common English letters and punctuation are encoded as a single byte…
    - 65 is “A”, 97 is “a”, 35 is “#”, etc.
  - 0 through 31 are *control characters*
    - 8 is backspace
    - 4 is “end of transmission”
    - 10 is **line feed (LF)**
    - 13 is **carriage return (CR)**



- You’re probably familiar with “\n” — newline
  - On Unix-like systems, this is really LF — does both
  - On Windows, means CR LF
  - Open a file in text mode in Python (and other languages), and it does translation
  - If you ever open up a file and every line ends with “^M” — those are the carriage returns — this was a Windows file and you’re on a Unix-like machine

Inside an HTTP 1.0/1.1 request

- “Plain text” (“Latin 1” encoding)
  - Lines separated with CR LF
- Request line:
  - Method - “action” to perform. GET/HEAD/POST/…
  - Resource - e.g., which thing to fetch
  - Protocol version - either 1.0 or 1.1
- Request headers:
  - Provide additional information or modify request
  - Some required; many optional
- Blank line
- Body:
  - Optional data
  - Used when submitting data (e.g., a form via POST)

Request for
http://www.someschool.edu/main/about.html

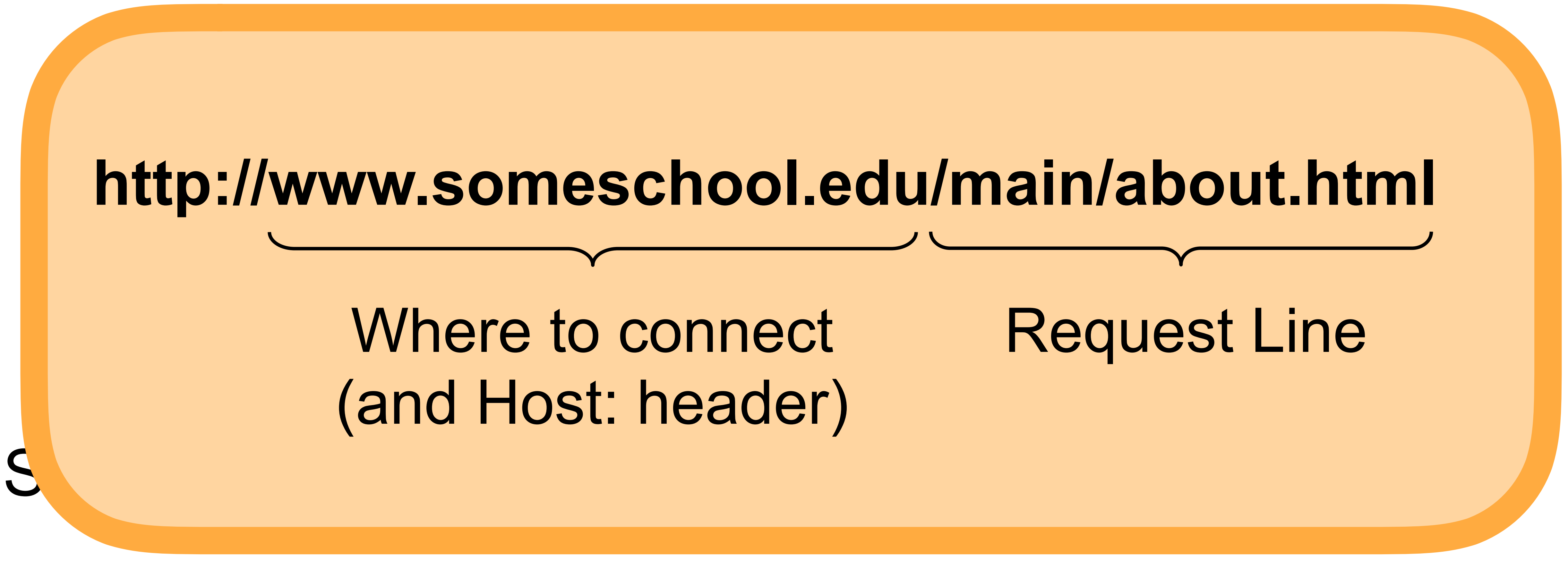
```
GET /main/about.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: en
(blank line)
(body, if there is one)
```

Inside an HTTP 1.0/1.1 request

- “Plain text” (“Latin 1” encoding)
  - Lines separated with CR LF
- Request line:
  - Method - “action” to perform. GET/HEAD/POST/…
  - Resource - e.g., which thing to fetch
  - Protocol version - either 1.0 or 1.1
- Request headers:
  - Provide additional information or modify request
  - Some required; many optional
- Blank line
- Body:
  - Optional data
  - Used when submitting data (e.g., a form via POS

Request for
http://www.someschool.edu/main/about.html

```
GET /main/about.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: en
(blank line)
(body, if there is one)
```



http://www.someschool.edu/main/about.html

Where to connect
(and Host: header)

Request Line

Inside an HTTP 1.0/1.1 response

- Status line:
  - Protocol version - either 1.0 or 1.1
  - Status code - 2xx=success, 4xx=error, …
  - Reason - Human-readable
- Response headers:
  - Provide additional information
- Blank line
- Body:
  - Optional data — but very common!
  - e.g., it’s the content of `about.html`!

Request for
http://www.someschool.edu/main/about.html

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 2006 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 2006 ...
Content-Length: 6821
Content-Type: text/html
 (blank line)
<html>
<head>
<title>About Some School</title>
</head>
...
```

Questions?

HTTP Methods (Common)

- GET
  - The classic!
  - Request to download some object
  - No body on request, body of reply is the requested object

- POST
  - Send data from client to server
  - e.g., submitting a web form, adding item to shopping cart, etc.
  - Body on request and often on response too

- HEAD
  - Basically same as a GET except you *don’t want the body* (just headers)
  - Used to, e.g., see if something exists, when it was modified, etc.

HTTP Response status codes (selected)

- 1xx - Informational
  - None defined in HTTP 1.0, only a couple in HTTP 1.1

- 2xx - Successful
  - ⭐ 200: OK (e.g., here's the web page you requested…)

- 3xx - Redirection
  - 301: Moved Permanently (Location header tells you new URL)
  - ⭐ 304: Not Modified (not really a redirection; we'll revisit this one)

- 4xx - Client Error
  - 400: Bad Request (catchall for when client messes up, e.g., didn't include a required header)
  - 401: Unauthorized (the resource requires a password or something)
  - ⭐ 404: Not Found (the bane of the early 2000s web, though funny/creative ones helped)

- 5xx - Server Error
  - 500: Internal Server Error (catchall for when server configuration goes wrong)

HyperText Transfer Protocol (HTTP)

- The basics of HTTP:

- Client-server architecture

- Client connects to server on well-known TCP port 80
- Client issues request
- Server issues reply

- Protocol is “stateless”

} We’ll come back to this.

Questions?

Where do we go from here?

- We've described the basics…
  - .. what else do you want?!

- Users
  - Fast! (Performant)
  - Highly available!
  - .. nobody likes a slow or broken site!

- Content provider
  - Fast and highly available (make users happy!)
  - Scalable (stay fast and highly available even with lots of users/content)

Do these goals sound really familiar?

.. they're basically the same as DNS!

Solve them using same ideas:
replication and *caching*!

Plus: Make up for some TCP issues…

HTTP

Availability, scalability, and performance

HTTP: Availability, scalability, and performance

- Like with DNS, these topics are somewhat intertwined!
- We'll discuss three things here:
  - Caching
  - Content Delivery Networks (CDNs)
  - Interplay of HTTP and TCP

Web Caching

HTTP caching: Why does caching work?

- *Why* does caching work?
  - Exploits *locality of reference* AKA *principle of locality*
    - *Spatial locality* — If something is accessed, something near it will also probably be accessed
    - *Temporal locality* — If something is accessed, it'll probably be accessed again soon

    - Both were a factor if you took CS61C
    - One is much more relevant to web caching

HTTP caching: How well does caching work?

- *How well* does caching work?

  - Very well up to a point…
  - .. file popularity has high peak but long tail
    - Large overlap in highly popular content
    - But many unique requests
    - .. common to many types of cache

  - In the real world…
    - Content is increasingly dynamic (personalized feeds, many updates)

  - But there's still a lot of static content worth caching
    - Images, CSS stylesheets, JavaScript libraries, …

Everyone downloads the same viral memes…

.. but everyone has their own weird interests.

HTTP caching: How does caching work?

- *How* does caching work in HTTP?

- The key is in the headers…

- Response headers:
  - `Cache-Control`
  - `Expires`

HTTP caching: the Cache-Control header

- `Cache-Control` header used for lots of cache-related things
- Used for both requests and responses

- Most important use is for server (response) to specify `max-age`

  - It’s just a TTL in seconds — how long response can be cached

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

HTTP caching: the Expires header

- `Cache-Control` is only available in HTTP 1.1
- HTTP 1.0 uses `Expires` response header
  - It’s just a TTL in absolute time — when cached response becomes invalid
  - `Expires: `**`Thu, 31 Dec 2037 23:55:55 GMT`**
- Servers often send both `Cache-Control: max-age` and `Expires`

HTTP caching: How does caching work?

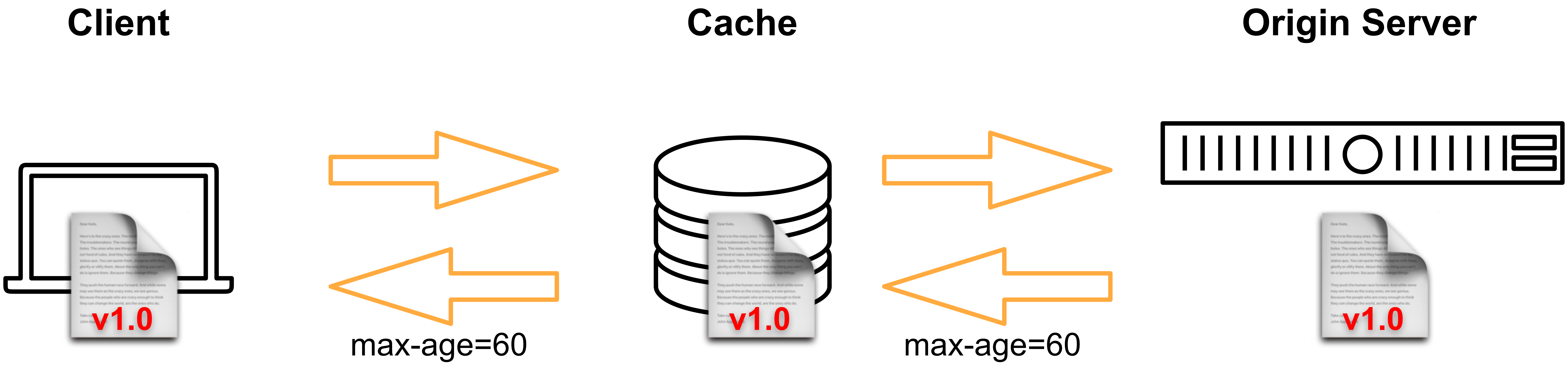
- *How* does caching work in HTTP?

- The key is in the headers…

- Response headers providing TTLs
    - `Cache-Control: max-age` (HTTP 1.1)
    - `Expires` (HTTP 1.0)

    - But TTLs aren’t always good enough!
    - .. server doesn’t necessarily *really* know when content will be updated
    - .. clients need a way to force skipping of caches!
        - `Cache-Control: no-cache` and `Pragma: no-cache` request headers

HTTP caching: How does caching work?



Client

Cache

Origin Server

v1.0

v1.0

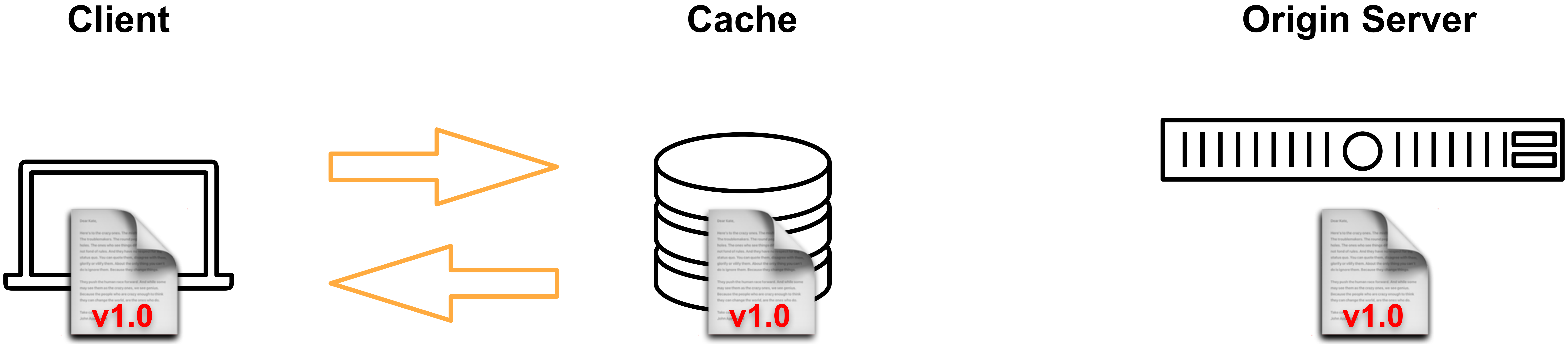
v1.0

max-age=60

max-age=60

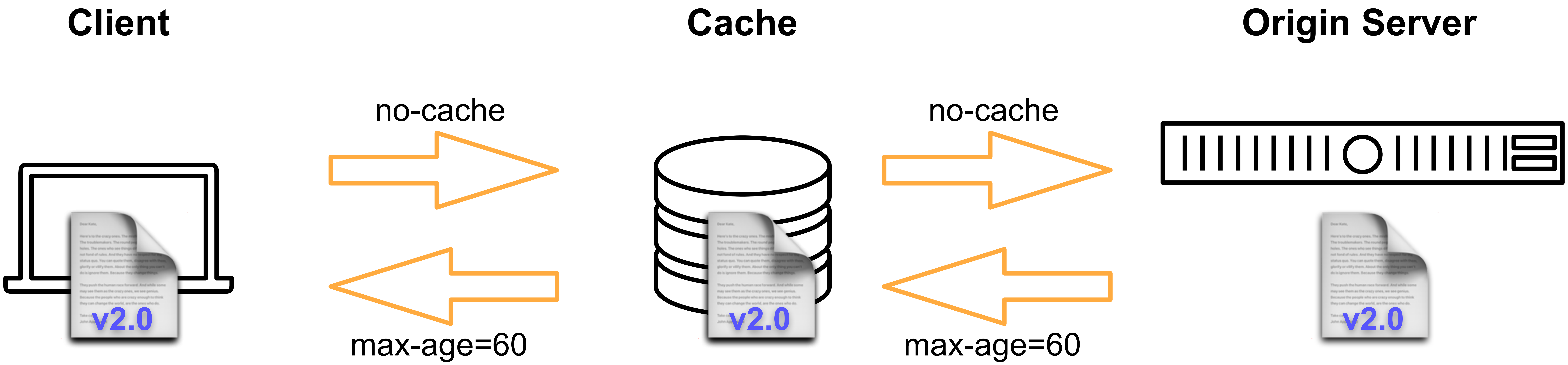
Client requests document; cached for one minute at t=0

HTTP caching: How does caching work?



Client requests document; cached for one minute at t=0
Document updated on server at t=1 — refresh would be stale until t=60!

HTTP caching: How does caching work?



Client requests document; cached for one minute at t=0
Document updated on server at t=1 — refresh would be stale until t=60!
User does “hard refresh” at t=10 (shift-click refresh in browser)

What if document hadn’t been updated? We just transferred it again for nothing!
v1.0 was already in the caches!

HTTP caching: How does caching work?

- Request header `If-Modified-Since:` **`<date>`**

  - If resource ***has*** changed since `<date>`:
    - Respond with latest version

  - If resource ***has not*** changed:
    - Respond with `304 - Not Modified`

    - Includes headers
    - But not the body

    - .. lets you know you’re up-to-date, but doesn’t waste bandwidth

HTTP caching: Typical caching interaction

- Client issues request for resource

- If resource in browser cache:
  - If cached version not expired (TTL > 0)
    - Assumed to be current — use version in browser cache
  - Else, cached version is expired
    - Send request using `If-Modified-Since: <date of cached version>`
    - If server's version is newer:
      - Respond with new version (`200` response)
    - If server's version has same date:
      - Respond with `Not Modified` (`304` response)
- Else, resource not in browser cache
  - Send request to server (with no `If-Modified-Since`)

HTTP caching: Typical caching interaction

- Client issues request for resource

- If resource in browser cache:
  - If cached version not expired (TTL > 0)
    - Assumed to be current — use version in browser cache
  - Else, cached version is expired
    - Send request using `If-Modified-Since: <date of cached version>`
    - If server's version is newer:
      - Respond with new version (`200` response)
    - If server's version has same date:
      - Respond with `Not Modified` (`304` response)
- Else, resource not in browser cache
  - Send request to server (with no `If-Modi`

What if server's version is ***older***?

HTTP caching: Typical caching interaction

- Client issues request for reso

- If resource in browser cache:
  - If cached version not expi
    - Assumed to be current — use version in browser cache
  - Else, cached version is expired
    - Send request using `If-Modified-Since: <date of cached version>`
    - If server’s version is newer:
      - Respond with new version (`200` response)
    - If server’s version has same date:
      - Respond with `Not Modified` (`304` response)
- Else, resource not in browser cache
  - Send request to server (with no `If-Modified-Since`)

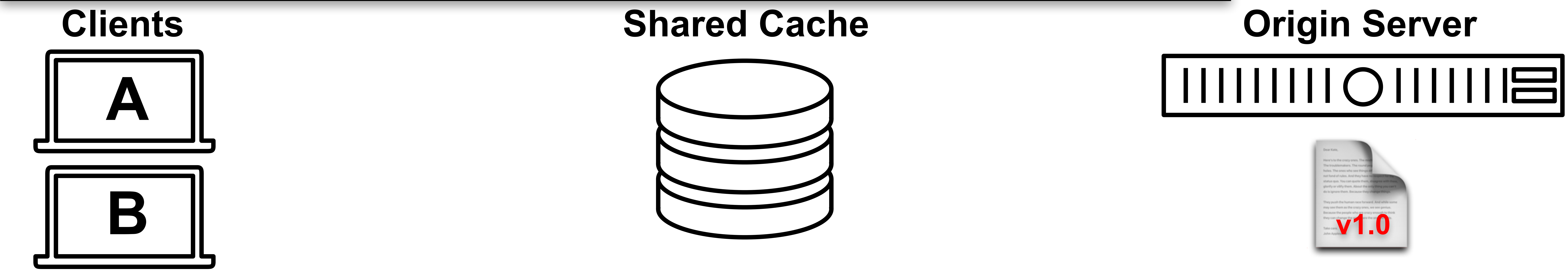
This example just looked at browser cache.

When browser actually makes requests, they may pass through other caches that use a similar algorithm!

Questions?

HTTP caching: Two-client example

Doc TTL is 5

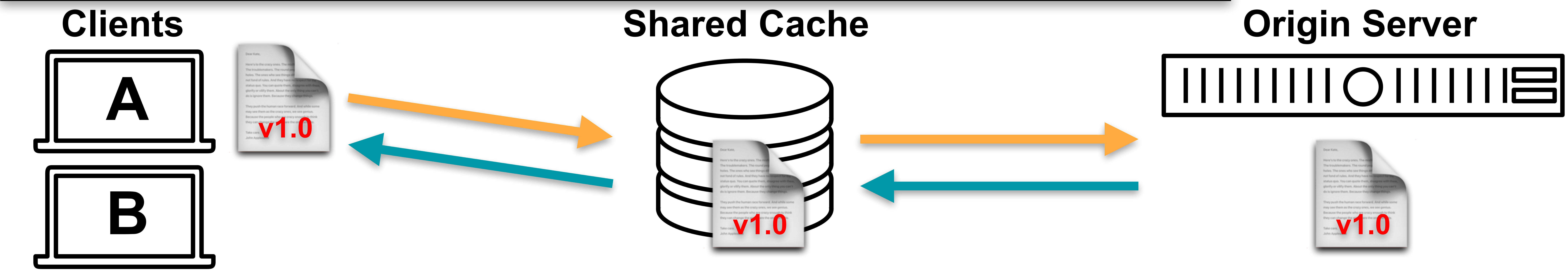


When?	What?	Result?

- Clients are downloading a document.
- Clients have local (browser) caches.
- Clients also share cache in network.
- Document TTL is 5 (minutes).

HTTP caching: Two-client example

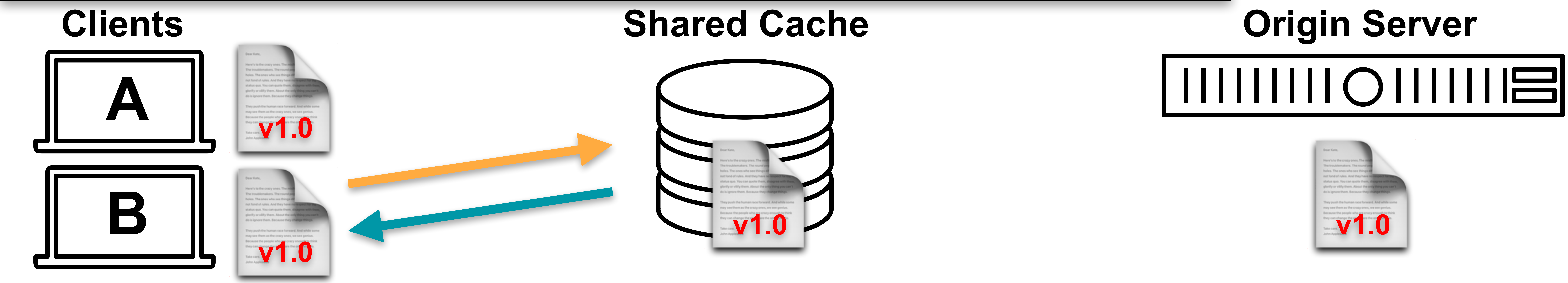
Doc TTL is 5



When?	What?	Result?
T=0	A requests	Fetched from Origin Server

HTTP caching: Two-client example

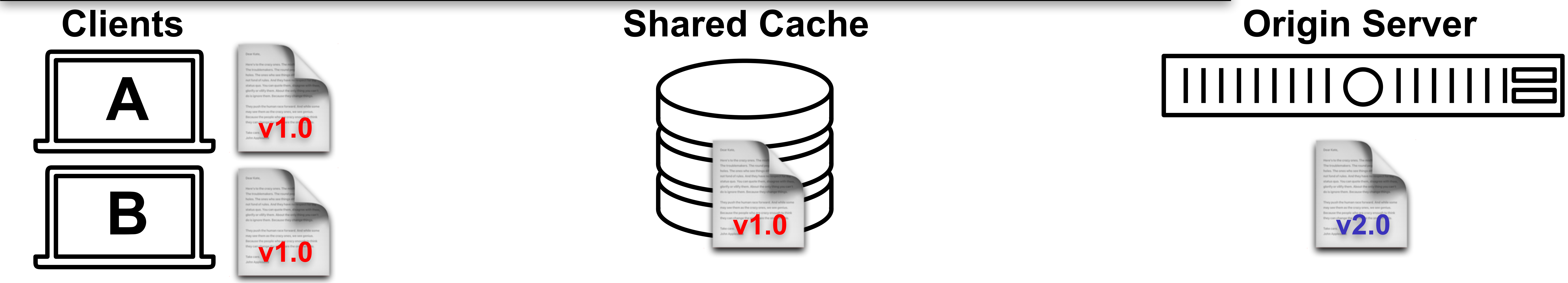
Doc TTL is 5



When?	What?	Result?
T=0	A requests	Fetched from Origin Server
T=1	B requests	Fetched from Shared Cache

HTTP caching: Two-client example

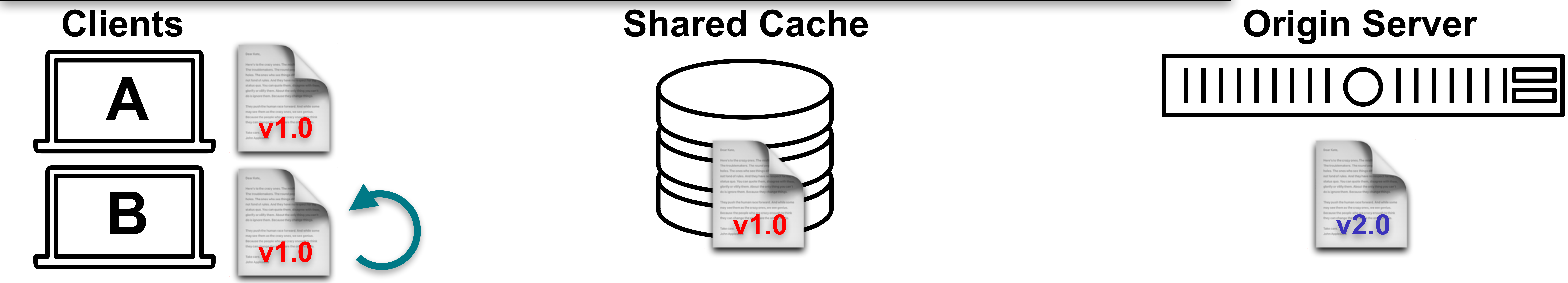
Doc TTL is 5



When?	What?	Result?
T=0	A requests	Fetched from Origin Server
T=1	B requests	Fetched from Shared Cache
T=2	Doc modified on server!	—

HTTP caching: Two-client example

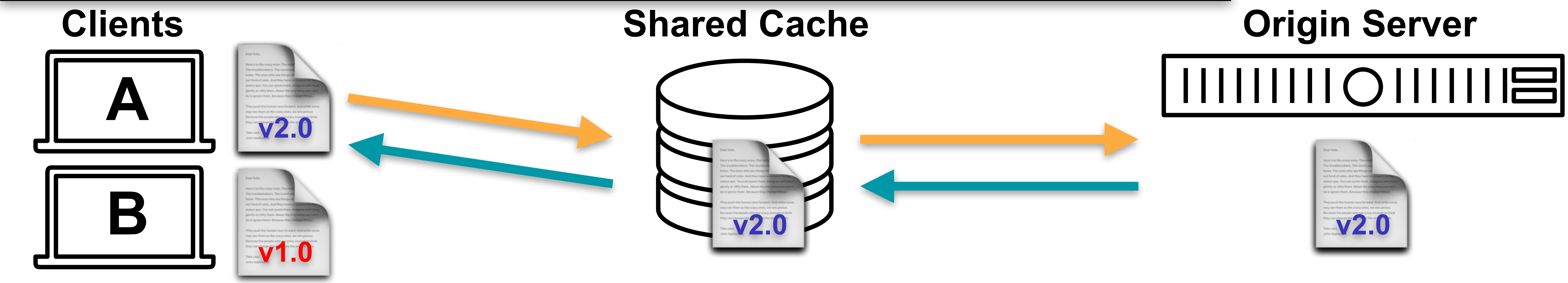
Doc TTL is 5



When?	What?	Result?
T=0	A requests	Fetched from Origin Server
T=1	B requests	Fetched from Shared Cache
T=2	Doc modified on server!	—
T=4	B requests	Fetched from Browser Cache

HTTP caching: Two-client example

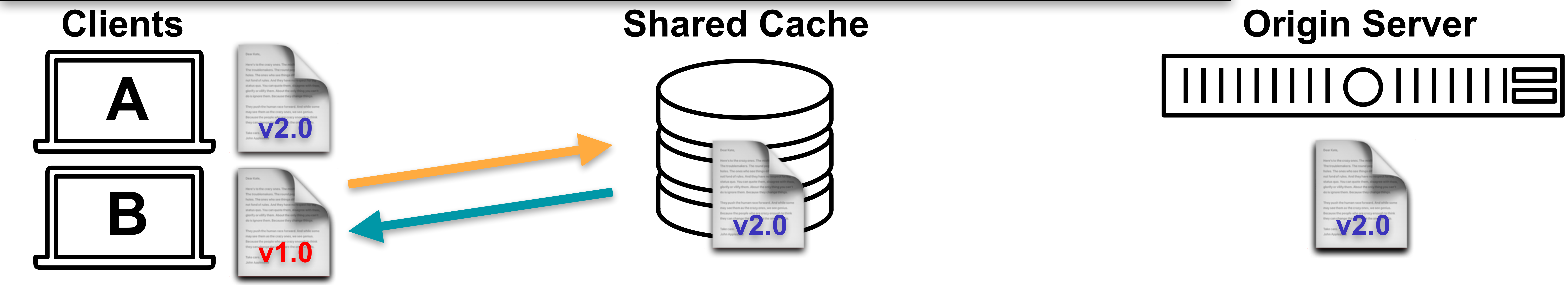
Doc TTL is 5



When?	What?	Result?
T=0	A requests	Fetched from Origin Server
T=1	B requests	Fetched from Shared Cache
T=2	Doc modified on server!	—
T=4	B requests	Fetched from Browser Cache
T=6	A requests	Client **A** sends If-Modified-Since (to Shared Cache) Shared Cache sends If-Modified-Since (to Origin Server) v2.0 fetched from Origin Server

HTTP caching: Two-client example

Doc TTL is 5



When?	What?	Result?
T=0	A requests	Fetched from Origin Server
T=1	B requests	Fetched from Shared Cache
T=2	Doc modified on server!	—
T=4	B requests	Fetched from Browser Cache
T=6	A requests	Client **A** sends If-Modified-Since (to Shared Cache) Shared Cache sends If-Modified-Since (to Origin Server) v2.0 fetched from Origin Server
T=7	B requests	Client **B** sends If-Modified-Since (to Shared Cache) v2.0 fetched from Shared Cache

Questions?

HTTP caching: Summary of important cache headers

- **Response** headers providing TTLs:
  - `Cache-Control: max-age` and `Expires`

- **Request** headers allowing overriding of TTLs:
  - `Cache-Control: no-cache` and `Pragma: no-cache`
  - Can be triggered by “shift-refresh”

- Allow requests that skip body if cache is up to date:
  - Request header `If-Modified-Since:` **`<date>`**

- Remember: you can have multiple caches along paths!

Questions?

HTTP caching: Final word on Cache-Control header

- A couple other important uses of `Cache-Control` in response…
    - `Cache-Control: no-store`
        - Don’t cache this!
        - Always request from origin server
        - e.g., for things like banking data
    - `Cache-Control: private`
        - Content only meant for one user
        - Okay to store in private (browser) cache
        - .. but don’t store it in shared proxy server cache!

HTTP caching: Where?

- We’ve discussed how caching works…
- .. but *where* are the caches?

- The client!

- Proxy servers

Proxy Servers

- *Proxy server*: A server that makes requests on behalf of a client

  - The caches we saw in previous examples fit that definition
    - .. *caching* is a major feature of web proxy servers

  - Also often used to *enforce policy*
    - .. company blocks all traffic except through proxy
    - .. proxy has whitelist/blacklist

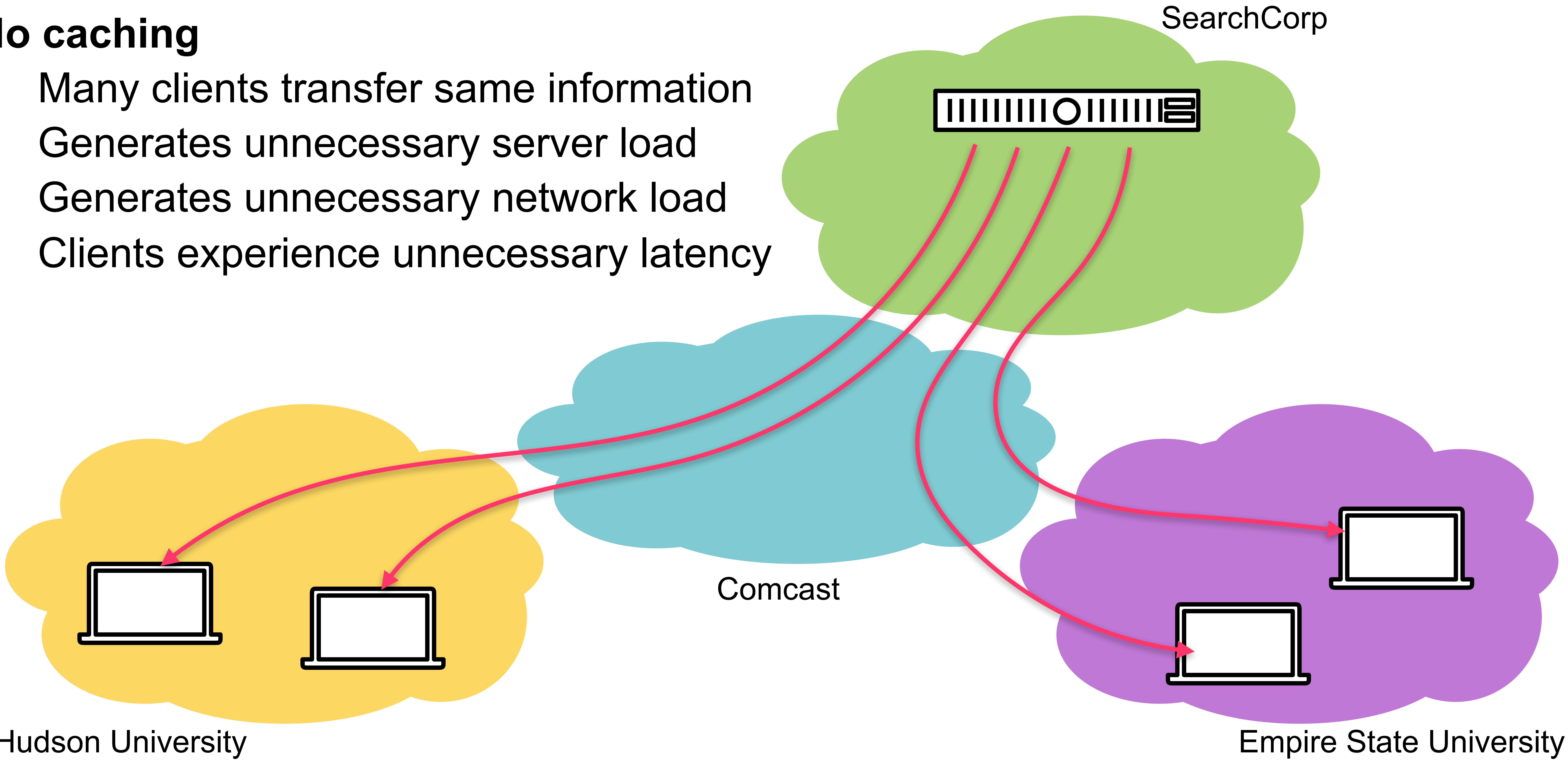
  - Also often used to do *load balancing*
    - .. request arrives at proxy
    - .. it redirects it to one of several equivalent servers

  - Note: our focus is the web, but other protocols have proxy servers too

HTTP caching: Where?

No caching

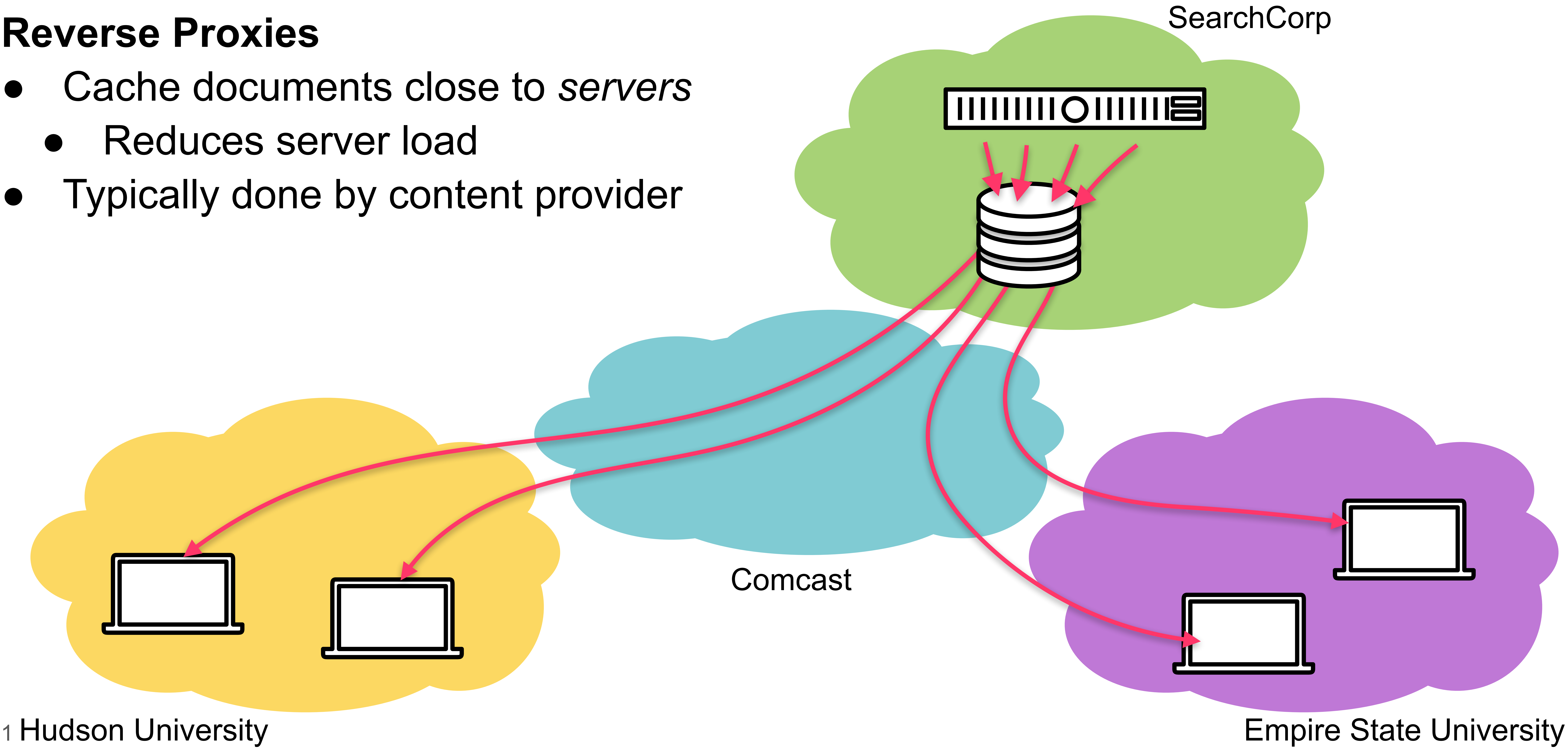
- Many clients transfer same information
- Generates unnecessary server load
- Generates unnecessary network load
- Clients experience unnecessary latency



HTTP caching: Where?

Reverse Proxies

- Cache documents close to *servers*
    - Reduces server load
- Typically done by content provider

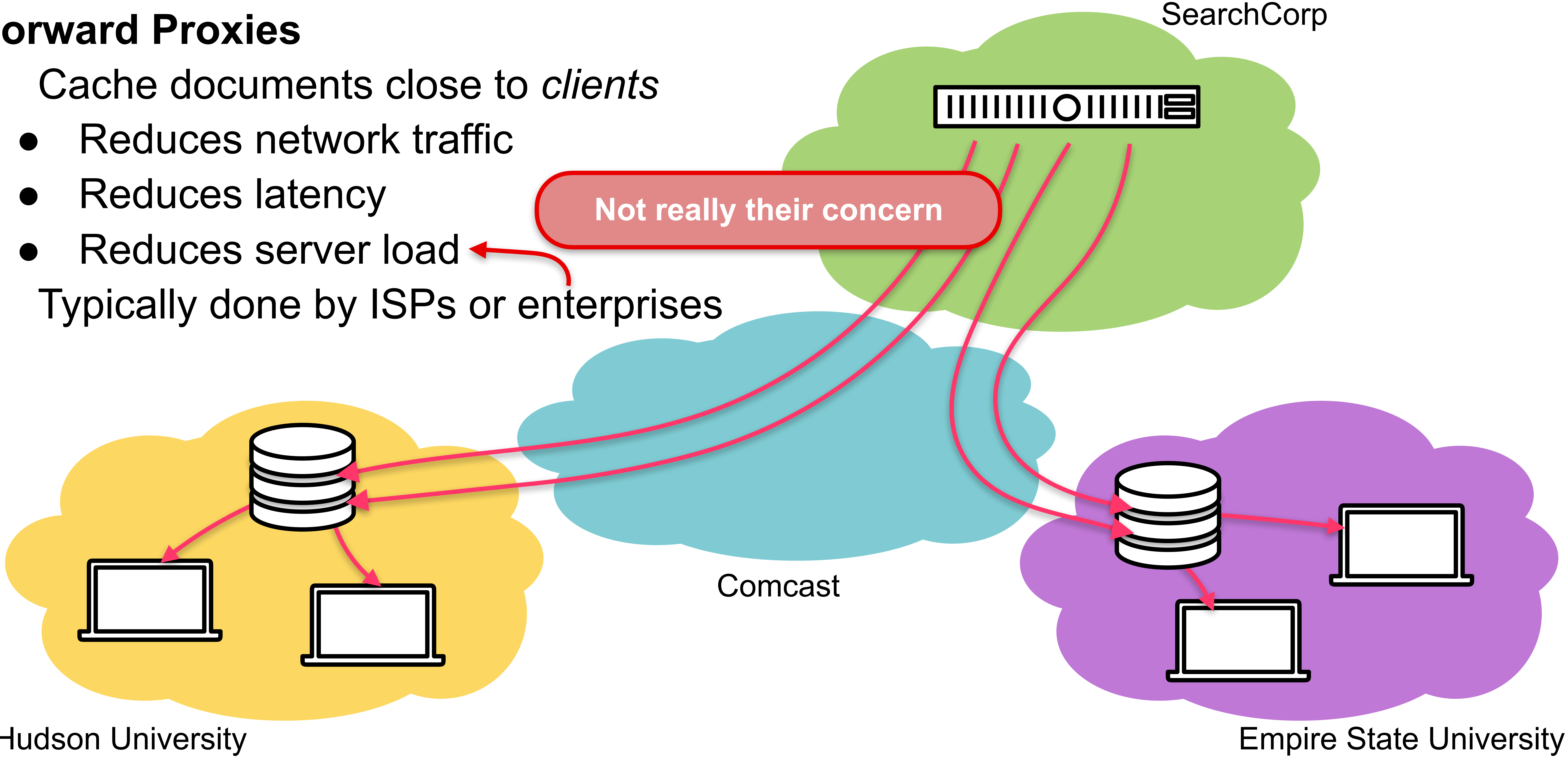


HTTP caching: Where?

Forward Proxies

- Cache documents close to *clients*
  - Reduces network traffic
  - Reduces latency
  - Reduces server load
- Typically done by ISPs or enterprises

Not really their concern



HTTP caching: Where?

- We’ve discussed how caching works…
- .. but *where* are the caches?

- The client!

- Proxy servers
  - Forward proxies (near client)
  - Reverse proxies (near server)

- Content Delivery Networks (CDNs)
  - This is its own subtopic!
  - Any questions before we move on to it?

Content Delivery Networks

CDNs

- *Replication* is a huge benefit to availability, scalability, and performance
  - We saw this with DNS!
  - Can spread the load
  - Places content closer to clients (less latency)

  - Caching is a form of opportunistic replication
    - .. but what if a given organization doesn’t have a forward proxy?
    - .. what if content provider and wants its content *always* replicated?

    - Idea: *Caching and replication as a service* — “CDNs 1.0”

CDNs “1.0”

- Large-scale distributed storage infrastructure
  - (Usually) administered by one entity
  - e.g., Akamai has 275,000+ servers in 136 countries

- How does content provider get its data onto Akamai’s servers?
- Two major ways
  - Pull
  - Push
  - .. we’ll come back to these in a moment

- Both typically used with DNS trick mentioned in previous lecture

CDNs “1.0”: The basic idea

- Content provider buys service from a CDN, e.g., Akamai
- CDN creates new domain names for the customer content provider
  - e.g., e12596.dscj.akamaiedge.net for cnn.com
  - The CDN’s DNS servers are authoritative for the new domains
- Content provider modifies its content so that embedded URLs reference the new domains
  - “Akamaize” content
  - e.g.: http://www.cnn.com/some-photo.jpg becomes http://e12596.dscj.akamaiedge.net/some-photo.jpg
- Initial request goes to CNN (e.g., for main http://www.cnn.com page)
  - .. but embedded links go to Akamai, which handles DNS resolution for URL
  - .. Akamai DNS servers pick one of their 275,000+ servers to serve it
    (based on IP geolocation, server load, etc. — see Lecture 17 - Intelligent indirection)

CDNs "1.0": The basic idea

- Content provider buys service from a CDN, e.g., Akamai

- CDN creates new domain names for the customer content provider
  - e.g., e12
  - The CDI

- Content prov e new domains
  - "Akamai
  - e.g.: http:/ et/some-photo.jpg

- Initial request goes to CNN (e.g., for main http://www.cnn.com page)
  - .. but embedded links go to Akamai, which handles DNS resolution for URL
  - .. Akamai DNS servers pick one of their 275,000+ servers to serve it
    (based on IP geolocation, server load, etc. — see Lecture 17 - Intelligent indirection)

DNS Pop Quiz

Q: What if CNN doesn't want to embed a weird Akamai domain name in its pages?

A: Add a CNAME record to CNN nameserver
(CNAME, cdn.cnn.com, e12596.dscj.akamaiedge.net)

CDNs “1.0”: The basic idea

- How does content provider get data onto CDN’s servers?

- *Pull*
    - Akamai servers act like a cache
    - Content provider gives CDN “origin” URL
    - When a client requests from Akamai
        - .. if cached, serve it
        - .. if not cached, request (“pull”) from origin, cache it, serve it

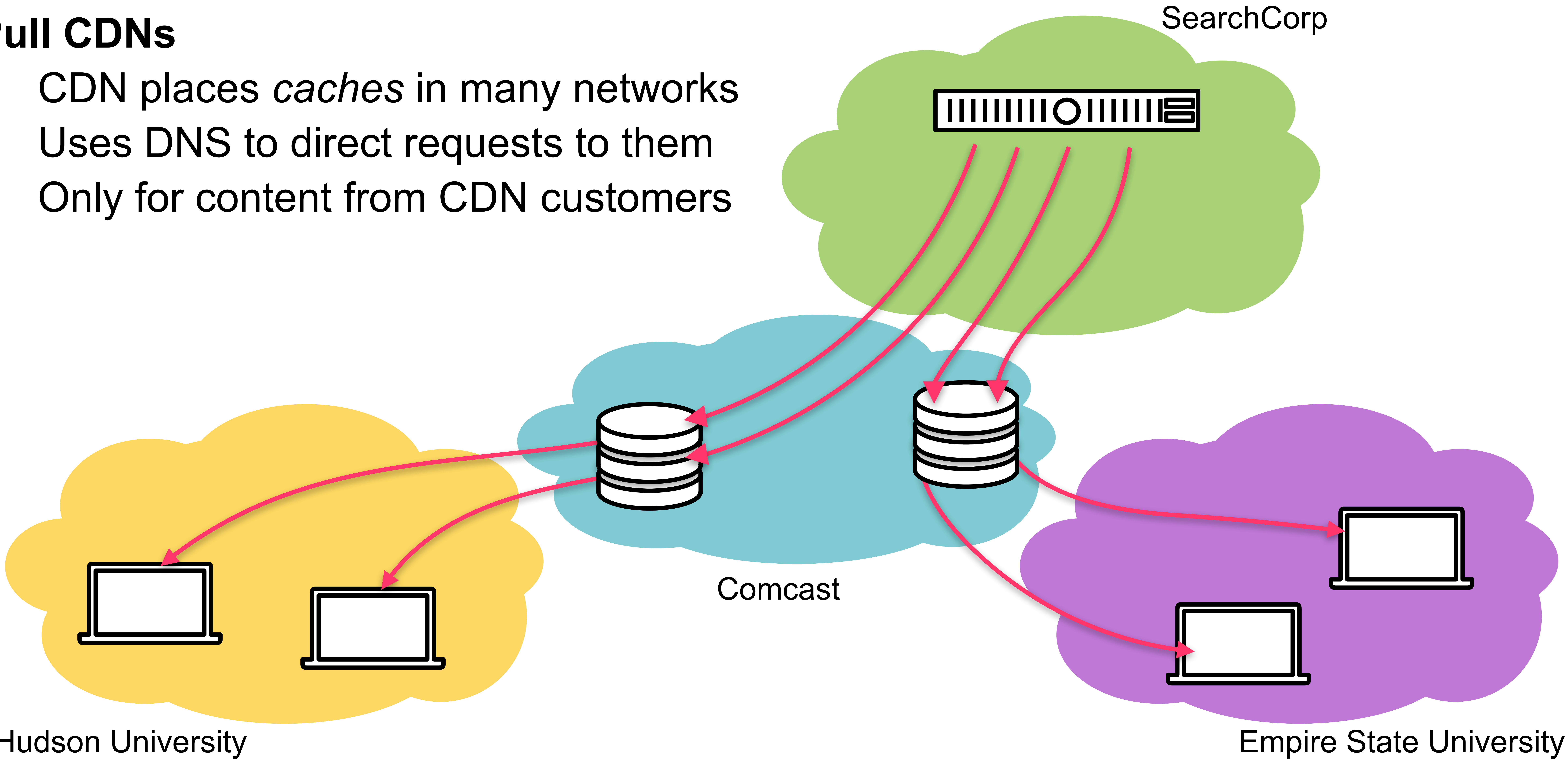
- *Push*
    - Akamai servers just act like normal servers
    - Content provider uploads content to CDN (“pushes” their content)
    - When a client requests from Akamai, just serve like any web server

- Various tradeoffs
    - Short version: pull is less work for content provider but push gives more control

CDNs “1.0”: The basic idea

Pull CDNs

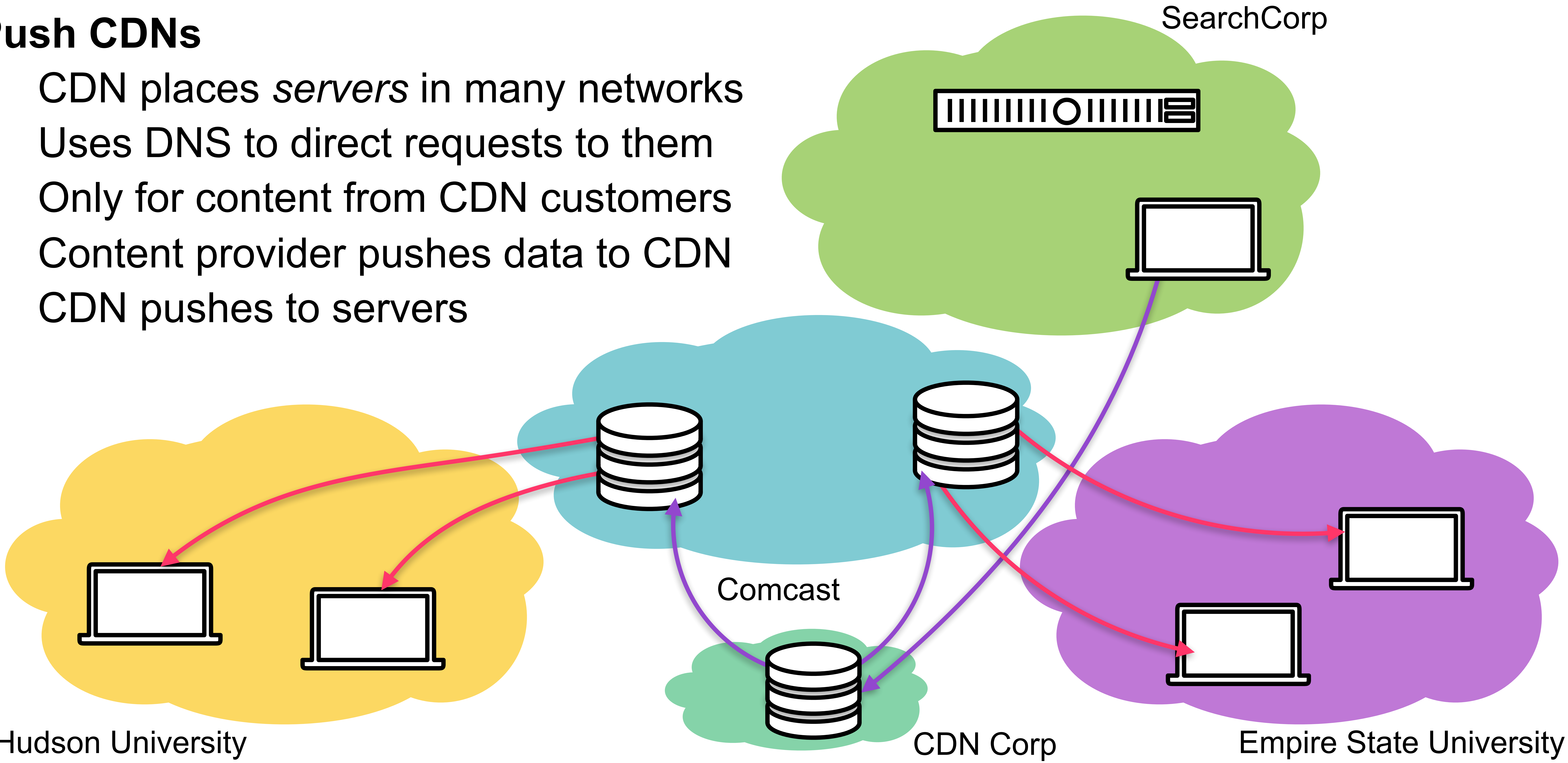
- CDN places *caches* in many networks
- Uses DNS to direct requests to them
- Only for content from CDN customers



CDNs “1.0”: The basic idea

Push CDNs

- CDN places *servers* in many networks
- Uses DNS to direct requests to them
- Only for content from CDN customers
- Content provider pushes data to CDN
- CDN pushes to servers



CDNs

- Clear to see how this works for static content (I called this “CDN 1.0”)
  - Replicate/cache on demand (pull)
  - Replicate manually by content provider (push)
  - Pick replica/cache server via clever DNS server

- What about dynamic content/features?
  - Constant evolution in this direction
  - A relatively hot commercial area!

Questions?

TCP and HTTP

TCP and HTTP

- Caching can be a big performance boost!

- But the way HTTP uses TCP also makes a big difference!

  - What am I talking about?
  - Let’s see…

TCP and HTTP: Observations

- Many web pages composed of multiple objects/resources
  - e.g., HTML file and a bunch of embedded images

- Many of the resources are pretty small — only a few packets
  - Small images
  - 304 responses (just checking if cache is up to date)
  - Etc.

  - Loading cnn.com resulted in about 40 responses that fit in a packet!

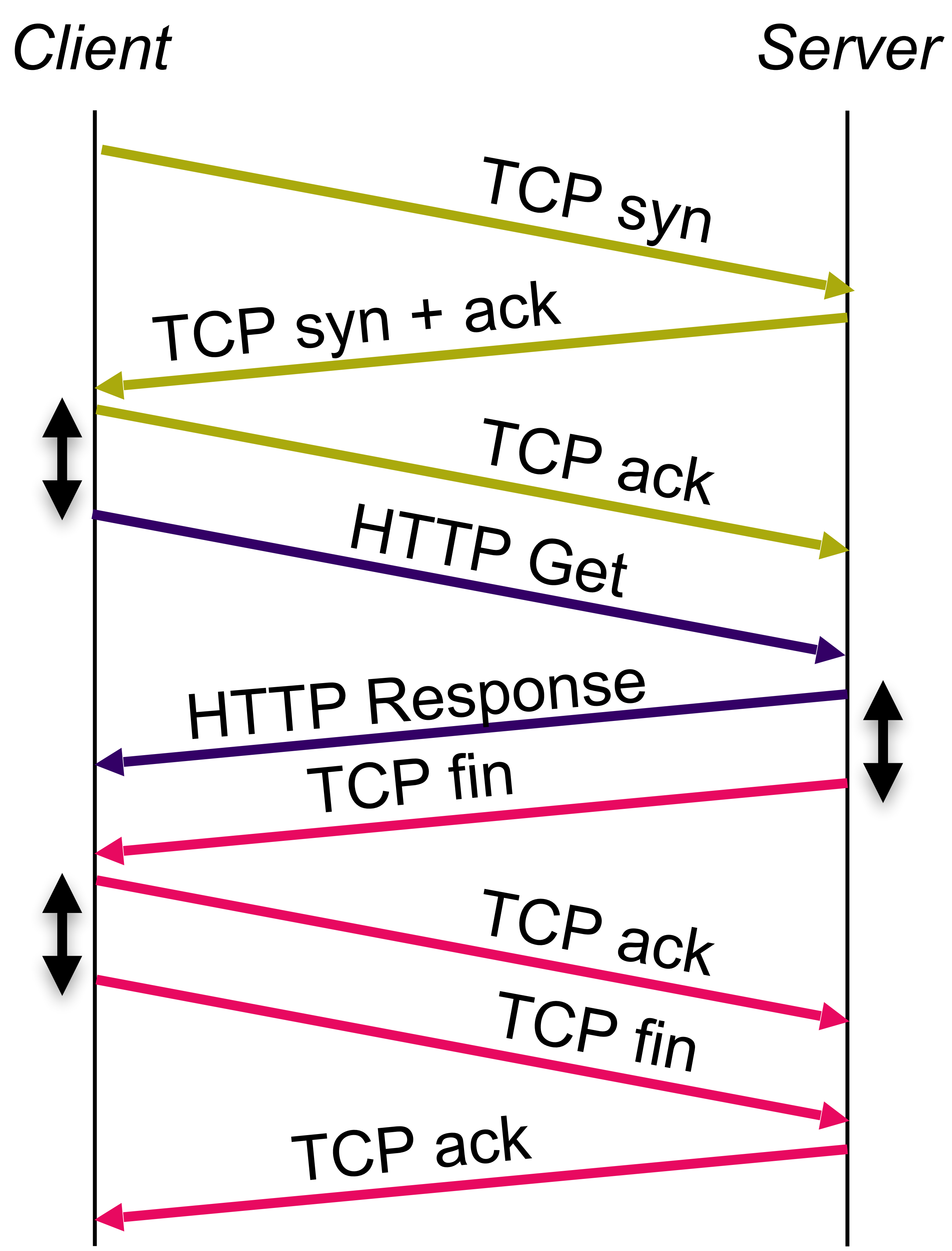
- TCP overheads fetching these can be very large!

HTTP Performance: TCP and HTTP

- Naive approach — one object at a time
  - Client creates TCP connection
  - Client sends ***request***
  - Server sends ***response***
  - Server closes connection

- Transmission delay is not the issue (<3ms at 5Mbps)
- Time dominated by RTTs (30ms RTT to Google)

- How many RTTs to download 40 small objects?
  - 2 · 40 = 80 RTTs = 2.4 seconds
  - Why not 2 RTTs per object? Why not 3?

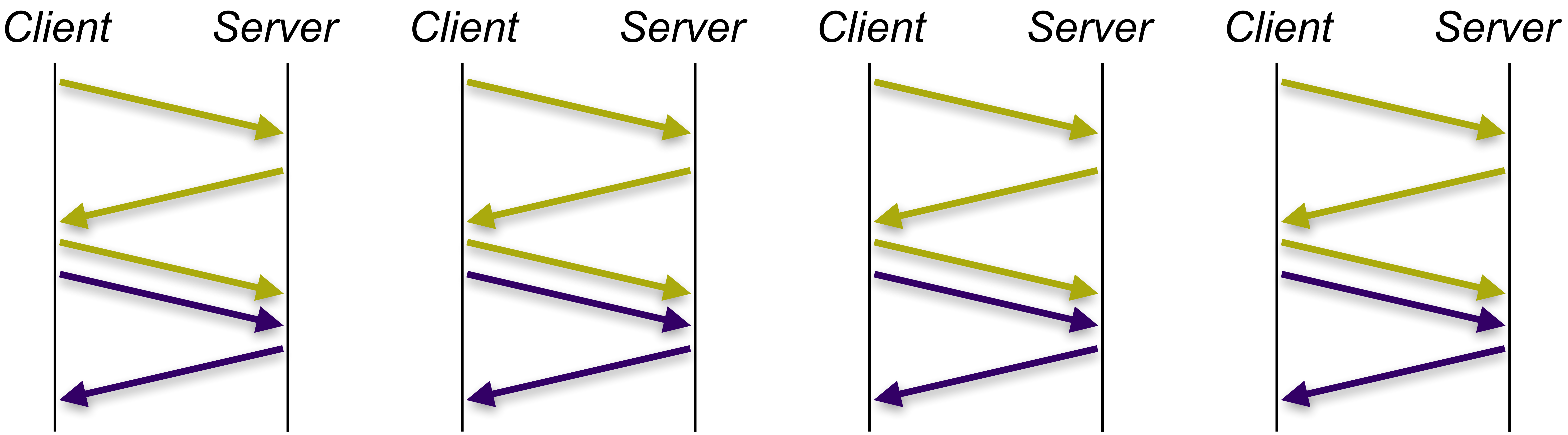


HTTP Performance: TCP and HTTP

- ***Concurrent requests***
  - Make several requests *in parallel*

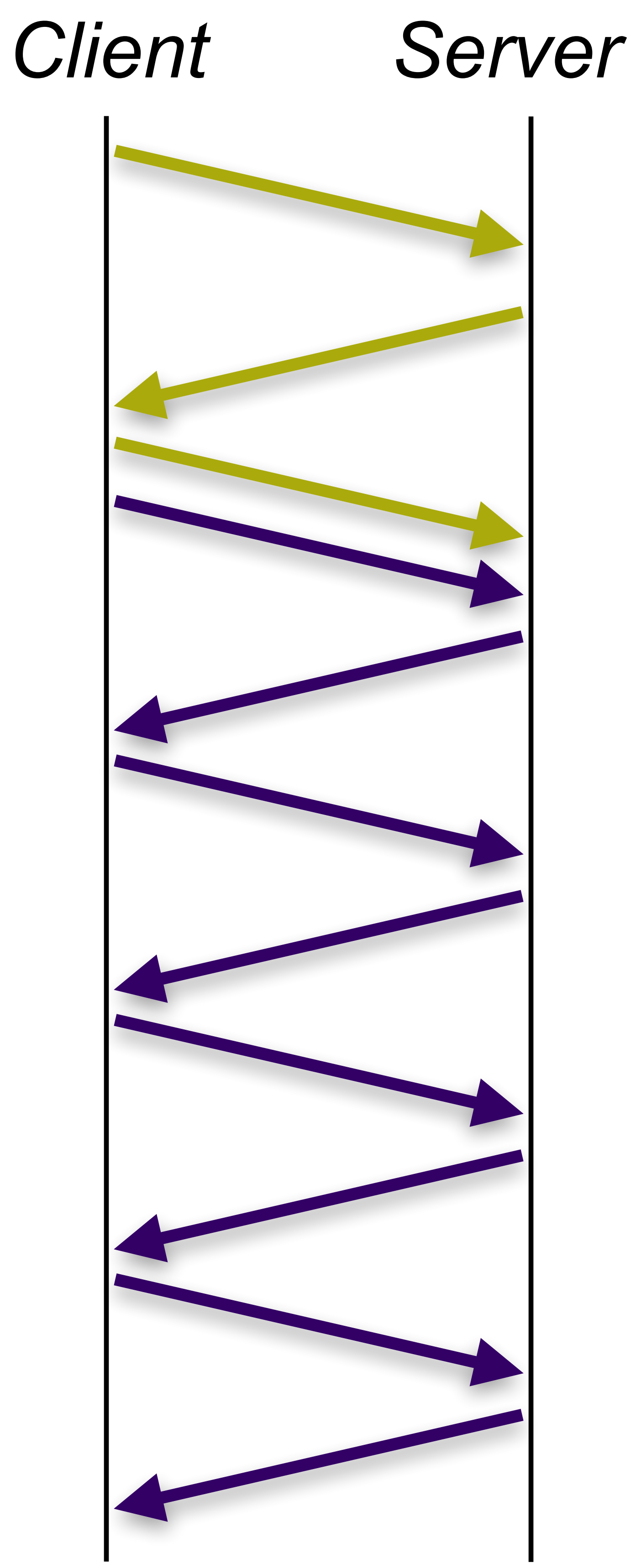
- How many RTTs to download 40 small objects, 4 at once?
  - 2 · 40/4 = 20 RTTs = 600ms (4x improvement)

- Browsers do this — limit has changed (was 6 per site for a long time?)



HTTP Performance: TCP and HTTP

- ***Persistent connections***
  - Maintain TCP connection across multiple requests
  - Client or server can tear down connection after idle period
- Performance advantages:
  - Avoid overhead of connection set-up and tear-down
  - Allow TCP congestion window to increase (next lectures)
- How many RTTs to download 40 small objects?
  - 40 + 1 = 41 RTTs = 1.23 seconds
  - With four concurrent persistent connections? 330ms
- Browsers do it — optional in HTTP 1.0; default in HTTP 1.1



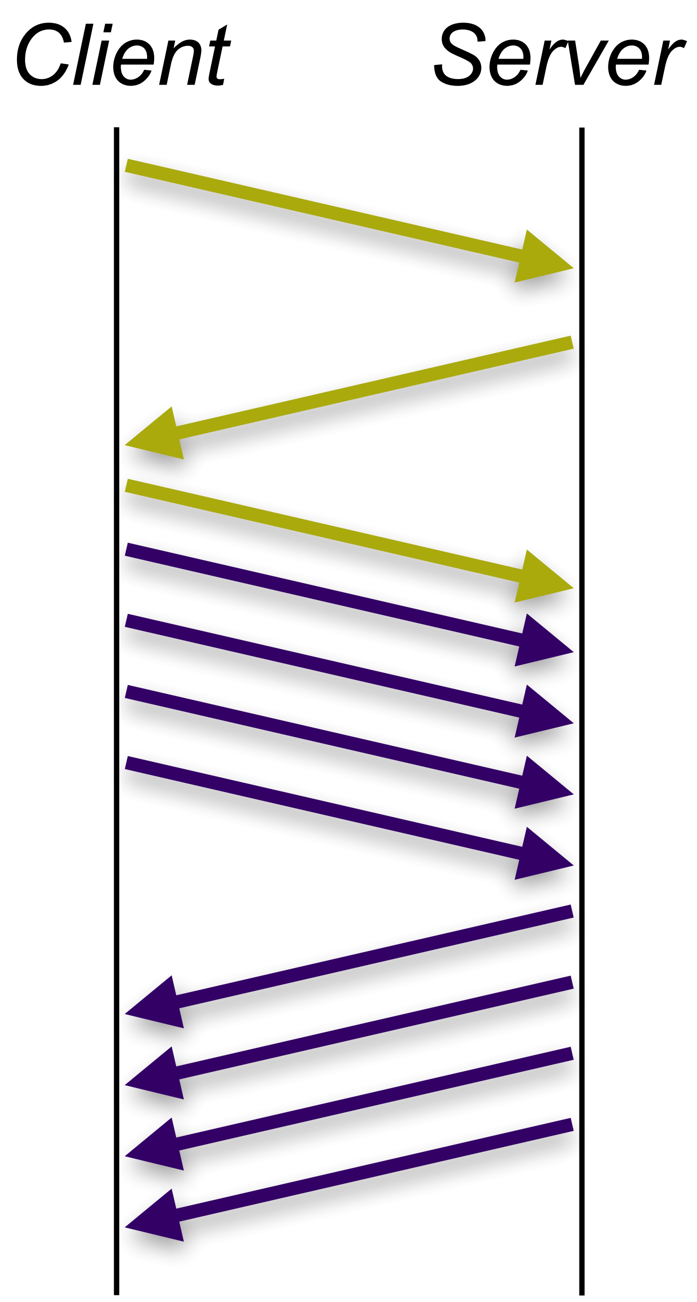
HTTP Performance: TCP and HTTP

- ***Pipelined connections***
  - Persistent connections to the next level!
  - Send multiple requests at once

- Performance advantages:
  - Reduces the RTTs
  - Multiple very small requests/responses can be coalesced into smaller number of larger packets

- How many RTTs to download 40 small objects?
  - 2! Probably dominated by transmission delay now!

- Appeared in HTTP 1.1
  - .. and promptly disabled

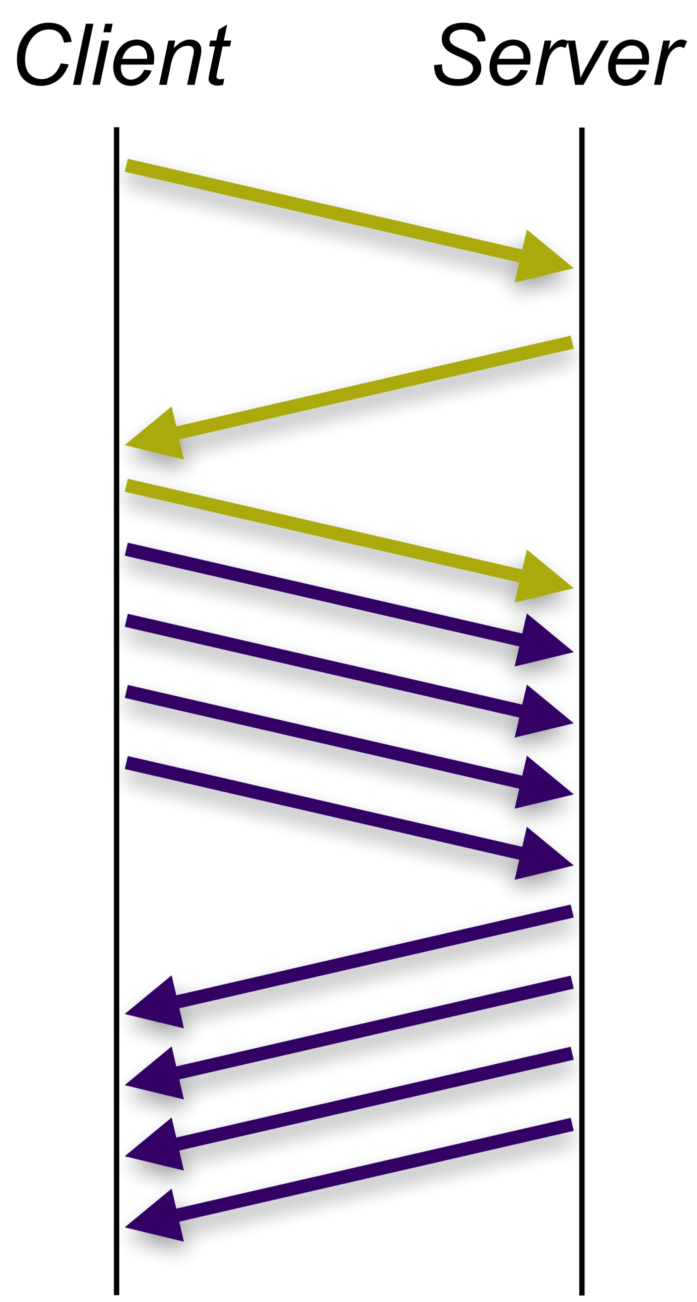


HTTP Performance: TCP and HTTP

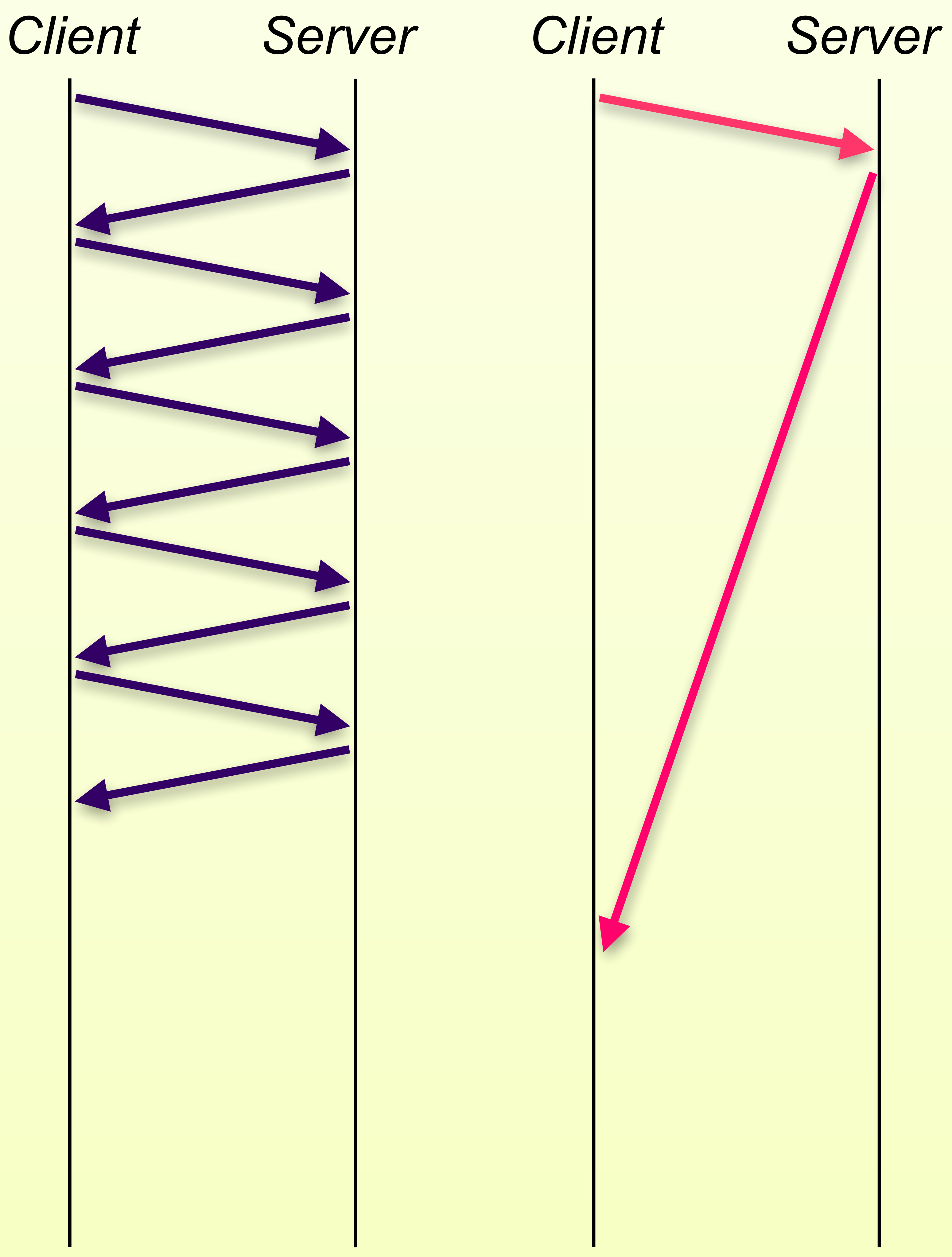
- Pipelined connections *aren’t actually used*
  - But they seemed like a huge win!
  - What happened?!
    - .. primarily two reasons

- Reason 1: Bugs!
  - One manifestation: images on page are swapped!
  - Often blamed on proxy servers
  - My guess: bad adaptation of multithreaded non-pipelined version

- Reason 2: *Head-of-line blocking*



HTTP Performance: TCP and HTTP



Two Persistent Connections

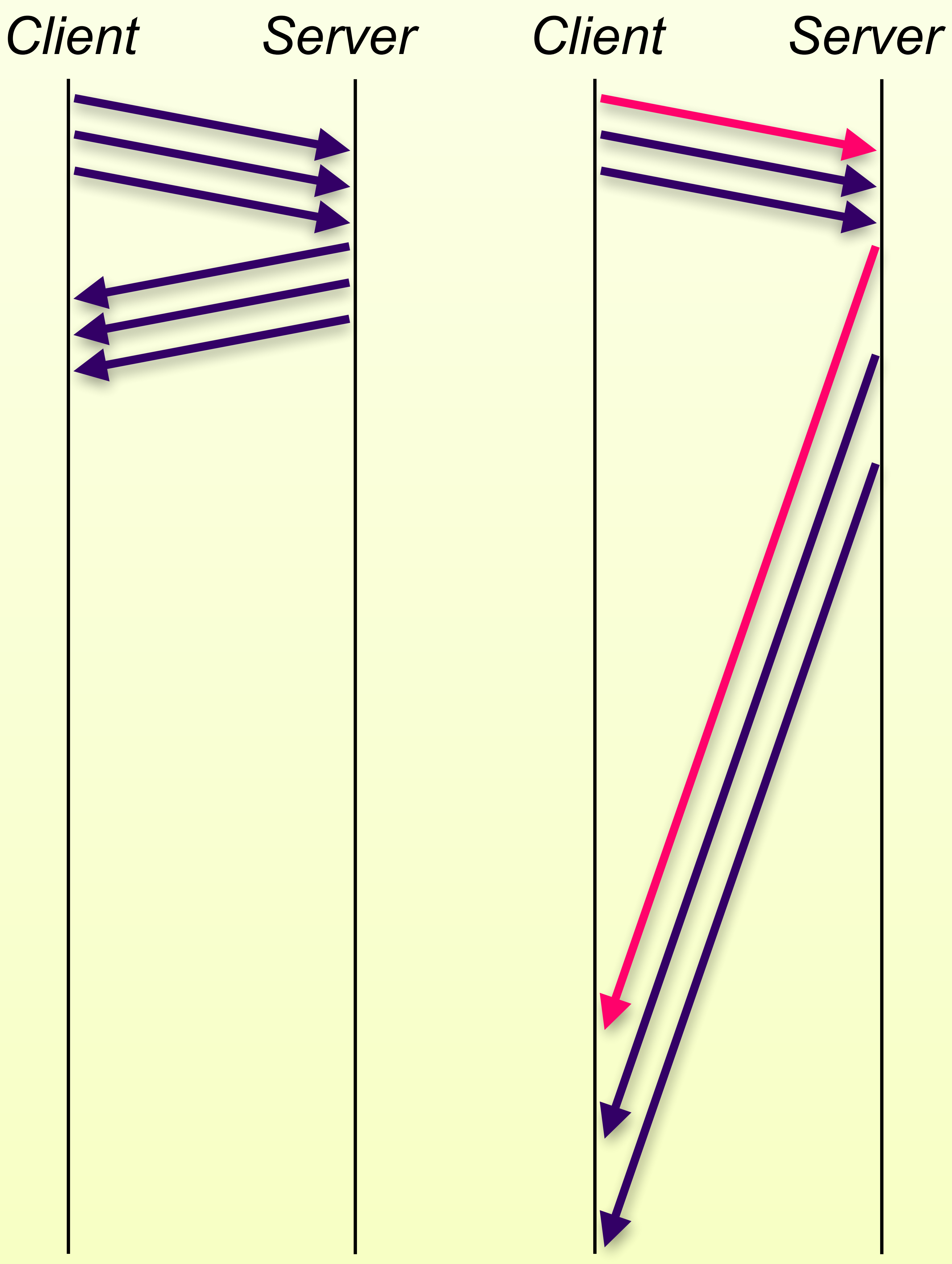
Downloading six objects

Persistent finishes 5 fairly quick, 1 slow

Pipelined finishes 3 very quick, 3 slow

…

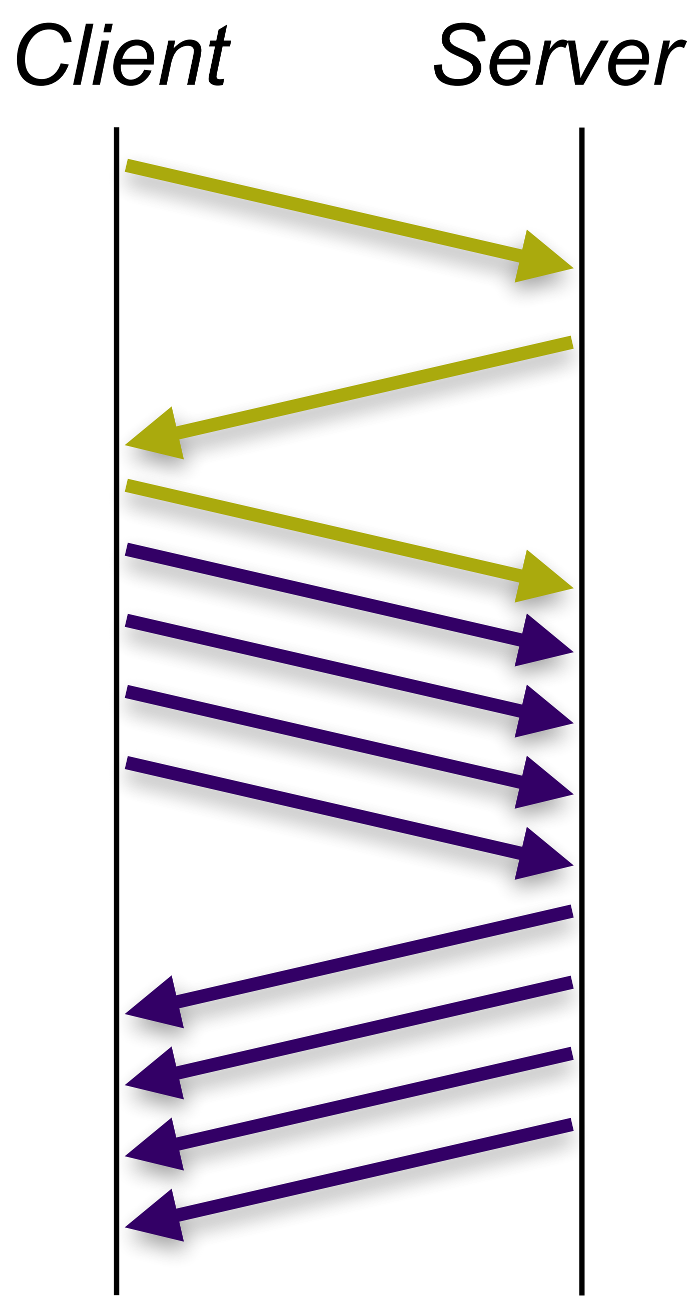
1 slow at the “head of the line” blocked 2 others



Two Pipelined Connections

HTTP Performance: TCP and HTTP

- Pipelined connections *aren't actually used*
  - But they seemed like a huge win!
  - What happened?!
    - .. primarily two reasons



- Reason 1: Bugs!
  - One manifestation: images on page are swapped!
  - Often blamed on proxy servers
  - My guess: bad adaptation of multithreaded non-pipelined version

- Reason 2: *Head-of-line blocking*
  - Small requests get stuck behind big one

- HTTP 2 replaced this with *multiplexing* with better results

HTTP Performance: TCP and HTTP

- Summing up…

- Single connection per small download can leaves performance on the floor!
  - RTTs kill your performance!

- Things you can do about it:
  - Concurrent connections
  - Persistent connections
  - Pipelined connections
  - .. and combinations thereof!

  Actually used today

  - (And multiplexed connections in HTTP 2&3)

- Why doesn't this apply to large downloads?
  - If transmission time dominates, only solution is get more bandwidth!

www.berkeley.edu (AWS)	50ms
eecs.berkeley.edu	30ms
cs.umass.edu	100ms
www.umass.edu (Akamai)	25ms
www.usp.br	300ms

Questions?

Have a good week!

Attributions

File:Mozilla dinosaur head logo.png, CC BY 3.0
https://commons.wikimedia.org/wiki/File:Mozilla_dinosaur_head_logo.png

File:Printer_dot_matrix_EPSON_VP-500.jpg, CC BY-SA 2.0
https://commons.wikimedia.org/wiki/File:Printer_dot_matrix_EPSON_VP-500.jpg

Many slides borrowed/adapted from earlier Berkeley CS168/EE122