EECS 489
Computer Networks

Winter 2024

Mosharaf Chowdhury

Material with thanks to Aditya Akella, Sugih Jamin, Philip Levis, Sylvia Ratnasamy, Peter Steenkiste, and many other colleagues.

Recap: End-to-end principle

- Functions that can be *completely* and *correctly* implemented *only* with the knowledge of application end host, should not be pushed into the network
- Fate sharing: fail together or don't fail at all
  - "it is acceptable to lose the state information associated with an entity if, at the same time, the entity itself is lost"

Agenda

- HTTP and the Web
- Improving HTTP Performance

The Web: Precursor

- 1945, Vannevar Bush, Memex
  - Concept of the web based on microfilms
- 1967, Ted Nelson, Project Xanadu
  - A world-wide publishing network to store information as connected literature
  - Coined the term “Hypertext”
- 1968, Douglas Engelbart, NLS (oN-Line System)
  - The mother of all demos

The Web: History

- World Wide Web (WWW): a distributed database of “pages” linked through Hypertext Transport Protocol (HTTP)
  - First HTTP implementation – 1990
    - Tim Berners-Lee at CERN
  - HTTP/0.9 – 1991
    - Simple GET command for the Web
  - HTTP/1.0 – 1992
    - Client/server information, simple caching

AWARD WINNER

Sir Tim Berners-Lee

ACM A. M. Turing Award (2016)

ACM Software System Award (1995)

2016 ACM A.M. Turing Award

The Web: History (cont’d)

- World Wide Web (WWW): a distributed database of “pages” linked through Hypertext Transport Protocol (HTTP)
  - HTTP/1.1 – 1996
    - Performance and security optimizations
  - HTTP/2 – 2015
    - Latency optimizations via request multiplexing over single TCP connection
    - Binary protocol instead of text
    - Server push

The Web: History (cont'd)

- World Wide Web (WWW): a distributed database of "pages" linked through Hypertext Transport Protocol (HTTP)
  - HTTP/3 – 2022
    - Built on top of QUIC, which is a user-space congestion control protocol on top of UDP
    - Solves the head-of-line (HOL) blocking problem when multiplexing over a single TCP connection

What does it consist of?

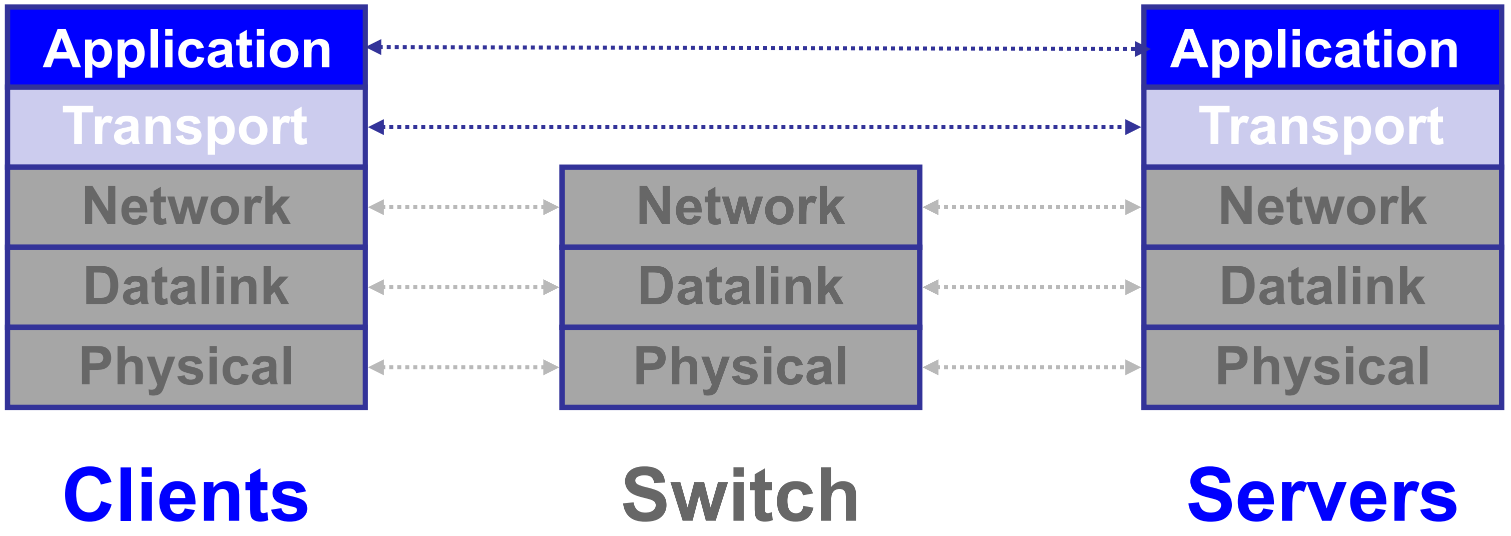
- Who uses it?
- Who provides the content?
- How do they communicate?

- How do we find the content?
- How is the content organized?
- How is it displayed?

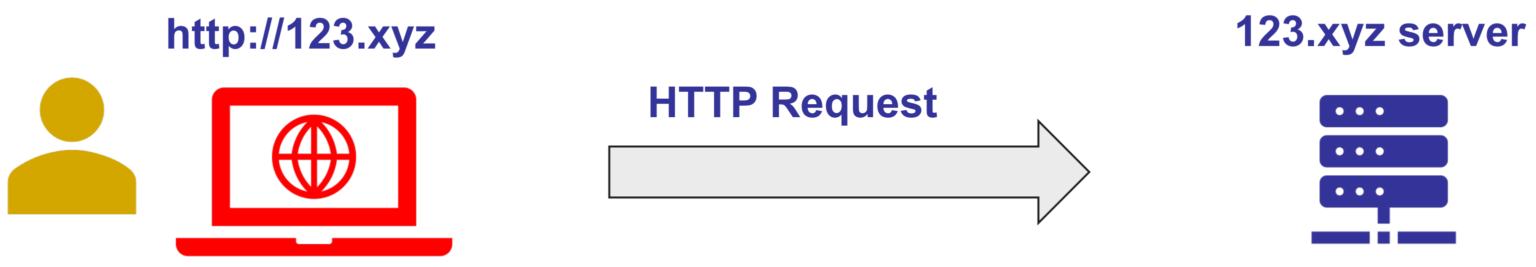
Web components

- Infrastructure:
  - Clients
  - Servers (DNS, CDN, Datacenters)
- Content:
  - URL: naming content
  - HTML: formatting content
- Protocol for exchanging information: HTTP

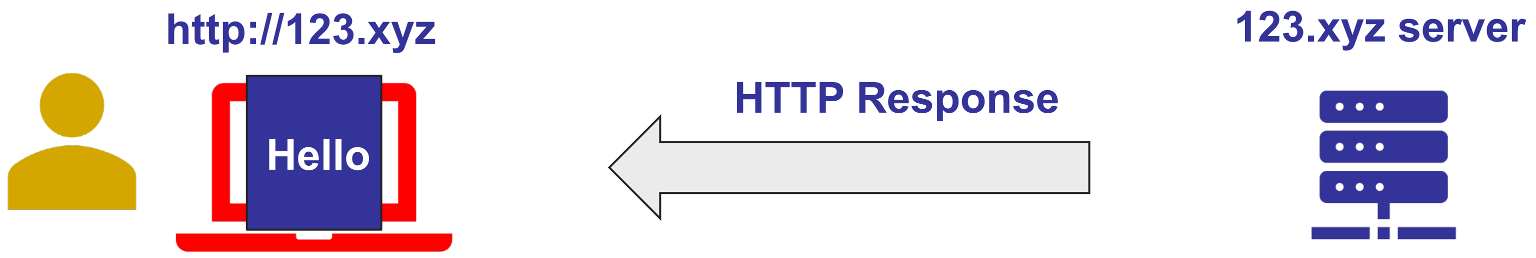
Why is there nothing about the network?



What we want



What we get



URL: Uniform Record Locator

- `protocol://host-name[:port]/directory-path/resource`

- Extend the idea of hierarchical hostnames to include anything in a file system
  - `https://github.com/mosharaf/eecs489/blob/f24/Slides/012224.pptx`
- Extend to program executions as well…
  - `https://www.google.com/search?q=eecs489`
  - Server-side processing can be included in the name

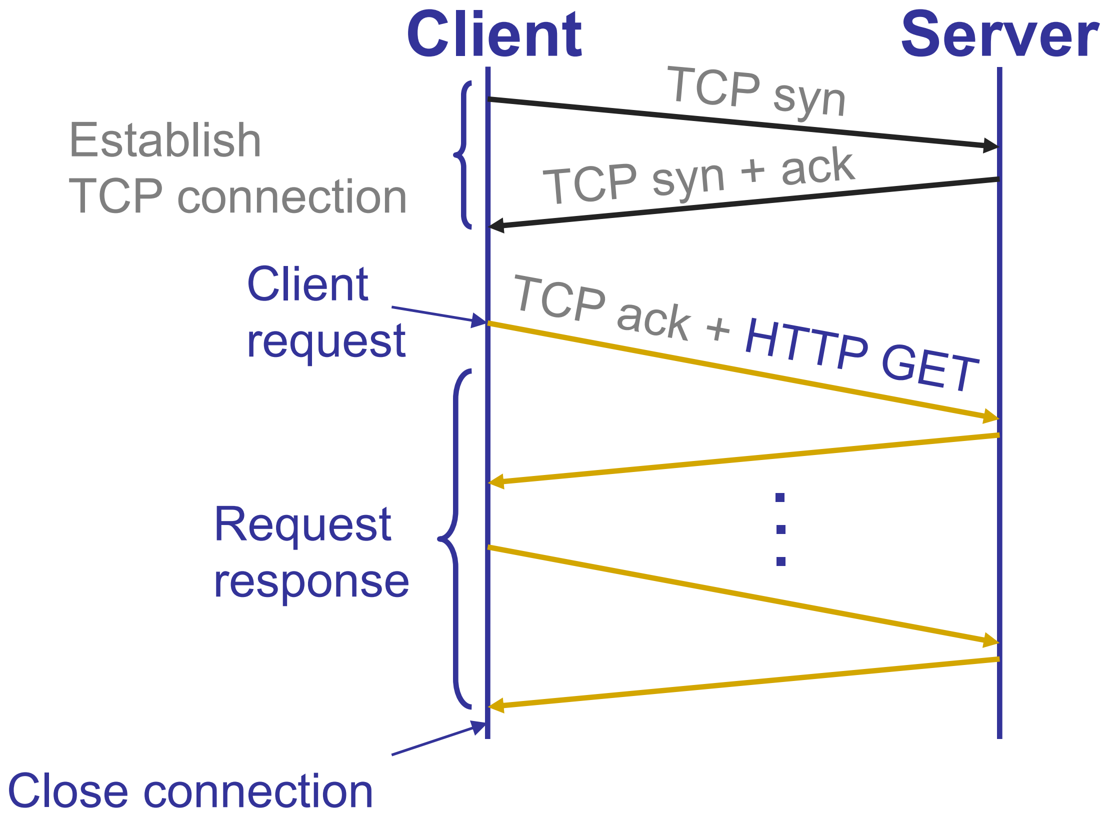
URL: Uniform Record Locator

- `protocol://host-name[:port]/directory-path/resource`
  - `protocol`: http, ftp, https, smtp, rtsp, *etc*.
  - `host-name`: DNS name, IP address
  - `port`*:* defaults to protocol's standard port
    - *E.g.,* http: 80, https: 443
  - `directory path`: hierarchical, reflecting file system
  - `resource`: Identifies the desired resource

Hyper Text Transfer Protocol (HTTP)

- Client-server architecture
  - Server is “always on” and “well known”
  - Clients initiate contact to server
- Synchronous request/reply protocol
  - Runs over TCP, Port 80
- Stateless
- ASCII format before HTTP/2 but Binary since then

Steps in canonical HTTP request/response

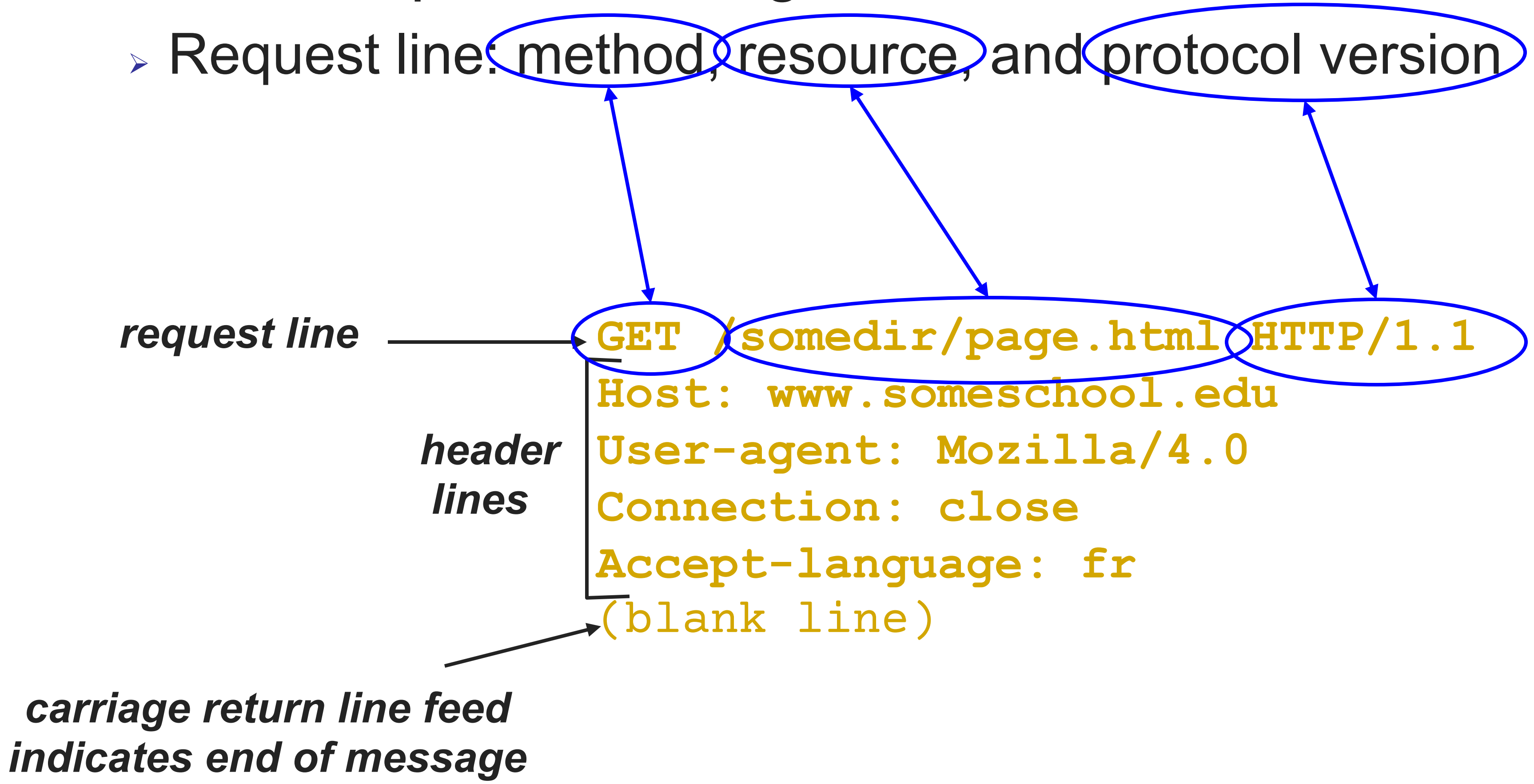


Method types (HTTP 1.1)

- GET, HEAD
- POST
  - Send information (e.g., web forms)
- PUT
  - Uploads file in entity body to path specified in URL field
- DELETE
  - Deletes file specified in the URL field

Client-to-server communication

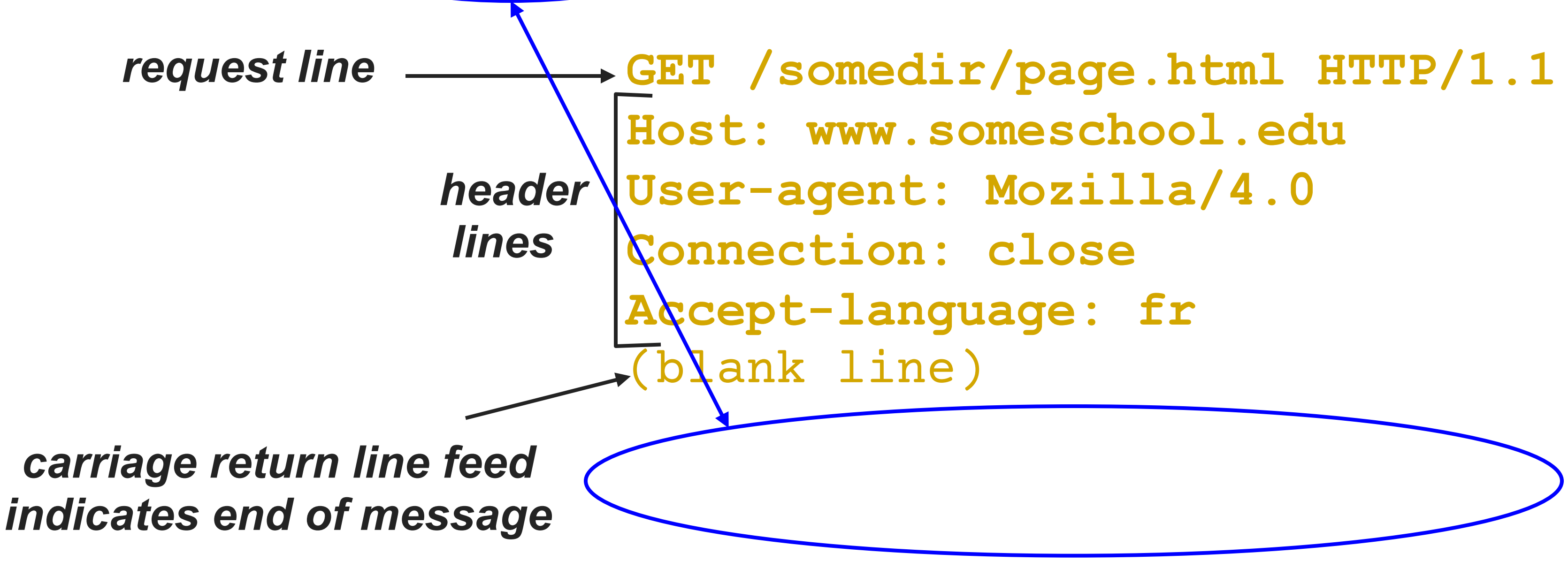
- HTTP Request Message
  - Request line: method, resource, and protocol version



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

Client-to-server communication

- HTTP Request Message
  - Request line: method, resource, and protocol version
  - Request headers: provide info or modify request
  - Body: optional data (e.g., to “POST” data to server)



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

Server-to-client communication

- HTTP Response Message
  - Status line: protocol version, status code, status phrase
  - Response headers: provide information
  - Body: optional data

status line
(protocol, status code, status phrase)

header lines

data
e.g., requested HTML file

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Jan 2017 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)
data data data data data ...
```

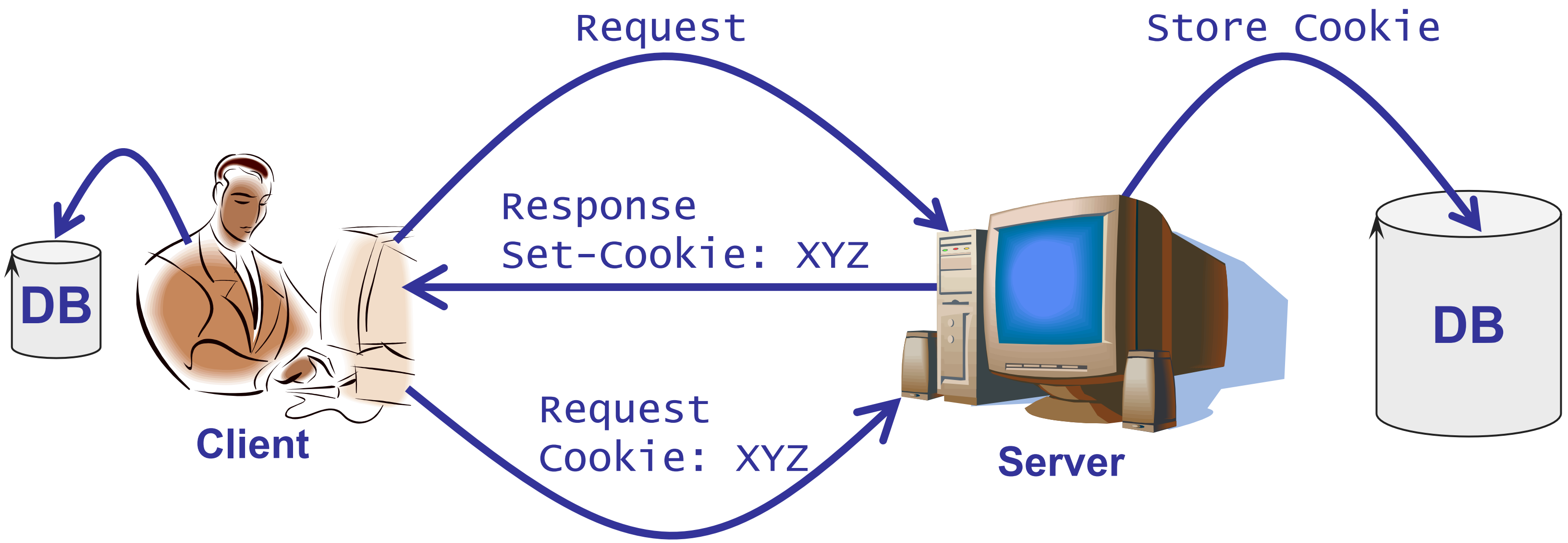
HTTP is stateless

- Each request-response treated independently
  - Servers not required to retain state
- Good: Improves scalability on the server-side
  - Failure handling is easier
  - Can handle higher rate of requests
  - Order of requests doesn't matter
- Bad: Some applications need persistent state
  - Need to uniquely identify user or store temporary info
  - e.g., Shopping cart, user profiles, usage tracking, …

How does a stateless protocol keep state?

State in a stateless protocol: Cookies

- Client-side state maintenance
  - Client stores small state on behalf of server
  - Client sends state in future requests to the server
- Can provide authentication



Beyond cookies

- Cookies provide excellent marketing opportunities and create concerns for privacy
  - Advertising companies tracks your preferences and viewing history across sites

- Many are trying to curtail the (mis)use of cookies
  - Example: Google Chrome is deprecating third-party cookies in 2024

5-MINUTE BREAK!

Announcements

- Assignment 1 is due on Jan 26, 2024

- Group formation for A2-A4 by Jan 27 2024
  - https://docs.google.com/forms/d/e/1FAIpQLSfnVI9BlnTZL07BtIb4BGhQm5V4rvQbgXAsGgsYH2TfJroHNg/viewform?usp=sf_link

Performance goals

- User
  - Fast downloads (not identical to low-latency communication!)
  - High availability
- Content provider
  - Happy users (hence, above)
  - Cost-effective infrastructure
- Network (secondary)
  - Avoid overload

Solutions?

Improve networking protocols including HTTP, TCP, etc.

- User
  - Fast downloads (not identical to low-latency communication!)
  - High availability
- Content provider
  - Happy users (hence, above)
  - Cost-effective infrastructure
- Network (secondary)
  - Avoid overload

Solutions?

Improve networking protocols including HTTP, TCP, etc.

- User
  - Fast downloads (not identical to low-latency communication!)
  - High availability
- Content provider
  - Happy users (hence, above)
  - Cost-effective infrastructure
- Network (secondary)
  - Avoid overload

Caching and replication

Solutions?

Improve networking protocols including HTTP, TCP, etc.

- User
  - Fast downloads (not identical to low-latency communication!)
  - High availability
- Content provider
  - Happy users (hence, above)
  - Cost-effective infrastructure
- Network (secondary)
  - Avoid overload

Caching and replication

Exploit economies of scale; e.g., webhosting, CDNs, datacenters

HTTP performance

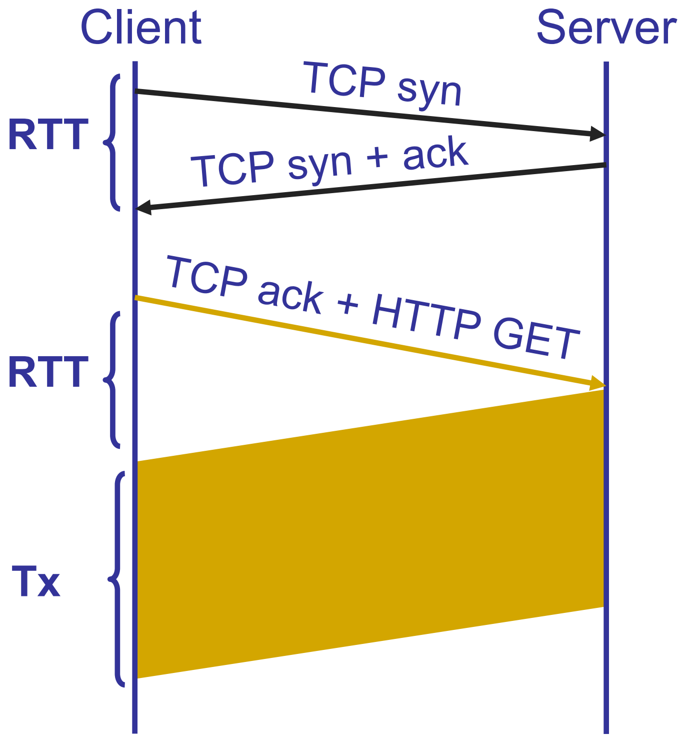
- Most Web pages have multiple objects
  - e.g., HTML file and a bunch of embedded images

- How do you retrieve those objects (naively)?
  - One item at a time

- New TCP connection per (small) object!

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 for TCP setup
  - 1 RTT for HTTP request and first few bytes
  - Transmission time
  - Total = 2RTT + Transmission Time

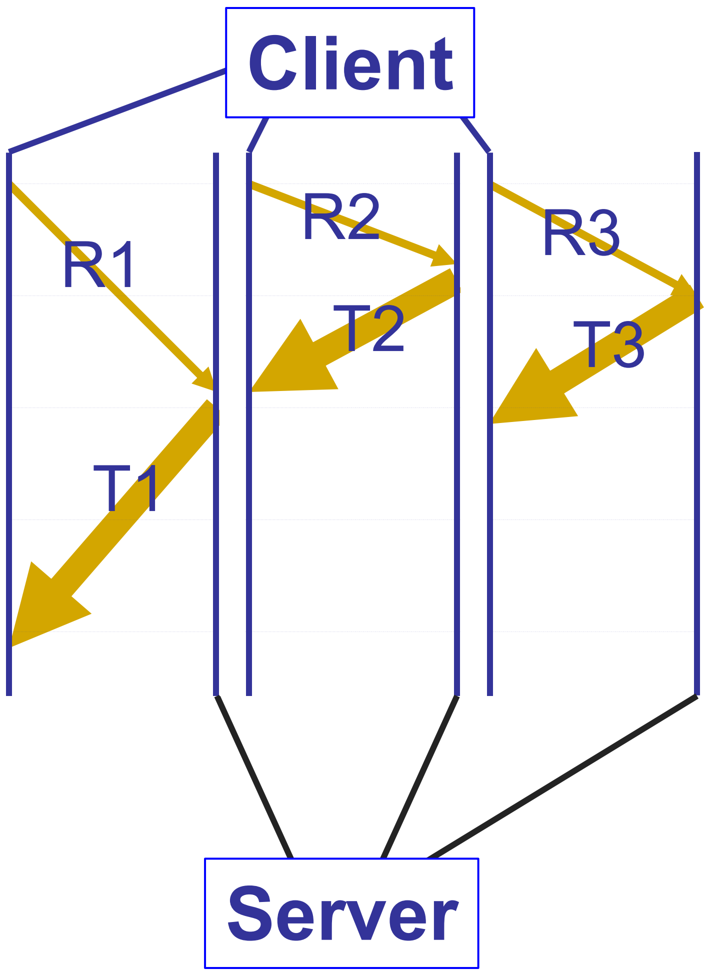


Non-persistent connections

- Default in HTTP/1.0
- 2RTT+Δ for each object in the HTML file!
  - One more 2RTT+Δ for the HTML file itself
- Doing the same thing over and over again
  - Inefficient

Concurrent requests and responses

- Use multiple connections in parallel
- Does not necessarily maintain order of responses



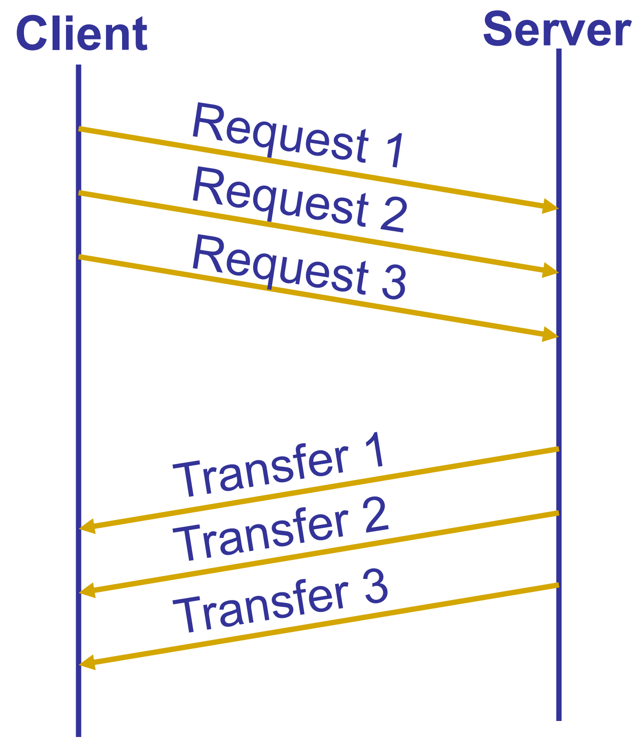
- Client = ☺
- Content provider = ☺
- Network = ☹ Why?

Persistent connections

- Maintain TCP connection across multiple requests
  - Including transfers subsequent to current page
  - Client or server can tear down connection
- Advantages
  - Avoid overhead of connection set-up and tear-down
  - Allow underlying layers (e.g., TCP) to learn about RTT and bandwidth characteristics
- Default in HTTP/1.1

Pipelined requests & responses

- Batch requests and responses to reduce the number of packets
  - Multiple requests can be contained in one TCP segment
- Data are sent in a FIFO manner
  - Can lead to head-of-line (HOL) blocking if many small responses follow a large one
  - Not supported by default by major browsers circa 2015
- Solution
  - Priority and preemption



Scorecard: Getting n small objects

- Time dominated by latency

- One-at-a-time: ~2n RTT
- m concurrent: ~2[n/m] RTT
- Persistent: ~ (n+1) RTT
- Pipelined: ~2 RTT
- Pipelined and Persistent: ~2 RTT first time; RTT later for another n from the same site

Scorecard: Getting n large objects each of size F

- Time dominated by TCP throughput B_C ($<= B_L$), where bottleneck link bandwidth is B_L
  - Assuming all TCP connections go through the same bottleneck link
- One-at-a-time: ~ nF/B_C
- m concurrent: ~ $nF/(mB'_C)$
  - Assuming each TCP connection gets the same throughput (B'_C), where $mB'_C <= B_L$
- Pipelined and/or persistent: ~ nF/B_C
  - The only thing that helps is higher throughput

Caching

- Why does caching work?
  - Exploits locality of reference
- How well does caching work?
  - Very well, up to a limit
  - Large overlap in content
  - But many unique requests
    - A universal story!
    - Effectiveness of caching grows logarithmically with size

Caching: How

- Modifier to GET requests:
  - `If-modified-since` – returns "not modified" if resource not modified since specified time

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
If-modified-since: Wed, 18 Jan 2017 10:25:50 GMT
(blank line)
```

Caching: How

- Modifier to GET requests:
  - `If-modified-since` – returns “not modified” if resource not modified since specified time
- Client specifies “`if-modified-since`” time in request
- Server compares this against “last modified” time of resource
- Server returns “Not Modified” if resource has not changed
- …. or a “OK” with the latest version otherwise

Caching: How

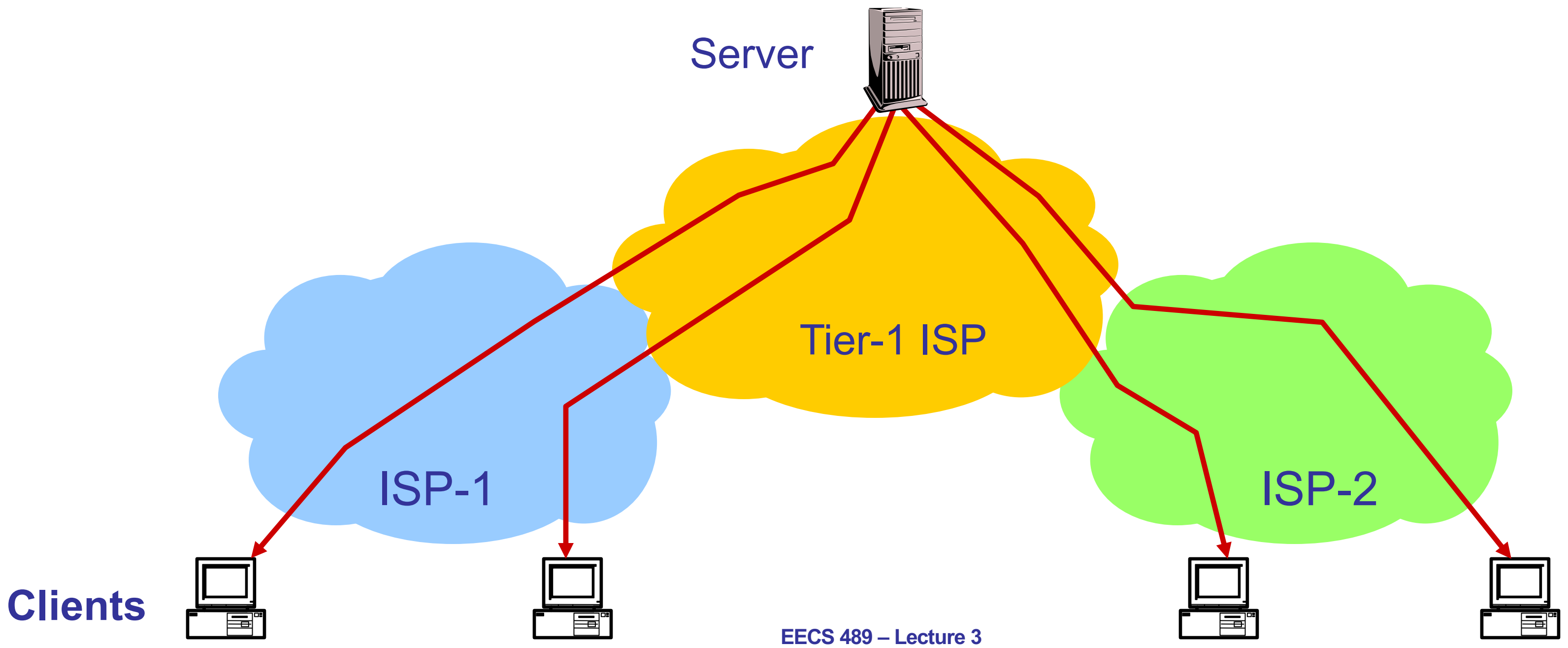
- Modifier to GET requests:
  - `If-modified-since` – returns “not modified” if resource not modified since specified time
- Response header:
  - `Expires` – how long it’s safe to cache the resource
  - `No-cache` – ignore all caches; always get resource directly from server

Caching: Where?

- Options
  - Client (browser)
  - Forward proxies
  - Reverse proxies
  - Content Distribution Network

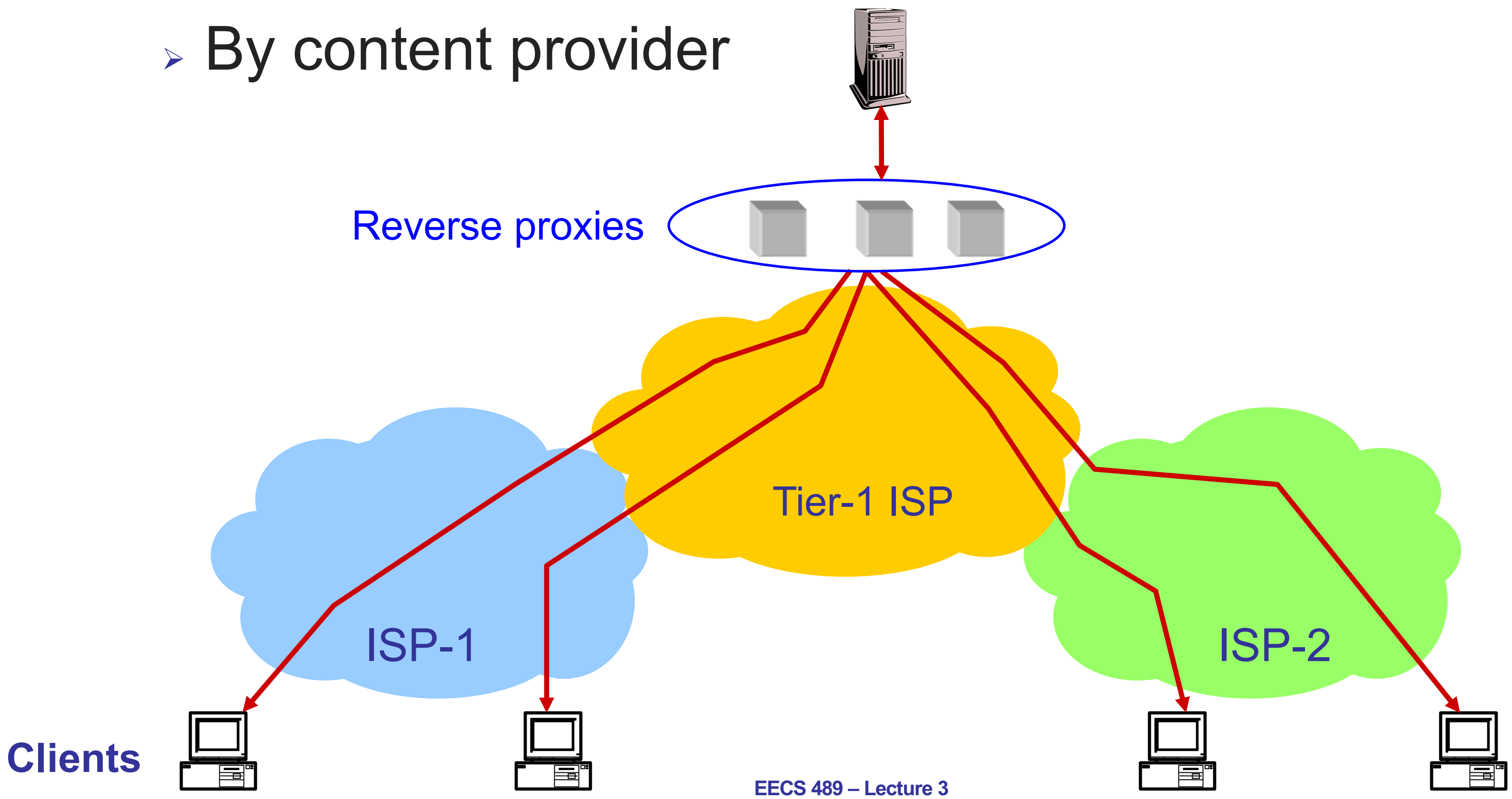
Caching: Where?

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



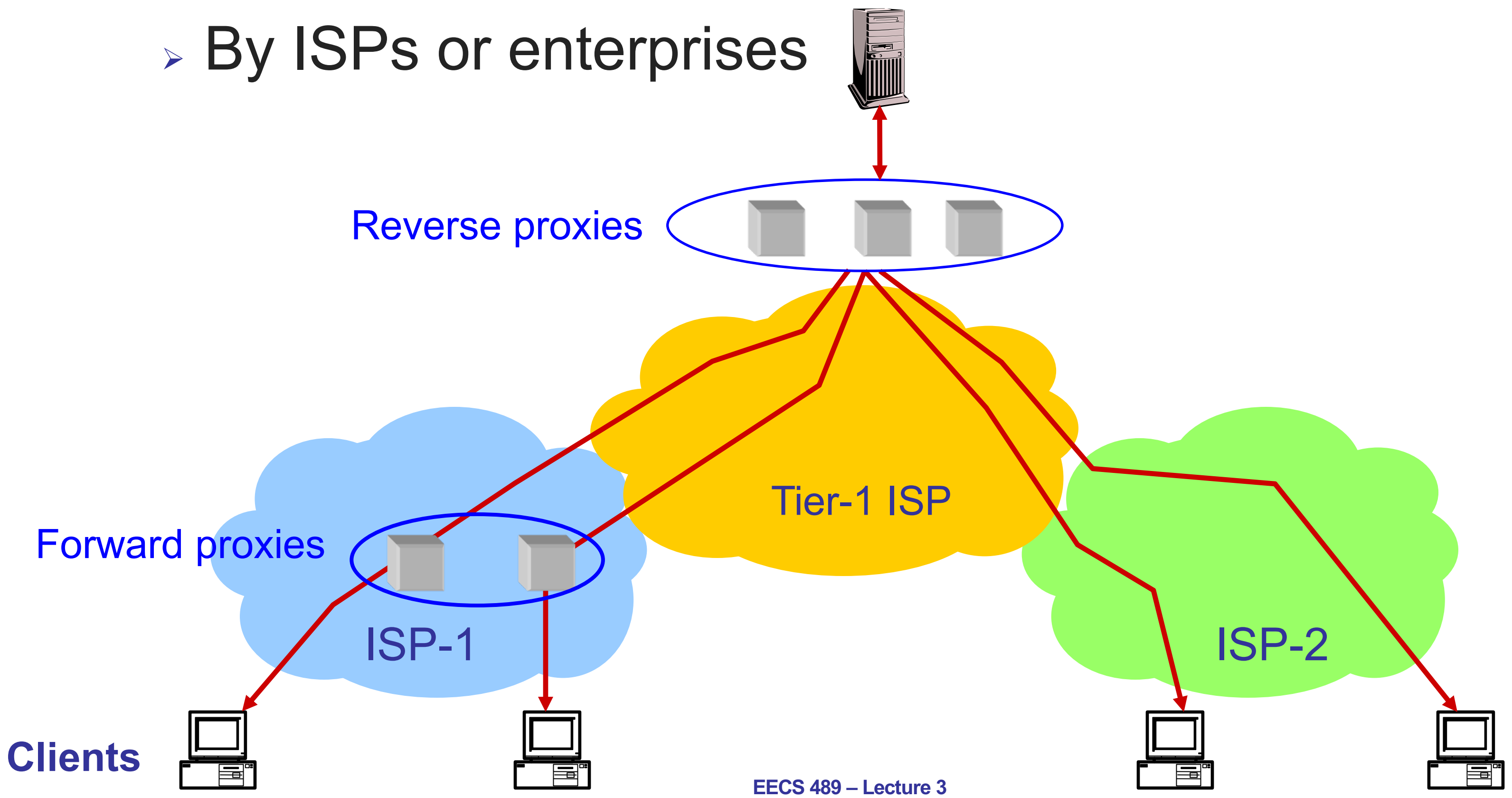
Caching with Reverse Proxies

- Cache documents close to server
  - Decrease server load
  - By content provider



Caching with Forward Proxies

- Cache documents close to clients
  - Reduce network traffic and decrease latency
  - By ISPs or enterprises



Summary

- HTTP/1.1
  - Text-based protocol
  - Replaced by binary HTTP/2 protocol, which has been replaced by HTTP/3 in 2022
- Many ways to improve performance
  - Pipelining and batching
  - Caching in proxies and CDNs
  - Datacenters