1945: Vannevar Bush

- “As we may think”, Atlantic Monthly, July, 1945.
- Describes the idea of a distributed hypertext system
- A ‘memex’ that mimics the “web of trails” in our minds

Dec 9, 1968: “The Mother of All Demos”

First demonstration of Memex-inspired system
Working prototype with hypertext, linking, use of a mouse…

Many other iterations before we got to the World Wide Web

- MINITEL in France. https://en.wikipedia.org/wiki/Minitel
- (Note that you don’t need to know any of this history for exams, this is just for the curious…)
1989: Tim Berners-Lee

1989: Tim Berners-Lee (CERN) writes internal proposal to develop a distributed hypertext system

- Connects “a web of notes with links”.
- Intended to help CERN physicists in large projects share and manage information

1990: TBL writes graphical browser for Next machines
1992-1994: NCSA/Mosaic/Netscape browser release

Lots of Traffic!

What is an Exabyte?

<table>
<thead>
<tr>
<th></th>
<th>10^3</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kilo</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Mega</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Giga</td>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>Tera</td>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>Peta</td>
<td>15</td>
<td>50</td>
</tr>
<tr>
<td>Exa</td>
<td>18</td>
<td>60</td>
</tr>
<tr>
<td>Zetta</td>
<td>21</td>
<td>70</td>
</tr>
<tr>
<td>Yotta</td>
<td>24</td>
<td>80</td>
</tr>
</tbody>
</table>

Network 1,000,000,000,000,000,000,000 Bytes
Storage 1,099,511,627,776 MByte

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
Steps in HTTP Request/Response

Client-to-Server Communication
- HTTP Request Message
  - Request line: method, resource, protocol version
  - Request headers: provide information or modify request
  - Body: optional data (e.g., to "POST" data to the server)

Server-to-Client Communication
- HTTP Response Message
  - Status line: protocol version, status code, status phrase
  - Response headers: provide information
  - Body: optional 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 to Maintain 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

Performance Issues

Performance Goals

- **User**
  - fast downloads (not identical to low-latency commn.)
  - high availability
- **Content provider**
  - happy users (hence, above)
  - cost-effective infrastructure
- **Network (secondary)**
  - avoid overload

Solutions?

- **User**
  - fast downloads (not identical to low-latency commn.)
  - high availability
- **Content provider**
  - happy users (hence, above)
  - cost-effective delivery infrastructure
- **Network (secondary)**
  - avoid overload

Improve HTTP to compensate for TCP's weak spots
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, i.e., one “GET” per TCP connection
  - Solution used in HTTP 0.9, and 1
- New TCP connection per (small) object!
  - Lots of handshakes
  - Congestion control state lost across connections

Typical Workload (Web Pages)
- Multiple (typically small) objects per page
- File sizes
  - Heavy-tailed
  - Pareto distribution for tail
  - Lognormal for body of distribution
- Embedded references
- Number of embedded objects also Pareto
  \[ \text{Pr}(X>x) = \frac{(x/xm)^{-k}}{xm^{-k}} \]
- This plays havoc with performance. Why?
- Solutions?

Solutions?
- User
  - fast downloads (not identical to low-latency communication!)
  - high availability
- Content provider
  - happy users (hence, above)
  - cost-effective delivery infrastructure
- Network (secondary)
  - avoid overload

Caching and Replication
- Improve HTTP to compensate for TCP’s weak spots

Exploit economies of scale (Webhosting, CDNs, datacenters)
Improving HTTP Performance:

Persistent Connections

- Maintain TCP connection across multiple requests
  - Including transfers subsequent to current page
  - Client or server can tear down connection
- Performance advantages:
  - Avoid overhead of connection set-up and tear-down
  - Allow TCP to learn more accurate RTT estimate
  - Allow TCP congestion window to increase
    - i.e., leverage previously discovered bandwidth
- Drawback? Head of line blocking
  - A “slow object” blocks retrieval of all later requests, including “fast” objects
- Default in HTTP/1.1

Improving HTTP Performance:
Pipelined Requests & Responses

- Batch requests and responses to reduce the number of packets
- Multiple requests can be contained in one TCP segment
- Head of line blocking issues remains: a delay in Transfer 2 delays all later transfers

Improving HTTP Performance:
Concurrent Requests & Responses

- Use multiple connections in parallel
  - Speeds up retrieval by ~m
  - Does not necessarily maintain order of responses
  - Partially deals with HOL blocking
    - Client = 
    - Content provider = 
    - Network = Why?

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/Persistent: ~2 RTT first time, RTT later
Scorecard: Getting $n$ Large Objects

Time dominated by bandwidth

- One-at-a-time: $\sim nF/B$
- M concurrent: $\sim \frac{n}{m} F/B$
  - assuming shared with large population of users
  - and each TCP connection gets the same bandwidth
- Pipelined and/or persistent: $\sim nF/B$
  - The only thing that helps is getting more bandwidth.

Improving HTTP Performance: 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
- Trend: increase in dynamic content
  - E.g., customizing of web pages
  - Reduces benefits of caching
  - Some exceptions, e.g., video

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

Everywhere!
- Client
- Forward proxies
- Reverse proxies
- Content Distribution Network

Clients keep a local cache of recently accessed objects
- Clients often have a small number of web pages they access frequently
- Leads to reuse of logos, old content, java scripts, ...
- Cheap: no additional infrastructure needed
- But caching closer to server can lead to higher hit rates!
Improving HTTP Performance:
Caching with Reverse Proxies

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

Improving HTTP Performance:
Caching with Forward Proxies

- Cache documents close to **clients**
  - decrease latency
- Typically done by ISPs or enterprises
  - reduce provider traffic load

Improving HTTP Performance:
Caching: How to Avoid Stale Content

- Modifier to GET requests:
  - **If-modified-since** – returns “not modified” if resource not modified since specified time

```
GET /~ee122/fa13/ HTTP/1.1
Host: inst.eecs.berkeley.edu
User-Agent: Mozilla/4.03
If-modified-since: Sun, 27 Oct 2013 22:25:50 GMT
<CRLF>
```

- 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

Improving HTTP Performance:
Caching: Helping the Cache

- 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
Improving HTTP Performance:

**Replication**

- Replicate popular Web site across many machines
  - Spreads load on servers
  - Places content closer to clients
    - Helps when content isn’t cacheable
- Problem: Want to direct client to particular replica
  - Balance load across server replicas
  - Pair clients with nearby servers
- Common solution:
  - DNS returns different addresses based on client’s geo location, server load, etc.

**Content Distribution Networks**

- Caching and replication as a service
- Large-scale distributed storage infrastructure (usually) administered by one entity
  - e.g., Akamai has servers in 20,000+ locations
- Combination of (pull) caching and (push) replication
  - **Pull**: Direct result of clients’ requests
  - **Push**: Expectation of high access rate
  - Also do some processing
    - Handle *dynamic* web pages
    - Transcoding

Recall:

**CDN Example – Akamai**

- Akamai creates new domain names for each client
  - e.g., a128.g.akamai.net for cnn.com
- The CDN’s DNS servers are authoritative for the new domains
- The client content provider modifies its content so that embedded URLs reference the new domains.
  - “Akamaize” content
    - e.g.: http://www.cnn.com/image-of-the-day.gif becomes http://a128.g.akamai.net/image-of-the-day.gif
- Requests for embedded objects are sent to CDN’s infrastructure...

Cost-Effective Content Delivery

- General theme: multiple sites hosted on shared physical infrastructure
  - efficiency of statistical multiplexing
  - economies of scale (volume pricing, etc.)
  - amortization of human operator costs
- Examples:
  - Web hosting companies
  - CDNs
  - Cloud infrastructure
Performance Issues

Are We Done Yet?

Some Challenges with HTTP 1.1

- Head of line blocking: “slow” objects delay later requests
- E.g., objects from remote storage versus objects in local memory
- Browsers open multiple TCP connections to achieve parallel transfers
- Increases throughput and reduces impact HOL blocking
- Increases load on servers and network
- HTTP headers are big
- Cost higher for small objects
- Objects have dependencies, different priorities
- Javascript versus images
- Extra RTTs for “dependent” objects

Example of Head of Line Blocking

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html

HTTP 2.0 to the Rescue

- Responses are multiplexed over single TCP connection
- Server can send response data whenever it is ready
- “Fast” objects can bypass slow objects – avoids HOL blocking
- Fewer handshakes, more traffic (help cong. ctrl., e.g., drop tail)
- Multiplexing uses prioritized flow controlled streams
- Urgent responses can bypass non-critical responses
- ≅ multiple parallel prioritized TCP connections, but over one TCP connection
- HTTP headers are compressed
- A PUSH features allows server to push embedded objects to the client without waiting for a client request
- Avoids an RTT
- Default is to use TLS – fall back on 1.1 otherwise

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html
HTTP/2 Multi-Streams Multiplexing

- Traffic sent as frames over prioritized streams
- Frames types: headers, data, settings, window updates and push promise
- Sender sends high priority frames first
- Frames are pulled from a per-stream queue when TCP is ready to accept more data
- Reduces queueing delay
- Each stream is flow controlled
- Receiver opens window faster for high priority streams
- Replicates TCP function but at finer granularity
- Clearly adds complexity to HTTP library

HTTP/2 Binary Framing

- Bit 0...7: Length
- Bit 8...15: Flags
- Bit 16...23: Stream Identifier
- Bit 24...31: Frame Payload

HTTP/2 Server Push

- Server can “push” objects that it knows (or thinks) the client will need
- Avoids delay of having client parse the page and requesting the objects (> RTT)
- But what happens if object is in the client cache – Oops!
- Server sends PUSH_PROMISE before the PUSH
- Client can cancel/abort the PUSH
- How does server know what to PUSH?
- Very difficult problem with dynamic content
- Javascripts can rewrite web page – changes URLs
- Also: benefits limited to objects from the origin server

HTTP 2 PUSH Features

- Server sends PUSH_PROMISE before the PUSH
- Client can cancel/abort the PUSH
- How does server know what to PUSH?
- Very difficult problem with dynamic content
- Javascripts can rewrite web page – changes URLs
- Also: benefits limited to objects from the origin server
Next Tuesday: Midterm Review

Use Piazza to request topics
Use midterm_review folder