HTTP
HTTP: HyperText Transfer Protocol

• Basis for fetching Web pages
Sir Tim Berners-Lee (1955–)

• Inventor of the Web
  • Dominant Internet app since mid 90s
  • He now directs the W3C

• Developed Web at CERN in ‘89
  • Browser, server and first HTTP
  • Popularized via Mosaic (‘93), Netscape
  • First WWW conference in ’94 ...

Source: By Paul Clarke, CC-BY-2.0, via Wikimedia Commons
Web Context
Web Protocol Context

• HTTP is a request/response protocol
  • Runs on TCP, typically port 80
  • Part of browser/server app
Fetching a Web page with HTTP

• Start with the page URL (Uniform Resource Locator): http://en.wikipedia.org/wiki/Vegemite

• Steps:
  1. Resolve the server to IP address (DNS)
  2. Set up TCP connection to the server
  3. Send HTTP request for the page
  4. Await HTTP response for the page
  5. Execute and fetch embedded resources, render
  6. Clean up any idle TCP connections
HTML

- Hypertext Markup Language (HTML)
  - Uses Extensible Markup Language (XML) to build a markup language for web content
  - Key innovation was the “hyperlink”, an element linking to other HTML elements using URLs
  - Also includes Cascading Style Sheets (CSS) for maintaining look-and-feel across a domain
  - “Browser wars” over specific standards
DOM (Document Object Model)

• Base primitive for HTML browsers
• Use HTML to create a tree of elements

• Embedded Javascript modifies the DOM based on:
  • User actions
  • Asynchronous Javascript
  • Other server-side actions
Lets explore a page

• [https://www.cs.washington.edu/](https://www.cs.washington.edu/)
Static vs Dynamic Web pages

• Static: Just static files, e.g., image
• Dynamic: Page content based on some computation
  • Javascript on client, PHP on server, or both
HTTP Protocol

- Originally simple; many options added over time
  - Text-based commands, headers

- Try it yourself: As a “browser” fetching a URL
  - Run “telnet <server name> 80”
  - Enter “GET /index.html HTTP/1.0”
  - Server will return HTTP response
## HTTP Protocol (2)

### Commands used in the request

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Read a Web page</td>
</tr>
<tr>
<td>HEAD</td>
<td>Read a Web page's header</td>
</tr>
<tr>
<td>POST</td>
<td>Append to a Web page</td>
</tr>
<tr>
<td>PUT</td>
<td>Store a Web page</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove the Web page</td>
</tr>
<tr>
<td>TRACE</td>
<td>Echo the incoming request</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Connect through a proxy</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Query options for a page</td>
</tr>
</tbody>
</table>

- **Fetch page**
- **Upload data**

Basically defunct
## HTTP Protocol (3)

- Codes returned with the response

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xx</td>
<td>Information</td>
<td>100 = server agrees to handle client's request</td>
</tr>
<tr>
<td>2xx</td>
<td>Success</td>
<td>200 = request succeeded; 204 = no content present</td>
</tr>
<tr>
<td>3xx</td>
<td>Redirection</td>
<td>301 = page moved; 304 = cached page still valid</td>
</tr>
<tr>
<td>4xx</td>
<td>Client error</td>
<td>403 = forbidden page; 404 = page not found</td>
</tr>
<tr>
<td>5xx</td>
<td>Server error</td>
<td>500 = internal server error error; 503 = try again later</td>
</tr>
</tbody>
</table>
Representational State Transfer (REST)

• Using HTTP for general network services

• RESTful APIs: An ideal for design of HTTP-based APIs

• Core tenets:
  • Stateless (no state on server)
  • Cacheable (individual URLs can be cached)
  • Layered (no visibility under REST hood)
Performance
PLT (Page Load Time)

• PLT is a key measure of web performance
  • From click until user sees page
  • Small increases in PLT decrease sales

• PLT depends on many factors
  • Structure of page/content
  • HTTP (and TCP!) protocol
  • Network RTT and bandwidth
Early Performance

• HTTP/1.0 used one TCP connection per web resource
  • Made HTTP very easy to build
  • But gave fairly poor PLT...
Reasons for Poor PLT

• Sequential request/responses, even when to different servers
• Multiple TCP connection setups to the same server
  • Multiple TCP slow-start phases
• Network is not used effectively
  • Worse with many small resources
Ways to Improve PLT

1. Reduce content size for transfer
   • Smaller images, gzip

2. Make better use of the network
   • Next

3. Avoid fetching same content
   • Caching and proxies [later]

4. Move content closer to client
   • CDNs [later later]
Better Network Use: Parallel Connections

• Browser runs multiple (say, 8) parallel HTTP instances
  • Server is unchanged; already handled concurrent requests for many clients

• How does this help?
  • Single HTTP wasn’t using network much ...
  • So parallel connections aren’t slowed much
  • Pulls in completion time of last fetch
Better Network Use: Persistent Connections

• Parallel connections compete with each other for network resources
  • 1 parallel client ≈ 8 sequential clients?
  • Exacerbates network bursts, and loss

• Persistent connections
  • Make 1 TCP connection to 1 server
  • Use it for multiple HTTP requests
Persistent Connections

One request per connection

Persistent connections

Persistent connections + pipelining
Persistent Connections (2)

• Widely used as part of HTTP/1.1
  • Supports optional pipelining
  • PLT benefits depending on page structure, but easy on network

But we didn’t stop there ....
Web Caching and Proxies
Web Caching

• Users often revisit web pages
  • Big win from reusing local copy, aka, caching

• Key question:
  • When is it OK to reuse local copy?
Locally Determine Validity of Cached Content

• Based on expiry information such as “Expires” header

• Or a heuristic (cacheable, fresh, not modified recently)

• Content is then available right away
Use Server to Validate Cached Content

• Based on “Last-Modified” header from server

• Or based on “Etag” header from server

• Content is available after 1 RTT (if connection open)
Web Caching: Putting it together

1: Request
2: Check expiry
3: Conditional GET
4a: Not modified
4b: Response
5: Response

Web browser
Cache
Web server
Program
Web Proxies

• Place intermediary between clients and servers

• Benefits for clients include a shared cache
  • Limited by secure / dynamic content
  • Also limited by “long tail”

• Organizational access policies too!
Web Proxies in Action

- Clients contact proxy; proxy contacts server
CDNs
Content Delivery Networks

• As the Web took off, traffic volumes grew and grew.
  1. Concentrated load on popular servers
  2. Led to congested networks
  3. Gave a poor user experience

• Idea:
  • Place popular content near clients
  • Helps with all three issues above
Before CDNs

• Sending content from the source server to 4 users takes $4 \times 3 = 12$ “network hops” in the example
After CDNs

• Sending content via replicas takes only $4 + 2 = 6$ “network hops”
After CDNs (2)

• Benefits assuming popular content:
  • Reduces source server, network load
  • Improves user experience
Popularity of Content

- Zipf’s Law: few popular items, many unpopular ones; both matter

Zipf popularity
(kth item is 1/k)

George Zipf (1902-1950)

How to place content near clients?

• Idea 1: Use browser and proxy caches
  • Helps, but limited to one client or clients in one organization
  • Want to place replicas across the Internet for use by all nearby clients

• Idea 2: Map clients to a nearby replica
  • Done via clever use of DNS
Content Delivery Network
Content Delivery Network (2)

• DNS gives different answers to clients
  • Tell each client the nearest replica (map client IP)
Business Model

• Clever model pioneered by Akamai
  • Placing site replica at an ISP is win-win
  • Improves site experience and reduces ISP bandwidth usage
CDNs Issues

• Performance: How accurate can the IP map be?

• Dynamic pages: What about dynamic content?

• Security: How to cache/forward encrypted content?

• Privacy: What about private information?