2- Application Level Protocols
HTTP 1.0/1.1/2
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
Web Context

Page as a set of related HTTP transactions

HTTP request
HTTP response

Document
Program
Database

youtube.com

Web server
www.cs.washington.edu
google-analytics.com

Web browser

Hyperlink

Web page
Web Protocol Context

- HTTP is a request/response protocol for fetching Web resources
  - Runs on TCP, typically port 80
    - HTTPS typically on port 443
  - Part of browser/server app
Static vs Dynamic Web pages

- Static web page is a file contents, e.g., image
- Dynamic web page is the result of program execution
  - Javascript on client, PHP on server, or both
HTTP Protocol

• Originally a simple protocol, with many options added over time

• HTTP transports typed data
  – TCP transports bytes

• HTTP is a request-response protocol
  – Client sends request message, server sends response message

• HTTP messages have a header and a payload section

• Header is encoded as text

• HTTP is carried over a TCP
Fetching a Web Page From a Browser

• Browser starts with the page URL:
  
  http://en.wikipedia.org/wiki/Vegemite

• Browser steps:
  – Resolve the server name to an IP address (DNS)
  – Set up a TCP connection to the server
  – Send HTTP request for the page
  – Wait for and then read HTTP response
  – (Assuming no errors) Process response data and render page
  – Clean up any idle TCP connections
HTTP Message Format

- Header is encoded as text
- Header is a sequence of lines
- Each line ends with \r\n
- Header ends with \r\n\r\n
- Payload length is given by either:
  - Content-length tag in header
  - Payload is encoded in a format that uses a sentinel (special value that marks the end)
Try It Yourself: View HTTP Request

• `$ nc -l 8080`  
  Opens a TCP socket on port 8080 and waits for an incoming connection

• Point a browser running on the same machine to  
  http://localhost:8080/first/second/third.html

• The output of the nc window is the HTTP request sent by the browser
Example HTTP Request

$ nc -l 8080
GET /first/second/third.html HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/*;q=0.8
DNT: 1
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8

What the browser sent
Try It Yourself: HTTP Response

```
$ nc www.washington.edu 80
GET / HTTP/1.0
Host: www.washington.edu

HTTP/1.1 200 OK
Date: Tue, 10 Oct 2017 15:58:14 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/1.0.1e-fips PHP/5.6.26 mod_pubcookie/3.3.4a mod_uwa/3.2.1
Last-Modified: Mon, 09 Oct 2017 21:45:12 GMT
ETag: "180cd3-c459-55b241ae94a00"
Accept-Ranges: bytes
Content-Length: 50265
Vary: Accept-Encoding,User-Agent
Connection: close
Content-Type: text/html

<!DOCTYPE html><html class="no-js"><head><meta content="IE=edge" http-equiv="X-UA-Compatible"/><title> UW Homepage </title><meta charset="utf-8"/><meta content="University of Washington" name="d.... <50265 bytes of data in all>
```
Try It Yourself: HTTP Response 2

$ nc uw.edu 80
GET / HTTP/1.0
Host: uw.edu

HTTP/1.1 301 Moved Permanently
Date: Tue, 10 Oct 2017 16:04:20 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/1.0.1e-fips PHP/5.6.26 mod_pubcookie/3.3.4a mod_uwa/3.2.1
Location: http://www.washington.edu/
Vary: Accept-Encoding
Content-Length: 385
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head><title>301 Moved Permanently</title></head><body><h1>Moved Permanently</h1><p>The document has moved <a href="http://www.washington.edu/">here</a>.</p><hr><address>Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/1.0.1e-fips PHP/5.6.26 mod_pubcookie/3.3.4a mod_uwa/3.2.1 Server at uw.edu Port 80</address></body></html>
# HTTP Protocol

## 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

Result 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; 503 = try again later</td>
</tr>
</tbody>
</table>
PERFORMANCE
Performance Measure: PLT (Page Load Time)

• PLT is the key measure of web performance
  – From click until user sees page

• PLT depends on many factors
  – Structure of page/content
  – HTTP (and TCP!) protocol
  – Network RTT and bandwidth
You Now Have a Shorter Attention Span Than a Goldfish

Kevin McSpadden
May 13, 2015

For more, visit TIME Health.

The average attention span for the notoriously ill-focused goldfish is nine seconds, but according to a new study from Microsoft Corp., people now generally lose concentration after eight seconds, highlighting the affects of an increasingly digitalized lifestyle on the brain.
Page Load Time Impact

From *How One Second Could Cost Amazon $1.6 Billion In Sales*, March 15, 2012
HTTP 1.0 (1996)

- HTTP/1.0 uses one TCP connection to fetch one web resource
  - Made HTTP very easy to build
  - But gave fairly poor PLT ...

- Framing?
  - Length?
  - Sentinel?
HTTP 1.0

- Many reasons why PLT is larger than necessary
  - Sequential request/responses, even when to different servers
  - Multiple TCP connection setups to the same server
Parallel Connections

• One simple way to reduce PLT
  – Browser runs multiple (8, say) HTTP instances in parallel
  – 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
HTTP 1.1 (1997)
Persistent Connections

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

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

- One request per connection
- Sequential requests per connection
- Pipelined requests per connection
Persistent Connections: Framing

• How are requests and responses framed?
  – Enforce use of content-length header field?
    • What if content is dynamically generated?
  – If not that, then what?
Persistent Connections

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

• How can we reduce PLT even more?
HTTP 2 (2015)

- HTTP 2 preserves the semantics of HTTP 1.0 / 1.1
  - Client still says GET and server still responds OK
- However, the requests are
  - encoded differently (compressed)
  - transferred differently (streams and frames)

- IETF RFC 7540, May 2015
  - Successor to Google’s SPDY protocol
This is the idea of how HTTP 2 fits in. A particular implementation might well combine HTTP 1.1 and HTTP 2.
HTTP 2 – Main Features

• Allows “real pipelining” of requests on persistent connections
  – We have to “name” each request explicitly so that we can match responses to requests
    • Why can’t we use ordering?

• Headers have gotten big
  – compress headers

• Servers can supply data that wasn’t requested
  – “server push”

• Clients can advertise priorities among their requests

Note: “real pipelining” allows the server to apply its own idea of priority, since it doesn’t have to reply in order
HTTP 2 – Streams and Frames

• A *connection* is a TCP connection between client and server
  – long lived, just like HTTP 1.1

• A *stream* is an ordered, bidirectional flow of information between client and server

• There is one connection between a client and server

• There is (roughly) one stream per HTTP request
HTTP 2 – Streams & Frames

Connection

Stream

Request message

Response message

Stream

Request message

Response message

...
Streams

• Each stream has a unique ID
  – Successive stream IDs must be increasing
  – When run out of stream IDs, have to create a new connection

• Race condition if both ends try to create stream IDs
  – Solution: “client” uses odd numbers, server uses evens

• A stream is created by sending a frame with a new stream ID
## Frame Types

<table>
<thead>
<tr>
<th>Frame type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>HTTP body</td>
</tr>
<tr>
<td>HEADERS</td>
<td>Header fields</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>Sender-advised priority of stream</td>
</tr>
<tr>
<td>RST_STREAM</td>
<td>Signal termination of stream</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>Configuration parameters for the connection</td>
</tr>
<tr>
<td>PUSH_PROMISE</td>
<td>Signal a promise (push) of referenced sources</td>
</tr>
<tr>
<td>PING</td>
<td>Measure roundtrip time and “liveness”</td>
</tr>
<tr>
<td>GOAWAY</td>
<td>Inform peer to stop creating streams for current connection</td>
</tr>
<tr>
<td>WINDOW_UPDATE</td>
<td>Connection flow control</td>
</tr>
<tr>
<td>CONTINUATION</td>
<td>Continue a segment of header block fragments</td>
</tr>
</tbody>
</table>
Simples encoding of an HTTP request

• Send a HEADER frame followed by zero or more CONTINUATION frames
  – Set END_HEADERS flag on last one
• Send DATA frames for request data, if needed
  – Set END_STREAM flag on last
• Response is the same, in reverse
Frame Header

- Length: length of payload
  - header is always 9 bytes
- Type: frame type
- Flags: depends on type
- R: reserved; "must be unset when sending and ignored when receiving"
- Stream ID: 0x0 is reserved for frames associated with the connection
  (not an individual stream)
• Padding is for security – obfuscate lengths
• Stream dependency – make this stream a child of named stream
  • If server can’t make progress on parent, assign resources proportional to weights to children
• Header block fragment – take the HTTP 1.1 header and compress it, then send it in chunks (if necessary)
• Frame header flags: END_HEADERS and END_STREAM
Figure 6: DATA Frame Payload
PRIORITY Frame

- E: exclusive bit – inserts this stream as only child of parent stream, moving existing children to be children of this stream
RST_STREAM Frame

• Ends a stream
  - Why is this useful?
    • Also have END_STREAM flag bit...
GOAWAY Frame

- Closes connection
- Provides largest id of any stream that the server may have acted on
  - Why?
PUSH_PROMISE Frame

- Allows server to send something not yet asked for
  - E.g., a style sheet or a javascript program or an embedded image
- Acts like a HEADERS frame
  - Can have CONTINUATIONs following for more header

Figure 11: PUSH_PROMISE Payload Format
PING Frame

- Is other end still there?
  - Responds with PING with ACK flag bit set

- Measure latency to other end
  - PING frames have highest priority...
• TCP does flow control on entire connection
  – but need flow control on a per stream basis as well
• HTTP 2 is supposed to be an optimized transport of HTTP requests
  – Needs to be backward compatible with HTTP 1/1.1

• Main problem:
  – How to tell if client and server can both speak HTTP 2?
    • Client could try HTTP 2 and then revert to 1.1
    • Client could start with HTTP 1.1 then upgrade to 2
Dynamically Upgrading to HTTP 2

- Client:

```plaintext
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```
Server Refuses Upgrade

- Server may simply not recognize the upgrade request if it isn’t HTTP 2 capable

HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

...
Server Wants to Upgrade

HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
HTTP 2 Wrap-up
Web Caching

• Users often revisit web pages
  – Big win from reusing local copy!
  – This is caching

• Key question:
  – When is it OK to reuse local copy?
Web Caching (2)

- Locally determine copy is still valid
  - Based on expiry information such as “Expires” header from server
  - Or use a heuristic to guess (cacheable, freshly valid, not modified recently)
  - Content is then available right away
Web Caching (3)

• Revalidate copy with remote server
  – Based on timestamp of copy such as “Last-Modified” header from server
  – Or based on content of copy such as “Etag” server header
  – Content is available after 1 RTT
Web Caching (4)

• Putting the pieces together:

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

Web browser
Cache
Program
Web server
Web Proxies

• Place intermediary between pool of clients and external web servers
  – Benefits for clients include caching and security checking
  – Organizational access policies too!

• Proxy caching
  – Clients benefit from larger, shared cache
  – Benefits limited by secure / dynamic content, as well as “long tail”
Web Proxies

- Clients contact proxy; proxy contacts server

![Diagram of Web Proxies]

- Near client
- Far from client
Content Delivery Networks

• As the web took off in the 90s, traffic volumes grew and grew. This:
  1. Concentrated load on popular servers
  2. Led to congested networks and need to provision more bandwidth
  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 to 4 users takes $4 \times 3 = 12$ “network hops” in the example
• Sending content via replicas takes only $4 + 2 = 6$ “network hops”
After CDNs

• Benefits assuming popular content:
  – Reduces server, network load
  – Improves user experience (PLT)
Popularity of Content

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

Zipf popularity (kth item is $\sim 1/k$)

Rank

Relative Frequency


George Zipf (1902-1950)
How to place content near clients?

- 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
  - Done by clever use of DNS
Content Delivery Network

CDN origin server

Distribution to CDN nodes

CDN node

Page fetch

Sydney

Boston

Amsterdam

Worldwide clients
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