HTTP/1.1 is awesome – you can’t argue with its deployed footprint.

But there are well-known performance limitations.
HTTP is not a good fit for TCP

• TCP is designed for long-lived, bulk transfers
  – High-handshake costs, TLS adds even more to startup costs
  – HTTP requests are short and bursty

• Parallelism needed, but:
  – Pipelining has problems with head-of-line-blocking, recovering from failures
  – More TCP connections, more client+server resources to manage the sockets, bandwidth consumed by TCP overhead
  – In practice, browsers limit to six concurrent connections
Parallelism Is Needed Because of Page Bloat

Growth of Average Web Page Size and Number of Objects - Jan 1995-July 2014

Total Bytes (K) vs Average Number of Objects

(Sources: Domenech 2007, Gomez 2006, Chorzinski 2010, Souders 2014)

From: https://www.webbloatscore.com/    See also: https://httparchive.org/reports/state-of-the-web
Parallelism Limits In Practice

<table>
<thead>
<tr>
<th>Name</th>
<th>Method</th>
<th>Status</th>
<th>Type</th>
<th>Time</th>
<th>Start Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>localhost</td>
<td>GET</td>
<td>200</td>
<td>text/html</td>
<td>17 ms</td>
<td></td>
</tr>
<tr>
<td>01.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>242 ms</td>
<td></td>
</tr>
<tr>
<td>02.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>243 ms</td>
<td></td>
</tr>
<tr>
<td>03.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>242 ms</td>
<td></td>
</tr>
<tr>
<td>04.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>241 ms</td>
<td></td>
</tr>
<tr>
<td>05.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>235 ms</td>
<td></td>
</tr>
<tr>
<td>06.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>235 ms</td>
<td></td>
</tr>
<tr>
<td>07.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>475 ms</td>
<td></td>
</tr>
<tr>
<td>08.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>563 ms</td>
<td></td>
</tr>
<tr>
<td>09.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>561 ms</td>
<td></td>
</tr>
<tr>
<td>10.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>561 ms</td>
<td></td>
</tr>
<tr>
<td>11.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>561 ms</td>
<td></td>
</tr>
<tr>
<td>12.jpeg</td>
<td>GET</td>
<td>202</td>
<td>image/jpeg</td>
<td>561 ms</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11-5. Staggered resource downloads due to six-connection limit per origin

From: https://hpbn.co/http1x/
HTTP Headers: Metadata >> Data

```
$ curl --trace-ascii -d '{"msg":"hello"}' http://www.igvita.com/api

== Info: Connected to www.igvita.com
=> Send header, 218 bytes
  POST /api HTTP/1.1
  User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 ... 
  Host: www.igvita.com
  Accept: */*
  Content-Length: 15
  Content-Type: application/x-www-form-urlencoded
=> Send data, 15 bytes (0xf)
{"msg":"hello"}

<= Recv header, 134 bytes
  HTTP/1.1 204 No Content
  Server: nginx/1.0.11
  Via: HTTP/1.1 GWA
  Date: Thu, 20 Sep 2012 05:41:30 GMT
  Cache-Control: max-age=0, no-cache
```

Here, 15 bytes of json + 352 bytes of request and response headers

1. HTTP request headers: 218 bytes
2. 15-byte application payload ("msg":"hello")
3. 204 response from the server: 134 bytes

From: https://hpbn.co/http2/
HTTP/1.1 Optimizations
Image Sprites

Send one large image of all flags, use CSS to “cut out” the flag you need

From: https://daniel.haxx.se/http2/
Inlining & Concatenation

- **Inlining**: send small images as base64

  ```html
  <img src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAAAAFCAYAAACNbyb1AAAAAEHELQVQ112P4//8/w38GIAXDIBKE0DHxg1jNBAO9TXL0Y4OHwAAAAABJRU5ErkJgg=" alt="Red dot" />  
  ```


- **Concatenation**: put all of your .js/.css files into a single, large .js/.css file
  
  - Probably sends more than you need
  
  - Small changes in one file means changes in the entire file
Domain Sharding

Six connections per domain,
But with the overhead of additional DNS lookups.

From: https://daniel.haxx.se/http2/
Evolution from SPDY to HTTP/2

- November 2009: Google begins work on SPDY to address performance limitations of HTTP/1.1
- September 2010: SPDY supported in Chrome
- January 2011: SPDY deployed for all Google services
- March 2012: Twitter supports SPDY
- March 2012: Call for proposals for HTTP/2
- June 2012: NGINX supports SPDY
- July 2012: Facebook announces planned support for SPDY
- November 2012: First draft of HTTP/2 (based on SPDY)
- August 2014: HTTP/2 draft-17 and HPACK draft-12 are published
- August 2014: Working Group last call for HTTP/2
- February 2015: IESG approved HTTP/2 and HPACK drafts
- May 2015: RFC 7540 (HTTP/2) and RFC 7541 (HPACK) are published

Google Deprecates SPDY

“HTTP/2's primary changes from HTTP/1.1 focus on improved performance. Some key features such as multiplexing, header compression, prioritization and protocol negotiation evolved from work done in an earlier open, but non-standard protocol named SPDY. Chrome has supported SPDY since Chrome 6, but since most of the benefits are present in HTTP/2, it’s time to say goodbye. We plan to remove support for SPDY in early 2016, and to also remove support for the TLS extension named NPN in favor of ALPN in Chrome at the same time. Server developers are strongly encouraged to move to HTTP/2 and ALPN.

We’re happy to have contributed to the open standards process that led to HTTP/2, and hope to see wide adoption given the broad industry engagement on standardization and implementation.”

High-level semantics of HTTP don’t change in HTTP/2, but the method of packaging and transport do.
Binary Framing Layer

No more hand-crafted telnet sessions – boo!!!!!

Figure 12-1. HTTP/2 binary framing layer

From: https://hpbn.co/http2/
Streams, Messages, Frames

Stream: bi-directional connection, with 1 or more messages

Message: logically complete request or response

Frame: typed, atomic unit of communication

Figure 12-2. HTTP/2 streams, messages, and frames

From: https://hpbn.co/http2/
Request & Response Multiplexing

- Interleave multiple requests in parallel without blocking on any one
- Interleave multiple responses in parallel without blocking on any one
- Use a single connection to deliver multiple requests and responses in parallel
- Remove unnecessary HTTP/1.x workarounds (such as concatenated files, image sprites, and domain sharding)
- Deliver lower page load times by eliminating unnecessary latency and improving utilization of available network capacity

Note: frames cannot be received out of order!

From: https://hpbn.co/http2/
Stream Dependencies & Weights

A gets ¾ of bandwidth, B gets ¼
A & B are dependent on the "root" stream (i.e., no dependencies)

C depends on D, service D first (weights trumped by dependency)

D before C, C before A & B, weight A & B as before

D before C, C & E equally
Before A & B, weight A & B as before

Figure 12-4. HTTP/2 stream dependencies and weights

From: https://hpbn.co/http2/
Server Push: 1 Request, N Responses

Conceptually similar to inlining, rel="preload", rel="prefetch", etc.
Can only push with same-origin policy.

From: https://hpbn.co/http2/

See discussion of HTTP/2 push in:

**Figure 12-5.** Server initiates new streams (promises) for push resources
Header Repetitiveness Allows Compression

Note: headers beginning with “:” are “pseudo-headers” (RFC 7540, 8.1.2.1); or “things-that-should-have-been-headers-in-HTTP/1.1” Pseudo-headers have to be listed before real headers.
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: (SETTINGS payload)

HTTP/1.1 200 OK
Content-length: 243
Content-type: text/html

(... HTTP/1.1 response ...)

(or)

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

(... HTTP/2 response ...)

Note:
“h2” = HTTP/2 over TLS
“h2c” = HTTP/2 over clear text TCP

From: https://hpbn.co/http2/
## 9 Byte Frame Header

### Header Types:
- **DATA** - Used to transport HTTP message bodies
- **HEADERS** - Used to communicate header fields for a stream
- **PRIORITY** - Used to communicate sender-advised priority of a stream
- **RST_STREAM** - Used to signal termination of a stream
- **SETTINGS** - Used to communicate configuration parameters for the connection
- **PUSH_PROMISE** - Used to signal a promise to serve the referenced resource
- **PING** - Used to measure the roundtrip time and perform "liveness" checks
- **GOAWAY** - Used to inform the peer to stop creating streams for current connection
- **WINDOW_UPDATE** - Used to implement flow stream and connection flow control
- **CONTINUATION** - Used to continue a sequence of header block fragments

---

### Table: 9 Byte Frame Header

<table>
<thead>
<tr>
<th>Bit</th>
<th>+0..7</th>
<th>+8..15</th>
<th>+16..23</th>
<th>+24..31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>R</td>
<td>Stream Identifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Frame Payload</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 12-7. Common 9-byte frame header*

Note: frames cannot be received out of order! Stream id, but not frame id.

---

**Note:** Technically, the Length field allows payloads of up to $2^{24}$ bytes (~16MB) per frame. However, the HTTP/2 standard sets the default maximum payload size of DATA frames to $2^{14}$ bytes (~16KB) per frame and allows the client and server to negotiate the higher value. Bigger is not always better; smaller frame size enables efficient multiplexing and minimizes head-of-line blocking.

From: [https://hpbn.co/http2/](https://hpbn.co/http2/)
Example Binary HTTP/2 Request

From: https://hpbn.co/http2/
HTTP/3 Network Stack

HTTP/2 optimizes within TCP context (e.g., binary, streams & frames), HTTP/3 replaces TCP

From: https://daniel.haxx.se/blog/2018/11/26/http3-explained/
HTTP/3

• “HTTP-over-QUIC” was renamed to “HTTP/3” (Nov 2018)
• HTTP/3 became Standard Track RFC in June 2022
  – Deployment is growing gradually
• Major changes:
  – Streams are moved from the HTTP layer to the QUIC layer
    • HTTP/2 fixes HTTP head-of-line blocking, but not TCP head-of-line blocking
      (i.e., streams in TCP can still be held up by dropped TCP packets)
  – Since streams are independent, header compression changes
  – There is no clear-text version of HTTP/3 (integral TLS 1.3)
  – QUIC has faster handshakes than TCP + TLS

https://http3-explained.haxx.se/