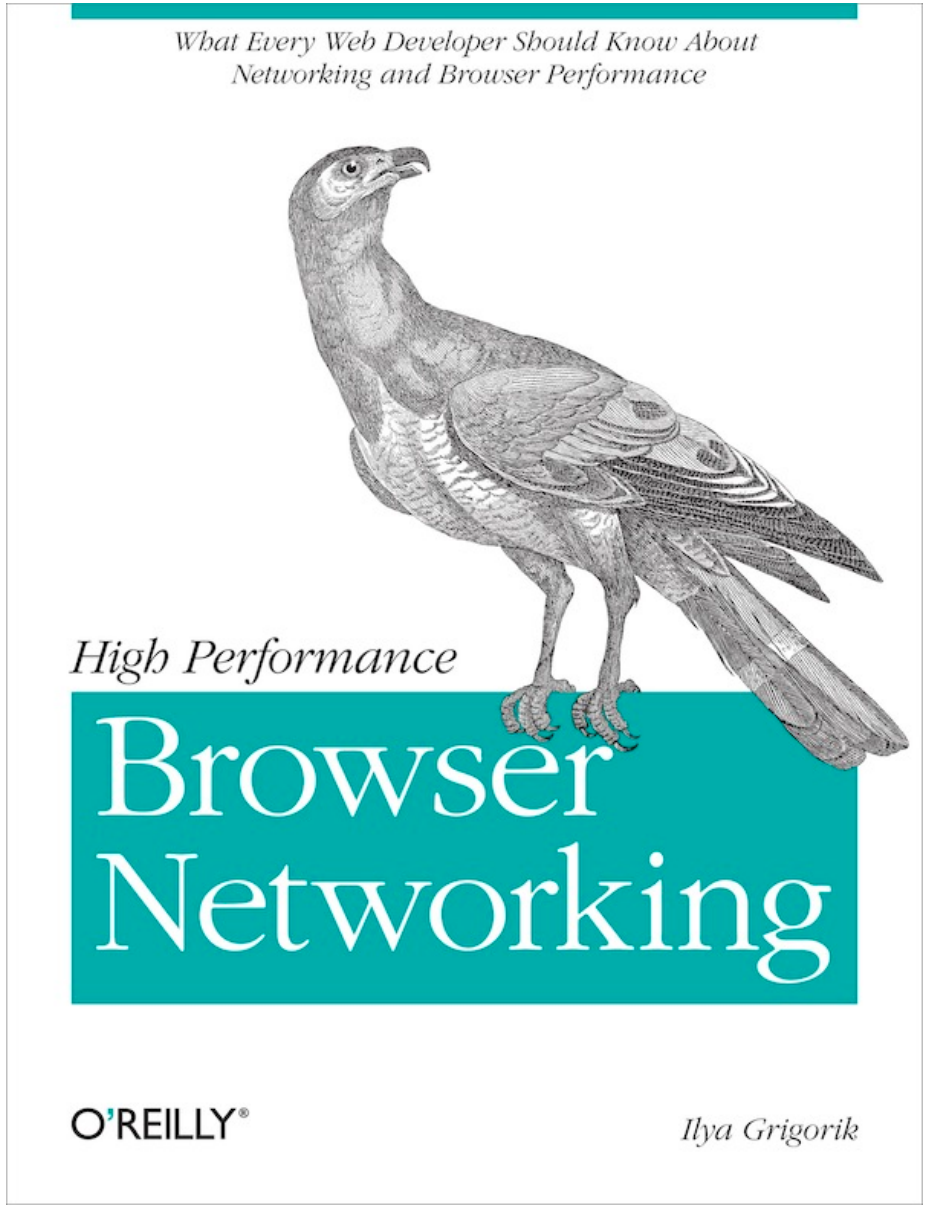
HTTP/2 all the things!

challenges, opportunities, and the exciting world ahead of us...

+Ilya Grigorik
@igrigorik

Who's this guy? :-)

- **Performance Engineer @ Google**
  - *Anything web perf related…*
- **Wrote HPBN (read @ hpbn.co)**
  - Radio → TCP → TLS → HTTP
  - Browser APIs: XHR, WS, WebRTC
  - …
- Blog: **igvita.com**
- Twitter: **@igrigorik**



```
$> telnet igvita.com 80
Connected to 173.230.151.99

GET /archive

Hypertext delivery with HTTP 0.9! - eom.
(connection closed)
```

HTTP 0.9 is the ultimate MVP *- one line, plain-text "protocol" to test drive the "www idea".*

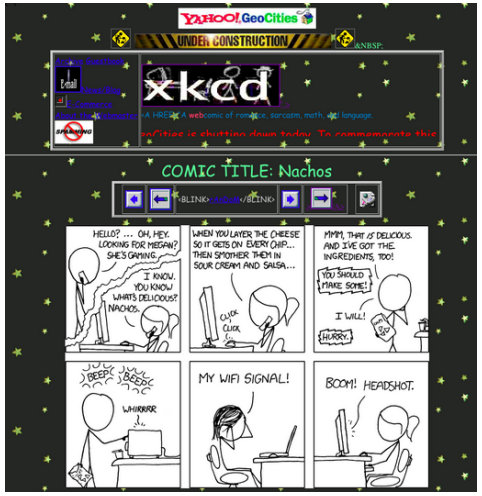
```
$> telnet ietf.org 80
Connected to 74.125.xxx.xxx

GET /rfc/rfc1945.txt HTTP/1.0
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Accept: */*

HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 137582
Last-Modified: Wed, 1 May 1996 12:45:26 GMT
Server: Apache 0.84

4 years of rapid iteration later... eom.
(connection closed)
```

HTTP 1.0 is an informational RFC *- documents "common usage" of HTTP found in the wild.*

```
$> telnet google.com 80
Connected to 74.125.xxx.xxx

GET /index.html HTTP/1.1
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked
Server: nginx/1.0.11
Content-Type: text/html; charset=utf-8
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache

100
<!doctype html>
(snip)
```

HTTP 1.1 ships as RFC standard in 1999 *- hyper*
{~~text~~}media all the things!

In the meantime...



Geocities ftw!
(circa HTTP/1.1)

- Web applications, not (just) pages.
- Rich media and multi-device layouts.

State of the HTTP nation...

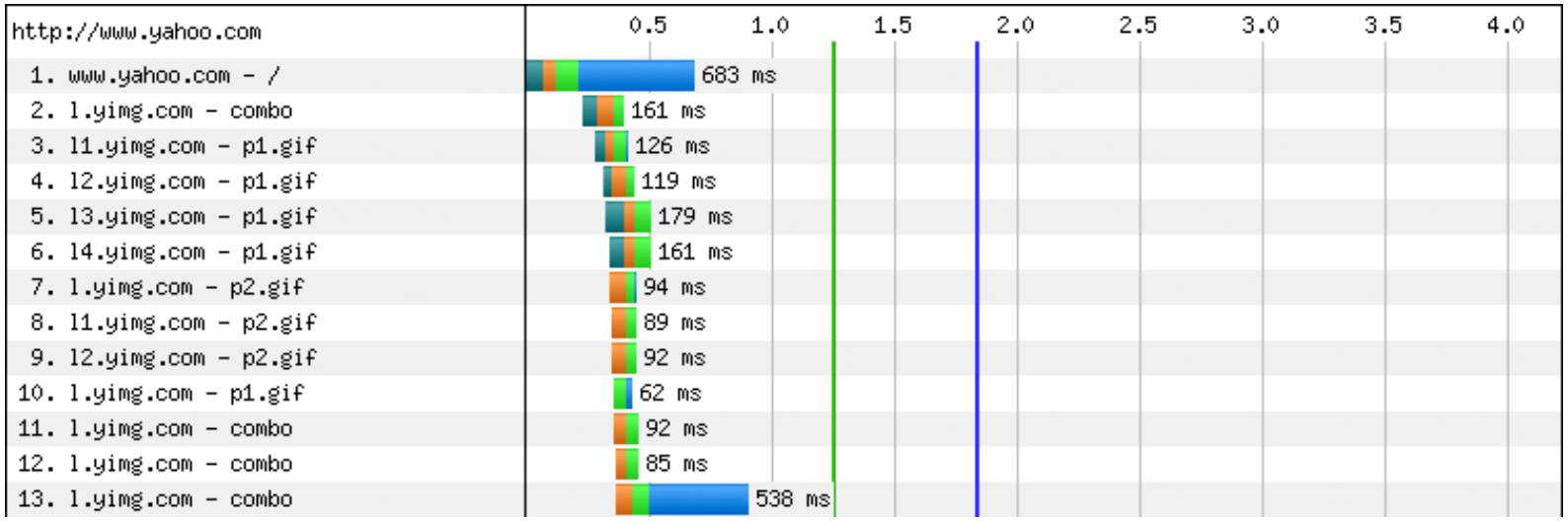
- **12** distinct hosts per page
- **78** distinct requests per page
- **1,232 KB** transferred per page

*Resulting in typical render times of **2.6-5.6 seconds**.*

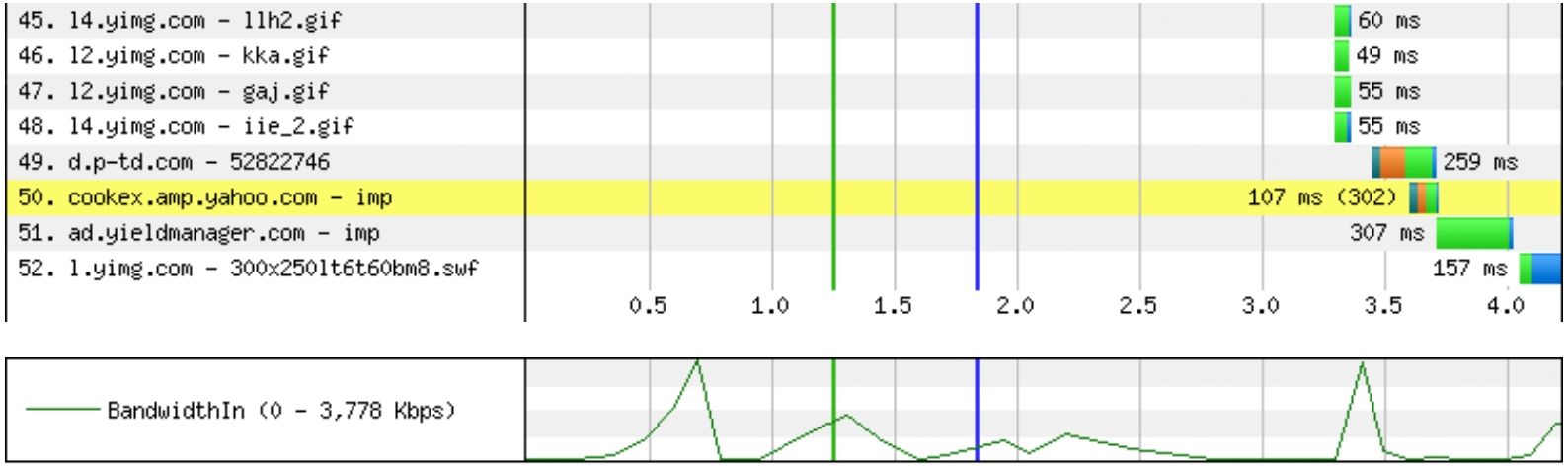
50th and 90th percentiles

* All numbers are medians, based on latest HTTP Archive crawl data.

Yahoo.com waterfall… BW is not the issue?



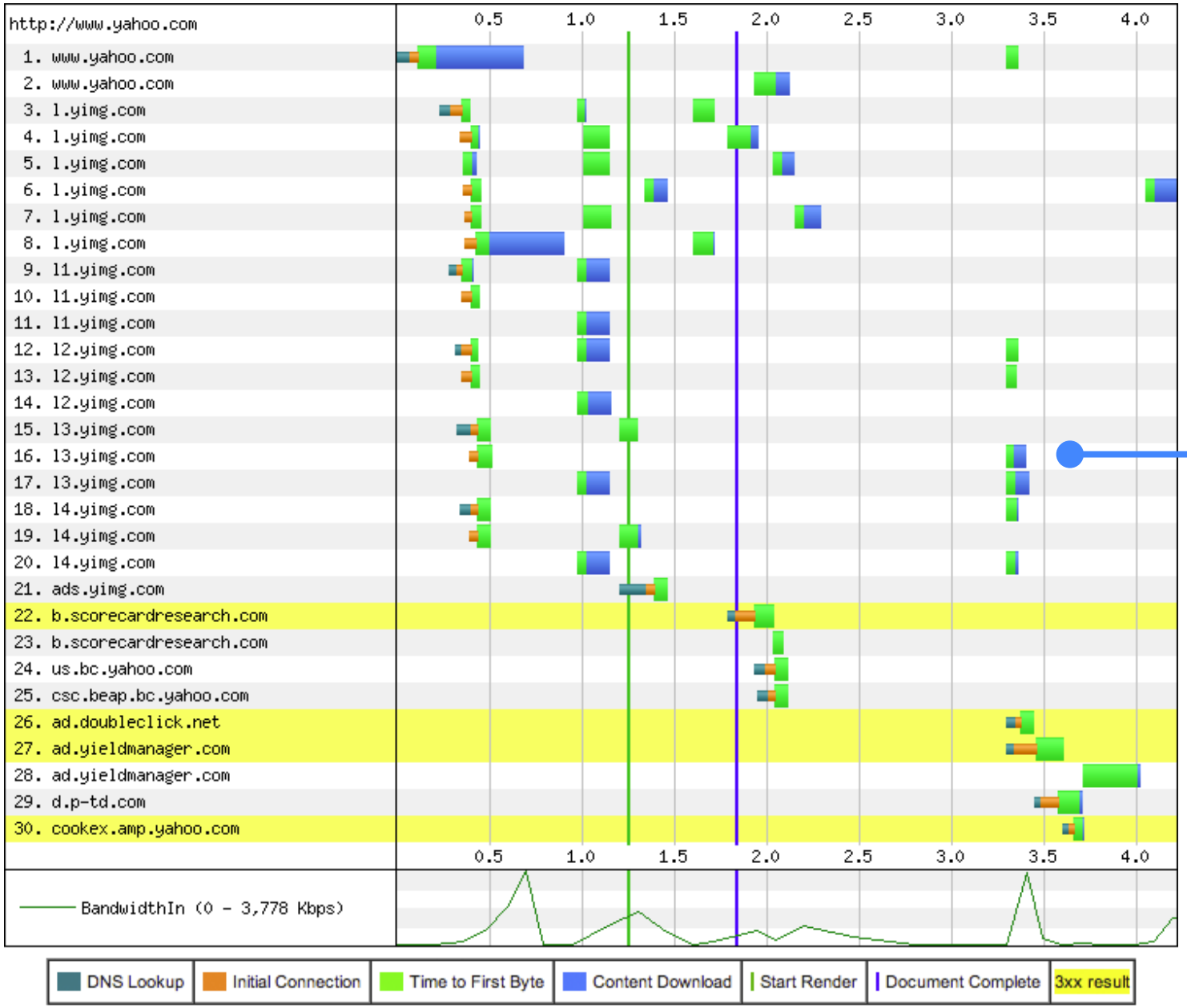
… (snip 30 requests) …



- 52 requests
- 4+ seconds

[Primer on Web Performance](#) (Chapter 10)

“Connection view” tells the story…



30 connections

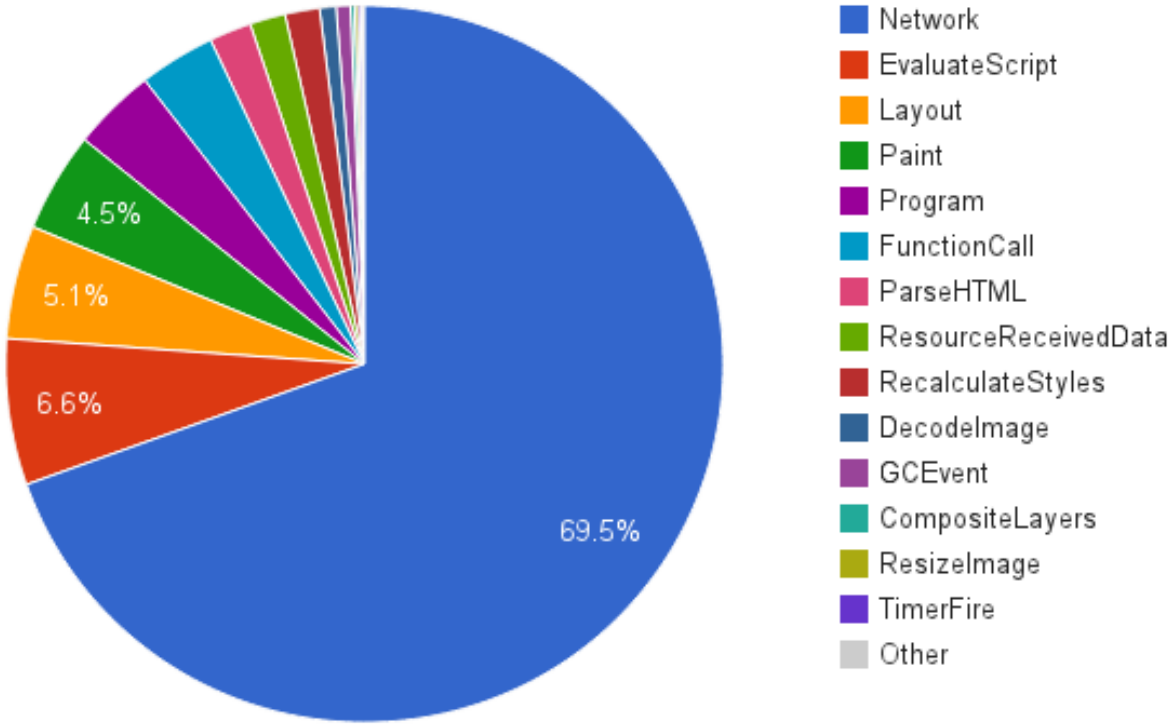
- *DNS lookups*
- *TCP handshakes*
- *…*

Transfer time (in blue)

***We’re not BW limited**, we’re literally idling, waiting on the network to deliver resources.*

Top 1M Alexa sites...

```
Total CPU time:          654,697s
Total page load time: 2,149,369s
Average CPU time:           735ms
Average page load time:   2,413ms
```




```
                Network:     1494671s  69.5%
         EvaluateScript:      141658s  6.6%
                 Layout:      109802s  5.1%
                  Paint:       96955s  4.5%
```

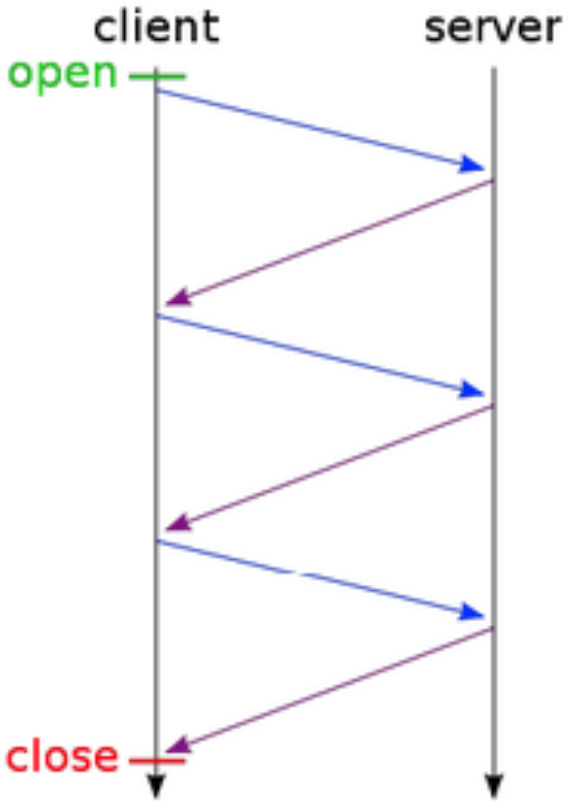
- Cable profile (5Mbps / 28 ms RTT)
- Main thread attribution in Blink
  - Measured via Telemetry

- **69.5% of time blocked on network**
- **6.6% of time blocked JavaScript**
- **5.1% blocked on Layout**
- **4.5% blocked on Paint**
- **...**

No surprises here... First page load is network (latency) bound!

[blink-dev thread]

HTTP/1.1 performance problems...

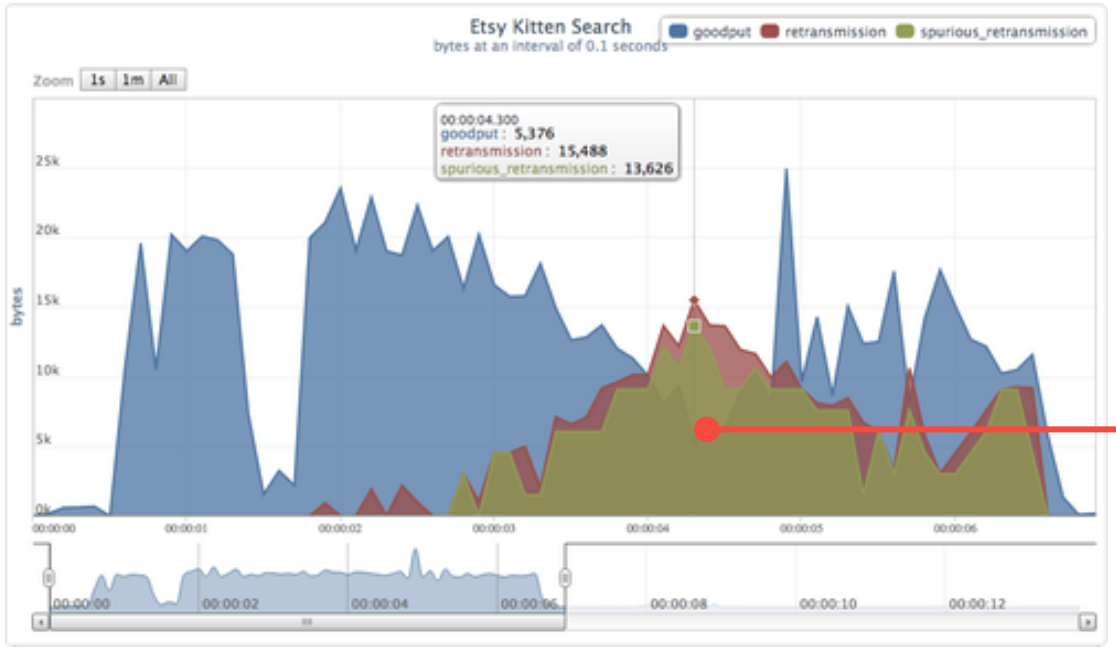


- **Limited parallelism**
  - *Maximum of 6 requests per origin*
  - *Pipelining does not work in practice*
  - *Competing TCP flows, spurious retransmissions*
  - *Extra handshakes, FDs, memory buffers, etc.*
- **Client-side request queuing**
  - *Head-of-line blocking*
  - *Delayed request dispatch*
- **High protocol overhead**
  - *~800 bytes of header + cookies*
  - *No compression of HTTP metadata*

Where there's a will, there's a way...

we're an inventive bunch, so we came up with some "optimizations" (read, "hacks")

Domain shard… all the things!



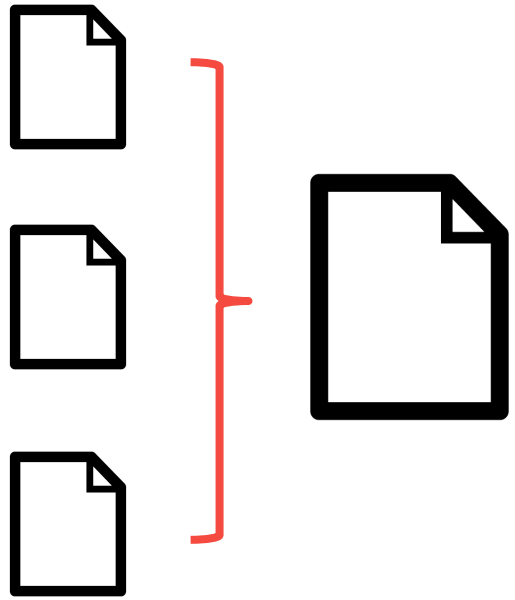
6 connections per origin
just add more origins, right?

Duplicate (spurious) data packets due to oversharding

Optimal number of shards? There is no such thing. Depends on particular page, device + network + network weather. Most sites overshard, and hurt themselves… Causing congestion, retransmissions, etc.

http://perf.fail/post/96104709544/zealous-sharding-hurts-etsy-performance

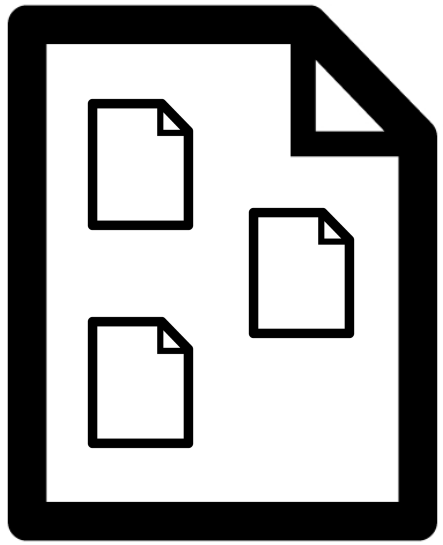
Concat… all the things!



"Reduce number of requests"...

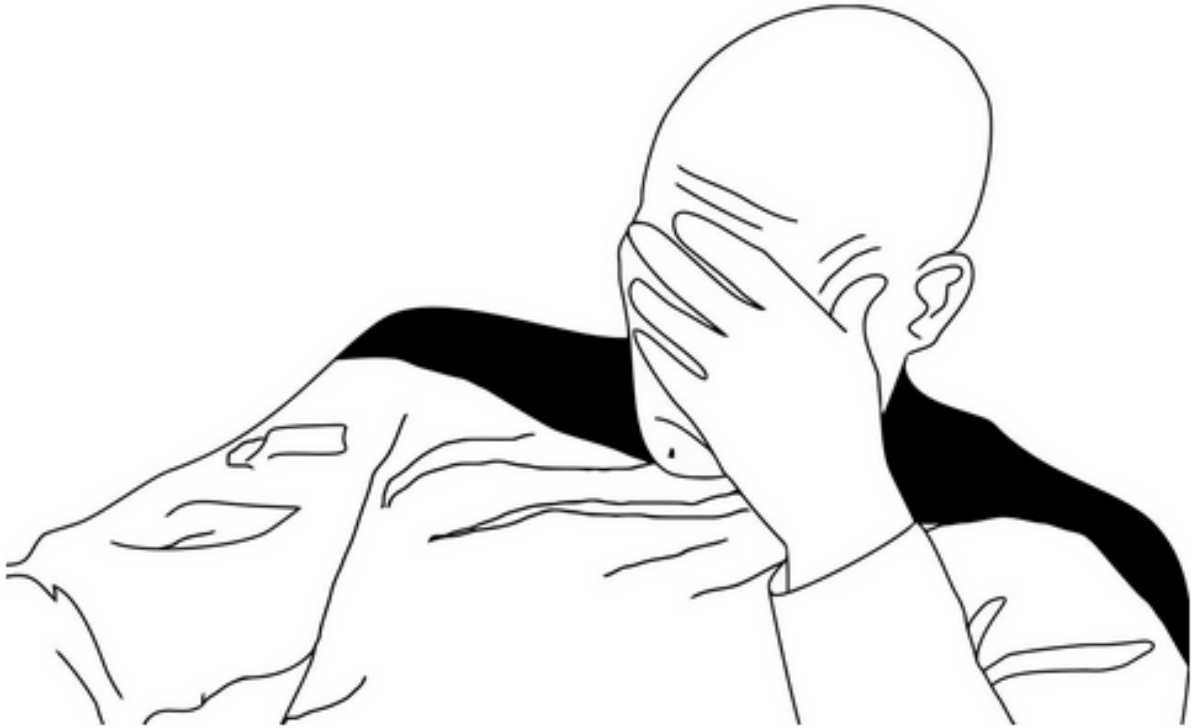
- **Large monolithic code chunks**
  - *e.g. most pages use <20% of CSS rules*
- **Expensive cache invalidations**
  - *e.g. single char update forces full fetch*
- **Delayed execution of JSS / CSS**
  - *e.g. must wait for entire JS file to arrive*
  - *e.g. must wait for entire CSS file to arrive*

Inline... all the things!



"Reduce number of requests"...

- **Duplicated resources**
  - *every page must embed the same resource*
  - *can't use the HTTP cache*
- **Breaks prioritization**
  - *inlined asset is "upgraded" to HTML priority*
  - *inflates the size of HTML document*



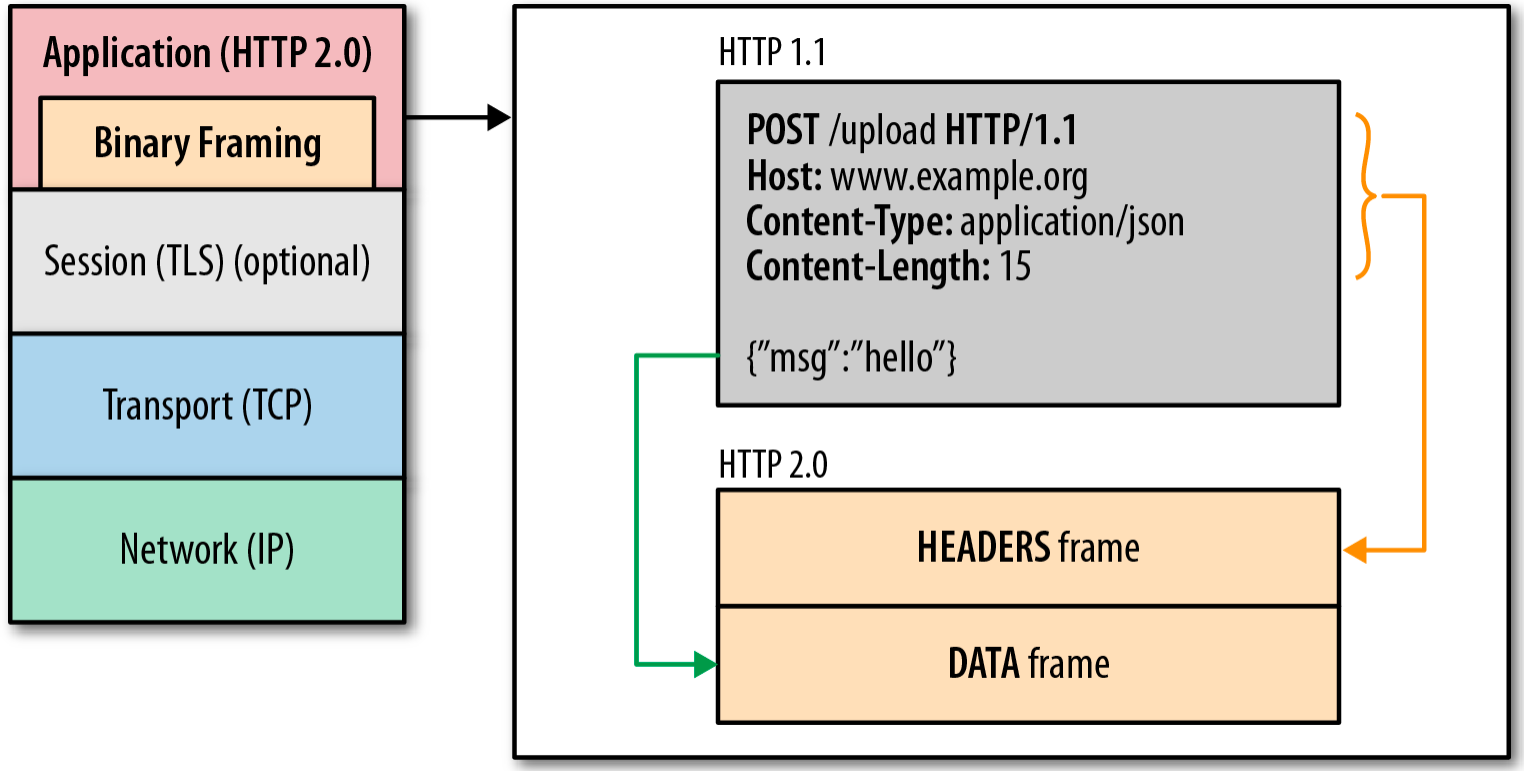
*Let's **fix HTTP** instead?*

*"**HTTP 2.0** is a protocol designed for **low-latency transport of content** over the World Wide Web"*

- Improve end-user perceived latency
- Address the "head of line blocking"
- Not require multiple connections
- Retain the semantics of HTTP/1.1

HTTP/2 in one slide…

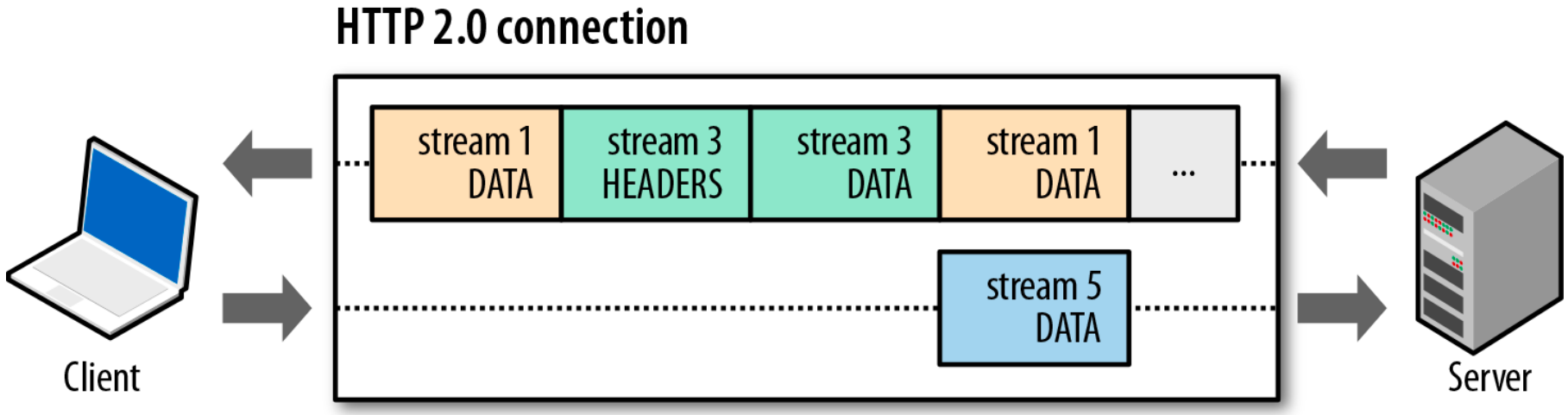
- **One TCP connection**
- **Request → Stream**
  - Streams are multiplexed
  - Streams are prioritized
- **Binary framing layer**
  - Prioritization
  - Flow control
  - Server push
- **Header compression**



*"... **we're not replacing all of HTTP** — the methods, status codes, and most of the headers you use today will be the same. Instead, **we're redefining how it gets used "on the wire" so it's more efficient**, and so that it is more gentle to the Internet itself ...."*

- Mark Nottingham (HTTPbis chair)

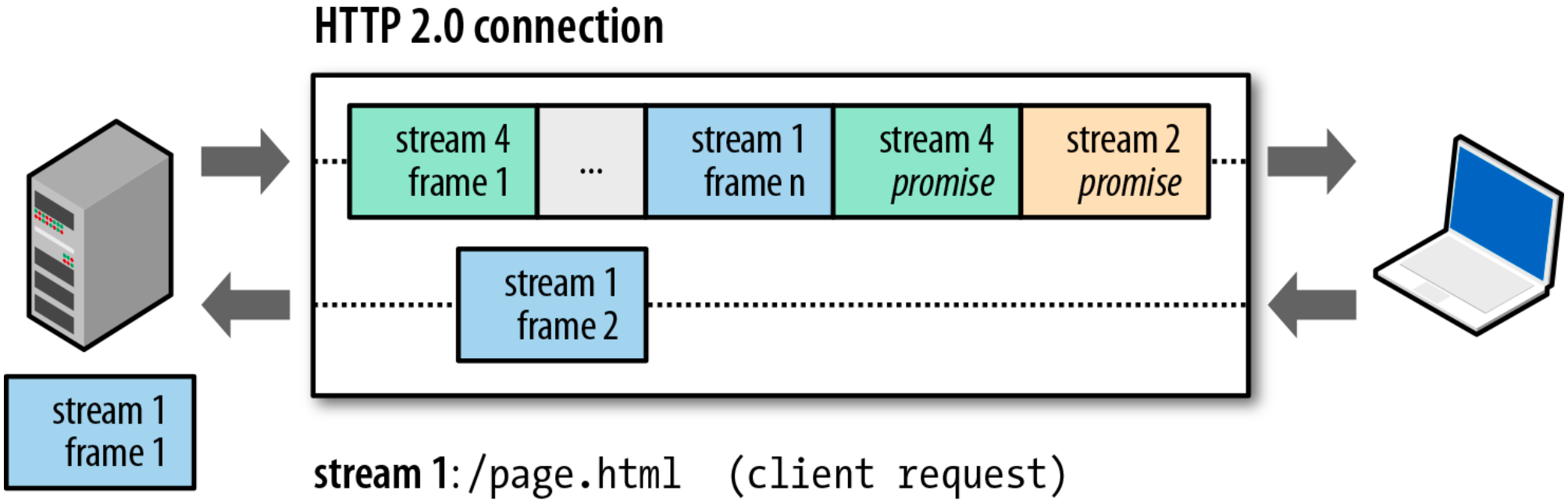
Basic data flow in HTTP 2.0...



- **Streams are multiplexed** by splitting communication into frames
  - All frames (e.g. HEADERS, DATA, etc) are sent over single TCP connection

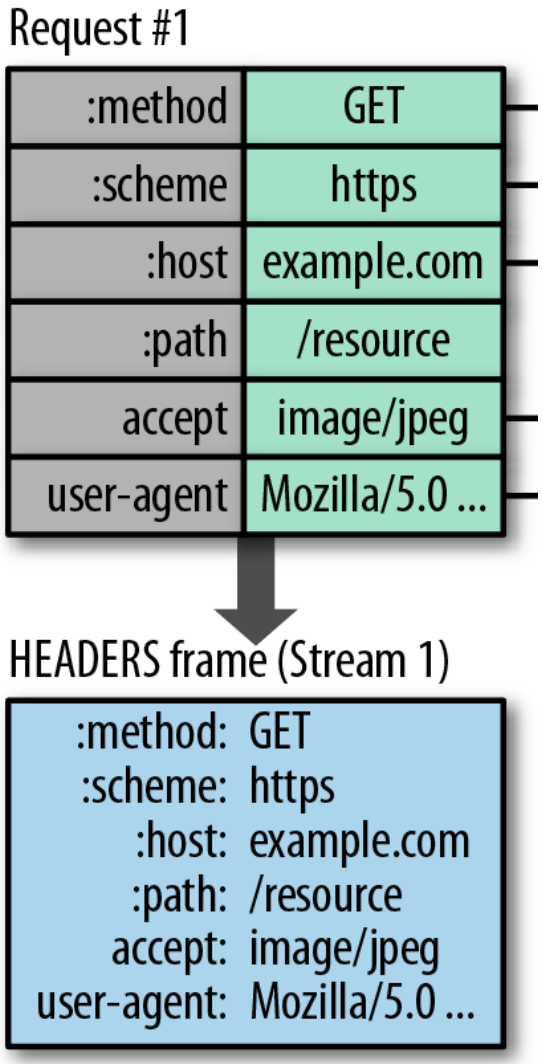
- **Frames are interleaved**
  - Frames are prioritized
  - Frames are flow controlled

Server push... is replacing inlining



Inlining is server push. *Except,* ***HTTP 2.0 server push is cacheable!***

HTTP/2 header compression



- *Both sides maintain "header tables"*
- *New requests "toggle" or "insert" new values into the table*

Byte cost of new stream: ***9 bytes! ****

** as low as 9 bytes for an identical request.*

- `min(request overhead)` = 9 bytes
- `max(parallelism)` = 100~1000+ streams
- `max(client queueing latency)` = 0 ms

But you already knew all that!

The more interesting part is how it changes web development...

Remove domain sharding for HTTP/2

Sharding hurts HTTP/2 performance

- *Breaks prioritization, flow control, etc.*

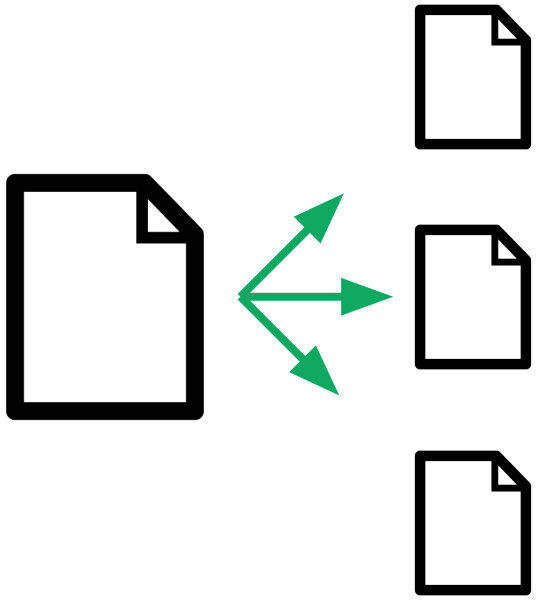
```
$> openssl s_client -connect google.com:443 |
     openssl x509 -noout -text |
     grep DNS
DNS:*.google.com, DNS:*.android.com, DNS:*.appengine.google.com, ...
```

Tip: use altName hosts to deploy domain sharding! *

- *HTTP/1.1 → opens new connection to each origin*
- *HTTP/2 → reuses the same connection for altName origins*

** Origin must be covered by the cert and resolve to same IP*

Remove spriting / concatenation logic...

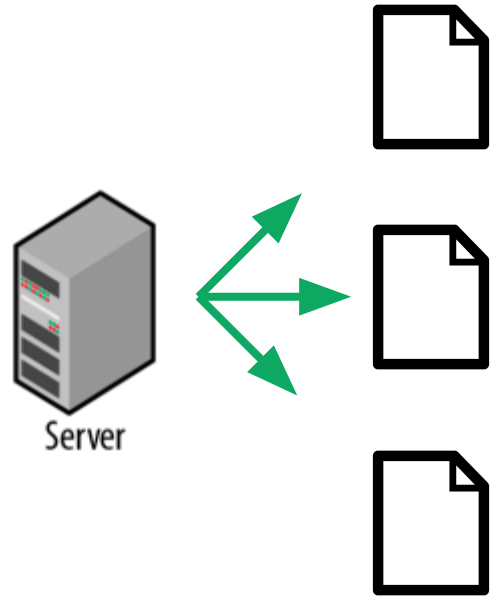


Streams are cheap, and no longer a constraint.

- **Deliver modular resources**
  - *aim to minimize resource churn*
  - *define granular caching strategy for each*
- **Conditional delivery based on protocol?**
  - *Combine for HTTP/1.1 clients **
  - *Granular resources for HTTP/2 clients*

** Need better tools / infrastructure to do conditional delivery*

Leverage server push instead of inlining...



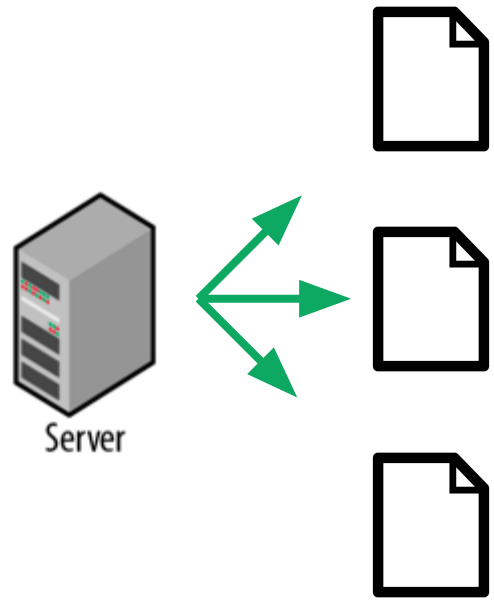
Server can respond with multiple replies!

- **Client** → I want /product/xyz
- **Server** → Ok, and you'll also need... style.css

- Pushed resource is cached independently
  - Use "smart push", don't push on every request
- Can remove RTT+ from critical path

- **Push... cache invalidations!**
  - push a "tombstone" record to invalidate

Jetty's "smart push" is a great strategy...



1. **Server observes incoming traffic**
   a. *Build a dependency model based on Referer*
   b. *e.g. index.html → {style.css, app.js}*

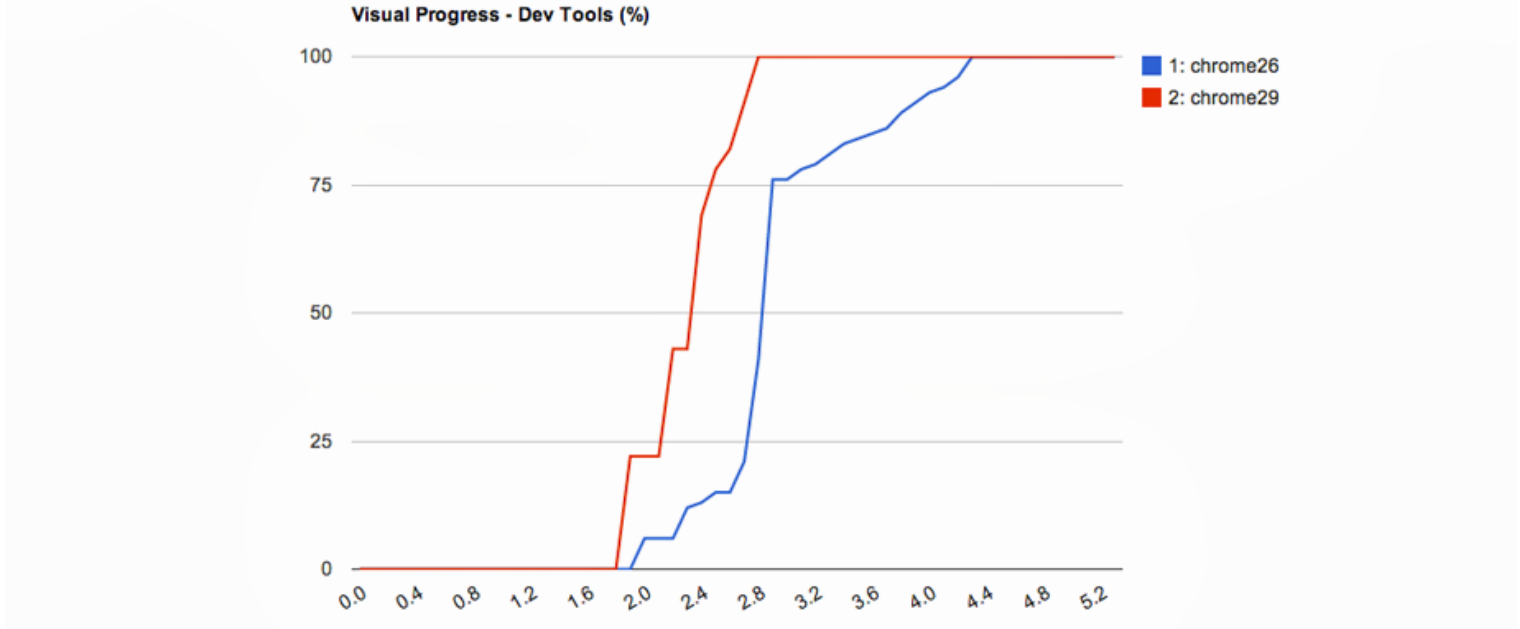
2. **Server initiates push for learned dependencies**
   a. *new client → GET index.html*
   b. *server → Push style.css, app.js*

Lots of room for experimentation + innovation!

*Servers need to be *much* smarter*

client is relinquishing a lot of control, badly implemented server → poor performance

Chrome 28+ does not delay stream dispatch (yay)



*"**Don't delay low priority requests when SPDY is available.** Check if the origin server supports SPDY. If so,*
start the request immediately."

Eliminates client queuing latency. Means the server must be smart about respecting client priorities!

SPDY resource scheduling

Prioritization is key to optimized rendering...

With HTTP/1.1 browsers held back requests… not with HTTP/2.

- *GET index.html, style.css, hero.jpg, other.jpg, more.jpg, …*

critical **low priority**

Critical resources should pre-empt others

- *Poorly implemented server: saturate the pipe with static image bytes!*
  - *e.g. SPDY/v2 implementation in nginx did not respect prioritization, and performance suffered… test your server!*

Smart++ server can optimize for each content type!

Don’t hold back all image bytes... send the first ~KB

- *Allows the browser to decode the image header and get dimensions*
- *Allows the browser to minimize reflows during layout*

Stream flow-control enables **fine-grained resource control** between streams. E.g…

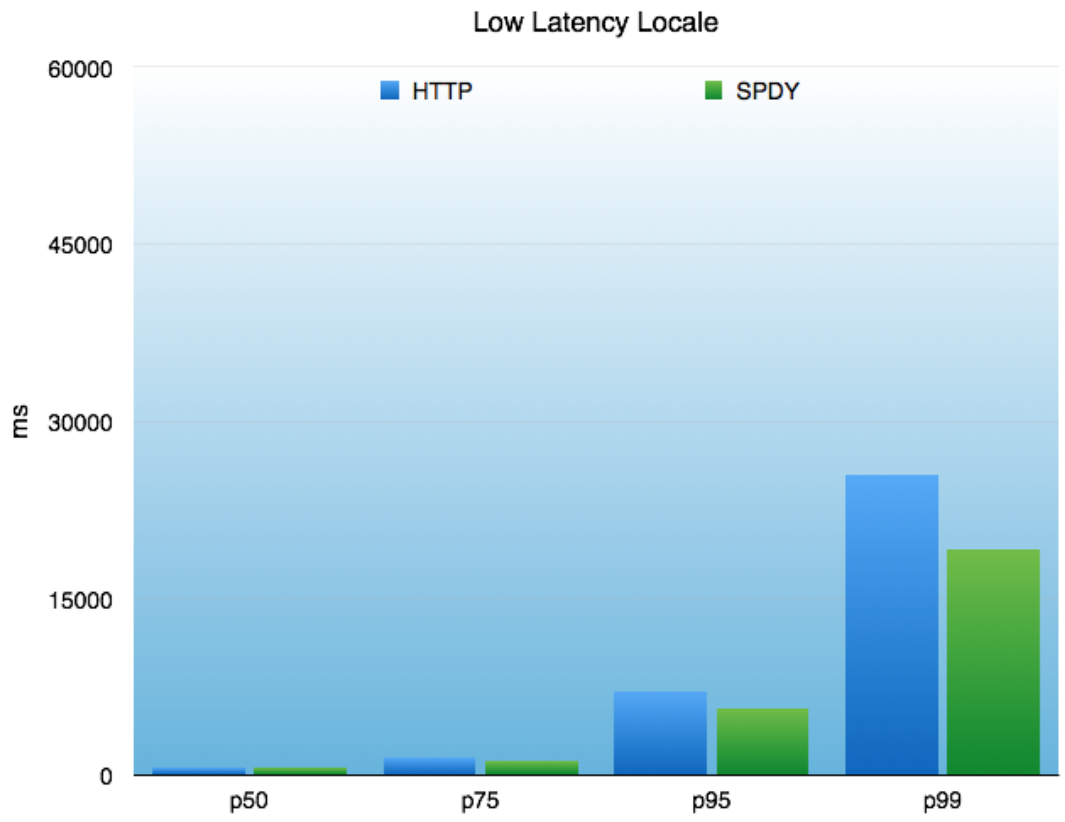
- **T(0):** I am willing to receive **4KB** of kittens.jpg.
- **T(0):** I am willing to receive **500KB** of critical.js
- …
- **T(n):** Ok, now send the **remainder** of kittens.jpg.

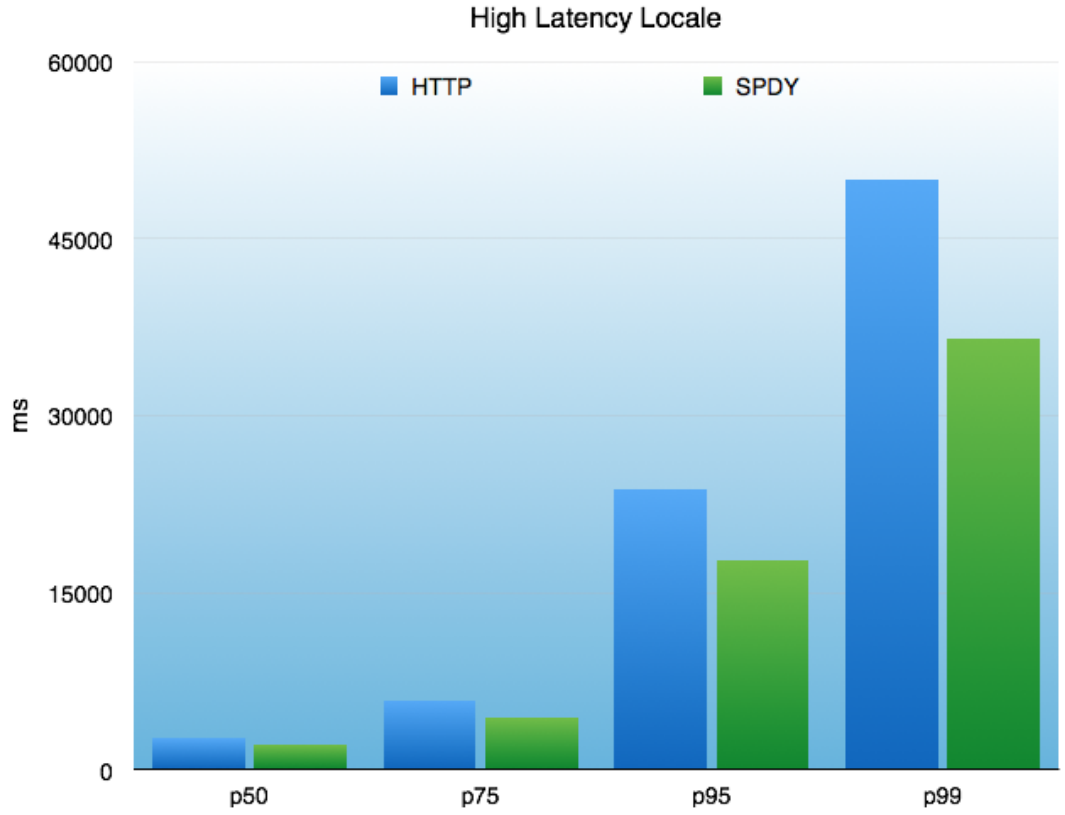
Client controls how and when the stream and connection window is incremented!

Real-world performance...

Your gains will vary based on site architecture, server, clients, ...

SPDY for API traffic @ Twitter





*"However, **we have measured as much as a 30% decrease in latency** in the wild for API requests carried over SPDY relative to those carried over HTTP. In particular, **we've observed SPDY helping more as a user's network conditions get worse**." - Twitter*

https://blog.twitter.com/2013/cocoaspdy-spdy-for-ios-os-x

HTTP/2 and SPDY

	Page load time improvement with SPDY enabled...			
	Google News	**Google Sites**	**Google Drive**	**Google Maps**
Median	**43%**	**27%**	**23%**	**24%**
95th percentile	**44%**	**33%**	**36%**	**28%**

Improvement over HTTP/1.1 + TLS

http://blog.chromium.org/2013/11/making-web-faster-with-spdy-and-http2.html

Fewer connections means that...

*"**SPDY also has advantages on the server:***

*SPDY requests **consume less resources** on the server*
*SPDY requests **consume less memory** but a bit more CPU*
*SPDY requires **fewer Apache worker threads**"*

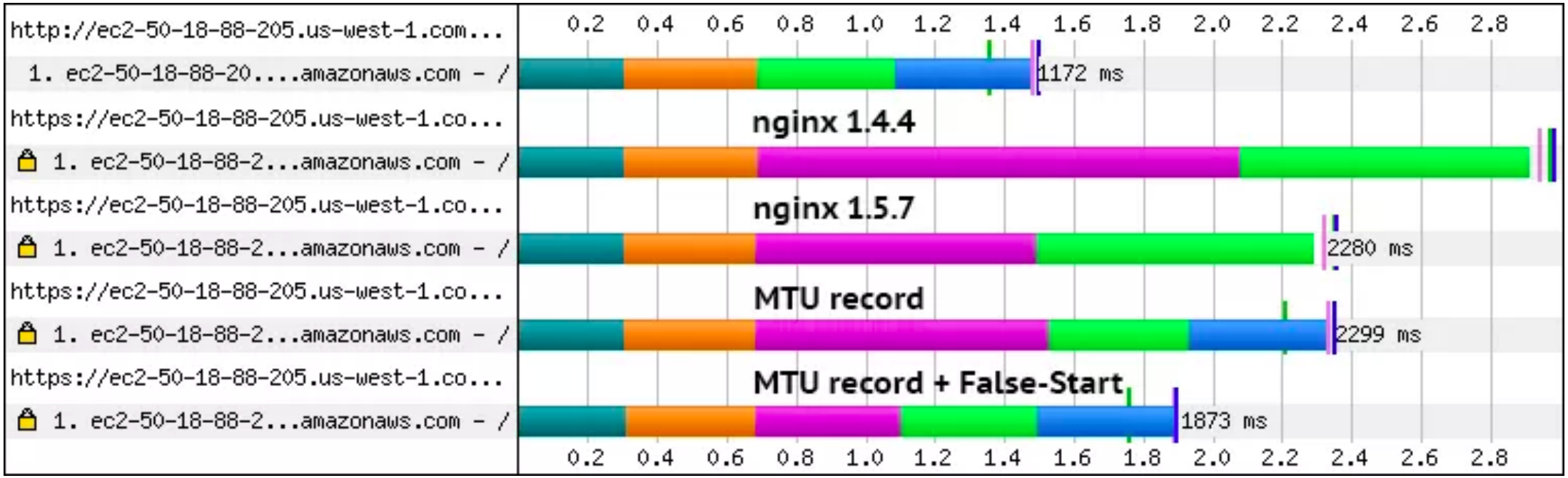
Hervé Servy, Neotys.

s/SPDY/HTTP2/g ... same results.

Speaking of TLS...

make sure your TLS stack is optimized!

Tuning Nginx TLS Time To First Byte (TTTFB)



- Pre 1.5.7: bug for 4KB+ certs, resulting in 3RTT+ handshakes
- 1.7.1 added ssl_buffer_size: 4KB record size remove an RTT
- 1.7.1 with NPN and forward secrecy → **1RTT handshake**

https://www.igvita.com/2013/12/16/optimizing-nginx-tls-time-to-first-byte/

	Session identifiers	Session tickets	OCSP stapling	Dynamic record sizing	ALPN / NPN	Forward secrecy	SPDY & HTTP/2
Apache	yes	yes	yes	no	no*	yes	mod_spdy
ATS	yes	yes	no	static	yes	yes	spdy/3.1
HAProxy	yes	yes	no	dynamic	yes	yes	no
IIS	yes	yes	yes	no	yes	yes	no
NGINX	yes	yes	yes	static	yes	yes	spdy/3.1
bud	no	yes	yes	static	yes	yes	no

- “Out of the box” TLS performance is poor… we need to fix this.
- No server is perfect, plenty of work to be done to improve perf.

	Session identifiers	Session tickets	OCSP stapling	Dynamic record sizing	ALPN / NPN	Forward secrecy	SPDY & HTTP/2
Akamai	yes	yes	yes	configurable (static)	yes	yes	spdy/3 spdy3.1 (opt-in)
CloudFlare	yes	yes	yes	4KB (static)	yes	yes	spdy/3.1
AWS ELB	yes	yes	no	no	no	yes	no
AWS CloudFront	no	yes	no	no	no	no	no
EdgeCast	no	no	yes	no	no	yes	no
Fastly	yes	yes	yes	no	no	yes	no
Google App Engine	yes	yes	no	dynamic	yes	yes	spdy/3.1
Heroku	yes	no	no	no	no	no	no
Limelight	yes	yes	no	no	no	yes	no
MaxCDN	yes	yes	no	no	yes	no	spdy/3.1

There is way too much red here… Bug your CDN about fixing this!

An optimized TLS deployment should...

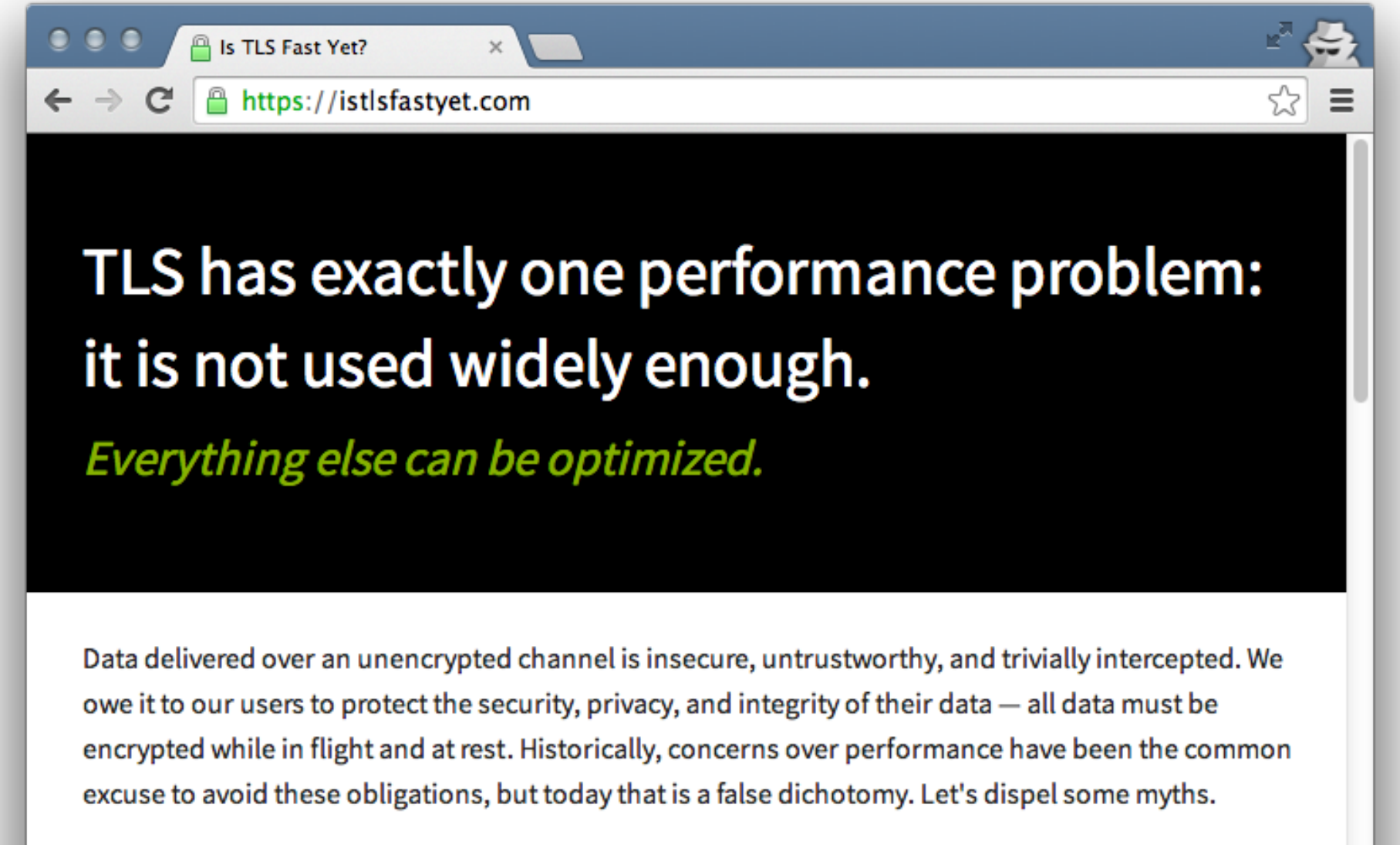
Deliver 1-RTT handshake 100% of the time

1. TLS False Start for new visitors
2. TLS resumption for returning visitors
3. Ensure that server is able to send full cert chain without blocking
4. OCSP stapling to avoid blocking

Optimize data delivery

1. Optimize record size to avoid unnecessary buffering delays
2. Leverage SPDY / HTTP/2 to further reduce latency and ops costs
    a. Leverage HTTP/2 optimizations: unshard, un-concat, etc

isTLSfastyet.com



Where to from here?

necessary steps to make HTTP/2 ubiquitous

Browser support is there, or coming soon...

- **Chrome M39 is shipping HTTP/2 (draft 14)**
  - *Coming in next stable release! Available in Canary today.*
  - *Google servers are also speaking HTTP/2*
- **Firefox 34 is shipping HTTP/2 (draft 14)**
  - *Coming in next stable release!*
- **IE supports HTTP/2 on Windows 10 Technical Preview**
  - *In the meantime, IE also supports SPDY v3*
- **Latest Safari suports SPDY v3**
  - *No official HTTP/2 announcements, but… I'm sure its coming.*

Wait, what about SPDY?

- SPDY was “experimental branch” of HTTP/2
- SPDY will be phased out now that we have HTTP/2
  - *All future and further work will be done within HTTP/2*

Server support is coming along as well...

- **nghttp2 is awesome!**
  - *Lots of projects built on top of nghttp2*
  - *Need to test TLS performance though.. :)*
- **Native Java, C#, Objective-C, Go, Python, Ruby, Erlang libraries**
  - *https://github.com/http2/http2-spec/wiki/Implementations*
- **Apache and Nginx are both WIP**
  - *No ETA for either project as of today*
  - *Looking for a good project to contribute to?*

tl;dr...

Site owners & developers

1. Remove sharding, concatenation, spriting...
2. Test your HTTP/2 server: prioritization, server push, etc.
3. Optimize your TLS deployment

Server & library developers

1. Respect prioritization and dependency hints
    a. *Aside: we need better server tests - QPS is not a good metric!*
2. Build smarter models: server push, content-type optimizations, etc.
3. Nudge / contribute HTTP/2 support in your favorite project / language

Slides
bit.ly/1rOWzXj

Thanks!

+Ilya Grigorik
@igrigorik