HTTP 2.0

Why now? What is it?

Ilya Grigorik - @igrigorik
Web Performance Engineer
Google

HTTPbis Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 22, 2014

M. Belshe
Twist
R. Peon
Google, Inc
M. Thomson, Ed.
Microsoft
A. Melnikov, Ed.
Isode Ltd

August 21, 2013

Hypertext Transfer Protocol version 2.0
draft-ietf-httpbis-http2-06

Abstract

This specification describes an optimized expression of the syntax of the Hypertext Transfer Protocol (HTTP). The HTTP/2.0 encapsulation enables more efficient use of network resources and reduced perception of latency by allowing header field compression and multiple concurrent messages on the same connection. It also introduces unsolicited push of representations from servers to clients.

This document is an alternative to, but does not obsolete the HTTP/1.1 message format or protocol. HTTP's existing semantics remain unchanged.

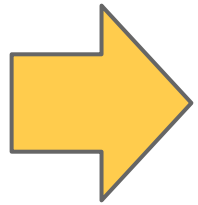
This version of the draft has been marked for implementation. Interoperability testing will occur in the HTTP/2.0 interim in Seatle, US, starting 2013-10-09.

*"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 2.0 goals

Delay	User reaction
0 - 100 ms	Instant
100 - 300 ms	Slight perceptible delay
300 - 1000 ms	Task focus, perceptible delay
1 s+	Mental context switch
10 s+	I'll come back later...

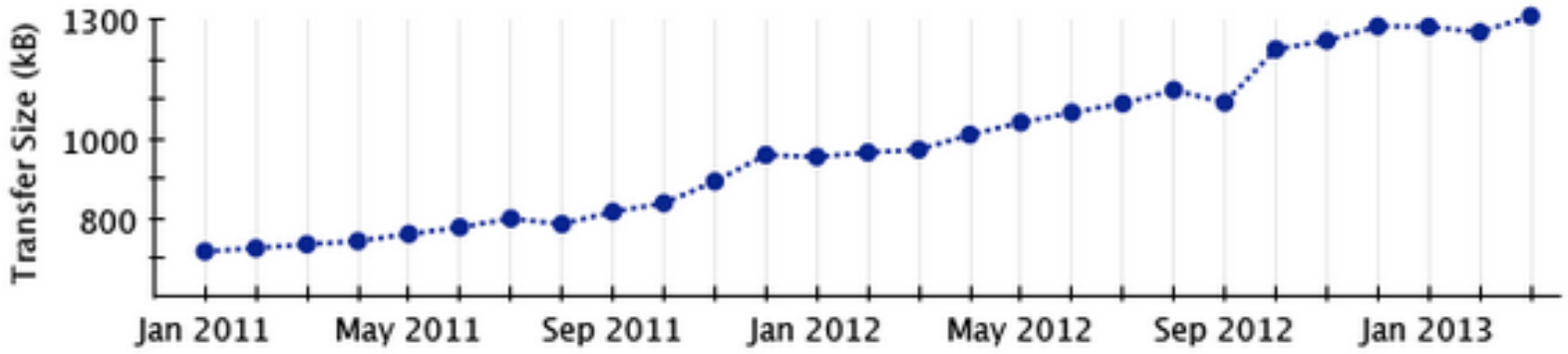


"1000 ms time to glass challenge"

- *Simple user-input must be acknowledged within ~100 milliseconds.*
- *To keep the user engaged, the task must complete within 1000 milliseconds.*

Ergo, our pages should render within 1000 milliseconds.

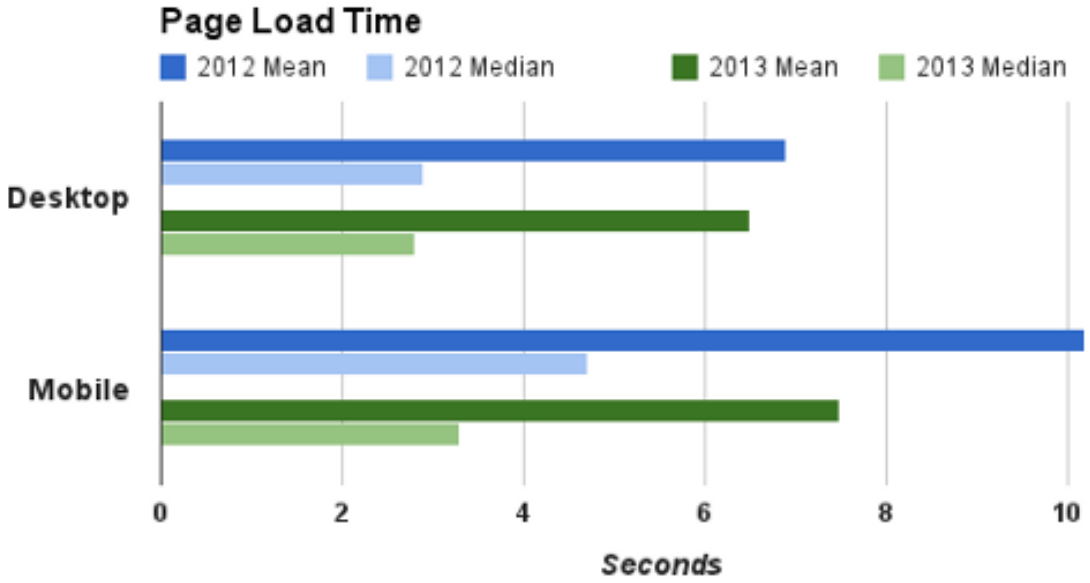
Our applications are complex, and growing...

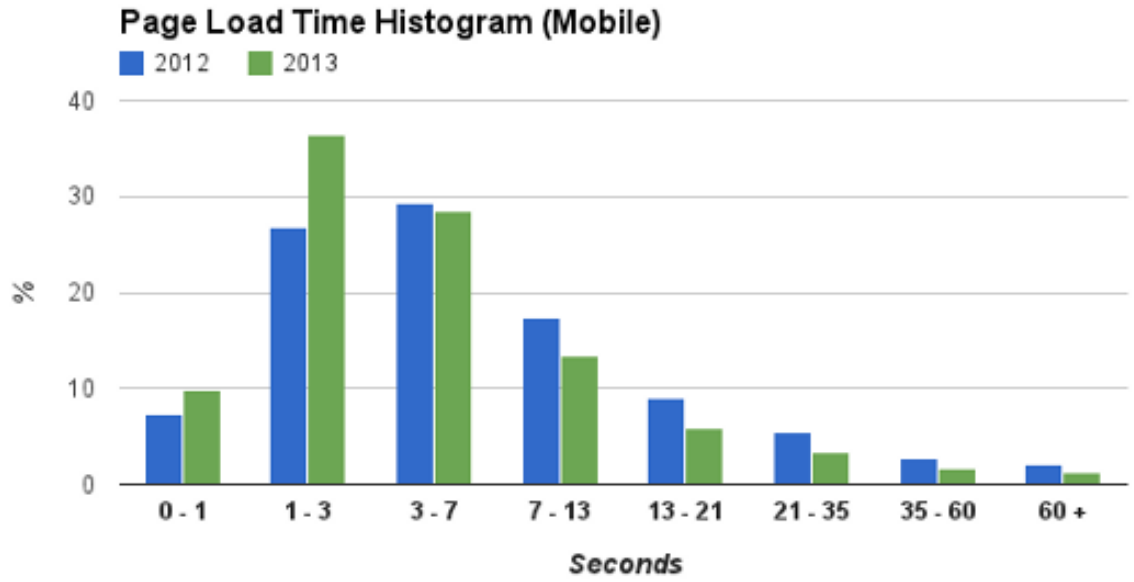


Content Type	Desktop		Mobile	
	Avg # of requests	Avg size	Avg # of requests	Avg size
HTML	10	56 KB	6	40 KB
Images	**56**	**856 KB**	**38**	**498 KB**
Javascript	**15**	**221 KB**	**10**	**146 KB**
CSS	5	36 KB	3	27 KB
Total	**86+**	**1169+ KB**	**57+**	**711+ KB**

Ouch!

HTTP Archive





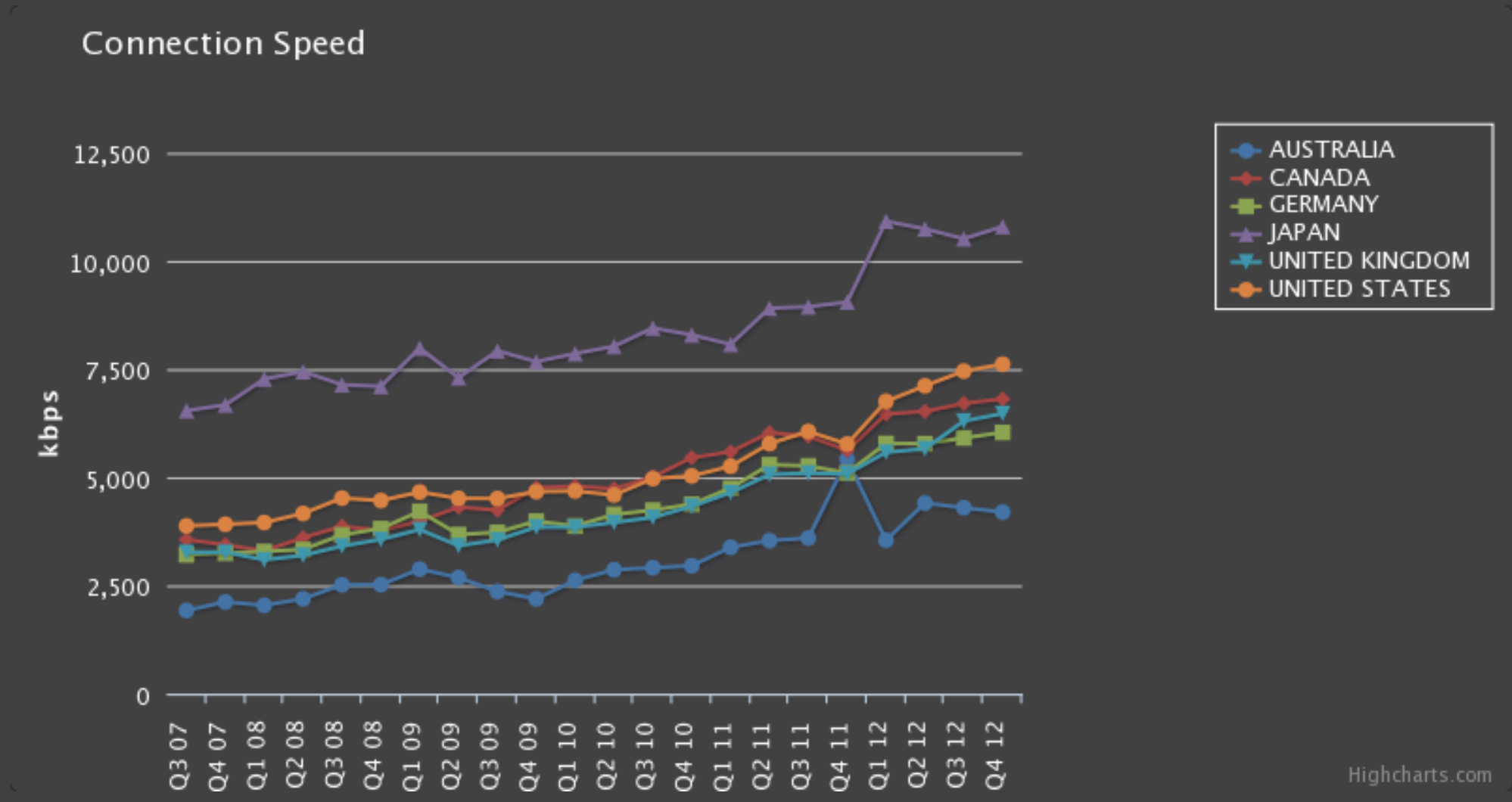
Desktop: ~3.1 s
Mobile: ~3.5 s

"It's great to see access from mobile is around 30% faster compared to last year."

[Is the web getting faster?](#) - Google Analytics Blog

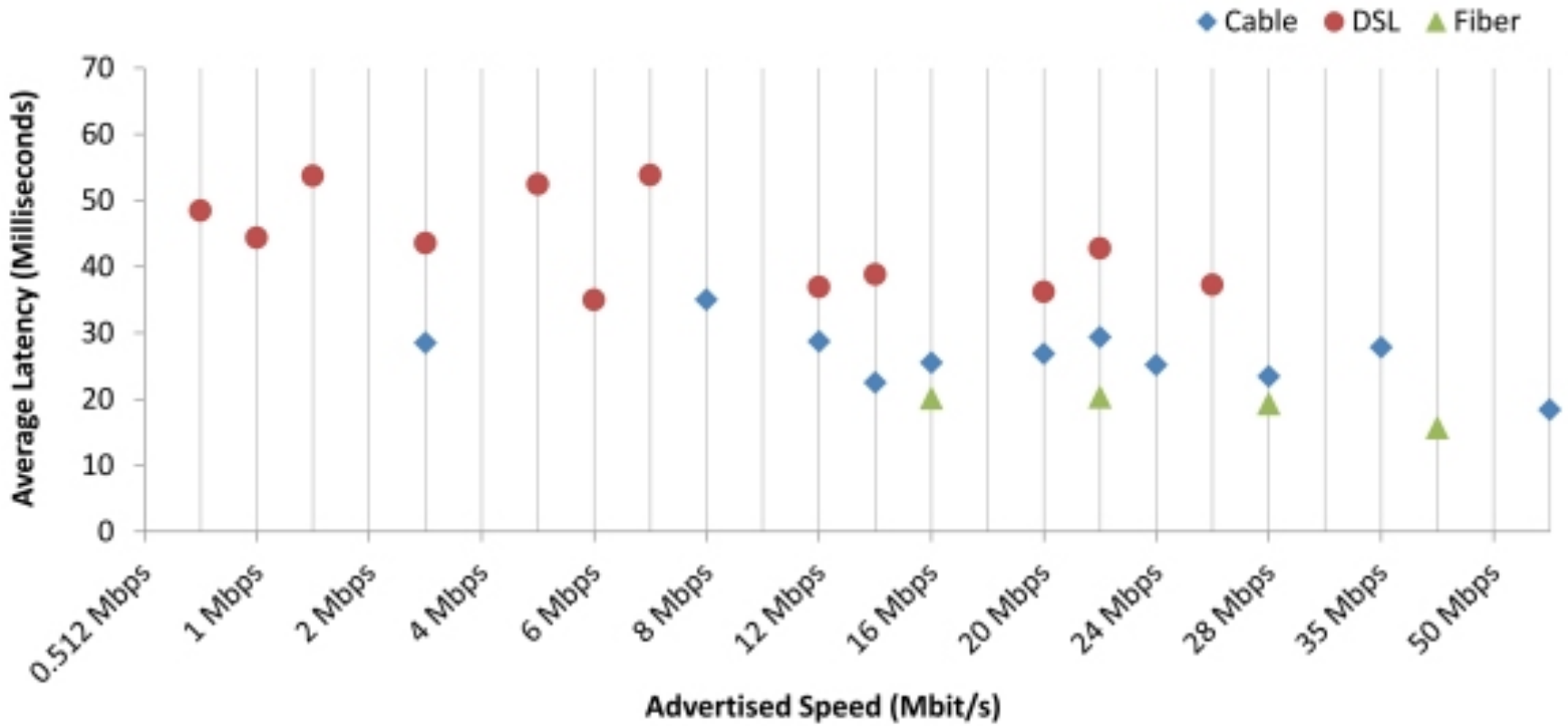
Great, network will save us?

Right, right? We can just sit back and…



Average connection speed in Q4 2012: ***5000 kbps+***

State of the Internet - Akamai - 2007-2012



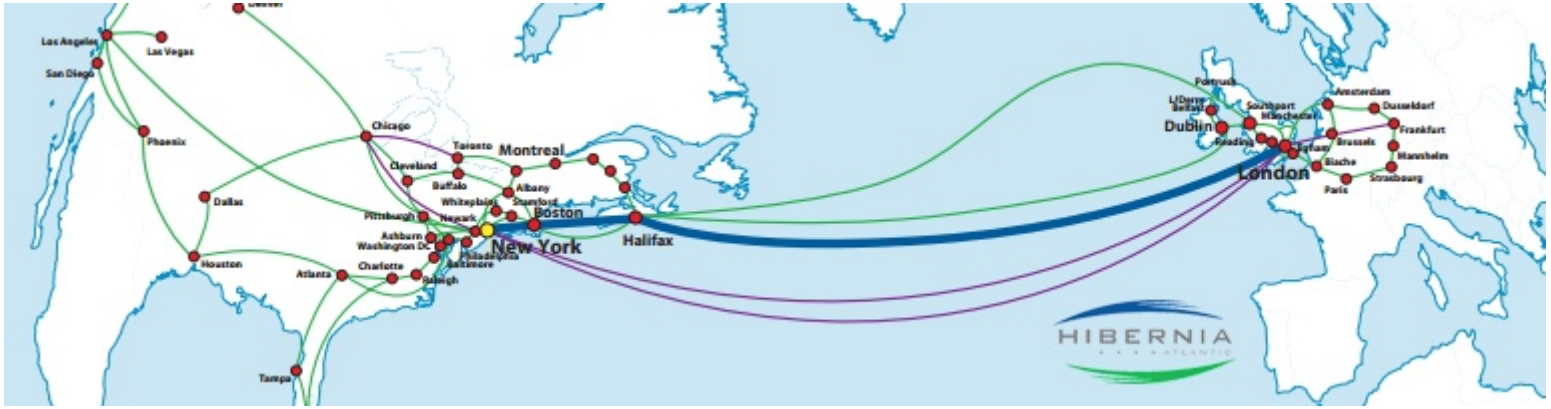
Fiber-to-the-home *services provided* ***18 ms*** *round-trip latency on average, while* ***cable-based*** *services averaged* ***26 ms****, and* ***DSL-based*** *services averaged* ***43 ms****. This compares to 2011 figures of 17 ms for fiber, 28 ms for cable and 44 ms for DSL.*

Measuring Broadband America - July 2012 - FCC

Worldwide: ~100 ms
US: ~50~60 ms

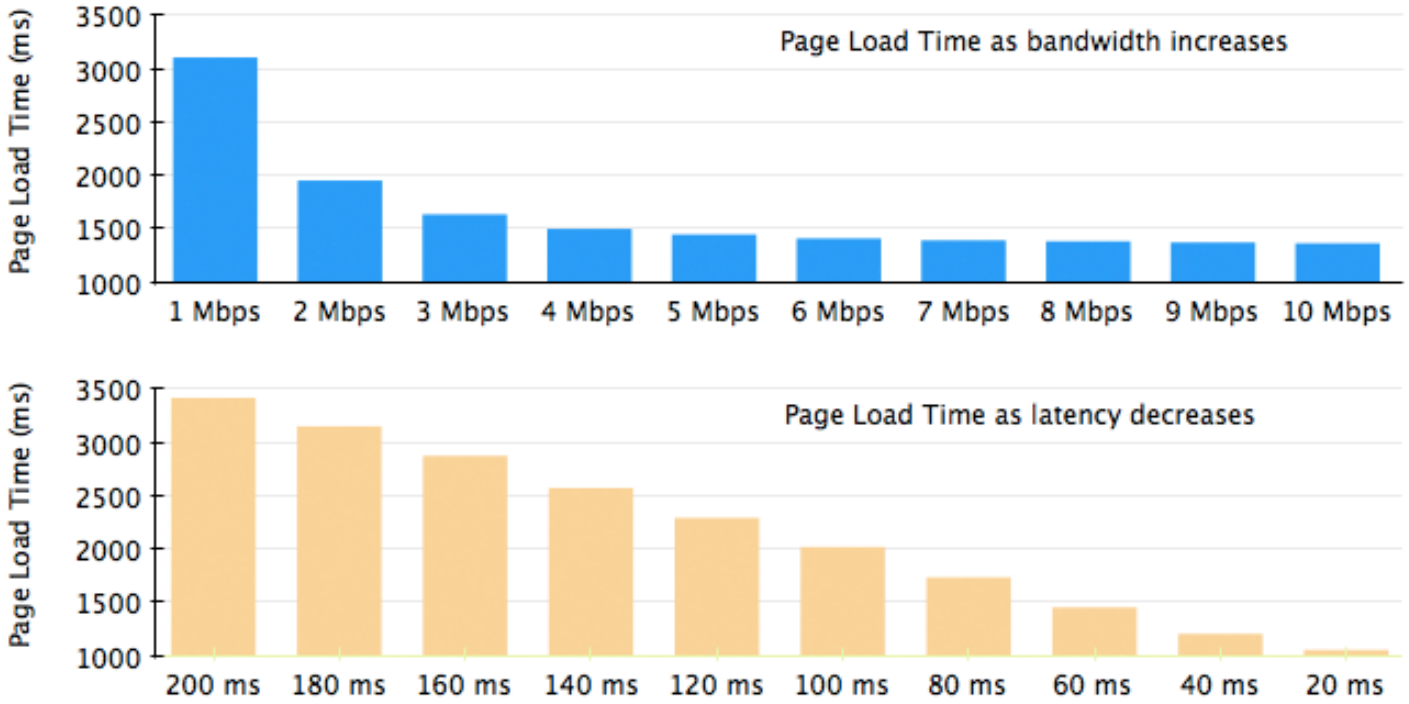
Average RTT to Google in 2012 was...

- **Improving bandwidth is "easy"...**
  - 60% of new capacity through upgrades in past decade + unlit fiber
  - *"Just lay more fiber..."*
- **Improving latency is expensive... impossible?**
  - Bounded by the speed of light - oops!
  - We're already within a small constant factor of the maximum
  - *"Shorter cables?"*



$80M / ms

Latency *vs.* Bandwidth impact on Page Load Time



Single digit perf improvement after 5 Mbps

Average household in is running on a **5 Mbps+** connection. Ergo, **average consumer would not see an improvement in page loading time by upgrading their connection.** (doh!)

[Bandwidth doesn't matter (much)] - Google

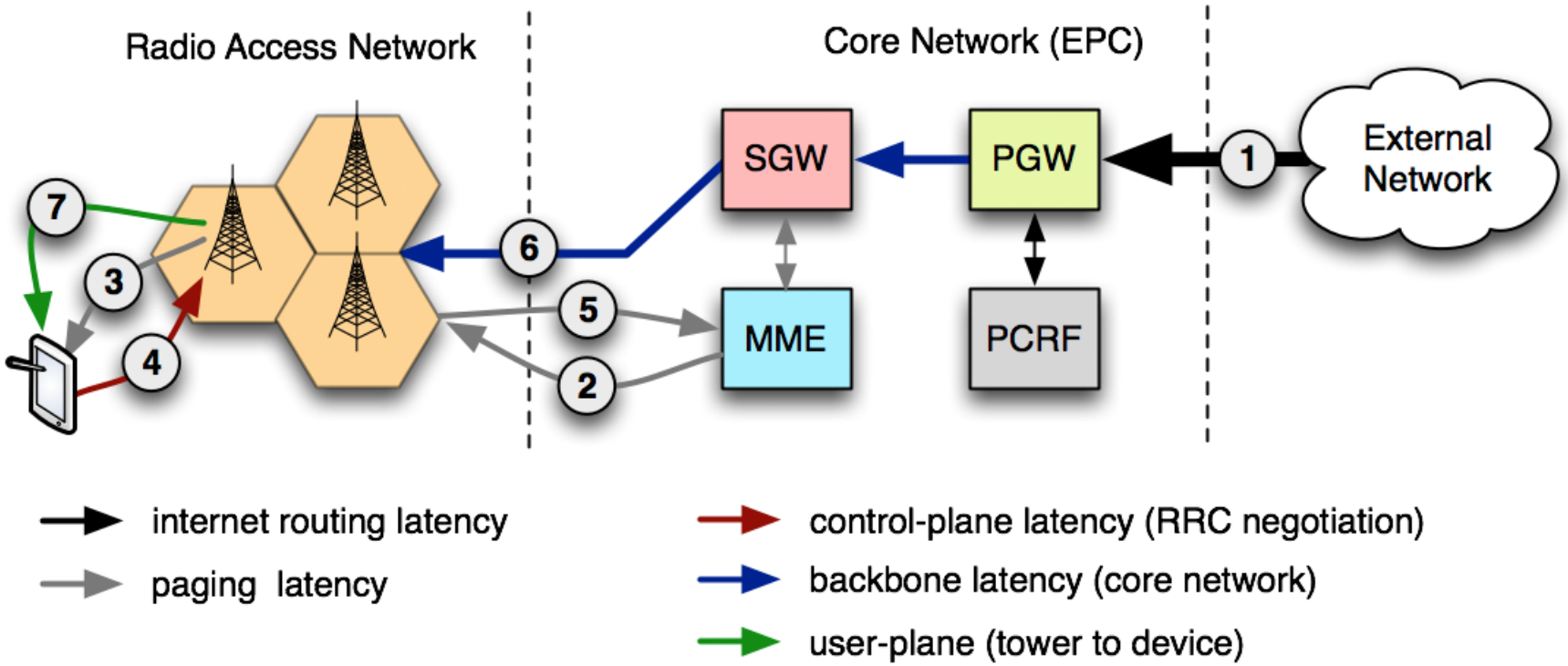
Bandwidth doesn't matter *(much)**

(for web browsing)

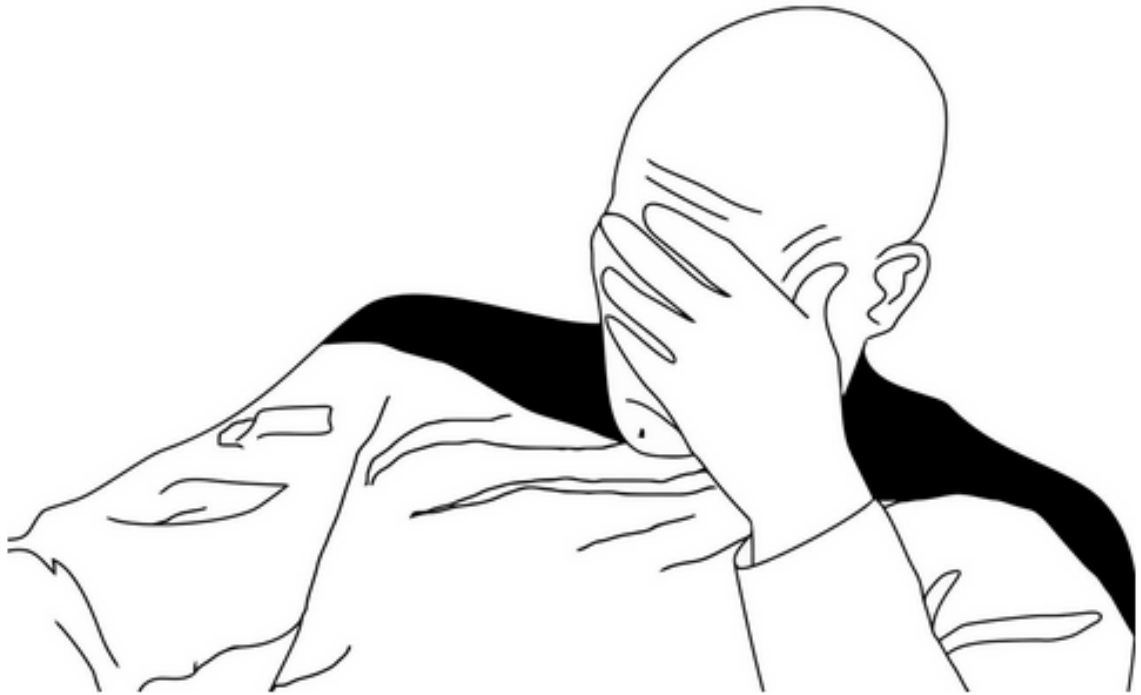
And then there’s mobile...

Variable downloads speeds, spikes in latency... but why?

Inbound packet flow



	LTE	HSPA+	HSPA	EDGE	GPRS
AT&T core network latency	40-50 ms	50-200 ms	150-400 ms	600-750 ms	600-750 ms



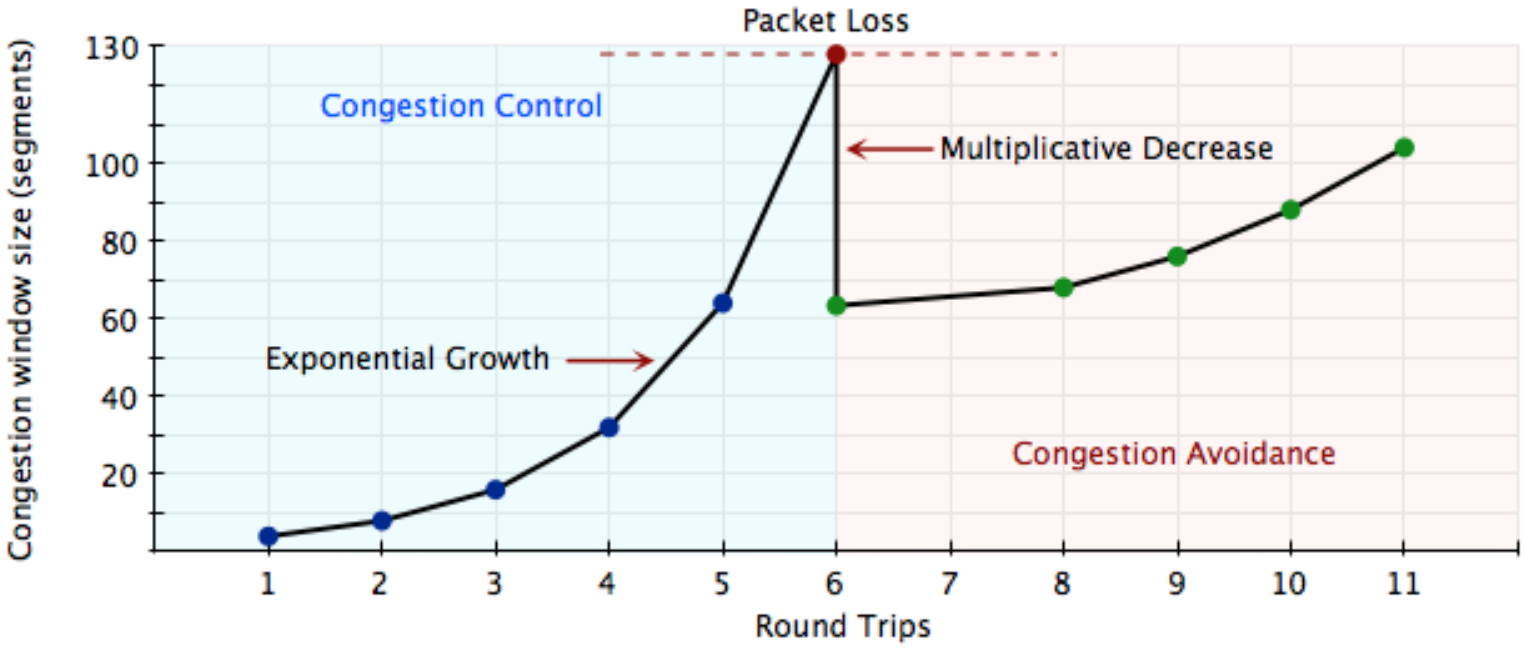
*... all that to send a **single TCP packet?***

OK. Latency is a problem.

But, how does it affect HTTP and web browsing in general?

TCP Congestion Control & Avoidance...

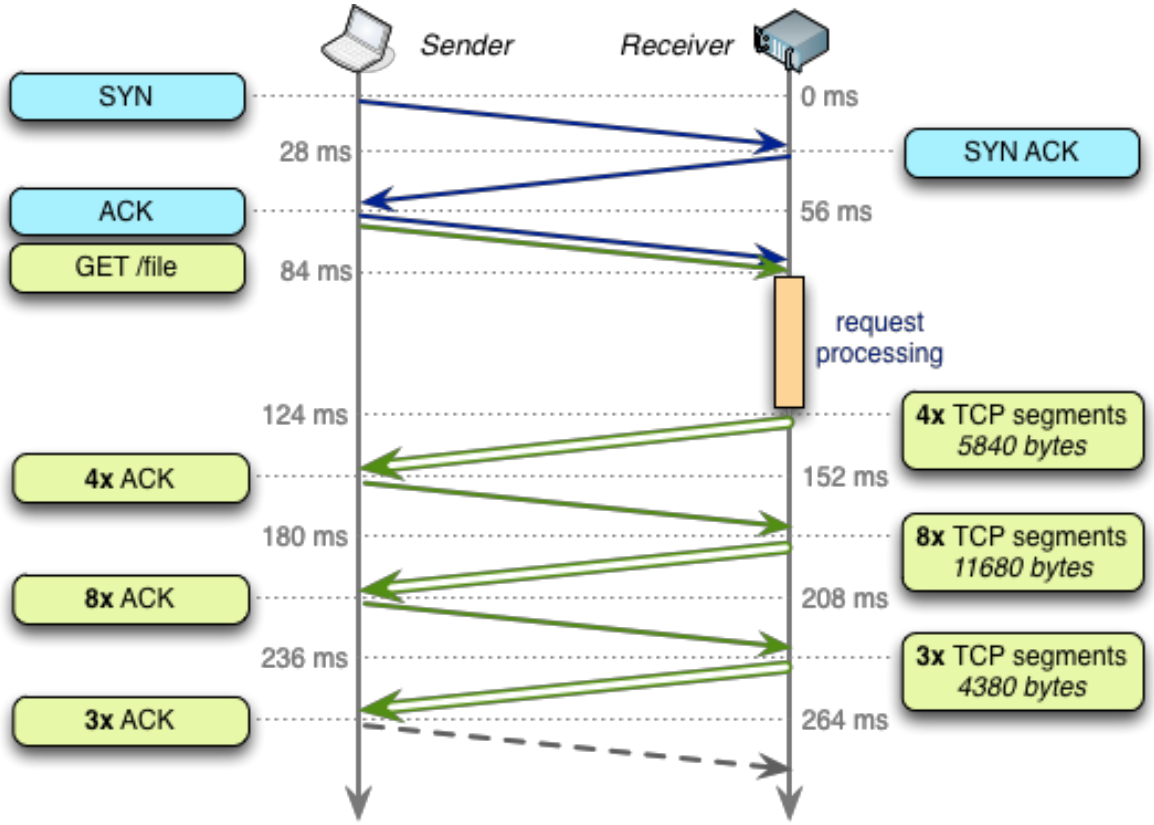
- TCP is designed to probe the network to figure out the available capacity
- TCP **does not** use full bandwidth capacity from the start!



TCP Slow Start is a feature, not a bug.

[Congestion Avoidance and Control]

Let's fetch a 20 KB file via a low-latency link (IW4)...



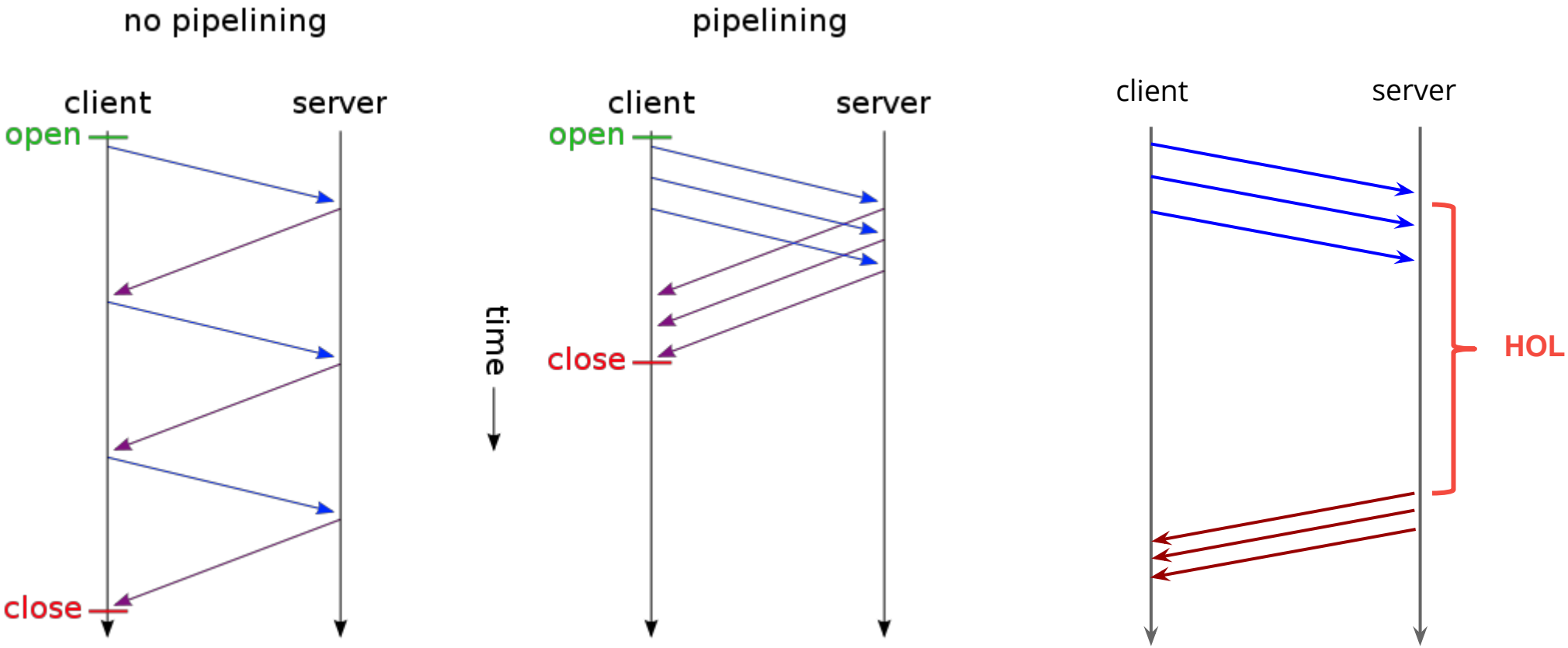
- **5 Mbps** connection
- **56 ms** roundtrip time (NYC > London)
- **40 ms** server processing time

4 roundtrips, or 264 ms!

Plus DNS and TLS roundtrips

[Congestion Avoidance and Control]

HTTP does not support multiplexing!



- **No pipelining:** request queuing
- **Pipelining*:** response queuing

- **Head of Line blocking**
  - It's a guessing game...
  - Should I wait, or should I pipeline?

Lets just open multiple TCP connections! Easy, right..?

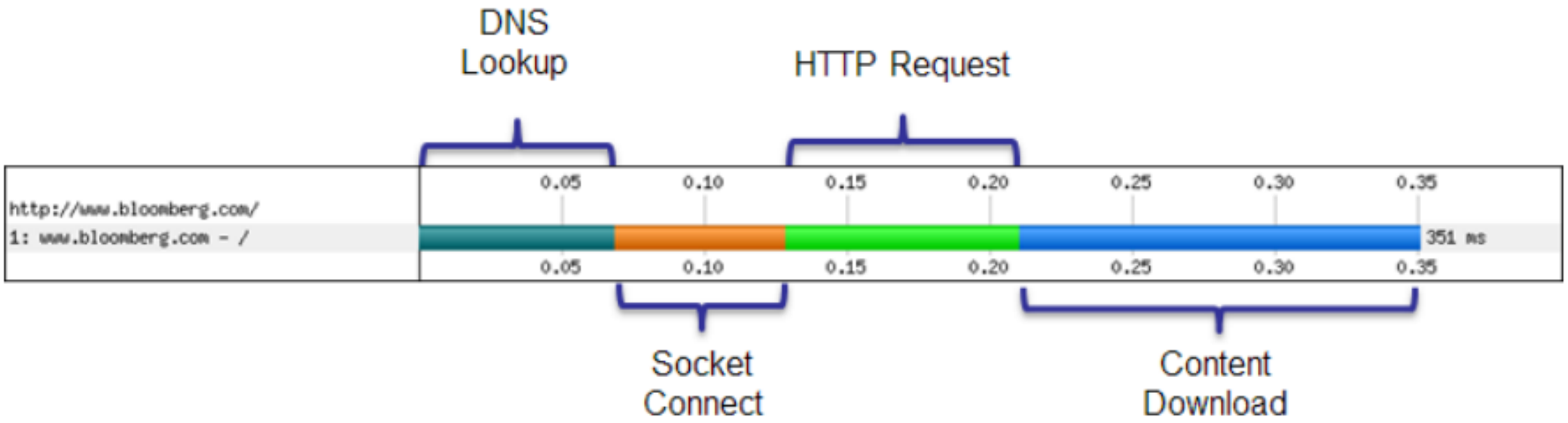
Top Desktop			
name	score	PerfTiming	Connections per Hostname
Chrome 20 →	12/16	yes	6
Firefox 14 →	13/16	yes	6
IE 8 →	7/16	no	6
IE 9 →	12/16	yes	6
Opera 12 →	10/16	no	6
RockMelt 0.9 →	13/16	yes	6
Safari 5.1 →	12/16	no	6

Top Mobile			
name	score	PerfTiming	Connections per Hostname
Android 2.3 →	8/16	no	9
Android 4 →	13/16	yes	6
Blackberry 7 →	11/16	no	5
Chrome Mobile 16 →	13/16	yes	6
IEMobile 9 →	11/16	yes	6
iPhone 4 →	10/16	no	4
iPhone 5 →	10/16	no	6
Nokia 950 →			
Opera Mobile 12 →	11/16	no	8

- **6 connections per host** on Desktop
- **6 connections per host** on Mobile (recent builds)

So what, what's the big deal?

The *(short)* life of a web request

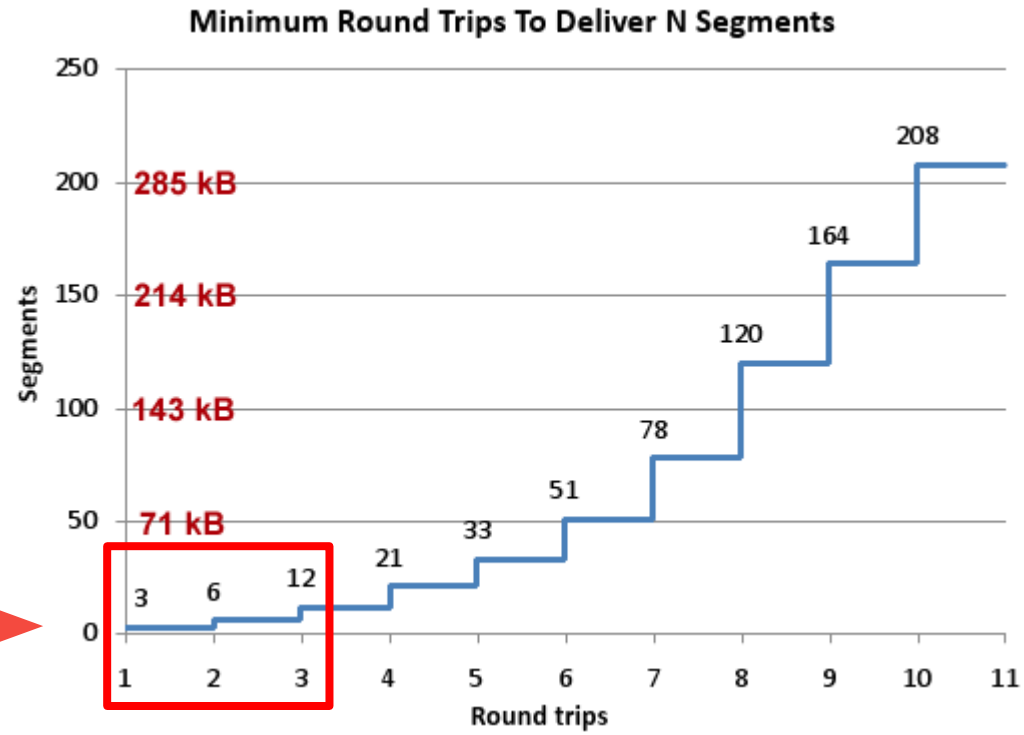


- *(Worst case)* **DNS lookup** to resolve the hostname to IP address
- *(Worst case)* **New TCP connection**, requiring a full roundtrip to the server
- *(Worst case)* **TLS handshake** with up to two extra server roundtrips!

- **HTTP request**, requiring a full roundtrip to the server
- **Server processing time**

HTTP Archive says...

- 1169 KB, 86 requests, ~15 hosts... **or ~14 KB per request!**
- Most HTTP traffic is composed of small, bursty, TCP flows.

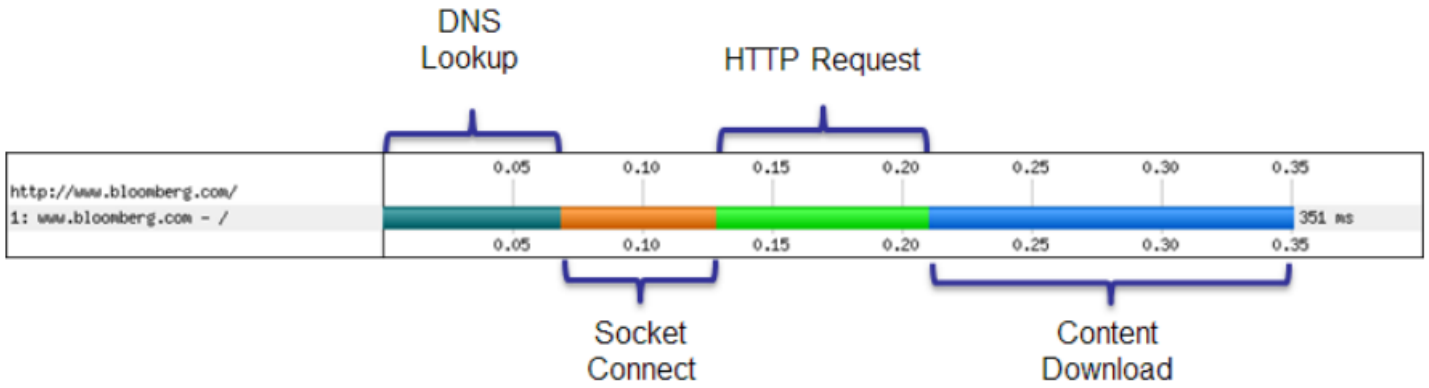


Where we want to be

You are here

1-3 RTT's

Let's fetch a 20 KB file via a 3G / 4G link...



	3G *(200 ms RTT)*	**4G** *(100 ms RTT)*
Control plane	*(200-2500 ms)*	*(50-100 ms)*
DNS lookup	200 ms	100 ms
TCP Connection	200 ms	100 ms
TLS handshake (optional)	(200-400 ms)	(100-200 ms)
HTTP request	200 ms	100 ms
Total time	**800 - 4100 ms**	**400 - 900 ms**

x4 (slow start)

One 20 KB HTTP request!

[Anticipate network latency overhead]

An Argument for Increasing TCP's Initial Congestion Window

Nandita Dukkipati Tiziana Refice Yuchung Cheng Jerry Chu Natalia Sutin
Amit Agarwal Tom Herbert Arvind Jain
Google Inc.
{nanditad, tiziana, ycheng, hkchu, nsutin, aagarwal, therbert, arvind}@google.com

ABSTRACT

TCP flows start with an initial congestion window of at most three segments or about 4KB of data. Because most Web transactions are short-lived, the initial congestion window is

for standard Ethernet MTUs (approximately 4KB) [5]. The majority of connections on the Web are short-lived and finish before exiting the slow start phase, making TCP's initial congestion window (*init_cwnd*) a crucial parameter in deter-

Updates CWND from 3 to 10 segments, or ~14960 bytes.

Default size on Linux 2.6.33+, but upgrade to 3.2+ for best performance.

Web pages. Popular Web browsers, including IE8 [2], Fire-

When there's a will, there is a way...

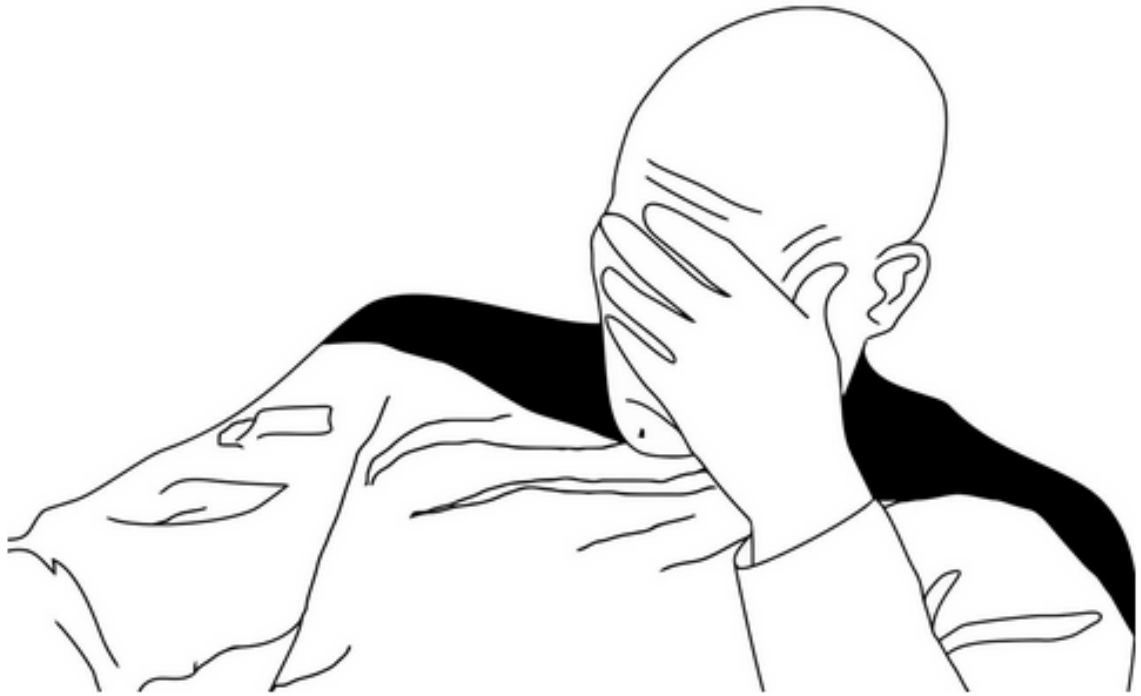
*web developers are an inventive bunch, so we came up with some **"optimizations"***

- **Concatenating files (JavaScript, CSS)**
  - Reduces number of downloads and latency overhead
  - Less modular code and expensive cache invalidations (e.g. app.js)
  - Slower execution (must wait for entire file to arrive)

- **Spriting images**
  - Reduces number of downloads and latency overhead
  - Painful and annoying preprocessing and expensive cache invalidations
  - Have to decode entire sprite bitmap - CPU time and memory

- **Domain sharding**
  - TCP Slow Start? Browser limits, Nah... 15+ parallel requests -- *Yeehaw!!!*
  - Causes congestion and unnecessary latency and retransmissions

- **Resource inlining**
  - Eliminates the request for small resources
  - Resource can't be cached, inflates parent document
  - 30% overhead on base64 encoding



*... why not **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

*(hopefully now you're convinced we **really** need it)*

Brief history of HTTP 2.0...

1. **Jan 2012** Call for Proposals for HTTP/2.0
2. **Oct 2012** First draft of HTTP/2.0, based on draft-mbelshe-httpbis-spdy-00
3. **Jul 2013** **First "implementation" draft (04) of HTTP 2.0**
4. Apr 2014 Working Group Last call for HTTP/2.0
5. Nov 2014 Submit HTTP/2.0 to IESG for consideration as a Proposed Standard

Earlier this month... interop testing in Hamburg!

- ALPN patch landed for OpenSSL
- Firefox implementation of 04 draft
- Chrome implementation of 04 draft
- Google GFE implementation of 04 draft (server)
- Twitter implementation of 04 draft (server)
- Microsoft (Katana) implementation of 04 draft (server)
- Perl, C#, node.js, Java, Ruby, ... and more.

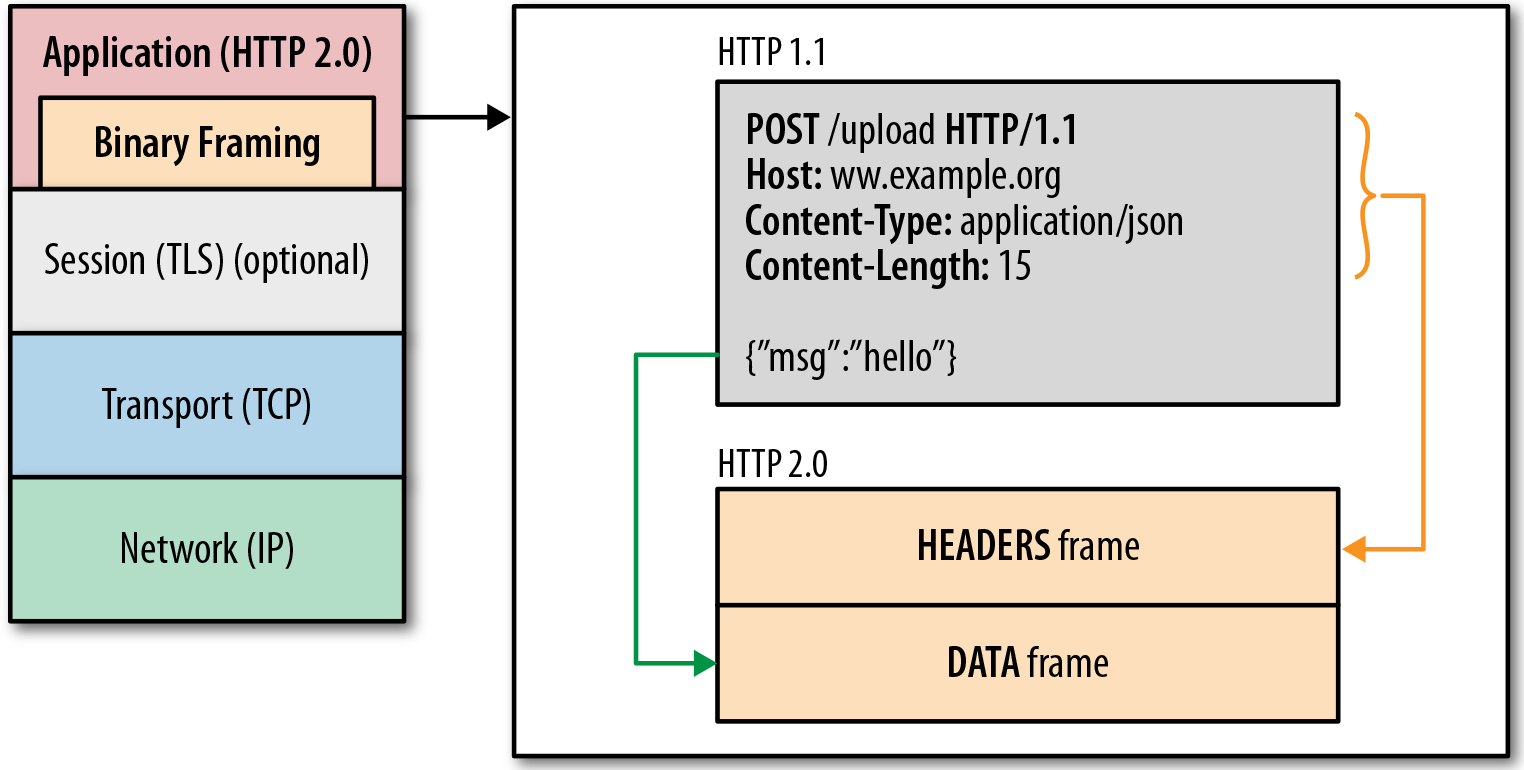
Moving fast, and (for once), everything looks on schedule!

HTTP 2.0 in a nutshell

- One TCP connection
- Request = Stream
  - Streams are multiplexed
  - Streams are prioritized

- (New) binary framing layer
  - Prioritization
  - Flow control
  - Server push

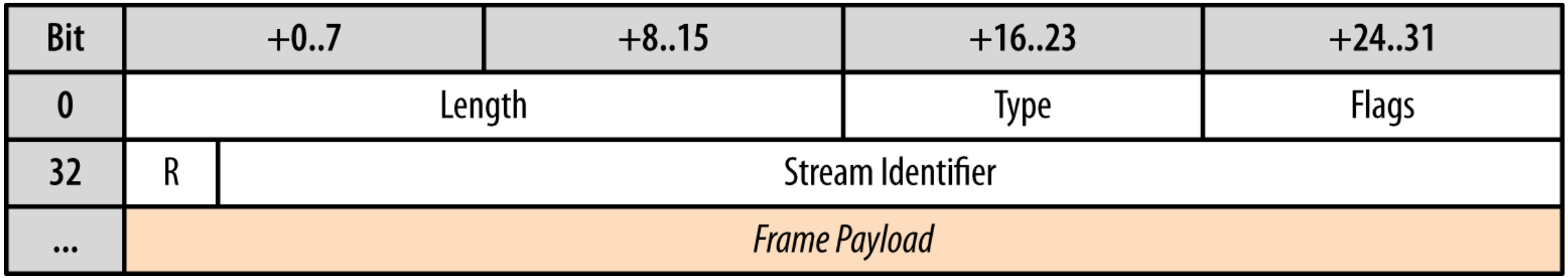
- Header compression



*"... **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 re-defining 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 (chair)

All frames have a common 8-byte header



- Length-prefixed frames
- **Type** indicates ... type of frame
  - *DATA, HEADERS, PRIORITY, PUSH_PROMISE, ...*
- Each frame may have custom **flags**
  - e.g. *END_STREAM*
- Each frame carries a **31-bit stream identifier**
  - After that, it's frame specific payload...

```
frame = buf.read(8)
if frame_i_care_about
  do_something_smart
else
  buf.skip(frame.length)
end
```

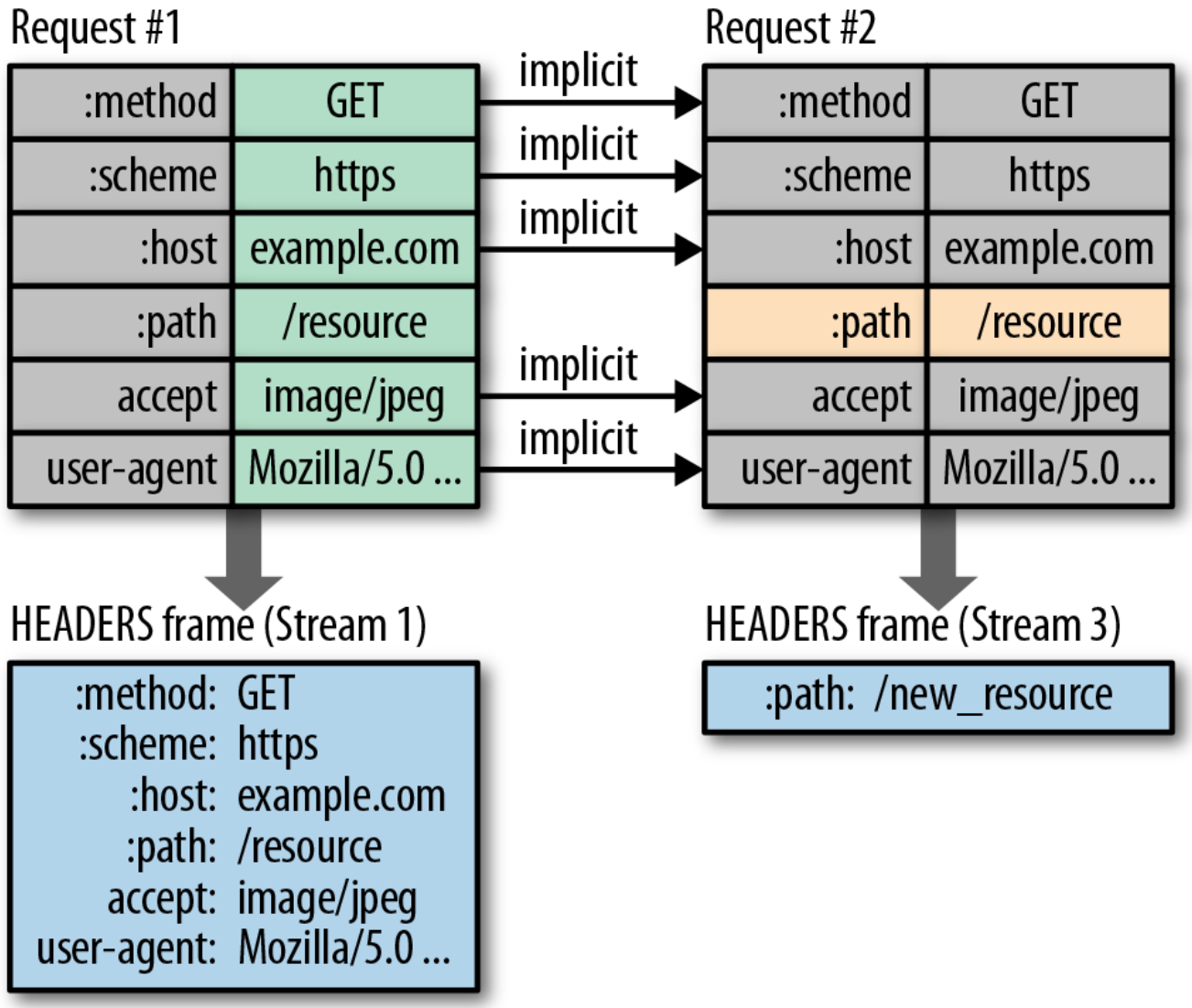
Opening a new stream with HTTP 2.0 (HEADERS)

Bit	+0..7	+8..15	+16..23	+24..31
0	Length		Type (1)	Flags
32	R	Stream Identifier		
64	X	Priority		
...	*Header Block*			

- Common 8-byte header
- Client / server allocate new stream ID
  - *client: odd, server: even*

- Optional 31-bit stream priority field
  - *Flags indicates if priority is present*
  - *2^31 is lowest priority*
- HTTP header payload
  - *see header-compression-01*

HTTP 2.0 header compression (in a nutshell)

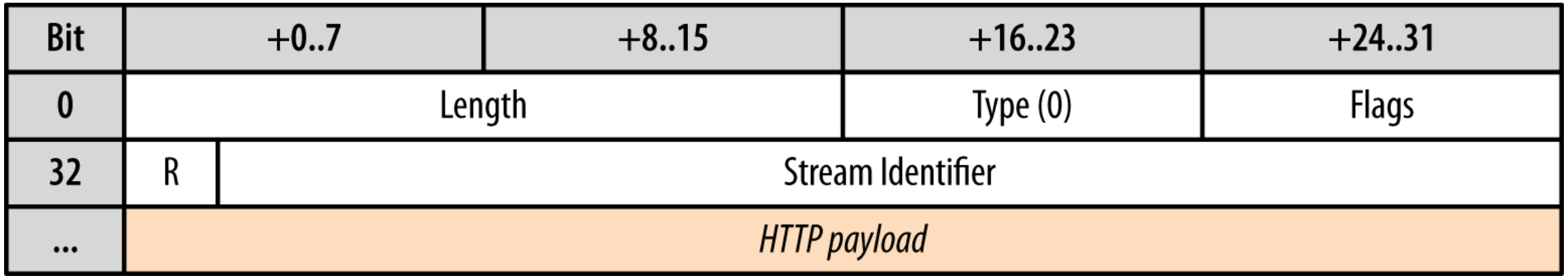


- Each side maintains “header tables”
- Header tables are initialized with common header key-value pairs

- New requests “toggle” or “insert” new values into the table
- New header set is a “diff” of the previous set of headers

- *E.g. Repeat request (polling) with exact same headers incurs no overhead (sans frame header)*

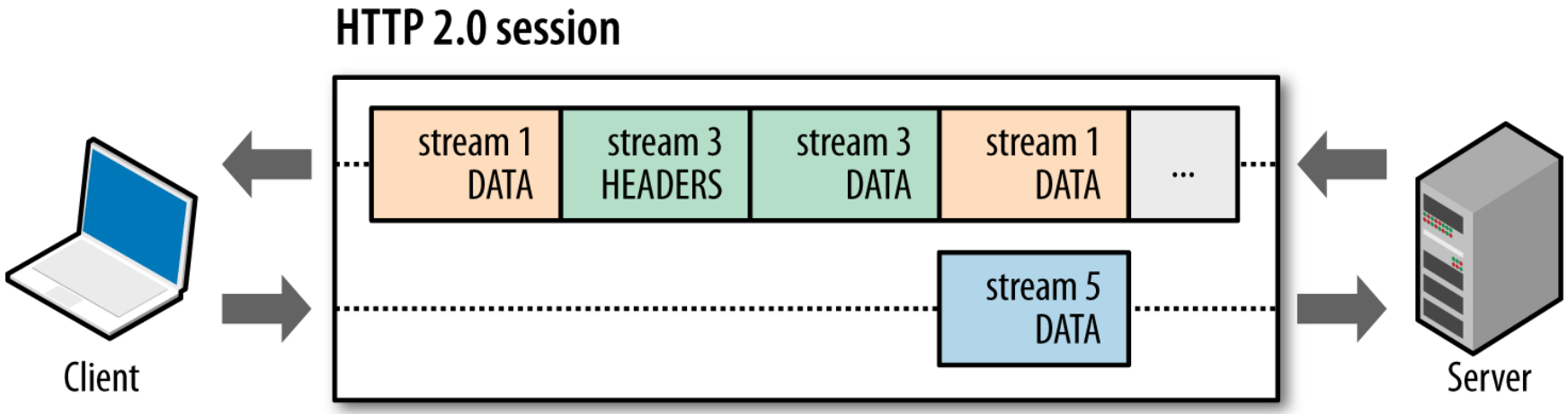
Sending application data with … DATA frames.



- Common 8-byte header
- Followed by application data…

- In theory, max-length = 2^16-1
- To reduce head-of-line blocking: **max frame size is 2^14-1** (~16KB)
  - Larger payloads are split into multiple DATA frames, last frame carries “END_STREAM” flag

Basic data flow in HTTP 2.0...

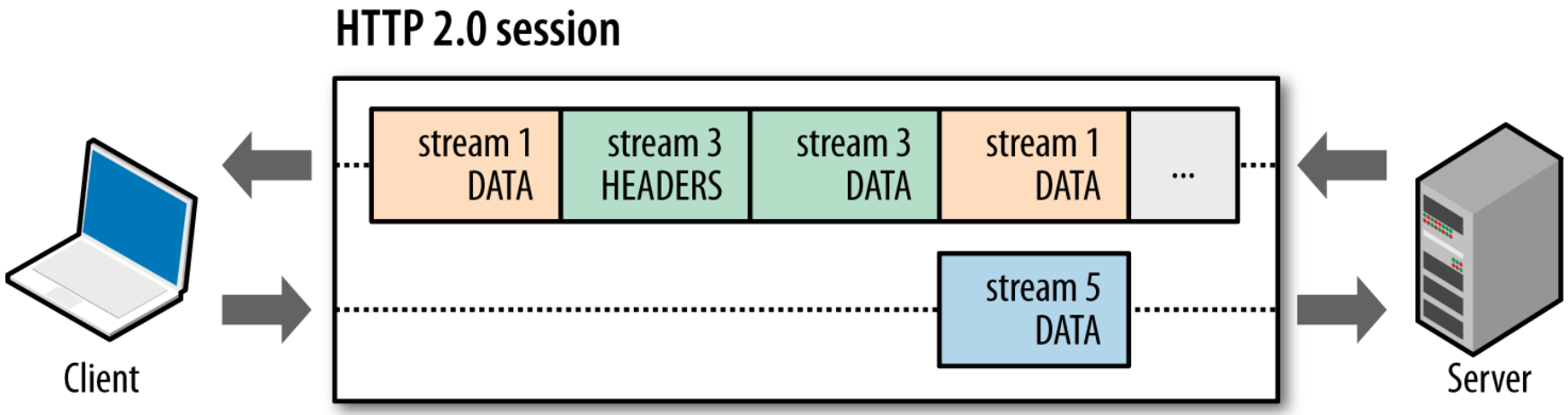


- Single TCP connection
- Streams are multiplexed by splitting communication into frames
  - e.g. HEADERS, DATA, etc.

- Frames are interleaved
  - Frames can be prioritized (by the server)
  - Frames can be flow controlled

- In diagram above: 3 active streams, all client initiated (odd).

Stream / Connection flow-control



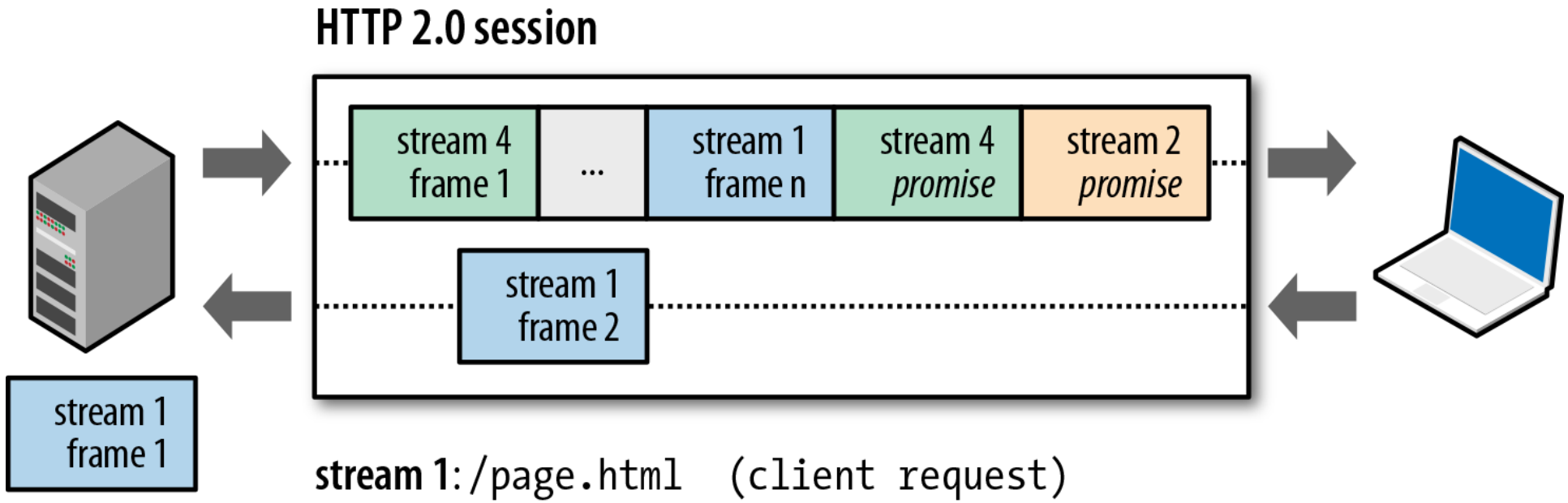
*We're multiplexing multiple streams **within** a single TCP connection!*

- Priority signals to the server the relative order of each stream
- Stream flow-control enables fine-grained resource control between streams
- Connection flow-control enables resource control between connections (e.g. proxies)

Very simple mechanism...

- Each stream and connection starts with 64KB window
- (only) DATA frames decrement the window
- Window size is updated by WINDOW_UPDATE frame

Server push... aka, replacement to inlining!



*If the server knows you'll need script.js, style.css, why not **push** it to the client?*

- Server initiates a stream via PUSH_PROMISE frame (similar to HEADERS)
- Must send PUSH_PROMISE to avoid client race condition (i.e. requesting same resource)
- Pushed resources are subject to same-origin policy
- **How do you know when to push?** Good question...

HTTP Upgrade flow from HTTP 1.x

```
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade
Upgrade: HTTP/2.0
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: HTTP/2.0

(... HTTP 2.0 response ...)
```

Alternatively, use **ALPN + TLS negotiation**:

1. Client advertises in ClientHello
   - **ProtocolNameList: http/2.0**
2. Server selects protocol and in ServerHello
   - **ProtocolName: http/2.0**

- **ALPN is the preferred negotiation method for HTTP 2.0**
  - Alas, proxies, intermediaries…

- ALPN negotiation is protocol agnostic and can be used for other applications also!

For an in-depth discussion on all of the above...

- *Optimizing **TCP** server stacks*
- *Optimizing **TLS** deployments*
- *Optimizing for **mobile** networks*
- *HTTP 2.0 features, framing, deployment...*
- *XHR, SSE, WebSocket, WebRTC, ...*

~~$29.99~~ **Read Online for Free**
Brought to you by Fluent Conference

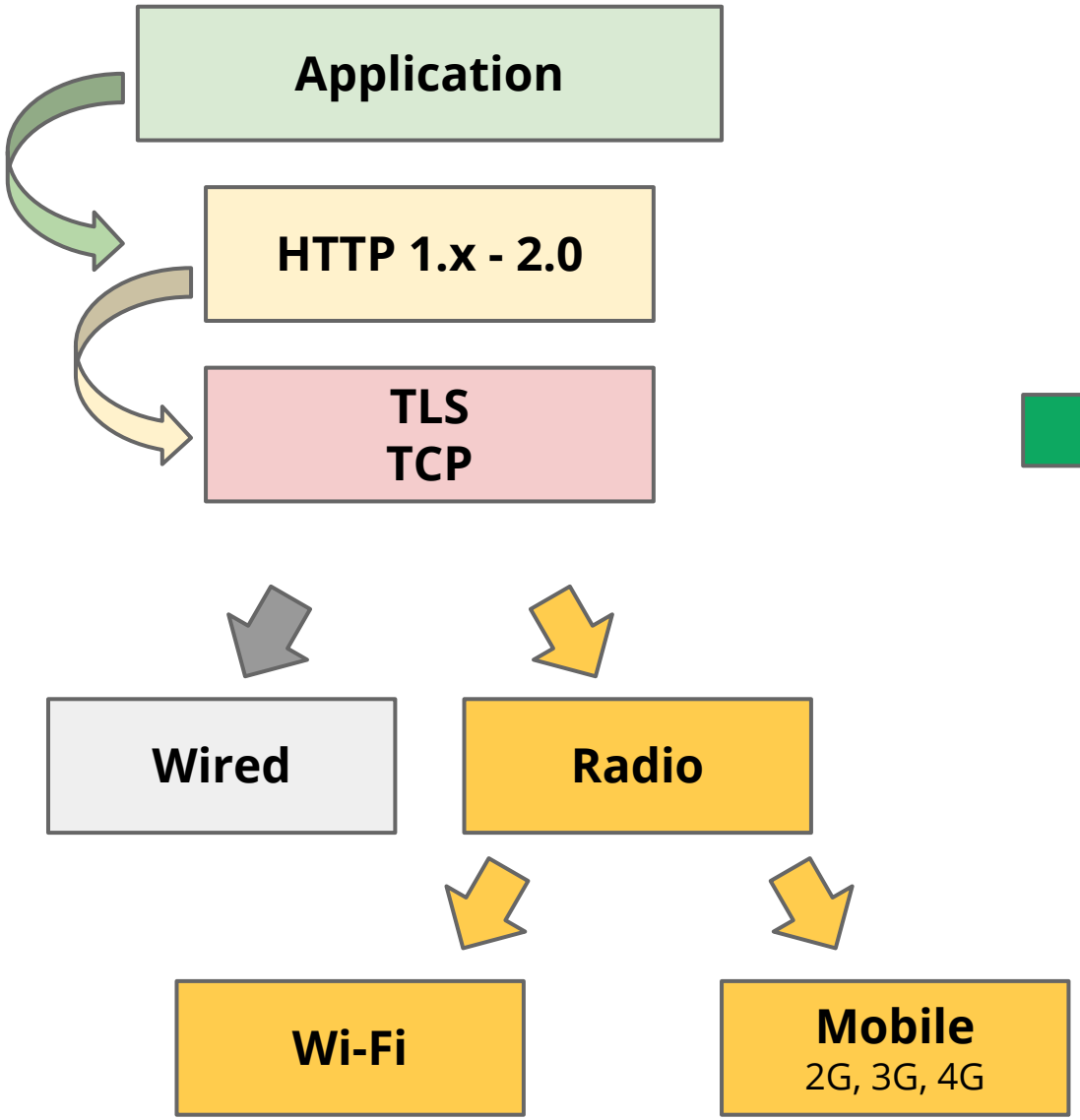
http://bit.ly/1fgTOsj

</shameless self promotion>

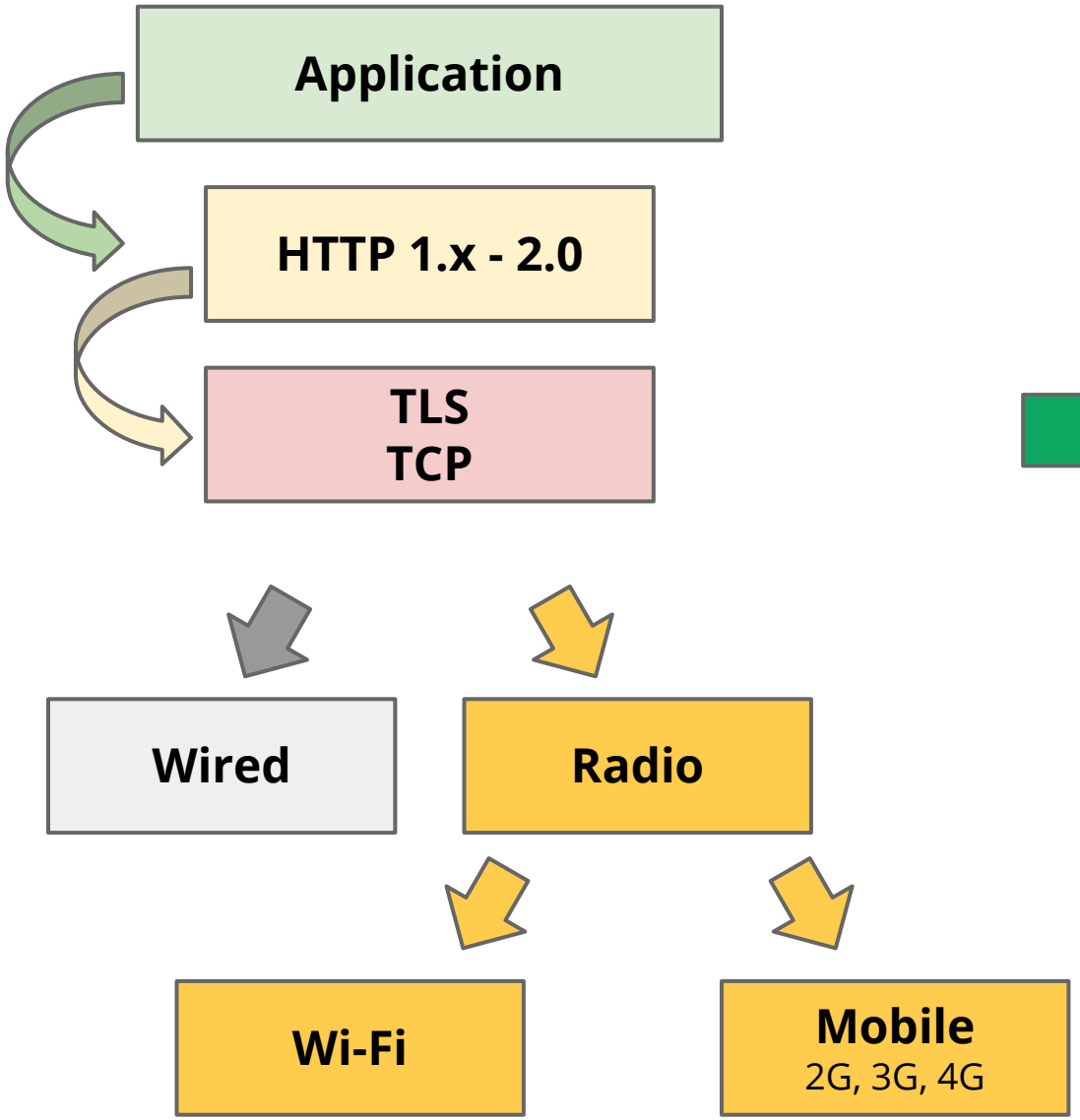
So, where does that leave us?

sounds great and all, but how do we adapt and adopt HTTP 2.0?

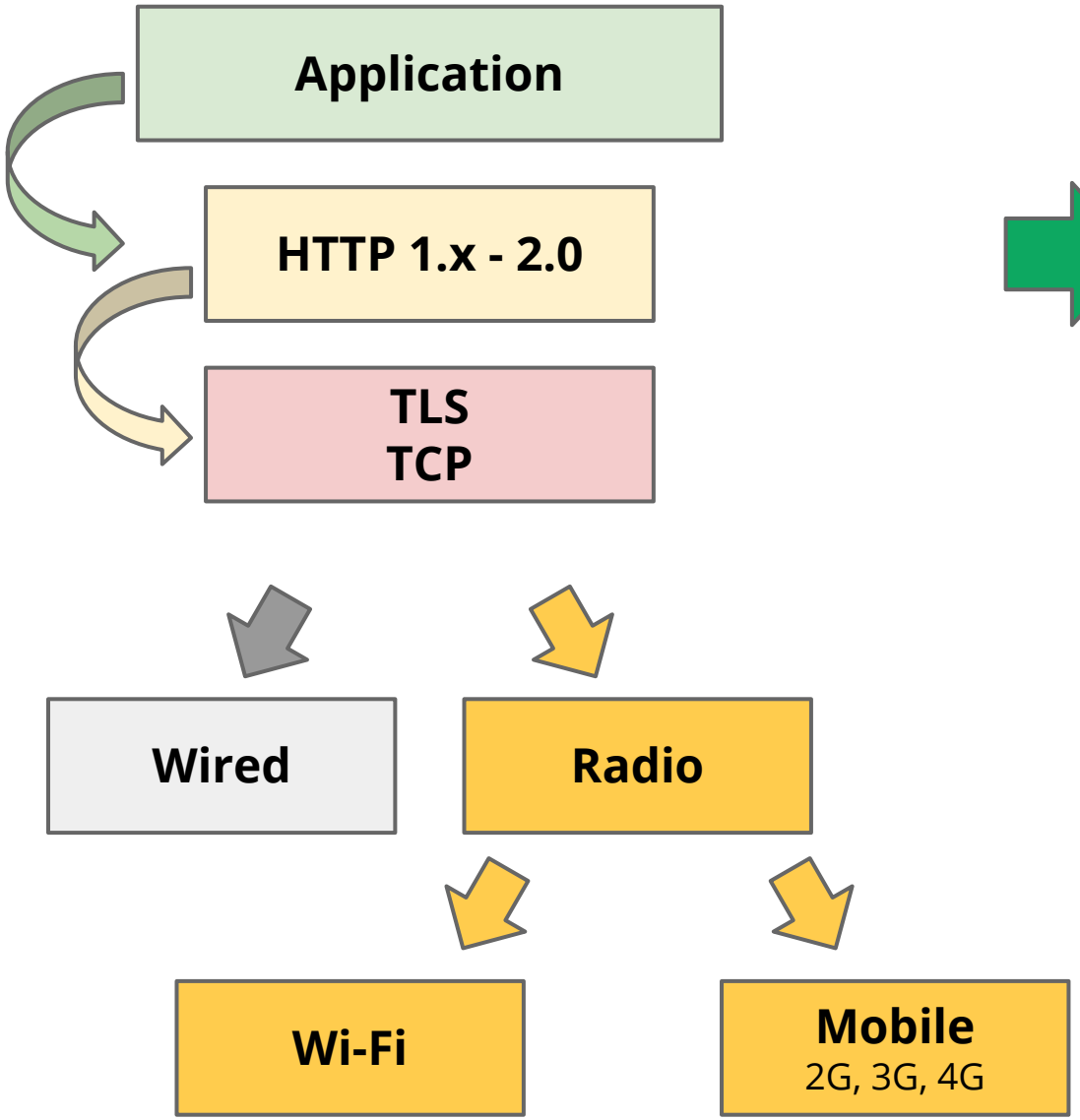
Let's work bottom up...



- Upgrade kernel: Linux 3.2+
- IW10 + disable slow start after idle
- TCP window scaling
- Position servers closer to the user
- Reuse established TCP connections
- Compress transferred data
- ....

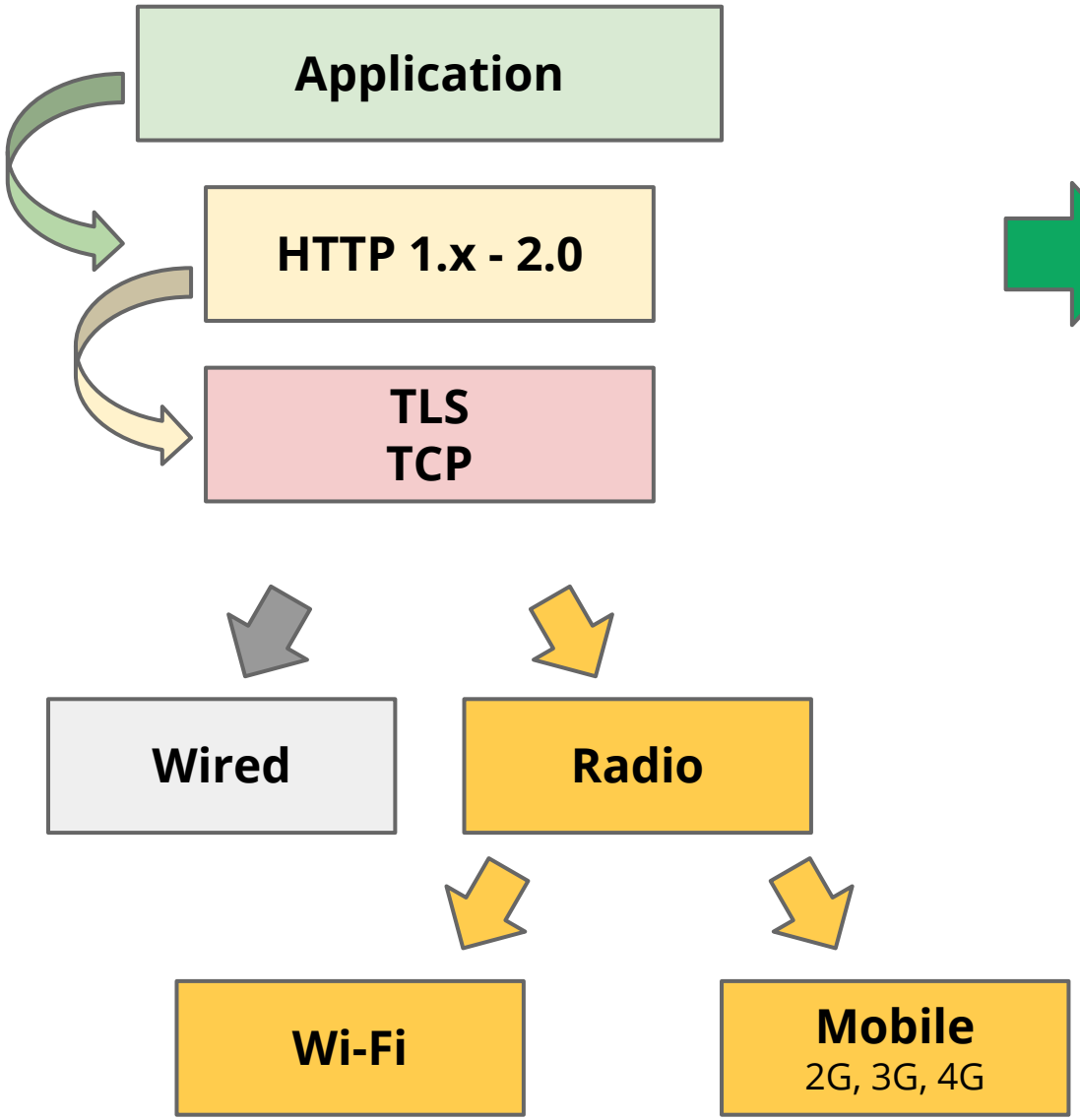


- Upgrade TLS libraries
- Use session caching / session tickets
- Early TLS termination (CDN)
- Optimize TLS record size
- Optimize certificate size
- Disable TLS compression
- Configure SNI support
- Use HTTP Strict Transport Security
- ....



HTTP 1.x hacks:

- Concatenate files (CSS, JS)
- Sprite small images
- Shard assets across origins
- Minimize protocol overhead
- Inline assets
- ....



HTTP 2.0 to the rescue!

- Unshard your assets (step 1)
- Undo other HTTP 1.x hacks... :-)
- Leverage server push
- ....

- Simpler applications, faster delivery, better caching, fewer server resources... \o/

Benefits

- Full request & response multiplexing
- Mechanism for request prioritization
- Stream and connection flow control

- Many small files? No problem
- Better TCP throughput
- Fewer TCP connections
- More efficient use of server resources
- Low overhead HTTP transfers
  - Header compression
  - Binary framing

Opportunities

- **Develop smarter servers**
  - Improved prioritization
  - Stream flow control
  - Smart resource push (mobile!)

- **Develop smarter clients**
  - Low latency, low overhead ...
  - Eliminate other RPC layers ...
  - Ready to use if you control both client and server

- **Help sites/apps migrate to HTTP 2.0**
  - HTTP 1.x will be around for a while
  - Smart proxies / load balancers

http://bit.ly/1fgTOsj

Questions?

Ilya Grigorik - @igrigorik
igvita.com

Slides @ http://bit.ly/18ZaMd7

Can haz SPDY?

Apache, nginx, Jetty, node.js, ...

Who supports SPDY?

- **Chrome**, since forever..
  - Chrome on Android + iOS
- **Firefox 13+**
- Next stable release of **Opera**

Server

- mod_spdy (Apache)
- nginx
- Jetty, Netty
- node-spdy
- ...

3rd parties

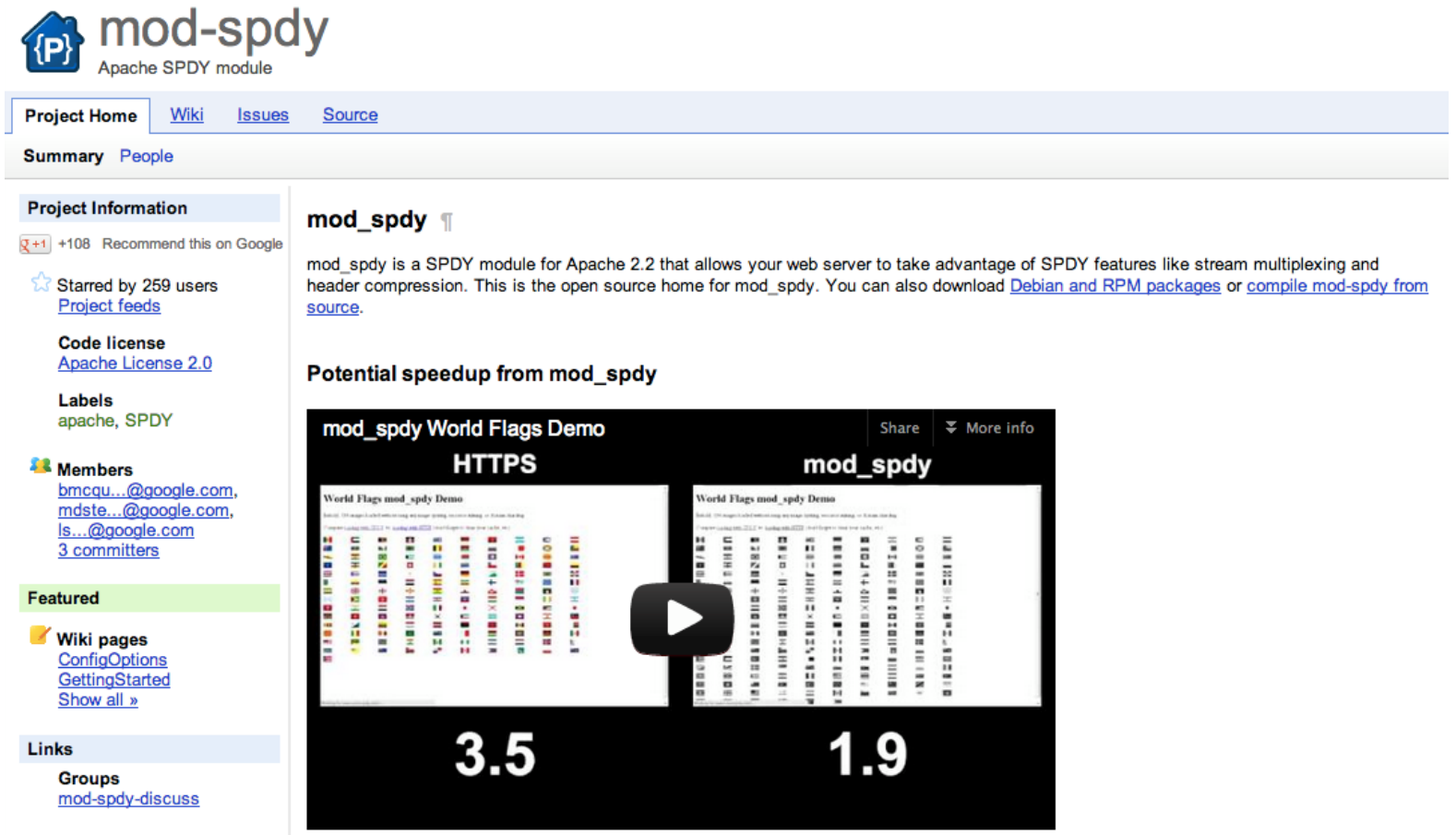
- Twitter
- Wordpress
- Facebook*

- Akamai
- Contendo
- F5 SPDY Gateway
- Strangeloop
- ...

All Google properties

- Search, GMail, Docs
- GAE + SSL users
- ...

Apache + SPDY



- mod_spdy is an open-source Apache module
- drop in support for SPDY

Installing mod_spdy in your Apache server

1

```
$ sudo dpkg -i mod-spdy-*.deb
$ sudo apt-get -f install
$ sudo a2enmod spdy

$ sudo service apache2 restart
```

2 **Profit**

- Configure mod_proxy + mod_spdy: https://gist.github.com/3817065
  - Enable SPDY for any backend app-server
  - SPDY connection is terminated by Apache, and Apache speaks HTTP to your app server

Building nginx with SPDY support

1

```
$ wget http://openssl.org/source/openssl-1.0.1c.tar.gz
$ tar -xvf openssl-1.0.1c.tar.gz

$ wget http://nginx.org/download/nginx-1.3.4.tar.gz
$ tar xvfz nginx-1.3.4.tar.gz
$ cd nginx-1.3.4

$ wget http://nginx.org/patches/spdy/patch.spdy.txt
$ patch -p0 < patch.spdy.txt
```

2

```
$ ./configure ... --with-openssl='/software/openssl/openssl-1.0.1c'
$ make
$ make install
```

3

Profit

http://blog.bubbleideas.com/2012/08/How-to-set-up-SPDY-on-nginx-for-your-rails-app-and-test-it.html

node.js + SPDY

1

```
var spdy = require('spdy'),
    fs = require('fs');

var options = {
  key: fs.readFileSync(__dirname + '/keys/spdy-key.pem'),
  cert: fs.readFileSync(__dirname + '/keys/spdy-cert.pem'),
  ca: fs.readFileSync(__dirname + '/keys/spdy-csr.pem')
};

var server = spdy.createServer(options, function(req, res) {
  res.writeHead(200);
  res.end('hello world!');
});

server.listen(443);
```

2

Profit

https://github.com/indutny/node-spdy

Jetty + SPDY

I <3 Java :-)

1. Copy X pages of maven XML configs
2. Add NPN jar to your classpath
3. Wrap HTTP requests in SPDY, or copy copius amounts of XML...
4. ...
5. Profit

http://www.smartjava.org/content/how-use-spdy-jetty

How do I use HTTP 2.0 today? Use SPDY...

- **Chrome**, since forever..
  - Chrome on Android + iOS
- **Firefox 13+**
- **Opera 12.10+**

Server

- mod_spdy (Apache)
- nginx
- Jetty, Netty
- node-spdy
- ...

3rd parties

- Twitter
- Wordpress
- Facebook

- Akamai
- Contendo
- F5 SPDY Gateway
- Strangeloop
- ...

All Google properties

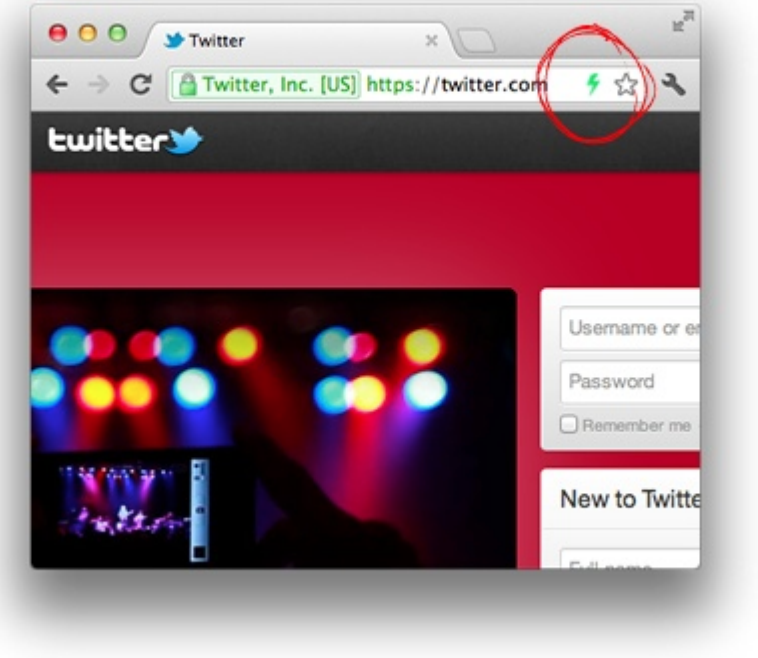
- Search, GMail, Docs
- GAE + SSL users
- ...

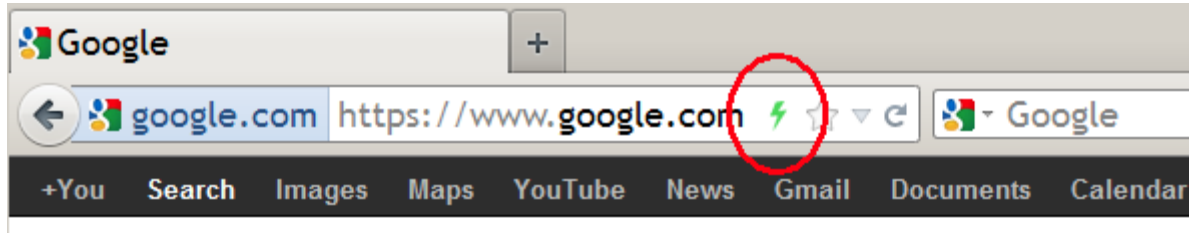
SPDY indicator(s)

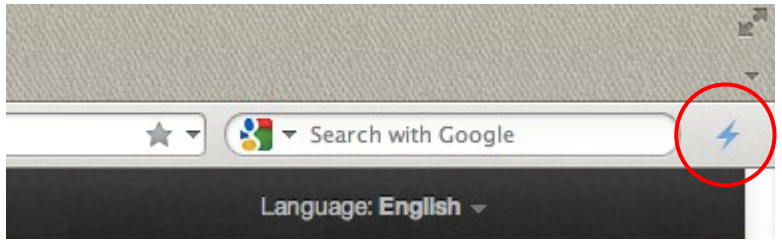
- Chrome SPDY indicator
- Firefox indicator
- Opera indicator

In Chrome console:

```
> window.chrome.loadTimes()
▼ Object
    commitLoadTime: 1350252136.934823
    finishDocumentLoadTime: 1350252137.397209
    finishLoadTime: 1350252137.529396
    firstPaintAfterLoadTime: 1350252137.611959
    firstPaintTime: 1350252137.523084
    navigationType: "Other"
    npnNegotiatedProtocol: "spdy/3"
    requestTime: 0
    startLoadTime: 1350252135.83449
    wasAlternateProtocolAvailable: false
    wasFetchedViaSpdy: true
    wasNpnNegotiated: true
  ▶ __proto__: Object
```







chrome://net-internals#spdy

Capturing network events (185) Stop Reset

- Capture
- Export
- Import
- Proxy
- Events
- Timeline
- DNS
- Sockets
- SPDY
- HTTP Pipelining
- HTTP Cache
- Tests
- HSTS
- Prerender

SPDY Status

- SPDY Enabled: true
- Use Alternate Protocol: true
- Force SPDY Always: false
- Force SPDY Over SSL: true
- Next Protocols: http/1.1,spdy/2,spdy/3

SPDY sessions

View live SPDY sessions

Host	Proxy	ID	Protocol Negotiatied	Active streams	Unclaimed pushed	Max	Initiated	Pushed	Pushed and claimed
0.docs.google.com:443	direct://	305272	spdy/3	1	0	100	80	0	0
clients4.google.com:443 apis.google.com:443 cbks0.google.com:443 clients1.google.com:443 clients2.google.com:443 docs.google.com:443 drive.google.com:443 encrypted-tbn0.gstatic.com:443 encrypted-tbn1.gstatic.com:443 encrypted-tbn2.gstatic.com:443 encrypted-tbn3.gstatic.com:443 khms0.google.com:443 khms1.google.com:443 maps-api-ssl.google.com:443	direct://	280013	spdy/3	0	0	100	3471	0	0

Try it @ https://spdy.io/ - open the link, then head to net-internals & click on stream-id