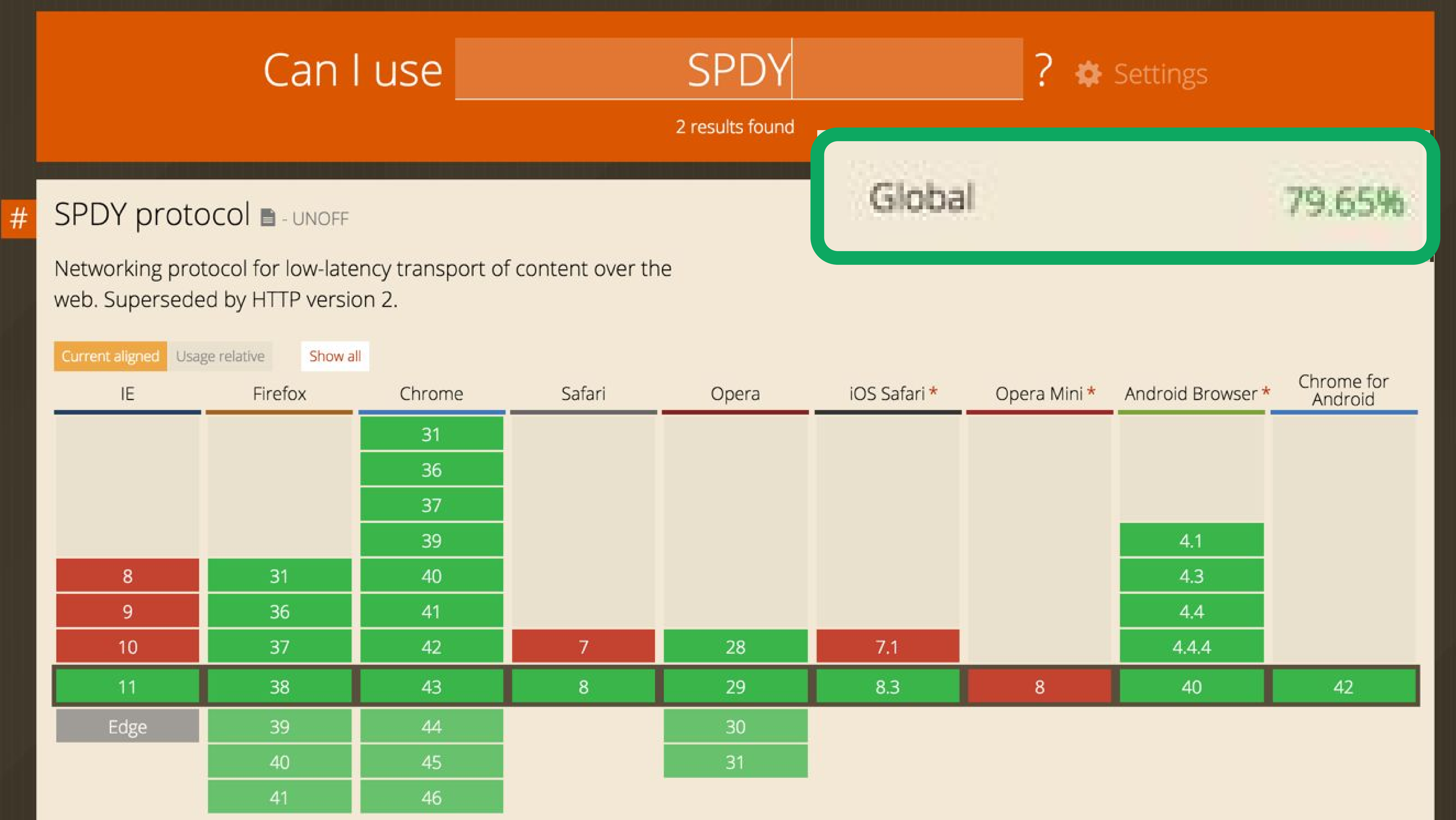
RFC 7540

HTTP/2 is here, let's optimize!

or, why (some) yesterday's best-practices are today's HTTP/2 anti-patterns.

+Ilya Grigorik
@igrigorik

O'REILLY® Velocity CONFERENCE



Can I use SPDY ?

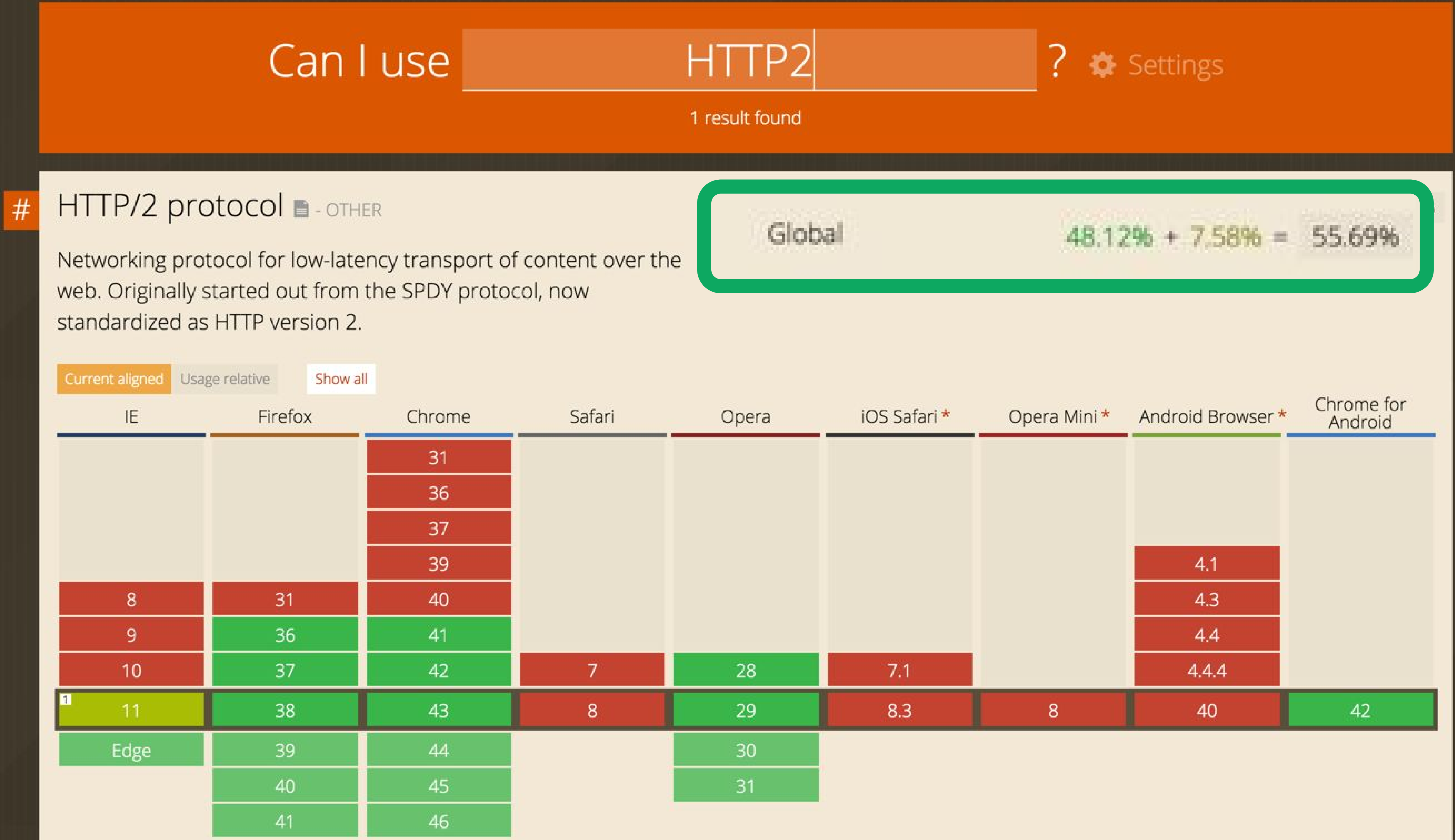
2 results found

Global 79.65%

SPDY protocol - UNOFF

Networking protocol for low-latency transport of content over the web. Superseded by HTTP version 2.

IE	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
		31						
		36						
		37						
		39					4.1	
8	31	40					4.3	
9	36	41					4.4	
10	37	42	7	28	7.1		4.4.4	
11	38	43	8	29	8.3	8	40	42
Edge	39	44		30				
	40	45		31				
	41	46						



Can I use HTTP2 ? Settings

1 result found

HTTP/2 protocol - OTHER

Global 48.12% + 7.58% = 55.69%

Networking protocol for low-latency transport of content over the web. Originally started out from the SPDY protocol, now standardized as HTTP version 2.

Current aligned Usage relative Show all

IE	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
		31						
		36						
		37						
		39					4.1	
8	31	40					4.3	
9	36	41					4.4	
10	37	42	7	28	7.1		4.4.4	
[1] 11	38	43	8	29	8.3	8	40	42
Edge	39	44		30				
	40	45		31				
	41	46						

"9% of all Firefox (M36) HTTP transactions are happening over HTTP/2. There are actually more HTTP/2 connections made than SPDY ones. This is well exercised technology."

Feb 18, 2015 - Patrick McManus, Mozilla

New TLS + NPN/ALPN connections in Chrome:

- *~27% negotiate HTTP/1*
- *~28% negotiate SPDY/3.1*
- *~45% negotiate HTTP/2*

May 26, 2015 - Chrome telemetry

HTTP/2 is here.
HTTP/2 is well tested.
HTTP/2 is replacing SPDY.

See RFC7540 (HTTP/2) and RFC7541 (HPACK). HTTP/2 has already surpassed SPDY in adoption, and Chrome will deprecate SPDY (and NPN) in early 2016.

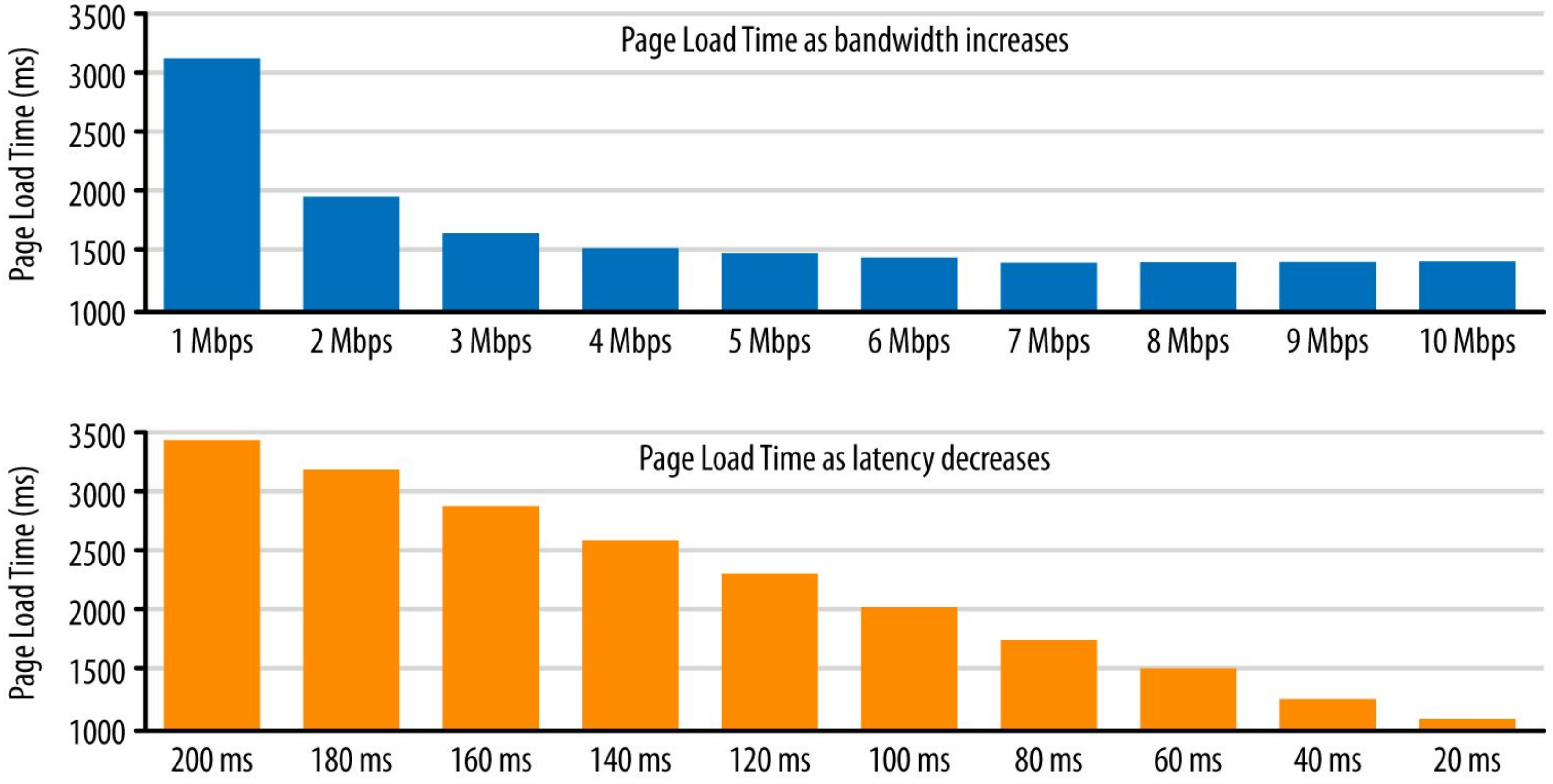
HTTP/2 in ~5 slides...

the what, why, and how behind the new protocol.

"HTTP/2 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

Latency vs Bandwidth impact on Page Load Time



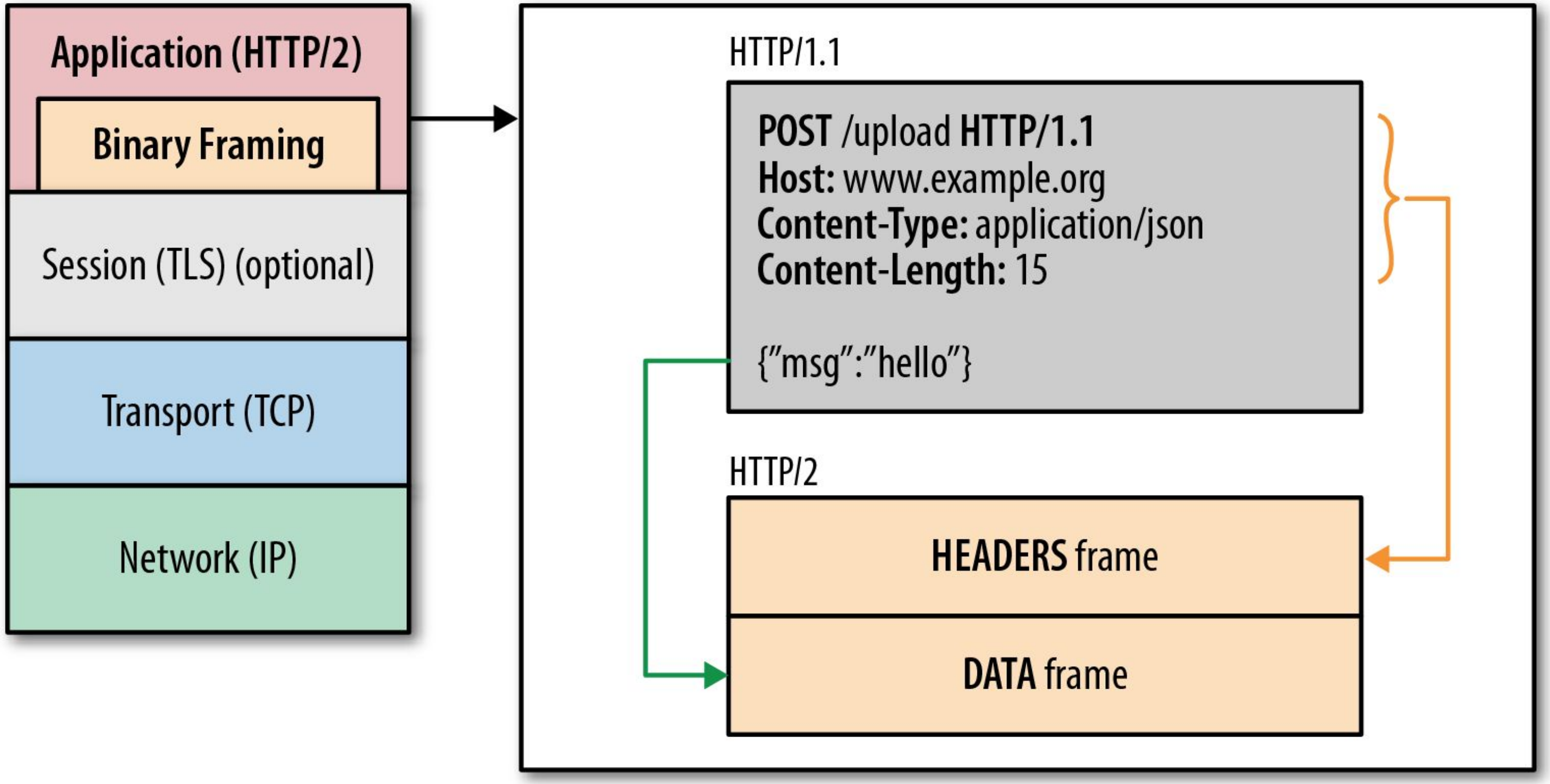
Single digit % perf improvement after 5 Mbps

Linear improvement in page load time!

“**To speed up the Internet at large, we should look for more ways to bring down RTT.** What if we could reduce cross-atlantic RTTs from 150 ms to 100 ms? This would have a larger effect on the speed of the internet than increasing a user’s bandwidth from 3.9 Mbps to 10 Mbps or even 1 Gbps.” - Mike Belshe

HTTP/2 in one slide...

1. **One TCP connection**
2. **Request → Stream**
   - ○ Streams are multiplexed
   - ○ Streams are prioritized
3. **Binary framing layer**
   - ○ Prioritization
   - ○ Flow control
   - ○ Server push
4. **Header compression (HPACK)**



Application (HTTP/2)

Binary Framing

Session (TLS) (optional)

Transport (TCP)

Network (IP)

HTTP/1.1

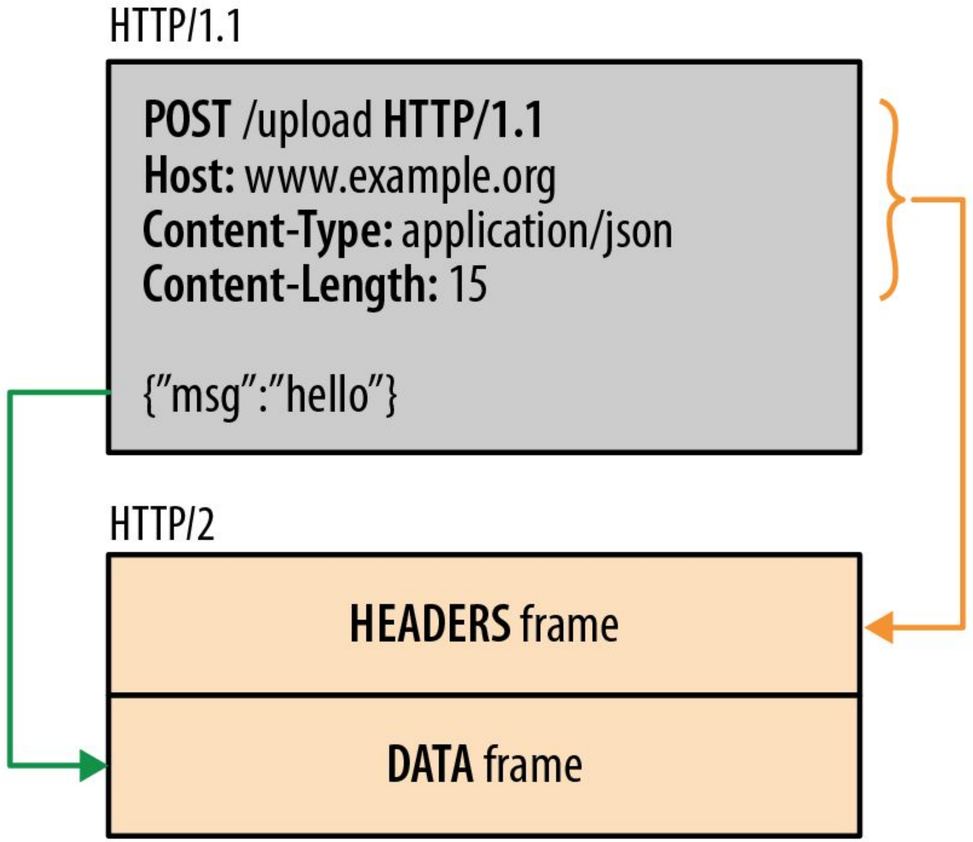
```
POST /upload HTTP/1.1
Host: www.example.org
Content-Type: application/json
Content-Length: 15

{"msg":"hello"}
```

HTTP/2

HEADERS frame

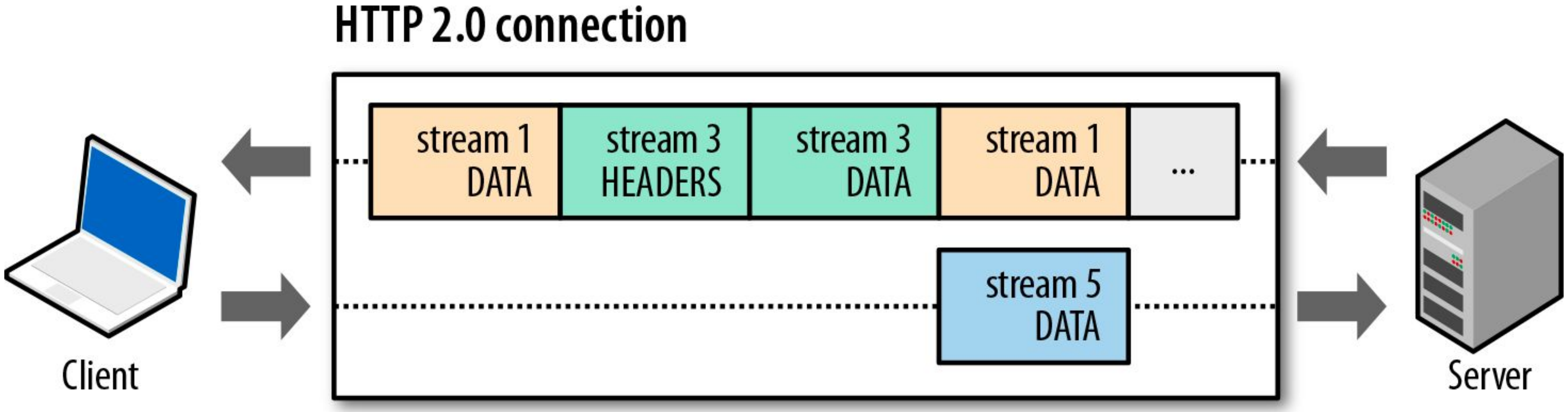
DATA frame

HTTP/2 binary framing 101



- **HTTP messages are decomposed into one or more frames**
  - HEADERS for meta-data
  - DATA for payload
  - RST_STREAM to cancel
  - ...
- **Each frame has a common header**
  - 9-byte, length prefixed
  - Easy and efficient to parse

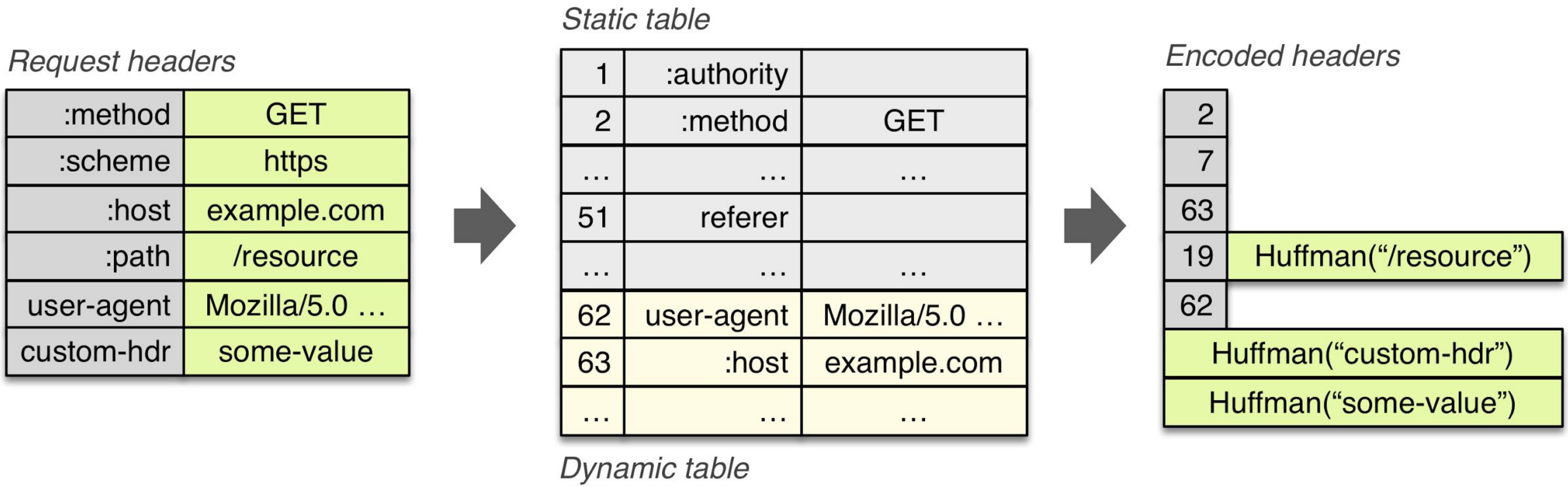
Basic data flow in HTTP 2.0...



Streams are multiplexed because frames can be interleaved

- All frames (e.g. HEADERS, DATA, etc) are sent over single TCP connection
- Frame delivery is prioritized based on stream dependencies and weights
- DATA frames are subject to per-stream and connection flow control

HPACK header compression



Request headers

:method	GET
:scheme	https
:host	example.com
:path	/resource
user-agent	Mozilla/5.0 …
custom-hdr	some-value

Static table

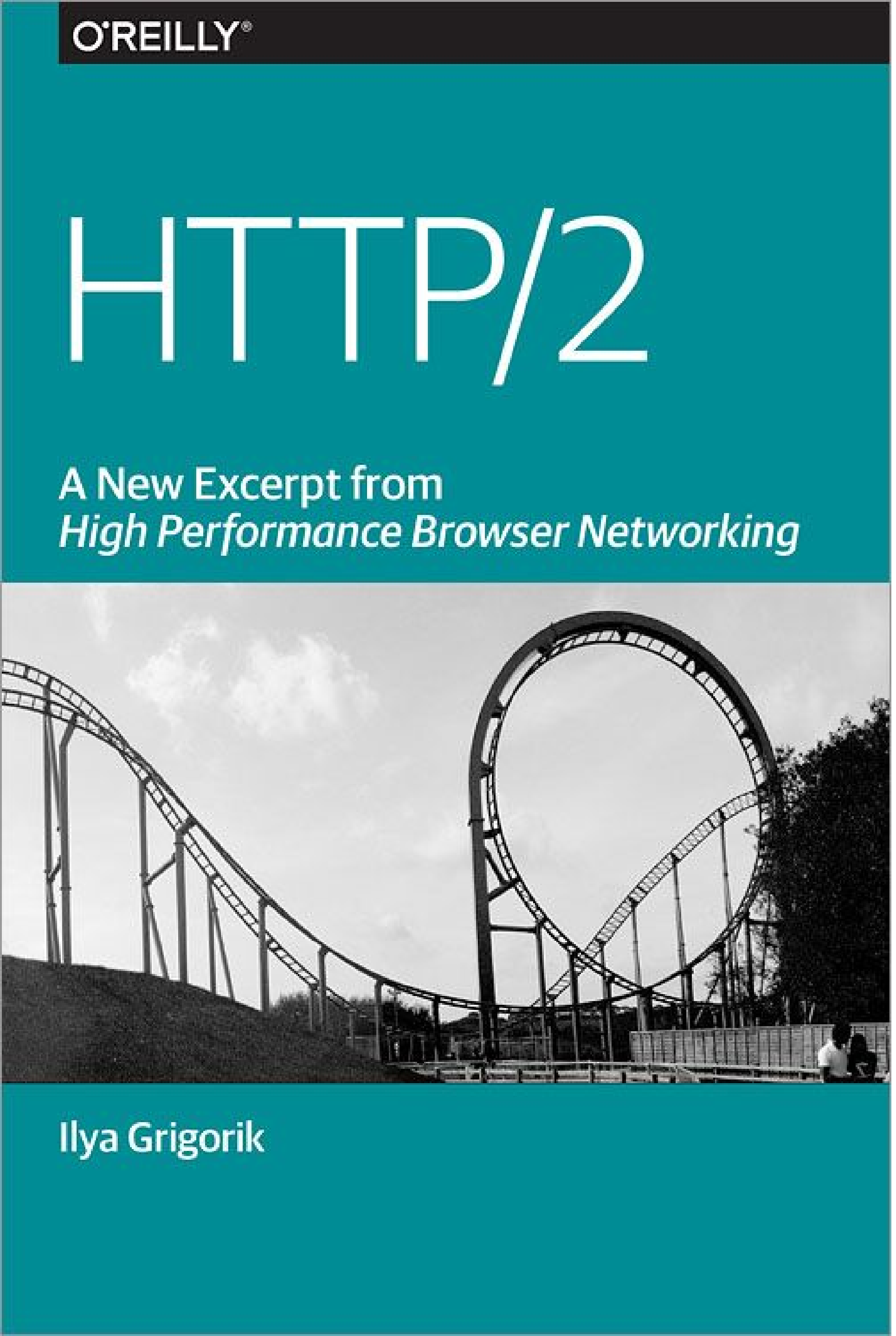
1	:authority	
2	:method	GET
…	…	…
51	referer	
…	…	…
62	user-agent	Mozilla/5.0 …
63	:host	example.com
…	…	…

Dynamic table

Encoded headers

2	
7	
63	
19	Huffman("/resource")
62	
Huffman("custom-hdr")	
Huffman("some-value")	

- *Literal values are (optionally) encoded with a static Huffman code*
- *Previously sent values are (optionally) indexed*
  - *e.g. "2" in above example expands to "method: GET"*



O'REILLY

HTTP/2

A New Excerpt from
High Performance Browser Networking

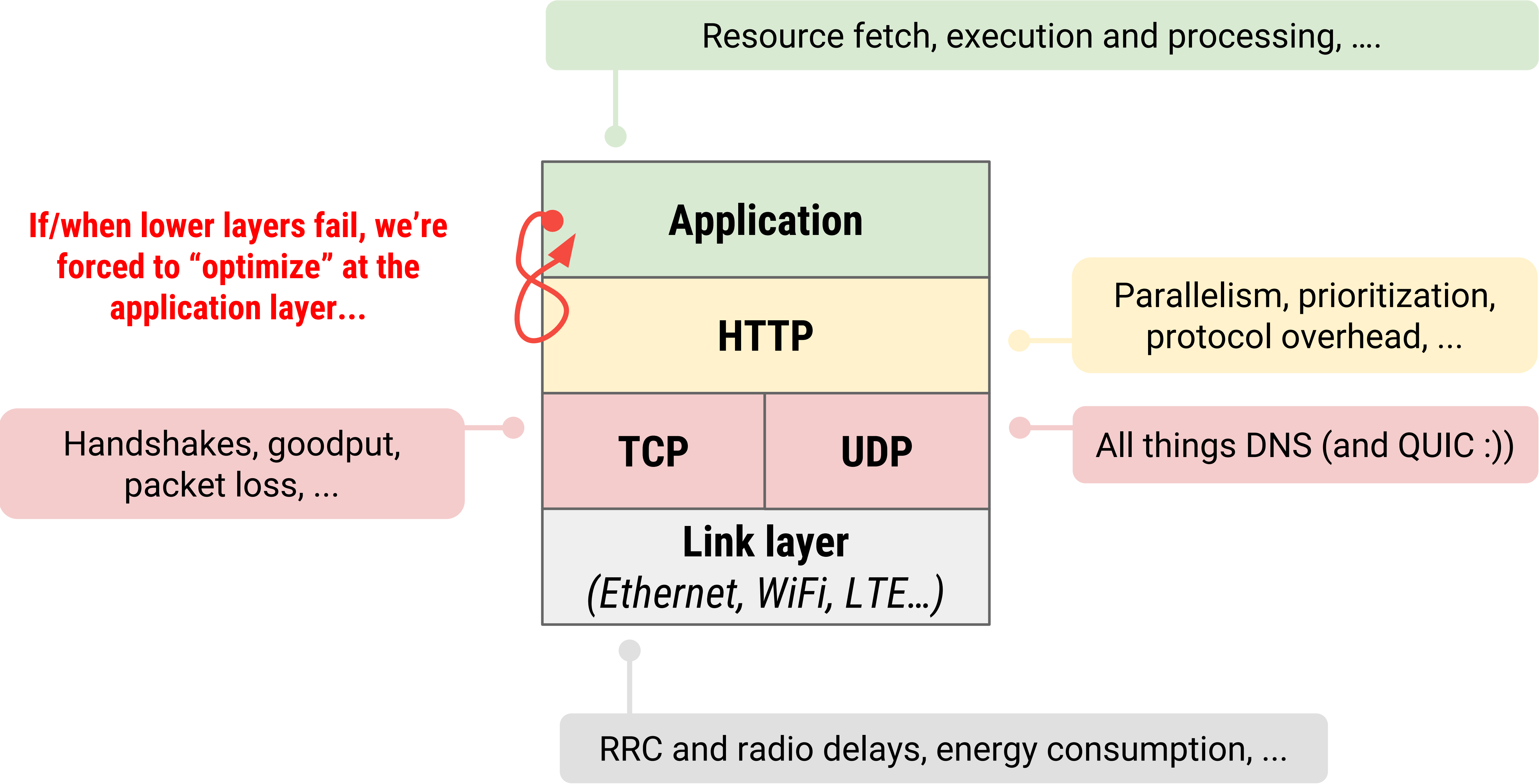
Ilya Grigorik

For a deep(er) dive on HTTP/2 protocol, grab the free book at the O’Reilly booth, or…

Read it online (free):
hpbn.co/http2

Optimizing (web) application delivery

let’s take a quick tour of our delivery pipeline...



If/when lower layers fail, we're forced to "optimize" at the application layer...

Resource fetch, execution and processing, ...

Application

HTTP

Parallelism, prioritization, protocol overhead, ...

TCP

UDP

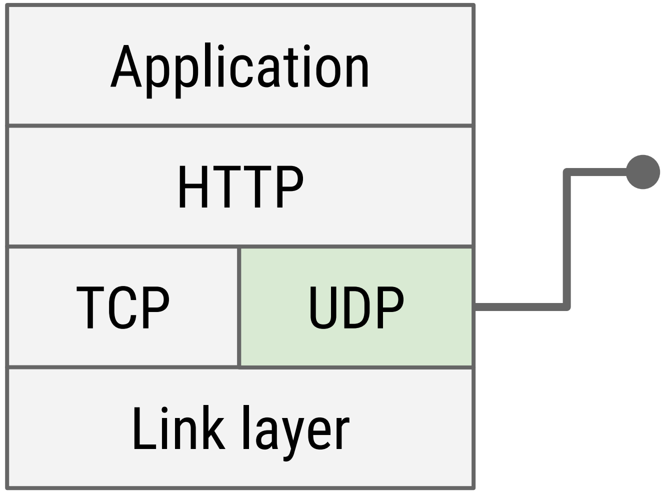
Handshakes, goodput, packet loss, ...

All things DNS (and QUIC :))

Link layer

(Ethernet, WiFi, LTE...)

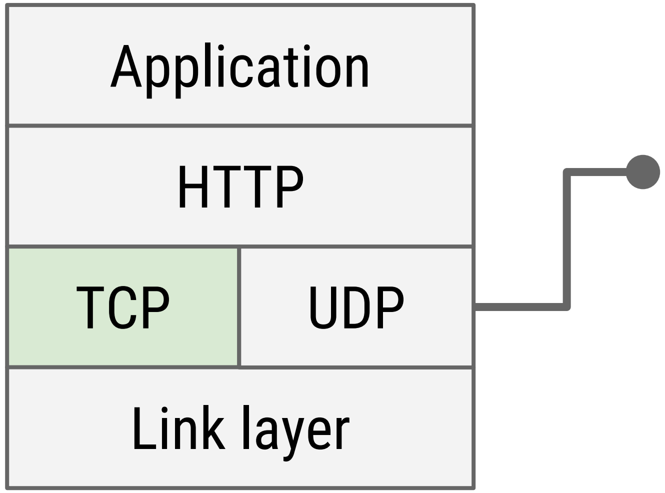
RRC and radio delays, energy consumption, ...



Reduce DNS lookups

Unresolved names block requests

✓ **HTTP/1.x** ✓ **HTTP/2**

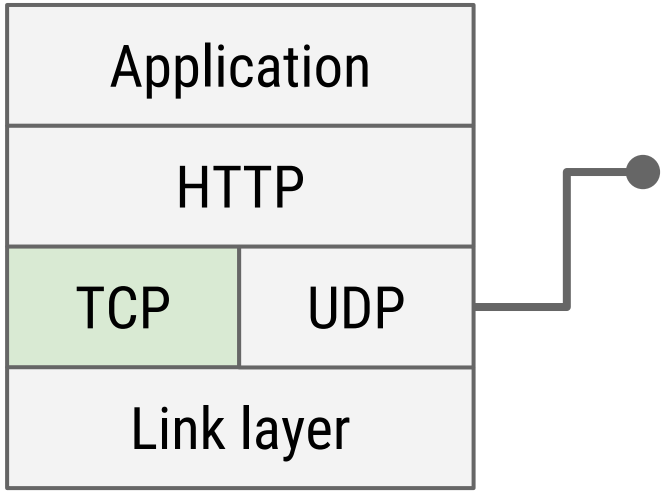


Reuse TCP connections

Connection are expensive

- handshake latency, resource overhead, ...

✓ **HTTP/1.x** ✓ **HTTP/2** — Single connection!

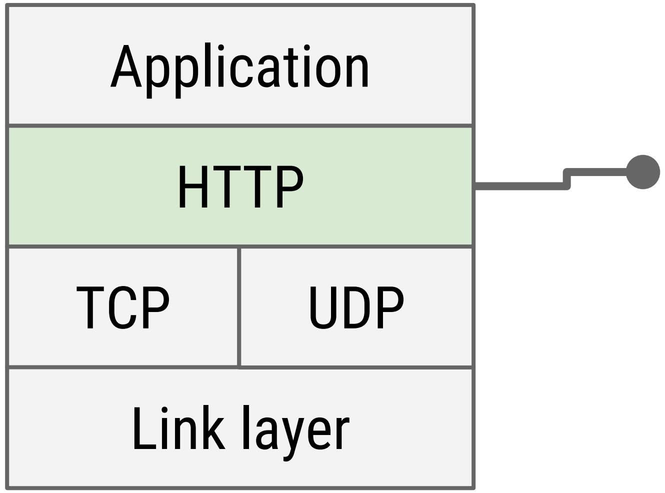


Use a Content Delivery Network

Page rendering is latency-bound (most of the time)

- lower roundtrip times are critical to optimize asset delivery

✓ **HTTP/1.x** ✓ **HTTP/2**

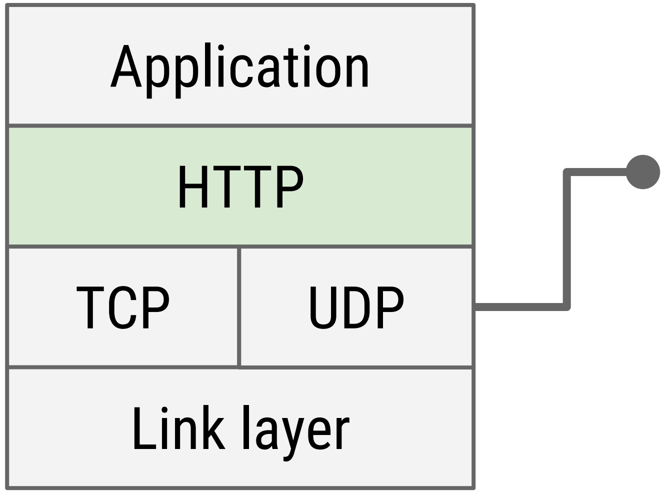


Minimize number of HTTP redirects

Each redirect restarts the fetch process

- cross-origin redirects are worst case: DNS, TCP, new HTTP request

✓ **HTTP/1.x** ✓ **HTTP/2**

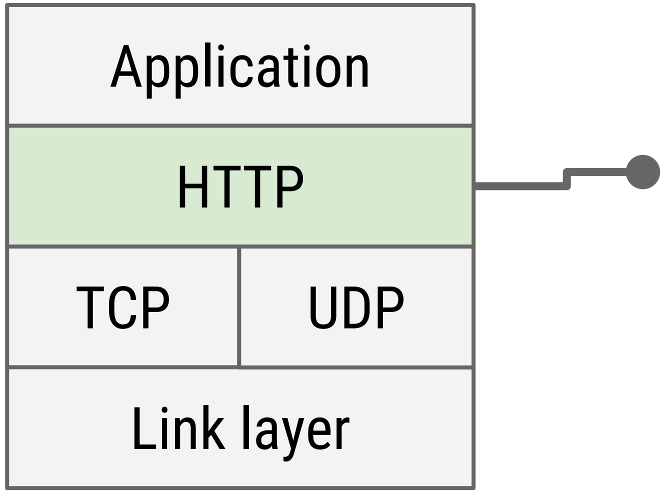


Eliminate unnecessary request bytes

Unnecessary metadata (e.g. headers) add up quickly

- 100+ requests, with a few KB each of headers… hundreds of KB's!

✓ HTTP/1.x **✓ HTTP/2** — HPACK helps...

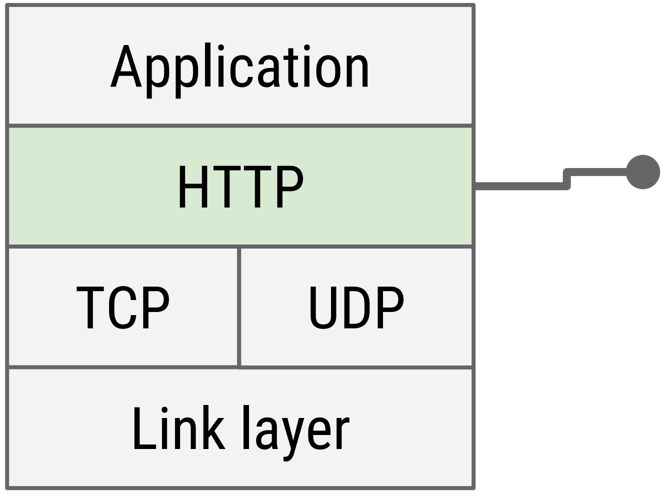


Compress assets during transfer

Bytes are slow and expensive to transfer...

- GZIP offers 40-80% savings on most assets - easy win.

✓ HTTP/1.x **✓ HTTP/2**

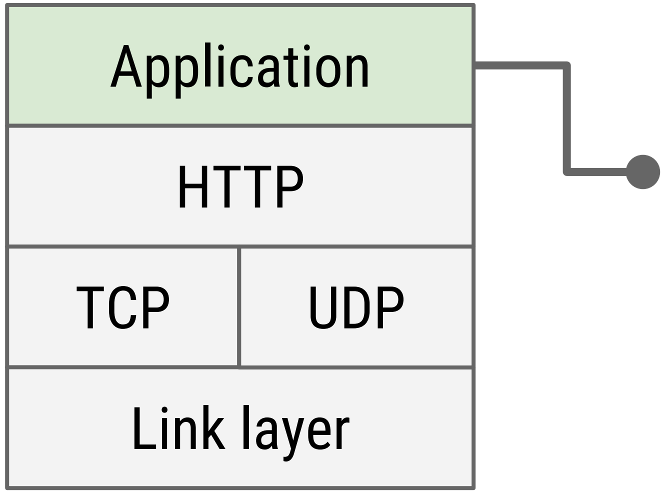


Cache resources on the client

Redundant data transfers are… redundant!

- Cache-Control and ETag's on each resource is a must.

✓ HTTP/1.x **✓ HTTP/2**



Eliminate unnecessary resources

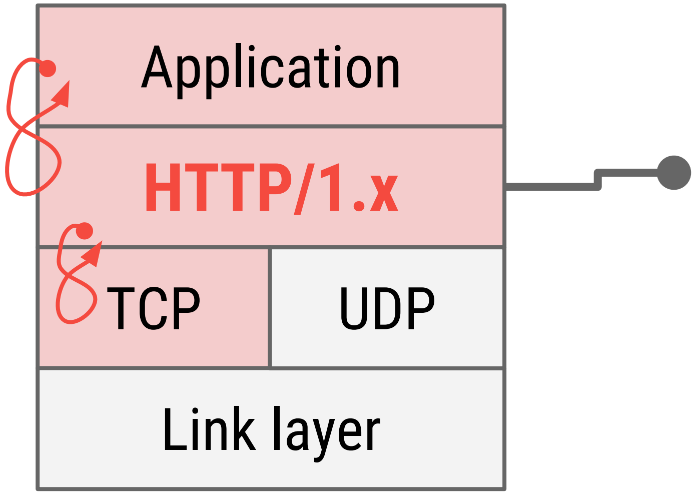
Fetch what you need, bytes are expensive

- Aggressive prefetching is expensive both on client and server.

✓ HTTP/1.x **✓ HTTP/2**

Evergreen performance best-practices

- → Reduce DNS lookups
- → Reuse TCP connections
- → Use a Content Delivery Network
- → Minimize number of HTTP redirects
- → Eliminate unnecessary request bytes
- → Compress assets during transfer
- → Cache resources on the client
- → Eliminate unnecessary resources



Limited parallelism

- Parallelism is limited by number of connections
- In practice, ~6 connections per origin

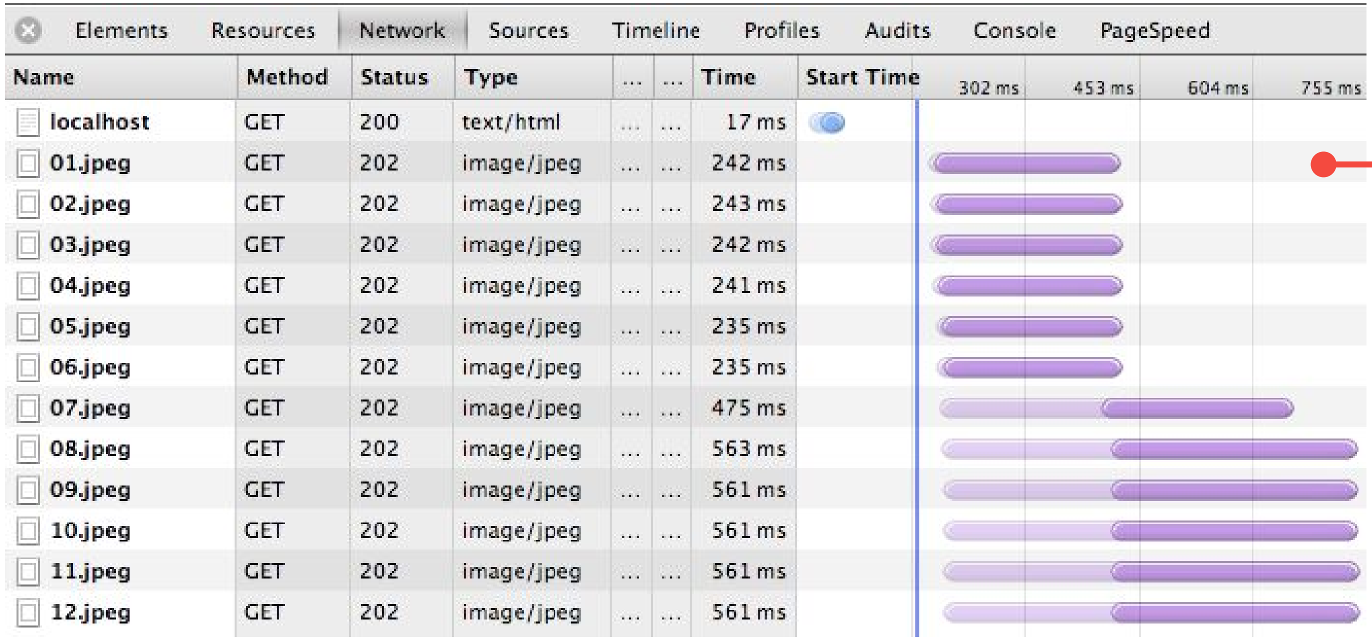
Head-of-line blocking

- Request queuing and delayed request dispatch on the client
- Strict response ordering on the server

High protocol overhead

- Header meta-data is uncompressed
- ~800 bytes of meta-data per request, plus cookies

Parallelism is limited by number of connections

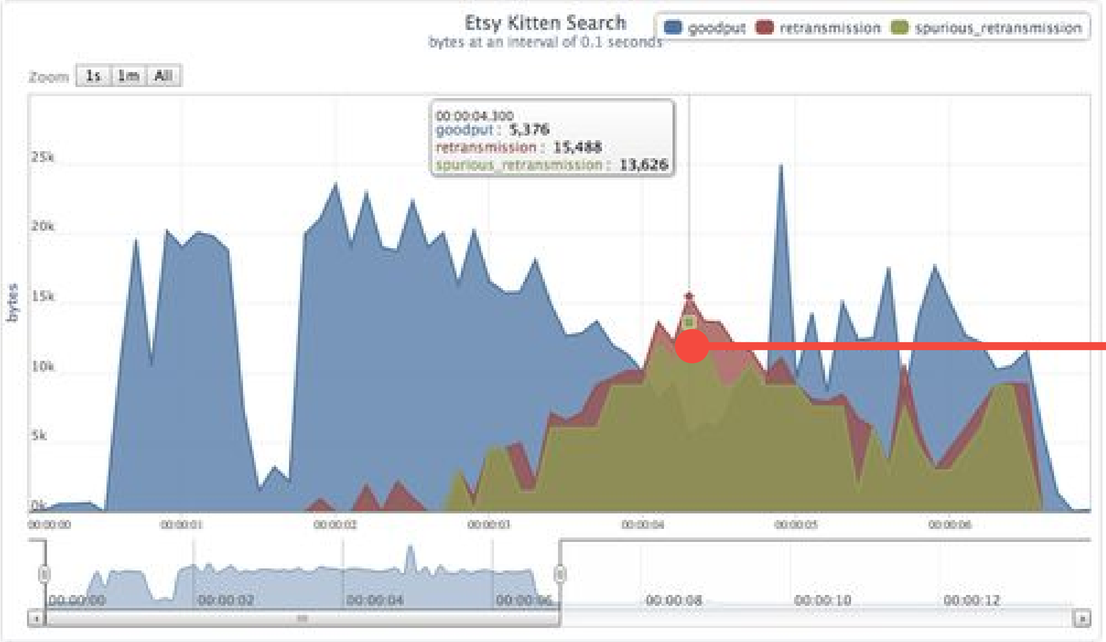


Name	Method	Status	Type	...	...	Time	Start Time
localhost	GET	200	text/html	...	...	17 ms	
01.jpeg	GET	202	image/jpeg	...	...	242 ms	
02.jpeg	GET	202	image/jpeg	...	...	243 ms	
03.jpeg	GET	202	image/jpeg	...	...	242 ms	
04.jpeg	GET	202	image/jpeg	...	...	241 ms	
05.jpeg	GET	202	image/jpeg	...	...	235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

~6 parallel downloads per origin

- Each connection incurs full TCP handshake
- Each connection incurs TLS handshake overhead (best case, resumed)
- Each connection occupies server/proxy resources (memory, CPU, etc)
- Each connection competes with others (broken congestion control)

No problem… domain shard all the things!



Duplicate (spurious) data packets lead to poor connection “goodput”

Congestion control is a bigger problem than you might think. Especially in emerging markets where many users are bandwidth+latency limited.

What’s the optimal number of shards?

Trick question, the answer depends on device + network + network weather + page architecture. Most sites **abuse** sharding, and hurt themselves… causing congestion, retransmissions, etc.

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

“One great metric around that which I enjoy is the fraction of connections created that carry just a single HTTP transaction (and thus make that transaction bear all the overhead). For HTTP/1 74% of our active connections carry just a single transaction - persistent connections just aren't as helpful as we all want. But in HTTP/2 that number plummets to 25%. That's a huge win for overhead reduction.”

Patrick McManus, Mozilla.

Report card: domain sharding

Introduced as a workaround for lack of multiplexing in HTTP/1. UA’s open ~6 connections per origin (6 parallel downloads), and sharding allows us to raise this number to… any number.

+ Enables (higher) parallelism for HTTP/1.x

- No “best” value for number of shards: wasted retransmissions & congestion
- Each connection has a resource cost: memory, CPU, etc
- Each connection competes with others for bandwidth: poor TCP performance
- Complicates our code and applications
- Breaks resource prioritization and flow control for HTTP/2
- Reduces HPACK compression benefits for HTTP/2

HTTP/1.x: domain sharding is abused, consider limiting use to **two** shards.

HTTP/2: unnecessary. Negative impact on performance. **Eliminate.**

Removing domain sharding for HTTP/2

HTTP/2 can coalesce connections on your behalf, if...

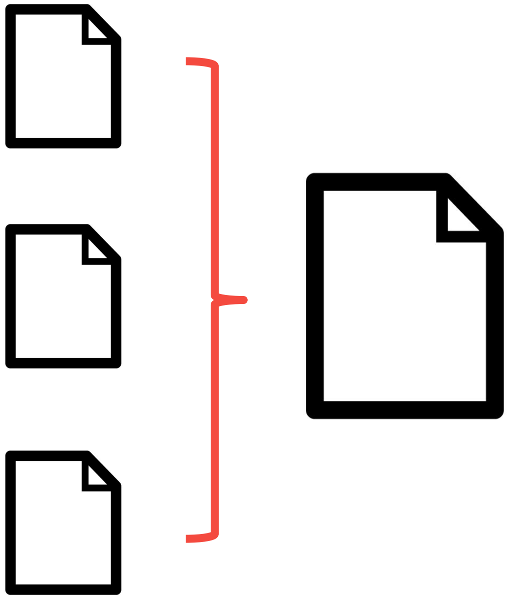
1. **TLS certificate is valid for both hosts**
2. **Hosts resolve to the same IP**

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

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

Using this technique you can get the best of both worlds with minimal work.

“Reduce requests” → concatenate all the things!



Head of line blocking is expensive, instead of fetching N small assets, let’s fetch fewer but bigger assets.. with same content!

+ Parallelism workaround for HTTP/1.x
+ Improved compression due to shared context

- Delayed processing and execution
  - e.g. must wait for entire (CSS, JS) response to arrive
- Expensive cache invalidations
  - single byte update forces full fetch
  - redundant data transfers on update

Yes, we've always had caching… But, we were never able to optimize for "churn" because small requests were too expensive. This is no longer the case.

- How expensive are your cache invalidations?
- Can you isolate high-velocity code from "stable" code?
- Can you assign different expiry times to these resources?

* churn: ratio of bytes (in cache vs new) we have to fetch when pushing an update.

Report card: concatenated assets

Introduced as a workaround for head-of-line blocking in HTTP/1. Concatenation allows us to fetch “multiple files” within one request.

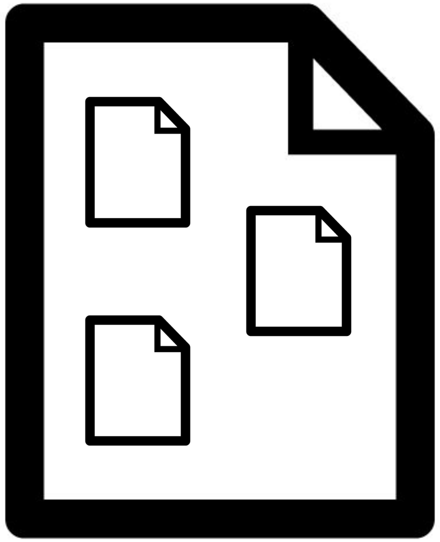
+ Improves HTTP/1.x performance
+ May deliver better resource compression
- Complicates our code and applications - e.g. extra build steps
- Breaks granular caching, updates, and revalidation
- Delays resource processing and execution
- Forces expensive cache invalidations

HTTP/1.x: use carefully and consider and optimize for invalidation costs (churn).

HTTP/2: avoid*. Ship small granular resources and optimize caching policies.

* Significant wins in compression are the only case where it might be useful.

“Reduce Requests” → inline all the things!



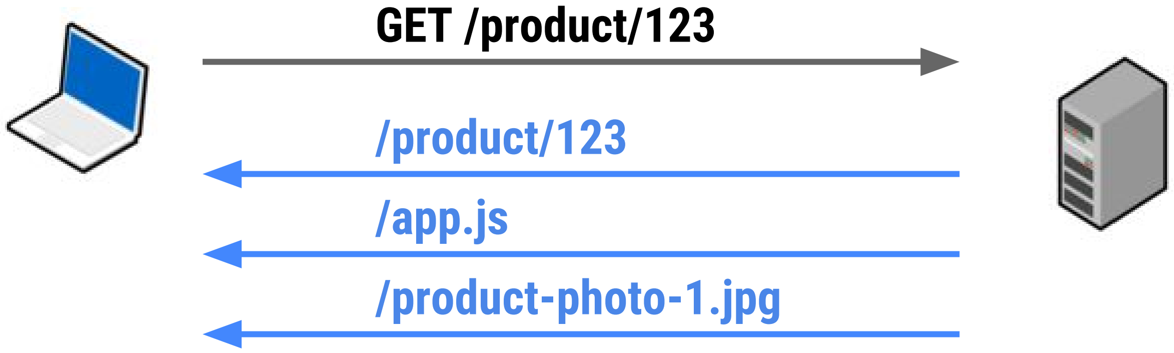
Head of line blocking is expensive, instead of fetching granular resources, just embed them inside others… to avoid requests.

+ Parallelism workaround for HTTP/1.x
+ Eliminates full request roundtrip

- Inlined resource can’t be cached independently
  - must duplicate bytes if resource is reused
  - invalidated if single byte in parent changes

- Breaks resource multiplexing and prioritization in HTTP/2
  - Inlined bytes are shipped with parent’s priority
  - Inlined bytes can’t be declined by the client

Inlining: because you'll also need…

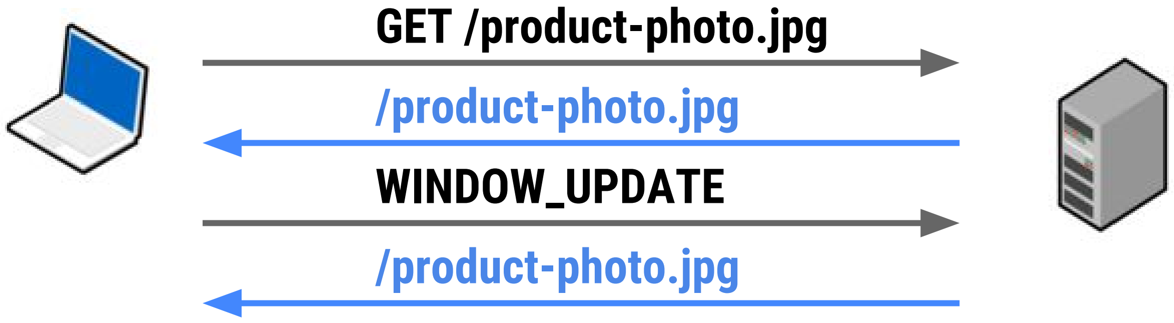


Server: *"You asked for /product/123, but you'll need app.js, product-photo-1.jpg, as well… I promise to deliver these to you. That is, unless you decline or cancel."*

Inlining is server push. Except, server push has nicer properties:

- + Delivers granular resources, which can be cached individually
- + Delivers granular resources, which can be mux'ed and prioritized correctly
- + Allows the client to opt-in / opt-out and control how and where it is used
  - + Server push is subject to flow control - e.g. "you can only push 5KB"
  - + Server push is optional and can be disabled by the client

Per-stream flow control + server push



- → **Client:** "I want first 20KB of photo.jpg"
- → **Server:** "Ok, 20KB… pausing stream until you tell me to send more."
- → **Client:** "Send me the rest now."

I want image geometry and preview, and I'll fetch the rest later...

- Flow control allows the client to pause stream delivery, and resume it later
- Flow control is a "credit-based" scheme
    - *Sending DATA frames decrements the window*
    - *WINDOW_UPDATE frames update the window*

http://chimera.labs.oreilly.com/books/1230000000545/ch12.html#_flow_control

Report card: resource inlining

Introduced as a workaround for head-of-line blocking in HTTP/1. Inlining is a latency optimization that eliminates full request roundtrip, and reduces “number of requests” in HTTP/1.

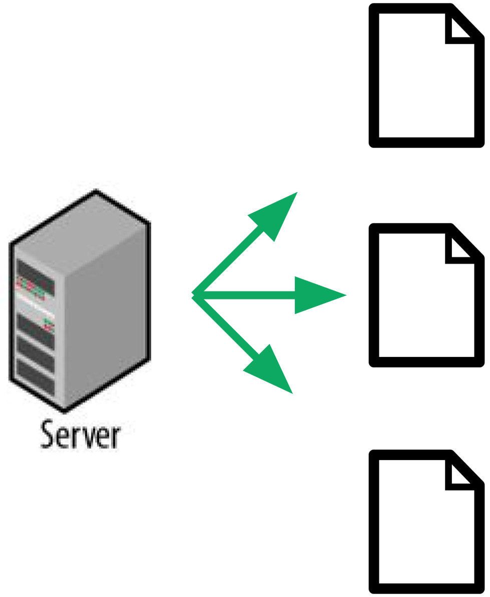
+ Removes full request roundtrip - i.e. “you’ll need this”
+ Parallelism workaround for HTTP/1.x
- Complicates our workflows, code and applications
- Breaks granular caching, updates, and revalidation
- Forces frequent and expensive invalidations

HTTP/1.x: use carefully and pay close attention to caching implications.

HTTP/2: replace with server push. The most naive strategy is still strictly better.

* Significant wins in compression are the only case where it might be useful.

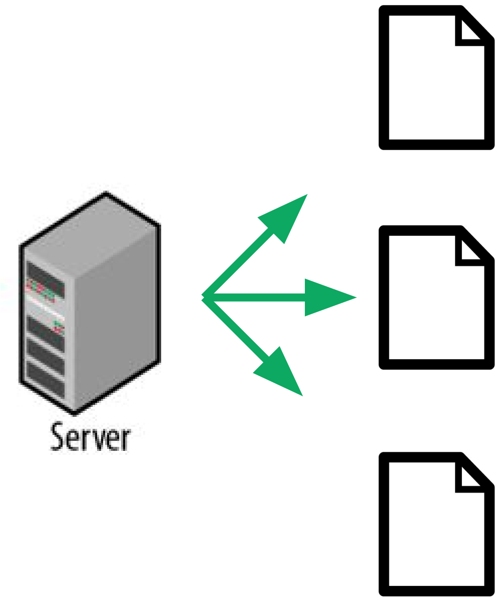
HTTP/2 server push instead of inlining...



Implementation and status

- Restricted to same-origin resources
- Opportunity for servers to become much smarter:
  - Don’t push on every request
  - Push based on observed traffic patterns
- UA’s iterating on implementation, e.g…
  - Auto RST_STREAM in-cache resources (works)
  - Pushing cache revalidations / invalidations? (TBD)
  - JavaScript API for processing server push? (TBD)

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



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

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

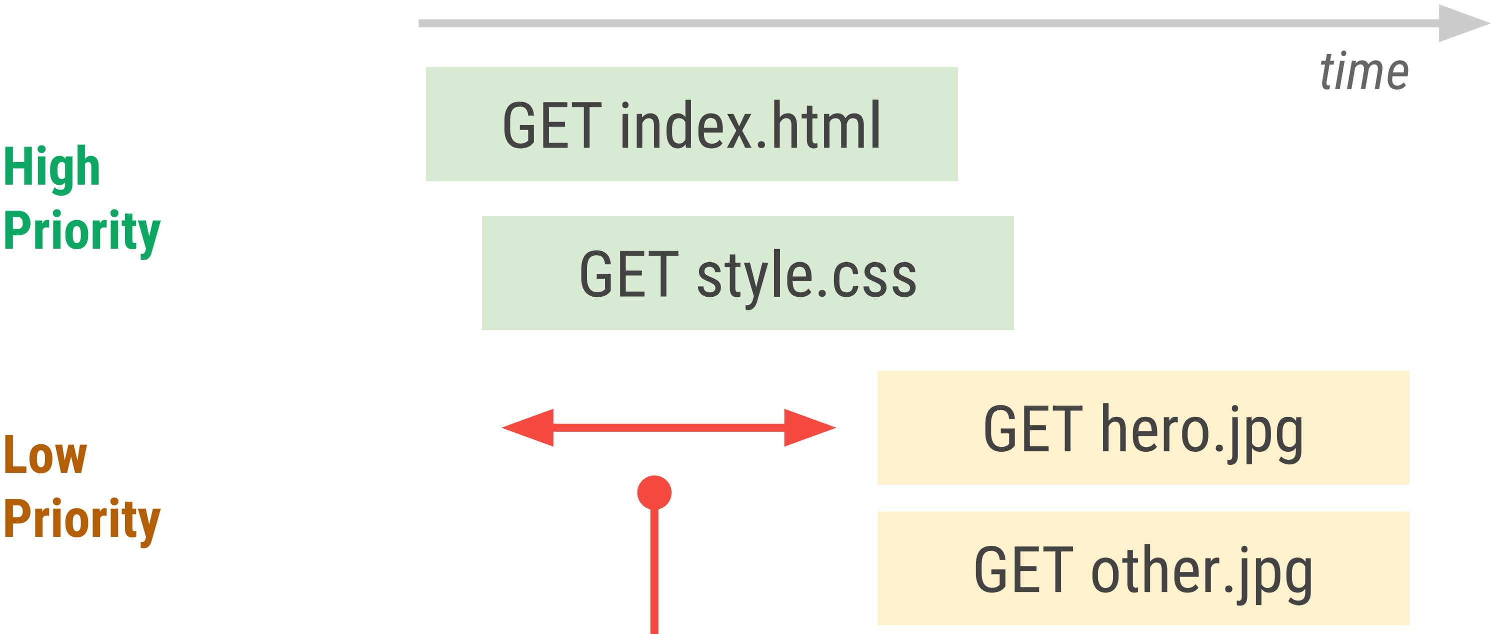
Lots of room for experimentation + innovation!

You... need to think about prioritization

*with HTTP/2 the browser is **relying on the server** to deliver data in an optimal way -- this is critical.*

Prioritization is key to optimized rendering...

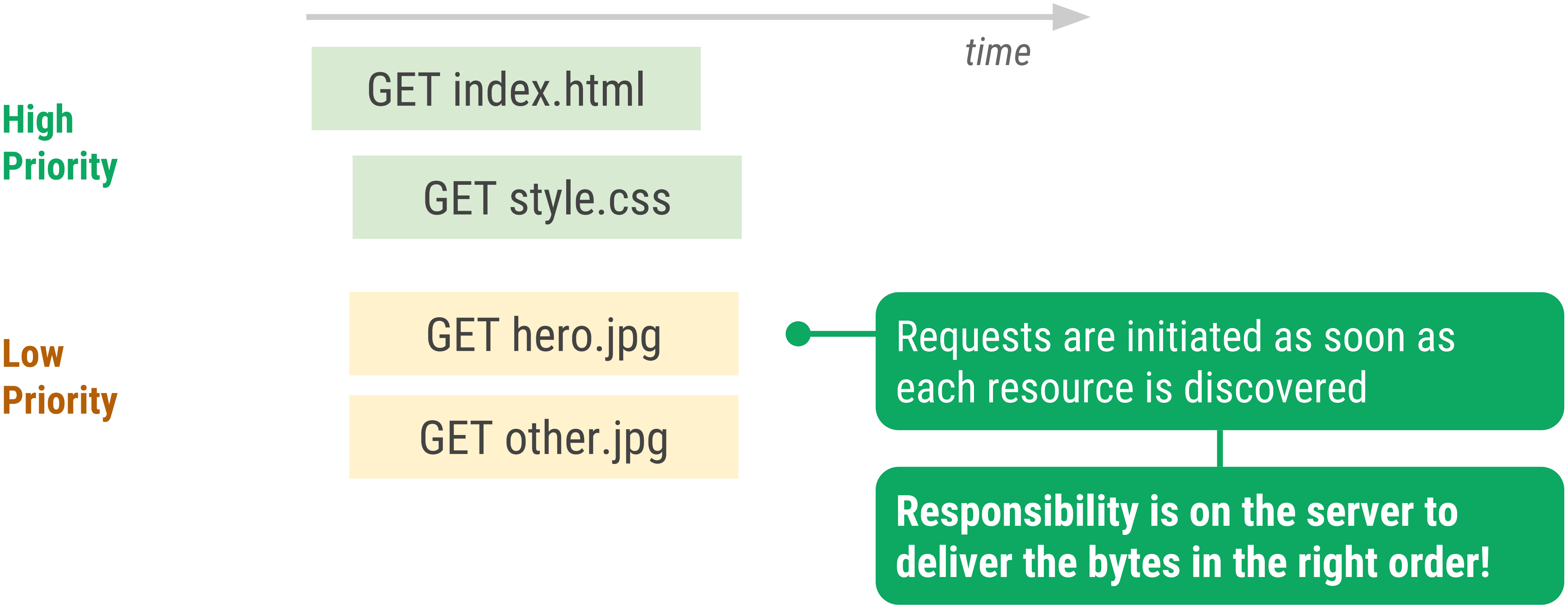
With HTTP/1.1 browsers “prioritizes” resources by holding a priority queue on the client and taking educated guesses for how to make the best use of available TCP connections… which delays request dispatch.



Delay dispatch, because we don’t want to “waste” bandwidth on images, we might have more critical resources, etc...

Prioritization is key to optimized rendering...

With HTTP/2 browsers prioritize requests based on type/context, and immediately dispatch the request as soon as the resource is discovered. The priority is communicated to the server as weights + dependencies.



time

High Priority

GET index.html

GET style.css

Low Priority

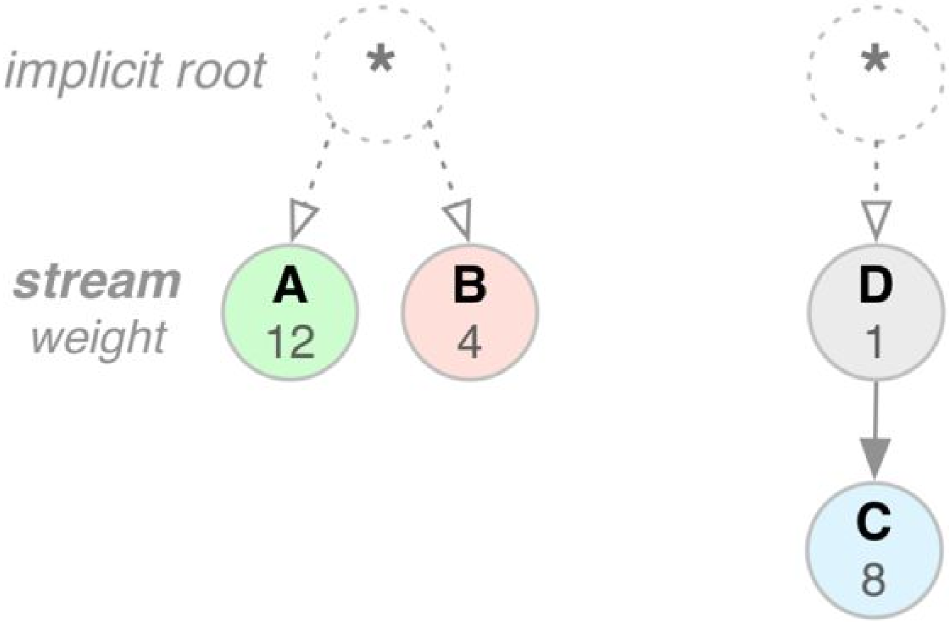
GET hero.jpg

GET other.jpg

Requests are initiated as soon as each resource is discovered

Responsibility is on the server to deliver the bytes in the right order!

Stream prioritization in HTTP/2...

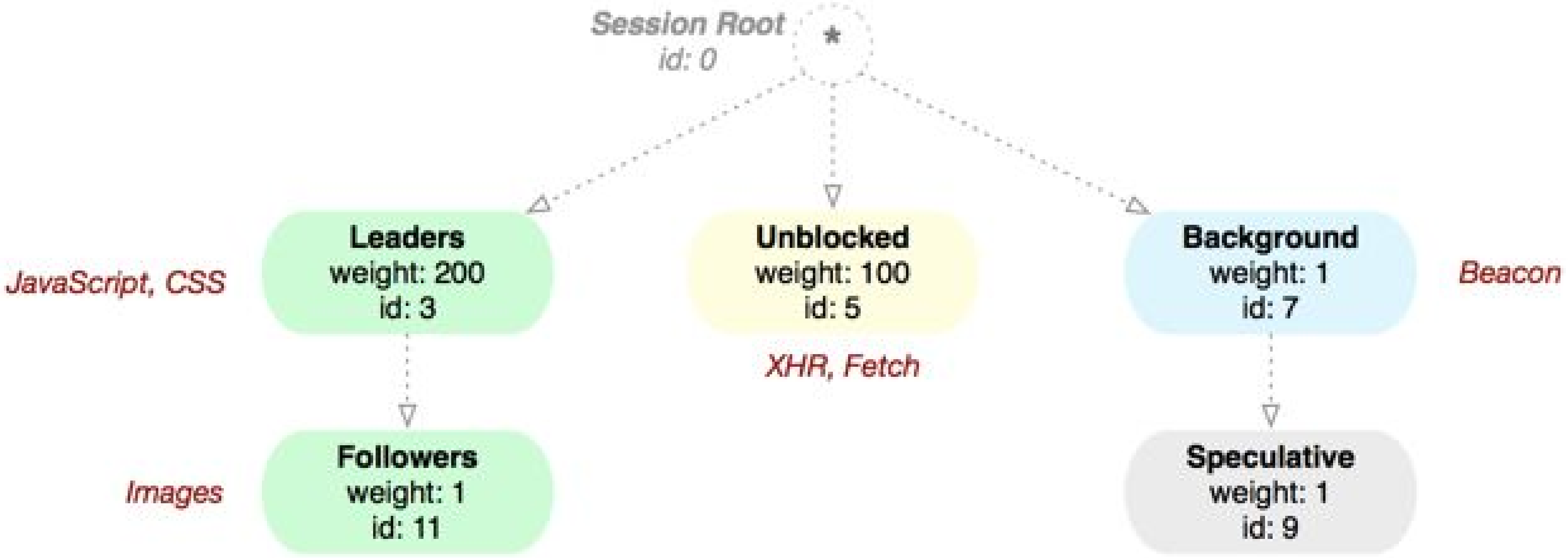


- **Each stream can have a weight**
  - [1-256] integer value
- **Each stream can have a dependency**
  - parent is another stream ID

1. E.g. style.css (“A”) should get 2/3rd’s of available resources as compared to logo.jpg (“B”)
2. E.g. product-photo-1.jpg (“D”) should be delivered before product-photo-2.jpg (“C”)

* Prioritization is an advisory hint to the server, it does not provide strict delivery semantics.

HTTP/2 prioritization in Firefox...



1. “Leaders” should get 2/3rd of resources, “Unblocked” bulk of the rest
2. “Leaders” should be prioritized ahead of “Followers” - e.g. HTML/CSS/JS ahead of images.
3. “Background” tasks should be done… in the background!

http://bitsup.blogspot.se/2015/01/http2-dependency-priorities-in-firefox.html

With HTTP/2 the **browser relies on the server** to deliver the response data in an optimal way.

It’s not just the number of bytes, or requests per second, but the order in which bytes are delivered. ***Test your HTTP/2 server very carefully.***

- Does it respect stream and connection flow control? (MUST)
- Does it support dependencies and weights? (MUST)
- Does it enable use of server push?

	HTTP/1.x	HTTP/2
Reduce DNS lookups	✓	✓
Reuse TCP connecti	✓	✓
Use a Content Delivery Netv		
Minimize number of HTTP redir		
Eliminate unnecessary request bytes		
Compress assets during transfer	✓	✓
Cache resources on the client	✓	✓
Eliminate unnecessary resources	✓	✓
Apply domain sharding	Revisit (max 2)	Remove
Concatenate resources	Carefully (caching)	Remove (compression)
Inline resources	Carefully (caching)	Remove, use server push

Pick your HTTP/2 server carefully!

… we need better test suites and benchmarks.

Learn more...

hpbn.co/http2

Slides...

bit.ly/http2-opt

+Ilya Grigorik
@igrigorik