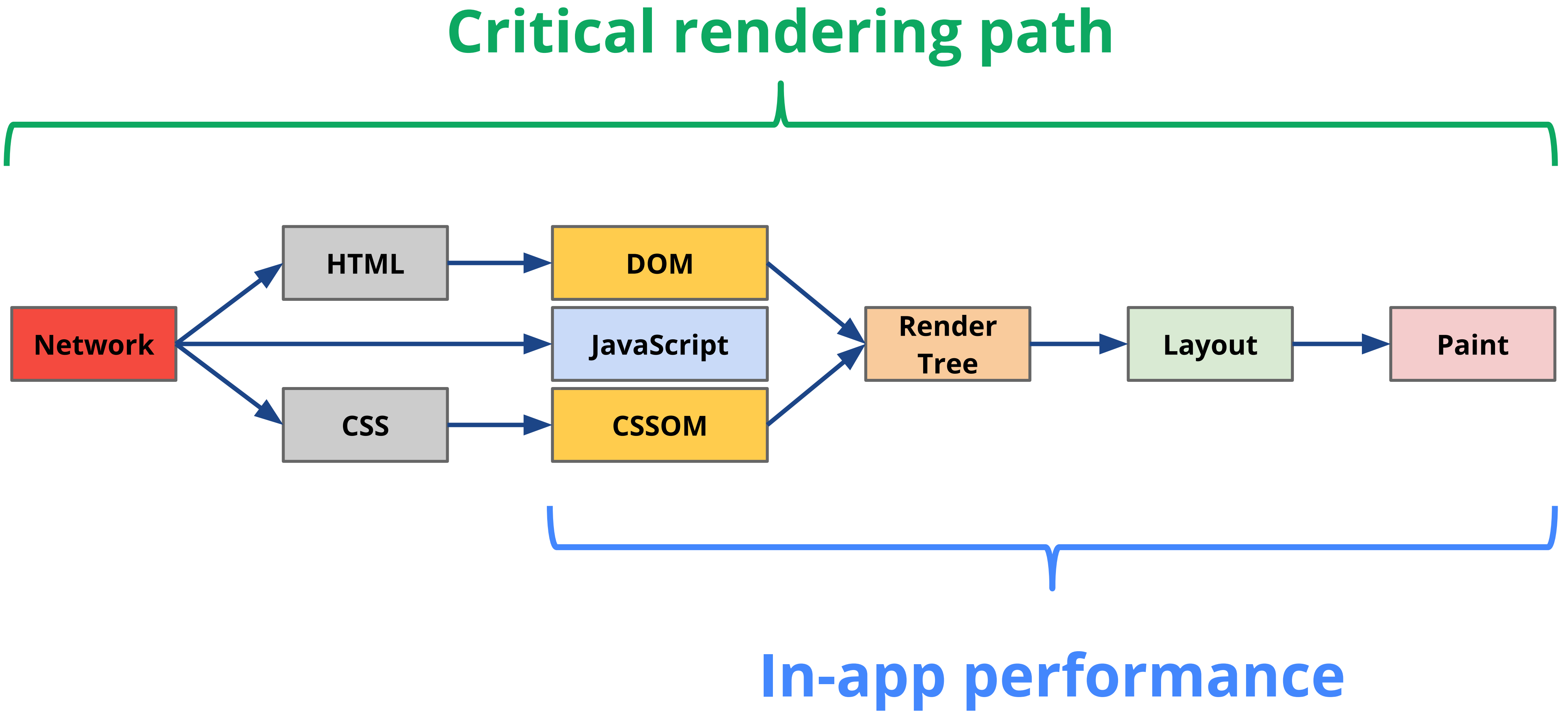
Building Faster Websites

crash course on web performance

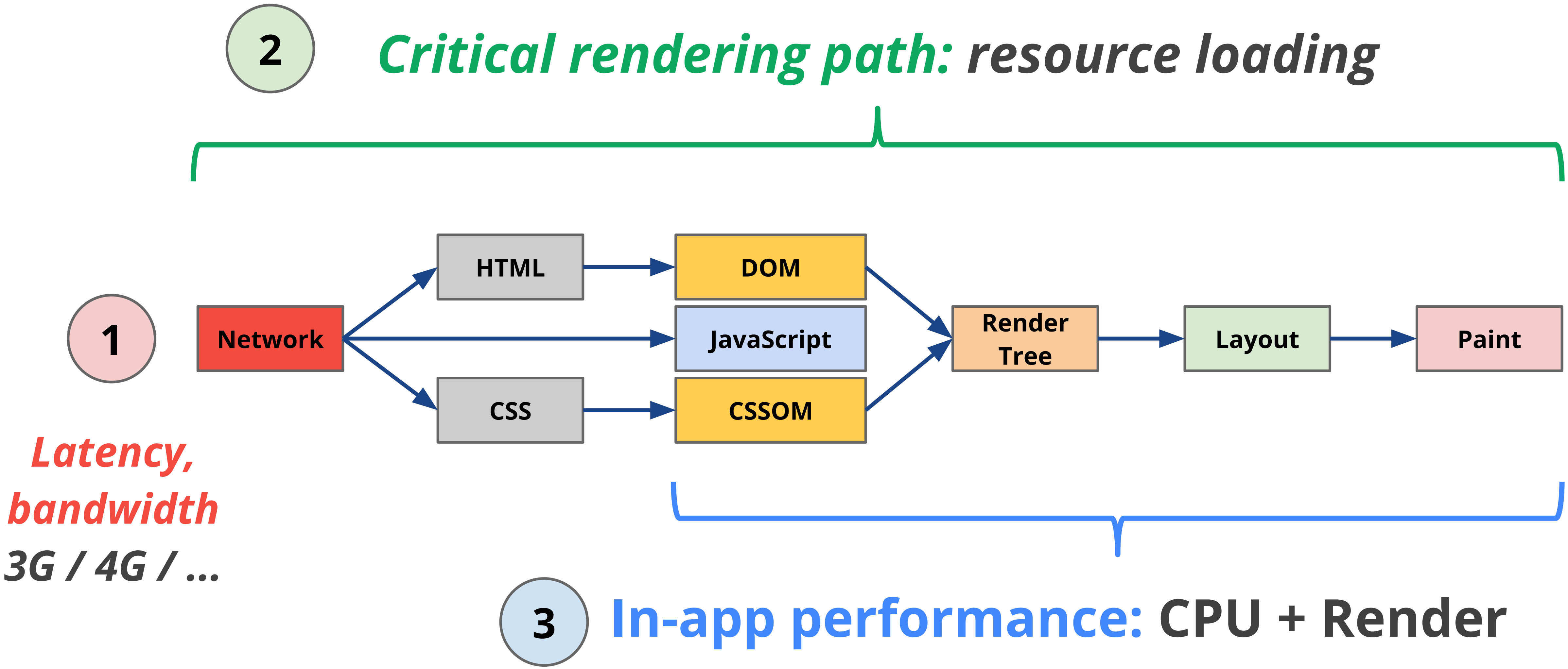
Ilya Grigorik - @igrigorik
Make The Web Fast
Google

Web performance in one slide...



Thanks. Questions?

Twitter @igrigorik
G+ gplus.to/igrigorik
Web igvita.com


2
Critical rendering path: resource loading
HTML
DOM
1
Network
JavaScript
Render Tree
Layout
Paint
CSS
CSSOM
Latency, bandwidth
3G / 4G / ...
3
In-app performance: CPU + Render

What's the impact of slow sites?

Lower conversions and engagement, higher bounce rates...

bing server delays experiment

	Distinct Queries/User	Query Refinement	Revenue/User	Any Clicks	Satisfaction	Time to Click (increase in ms)
50ms	-	-	-	-	-	-
200ms	-	-	-	-0.3%	-0.4%	500
500ms	-	-0.6%	-1.2%	-1.0%	-0.9%	1200
1000ms	-0.7%	-0.9%	-2.8%	-1.9%	-1.6%	1900
2000ms	-1.8%	-2.1%	-4.3%	-4.4%	-3.8%	3100

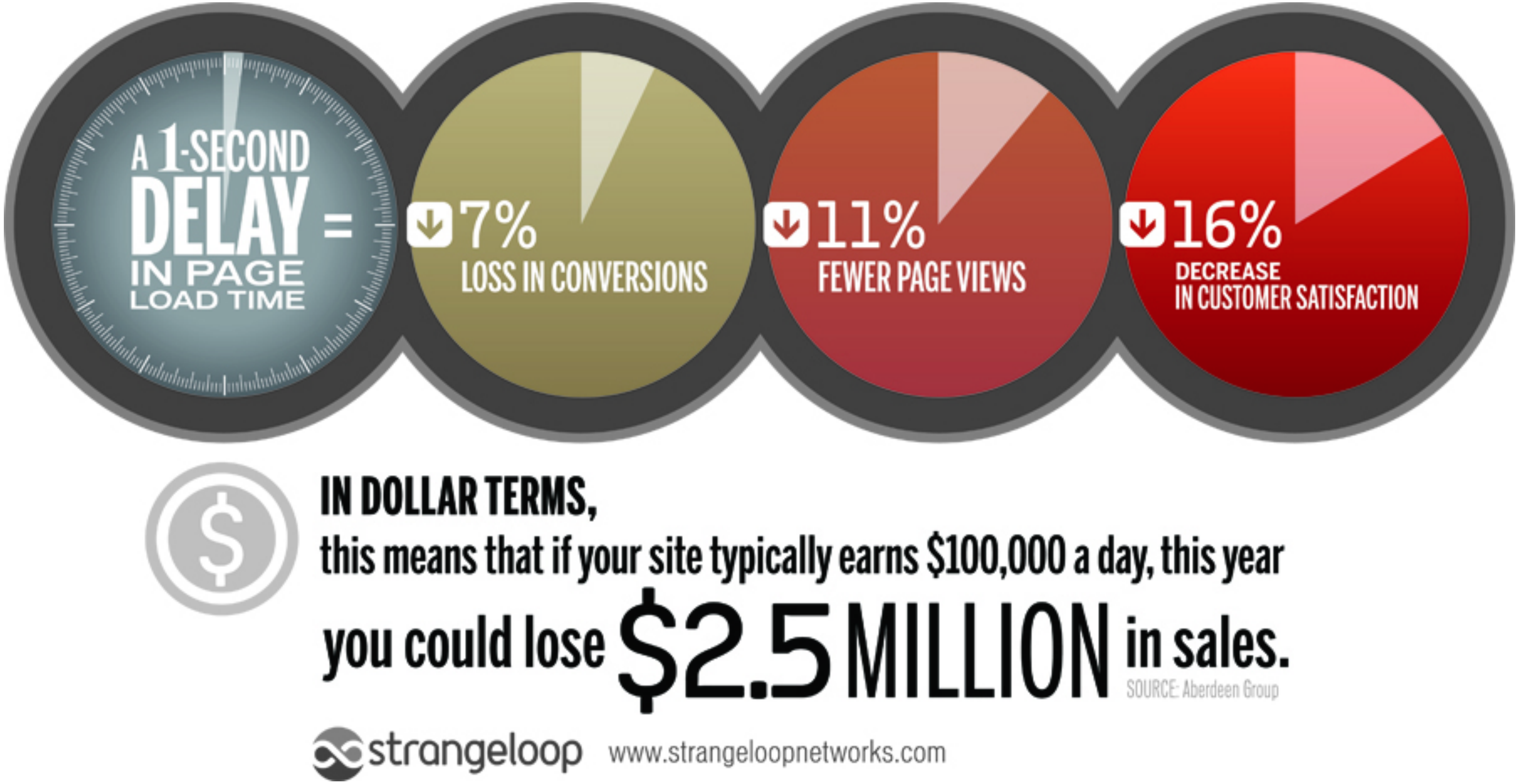
- Means no statistically significant change

"2000 ms delay reduced per user revenue by 4.3%!"

- Strong negative impacts
- Roughly linear changes with increasing delay
- Time to Click changed by roughly double the delay

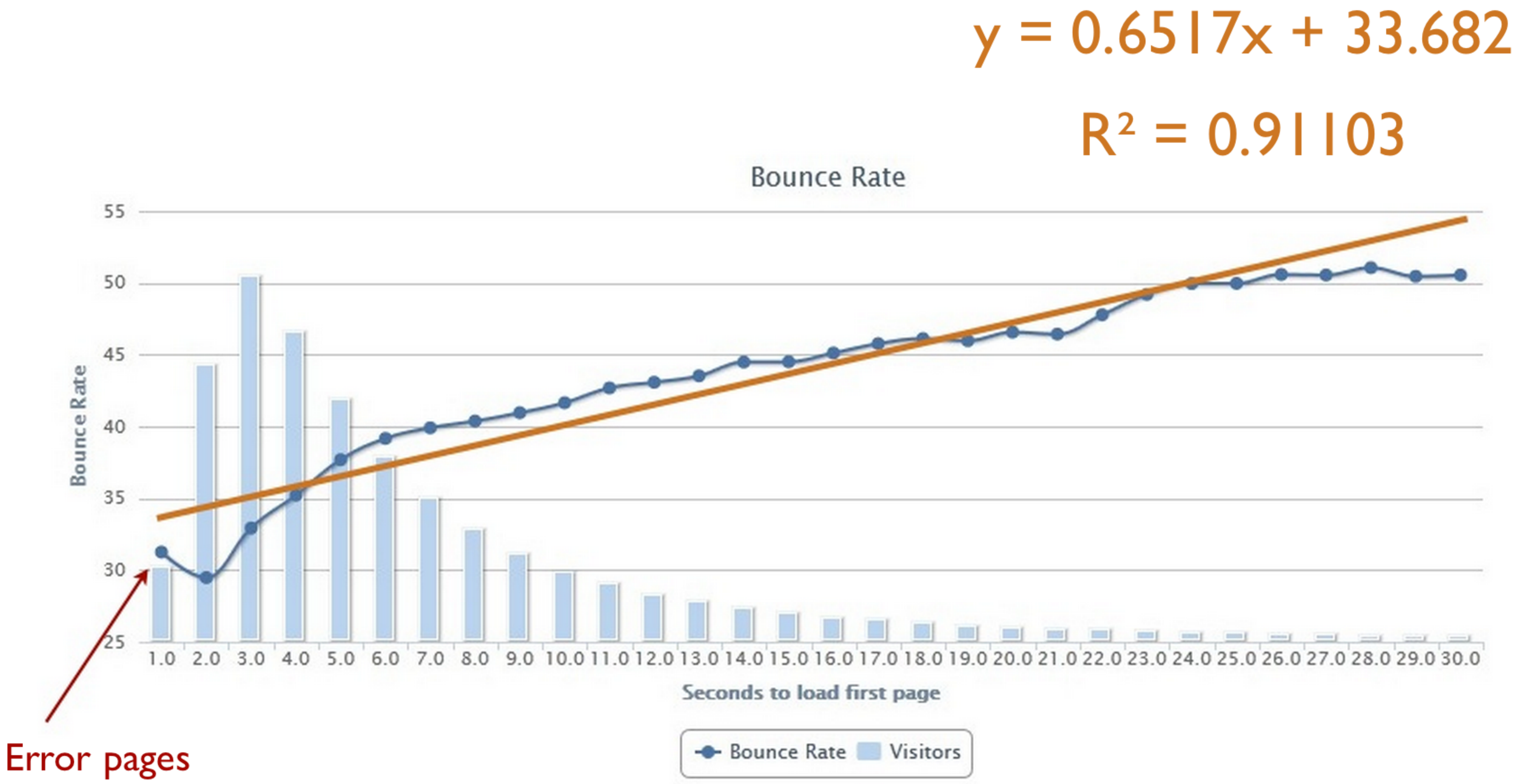
[Performance Related Changes and their User Impact]

Impact of 1-second delay...



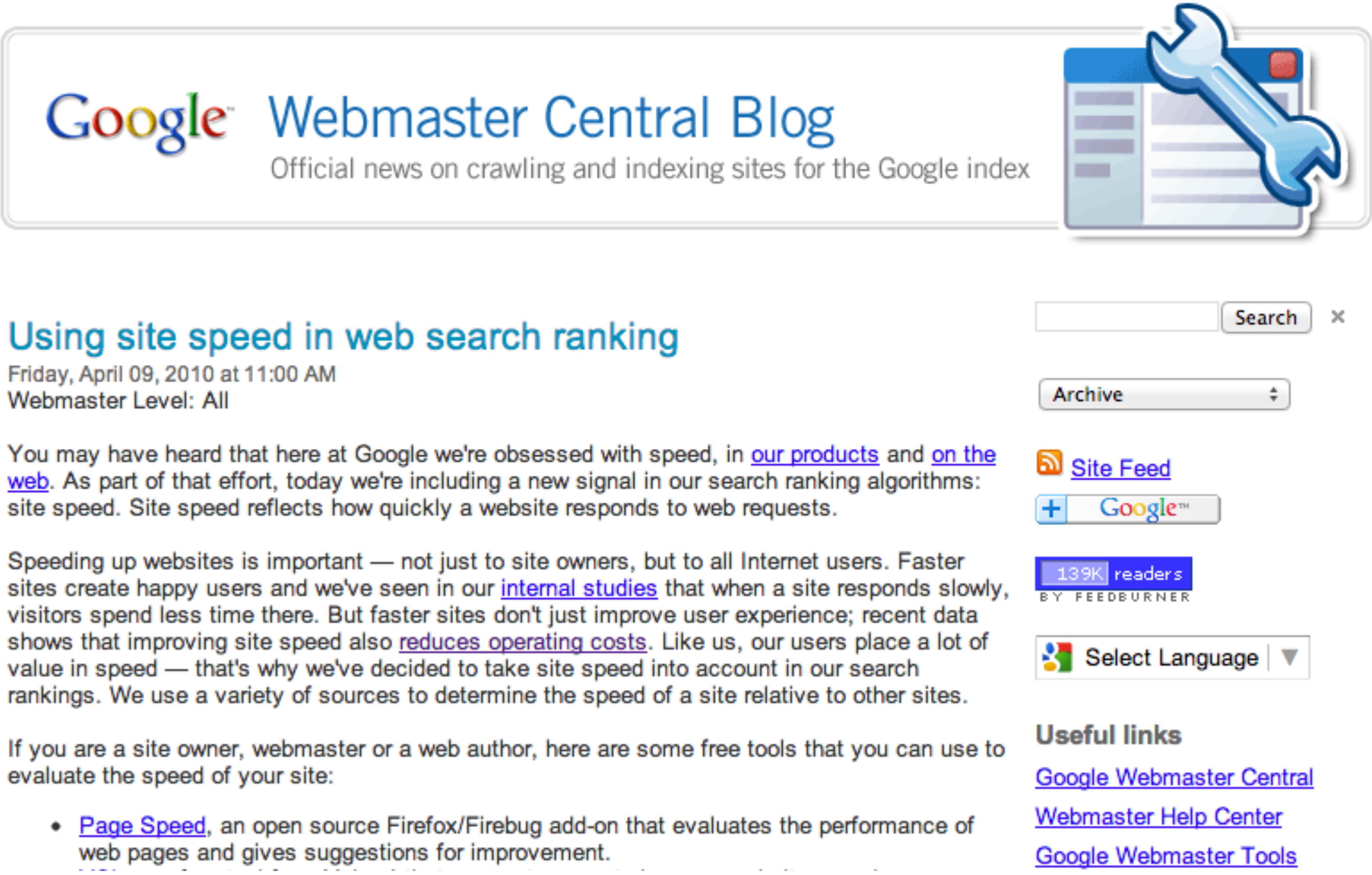
Impact of 1-second delay - Strangeloop

How speed affects bounce rate



[Yo ho ho and a few billion pages of RUM]()

Site speed is a signal for search



Google Webmaster Central Blog

Official news on crawling and indexing sites for the Google index

Using site speed in web search ranking

Friday, April 09, 2010 at 11:00 AM

Webmaster Level: All

You may have heard that here at Google we're obsessed with speed, in our products and on the web. As part of that effort, today we're including a new signal in our search ranking algorithms: site speed. Site speed reflects how quickly a website responds to web requests.

Speeding up websites is important — not just to site owners, but to all Internet users. Faster sites create happy users and we've seen in our internal studies that when a site responds slowly, visitors spend less time there. But faster sites don't just improve user experience; recent data shows that improving site speed also reduces operating costs. Like us, our users place a lot of value in speed — that's why we've decided to take site speed into account in our search rankings. We use a variety of sources to determine the speed of a site relative to other sites.

If you are a site owner, webmaster or a web author, here are some free tools that you can use to evaluate the speed of your site:

- Page Speed, an open source Firefox/Firebug add-on that evaluates the performance of web pages and gives suggestions for improvement.

Search

Archive

Site Feed

Google

139K readers

BY FEEDBURNER

Select Language

Useful links

Google Webmaster Central

Webmaster Help Center

Google Webmaster Tools

"We encourage you to start looking at your site's speed — not only to improve your ranking in search engines, but also to improve everyone's experience on the Internet."

Google Search Quality Team

Using site speed in web search ranking

Speed is a feature.

So, how are we doing today?

Okay, I get it, speed matters... but, are we there yet?

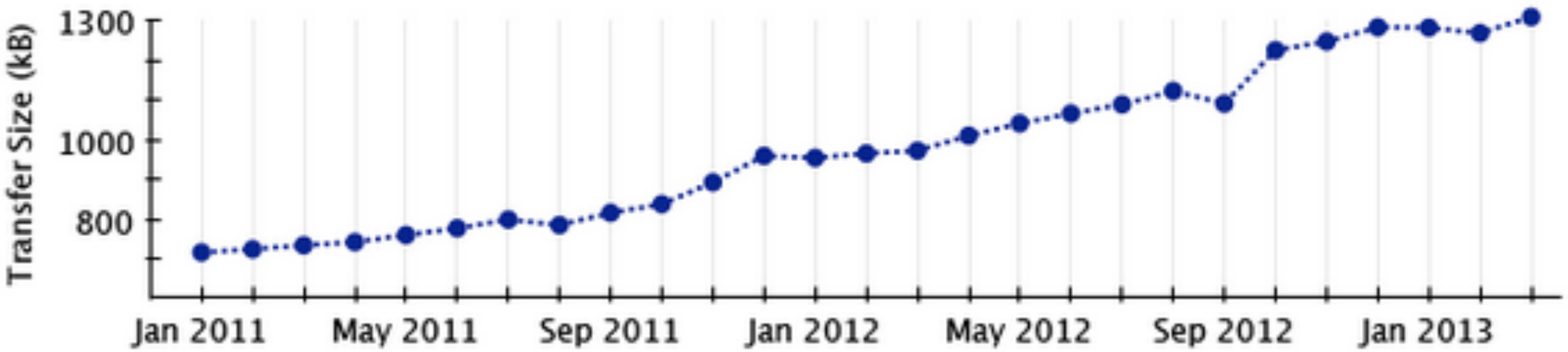
Delay	User reaction
0 - 100 ms	Instant
100 - 300 ms	Slight perceptible delay
300 - 1000 ms	Task focus, perceptible delay
1 s+	Mental context switch
10 s+	I'll come back later...

"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.

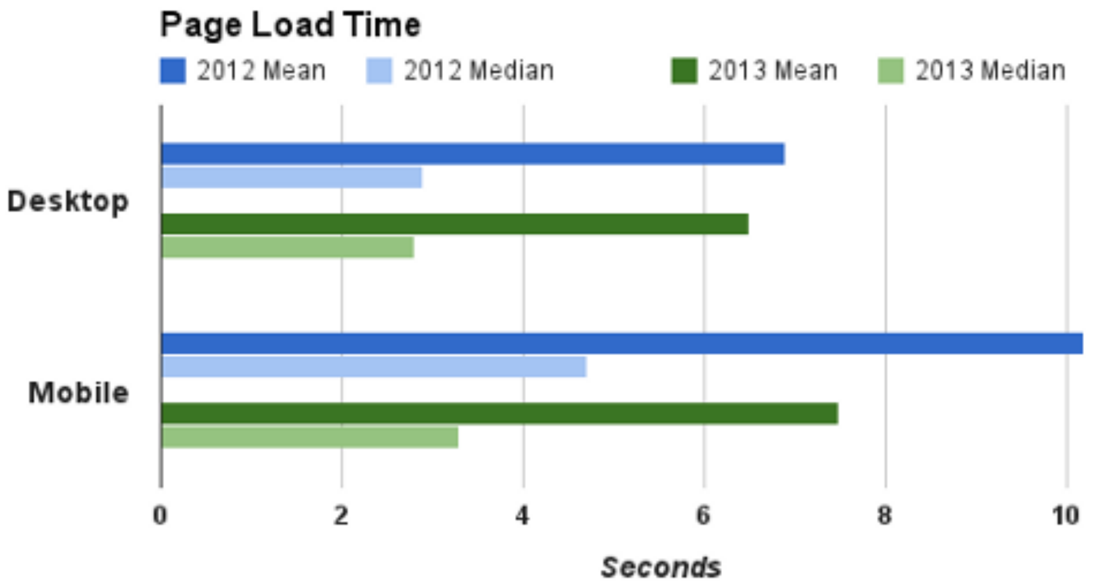
Our applications are complex, and growing...

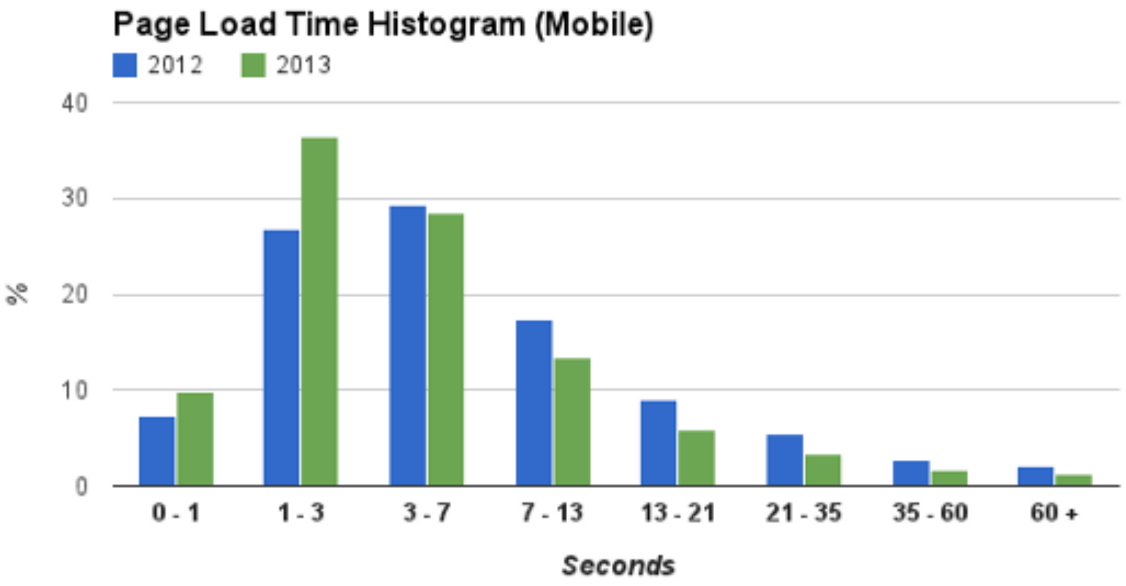


Content Type	Desktop		Mobile	
	Avg # of requests	**Avg size**	**Avg # of requests**	**Avg size**
HTML	10	56 KB	6	40 KB
Images	**56**	**856 KB**	**38**	**498 KB**
Javascript	**15**	**221 KB**	**10**	**146 KB**
CSS	5	36 KB	3	27 KB
Total	**86+**	**1169+ KB**	**57+**	**711+ KB**

Ouch!

HTTP Archive





Desktop: ~3.1 s

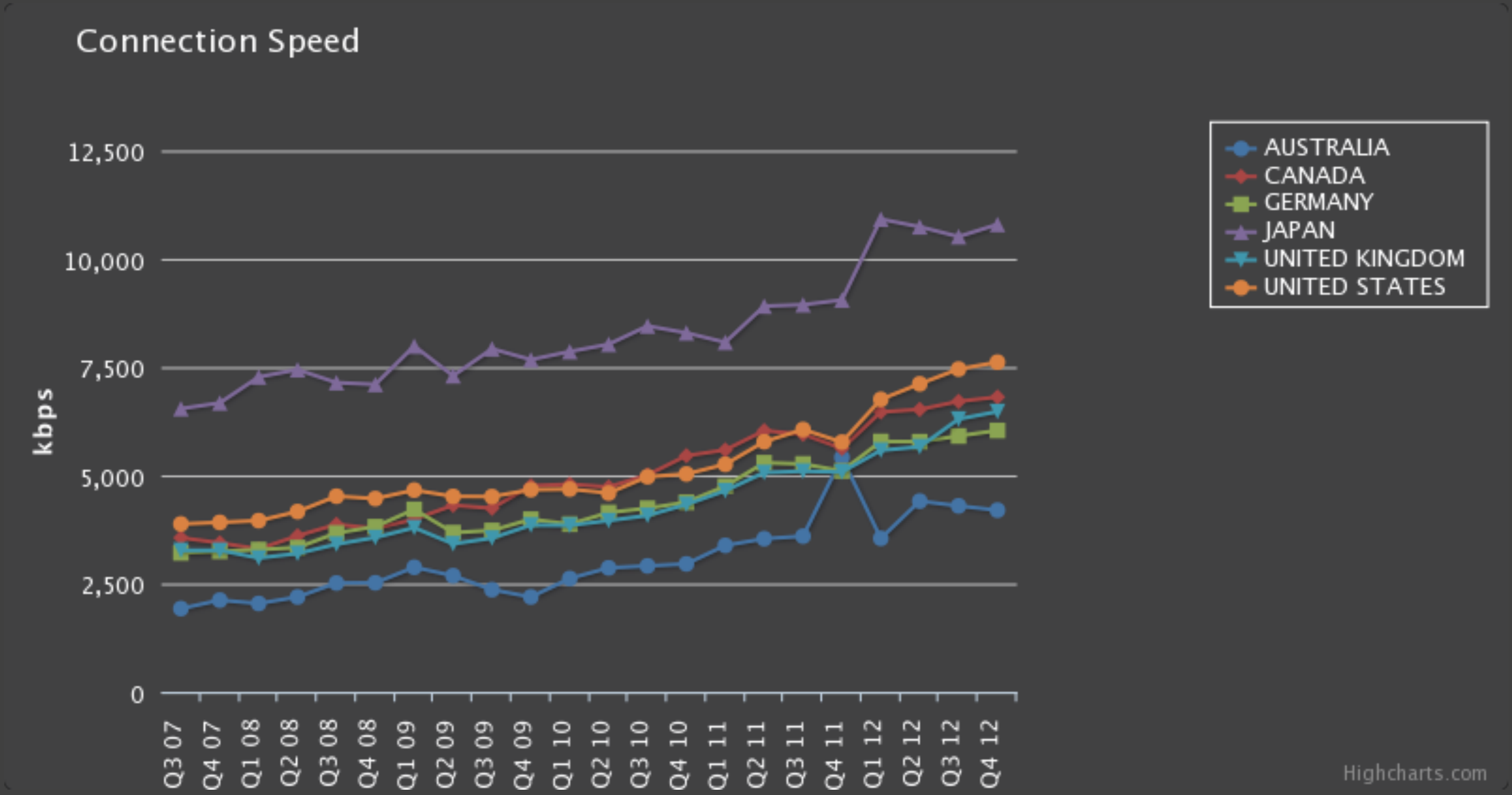
Mobile: ~3.5 s

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

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

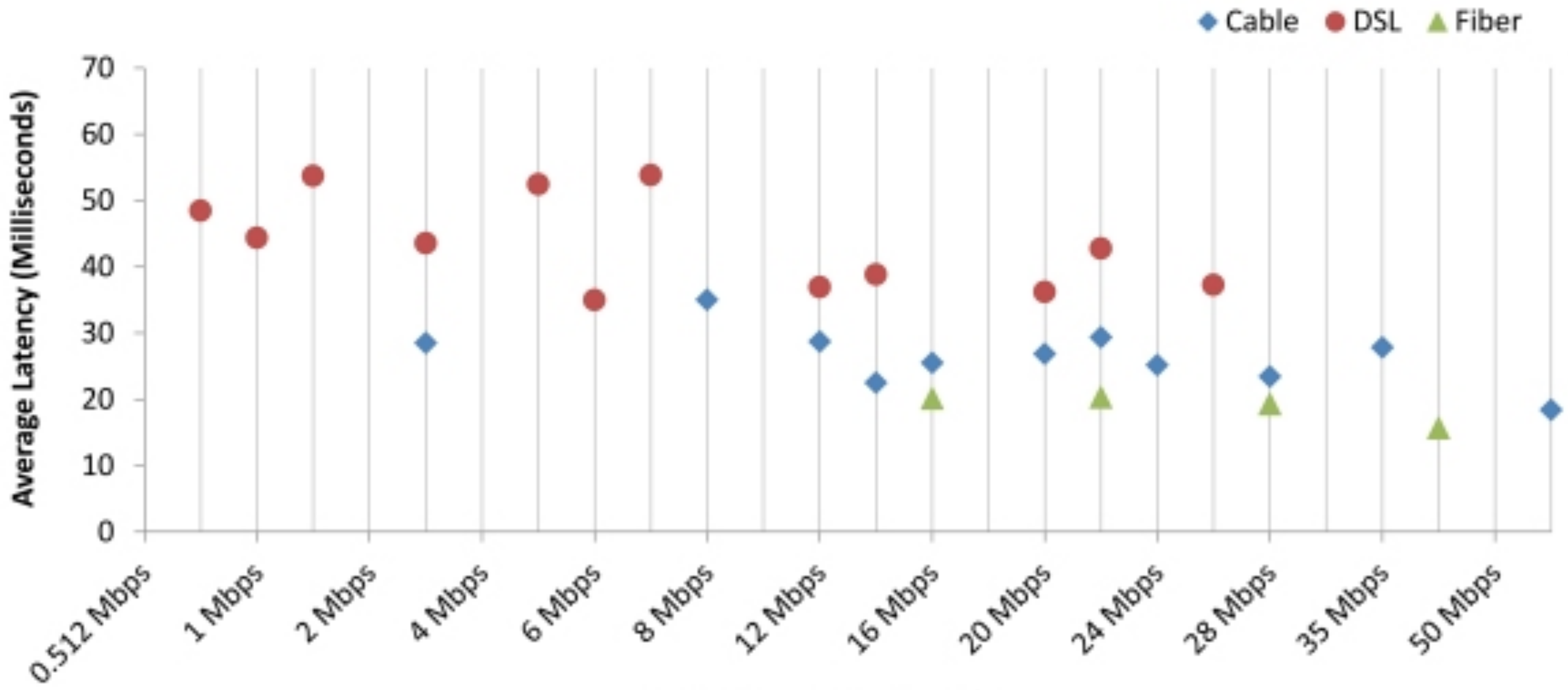
Great, network will save us?

Right, right? We can just sit back and...



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

State of the Internet - Akamai - 2007-2012



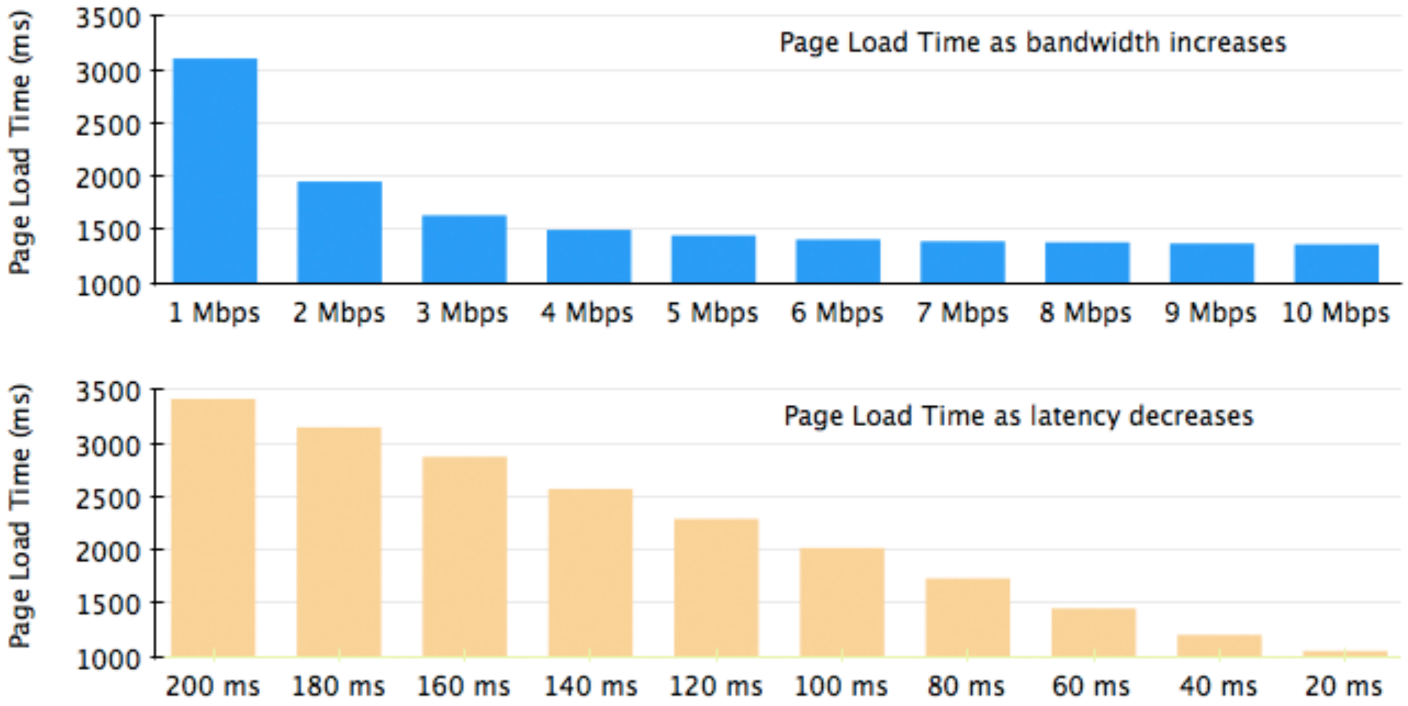
Fiber-to-the-home *services provided* ***18 ms*** *round-trip latency on average, while* ***cable-based*** *services averaged* ***26 ms****, and* ***DSL-based*** *services averaged* ***43 ms****. This compares to 2011 figures of 17 ms for fiber, 28 ms for cable and 44 ms for DSL.*

Worldwide: ~100 ms

US: ~50~60 ms

Average RTT to Google in 2012 was...

Latency *vs.* Bandwidth impact on Page Load Time



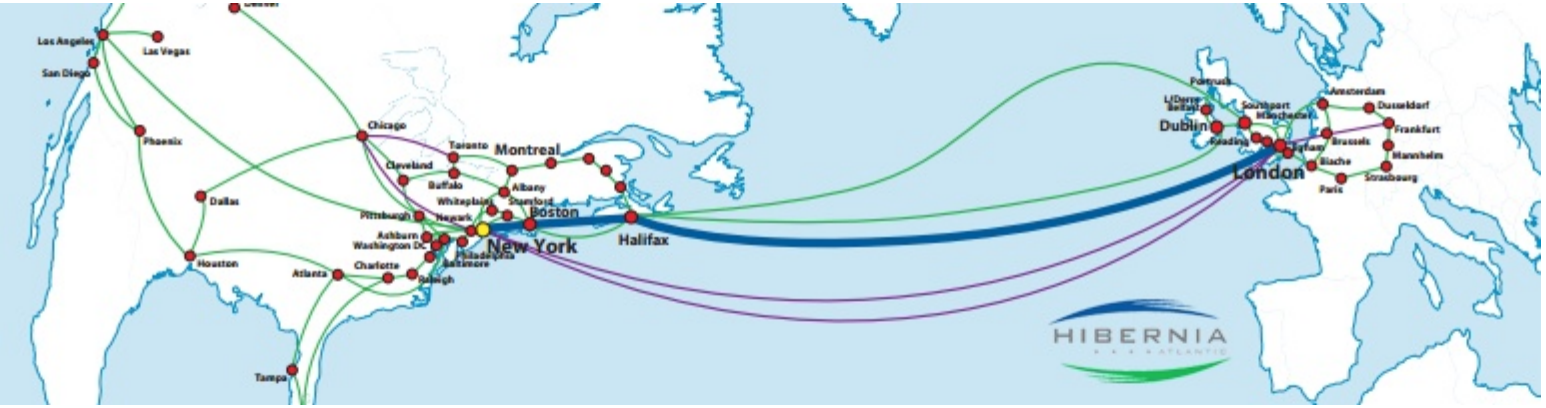
Single digit % perf improvement after 5 Mbps

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)*

- **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 is the new Performance Bottleneck]

Mobile, oh Mobile...

*"Users of the **Sprint 4G network** can expect to experience average speeds of 3 Mbps to 6 Mbps download and up to 1.5 Mbps upload with an **average latency of 150 ms**. On the **Sprint 3G** network, users can expect to experience average speeds of 600 Kbps - 1.4 Mbps download and 350 Kbps - 500 Kbps upload with an **average latency of 400 ms**."*

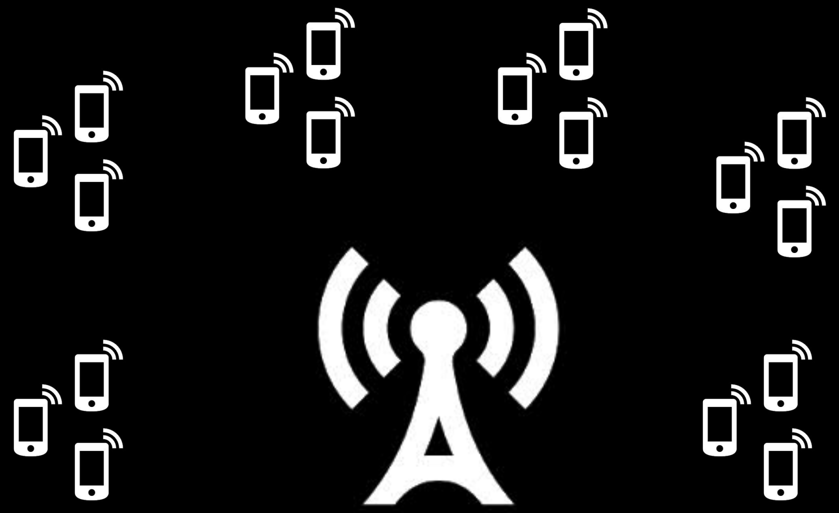
	3G	4G
Sprint	150 - 400 ms	150 ms
AT&T	150 - 400 ms	100 - 200 ms

AT&T

Why are mobile latencies so high?

... and variable?

Design constraint #1: "Stable" performance + scalability



- **Control** over network performance and resource allocation
- Ability to manage **10~100's of active devices** within single cell
- Coverage of much larger area

Design constraint #2: Maximize battery life

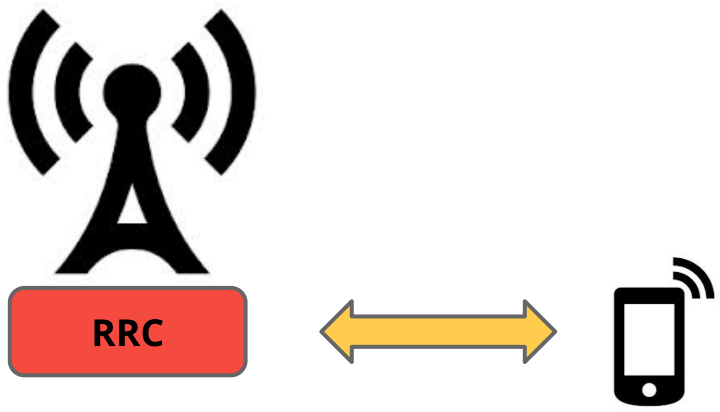
- Radio is the **second most expensive** component (after screen)
- Limited amount of available power (as you are well aware)

Radio Resource Controller

- **Phone:** Hi, I want to transmit data, *please?*
- **RRC:** OK.
    - Transmit in [x-y] timeslots
    - Transmit with Z power
    - Transmit with Q modulation

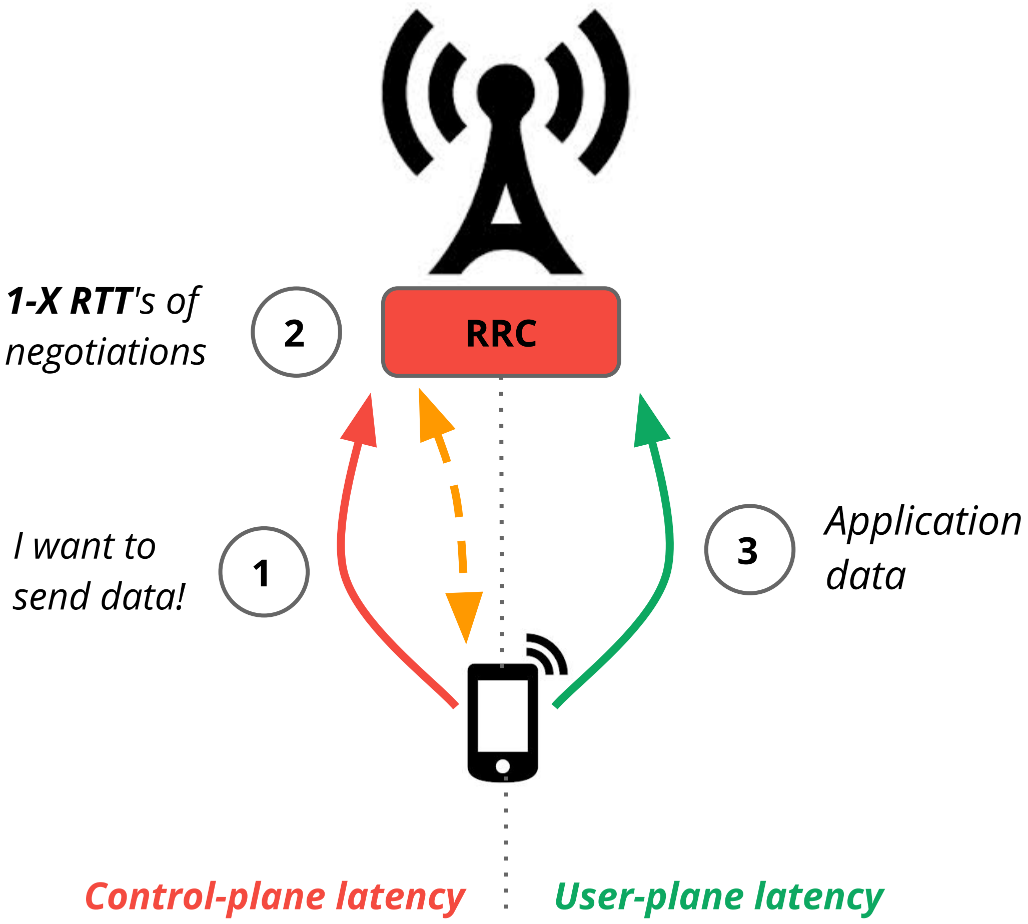
... (some time later) ...

- **RRC:** Go into low power state.



*All **communication and power management is centralized** and managed by the RRC.*

3G / 4G Control and User plane latencies

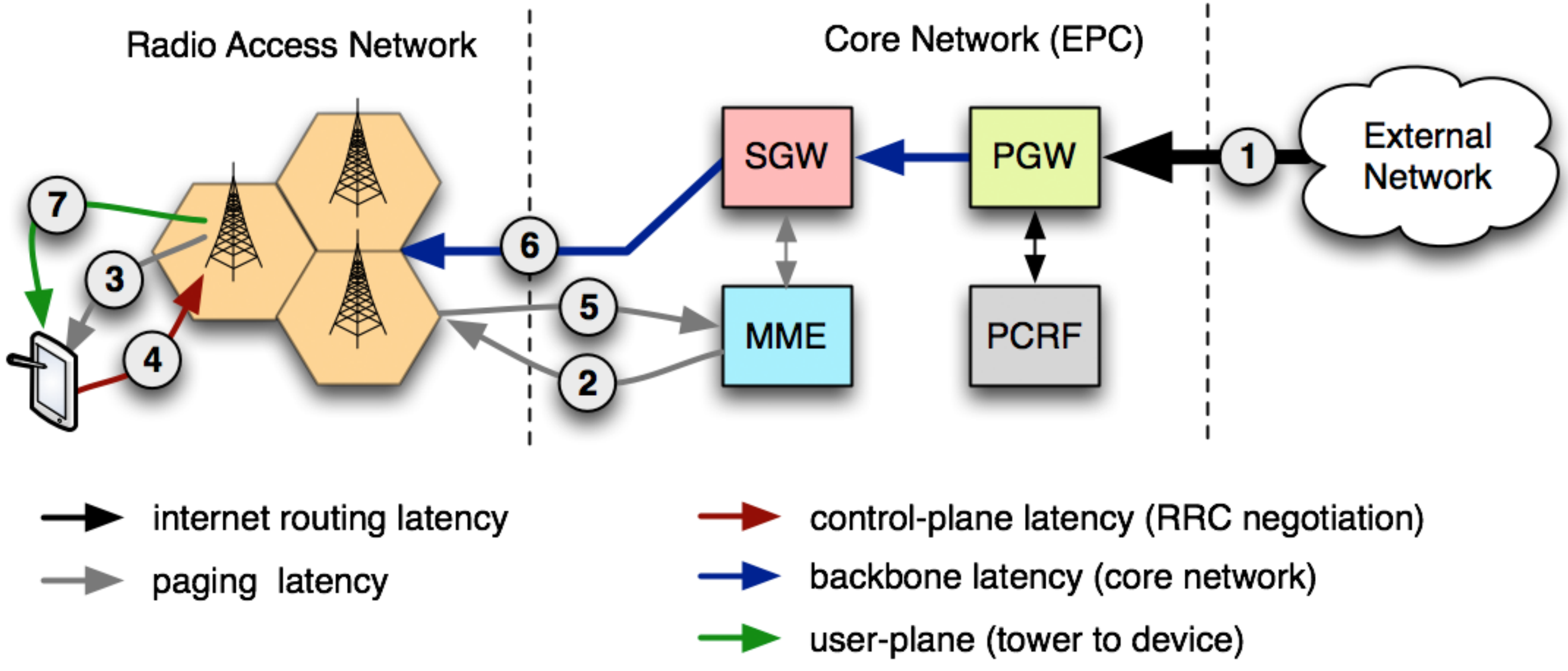


- There is a **one time** cost for control-plane negotiation
- **User-plane latency** is the one-way latency between packet availability in the device and packet at the base station

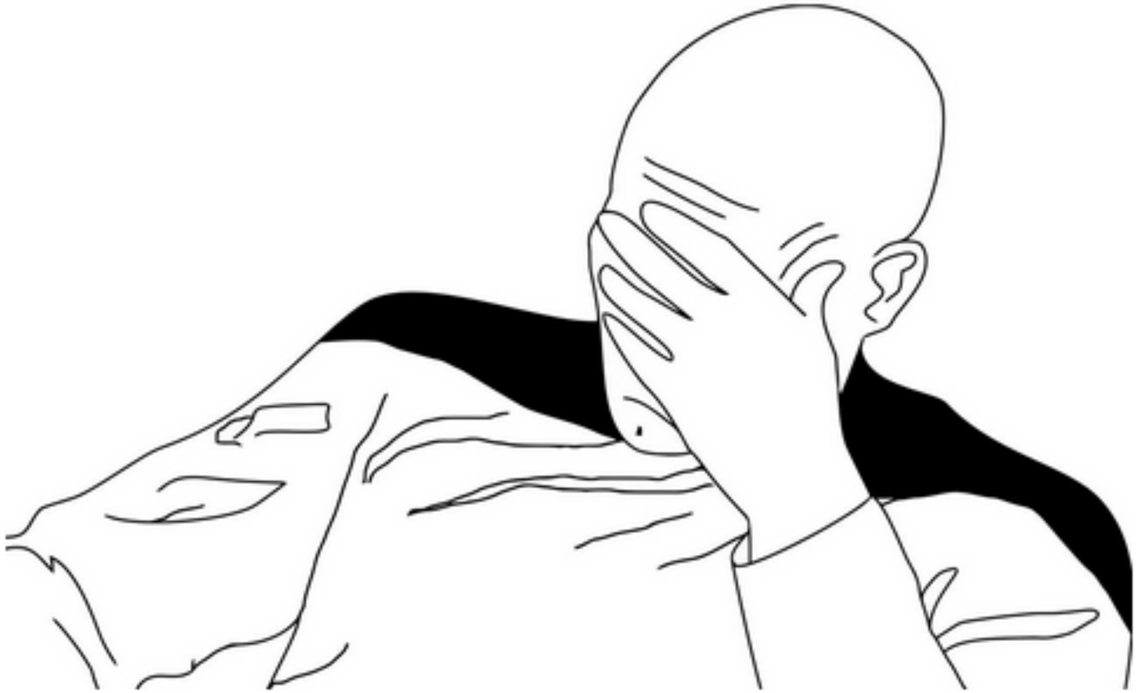
	LTE	HSPA+	3G
Idle to connected latency	< 100 ms	< 100 ms	**< 2.5 s**
User-plane one-way latency	< 5 ms	< 10 ms	< 50 ms

Same process happens for incoming data, just reverse steps 1 and 2

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



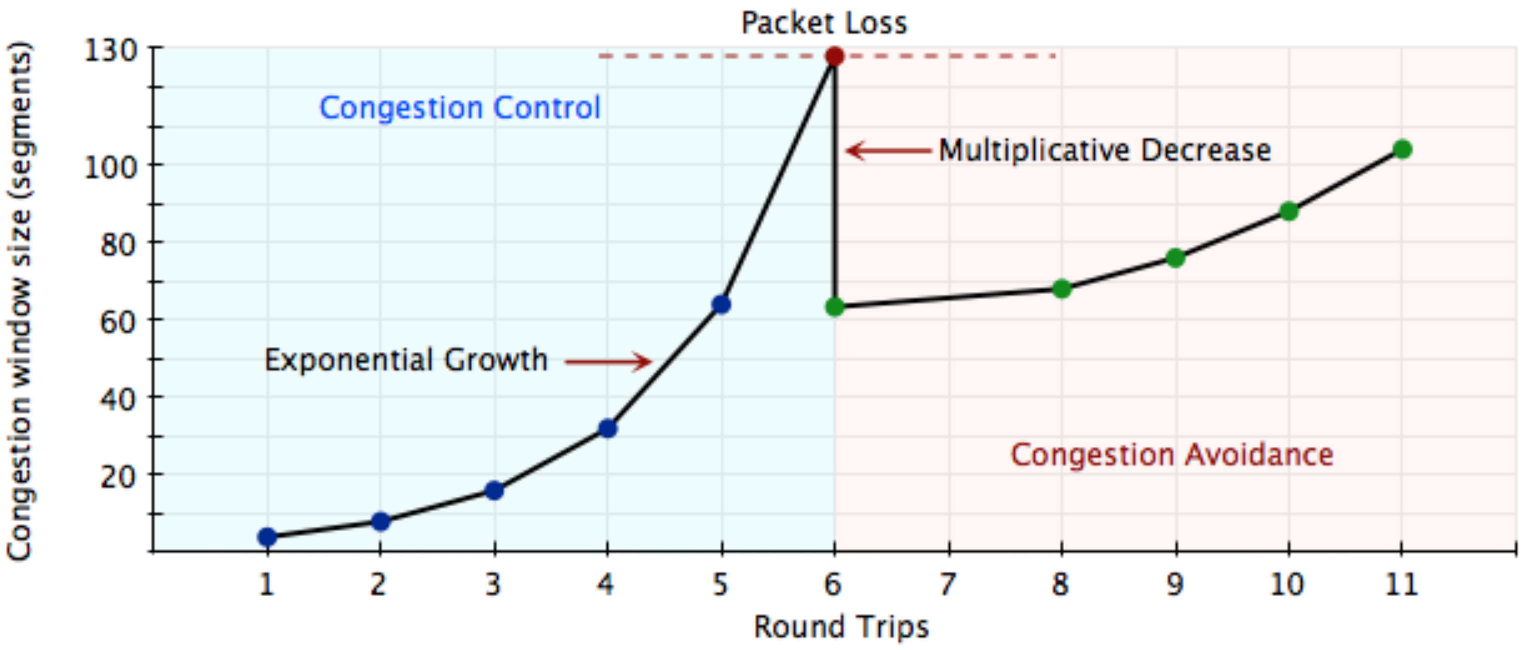
*... all that to send a **single TCP packet?***

Why is latency the bottleneck?

... what's the relationship between latency and bandwidth?

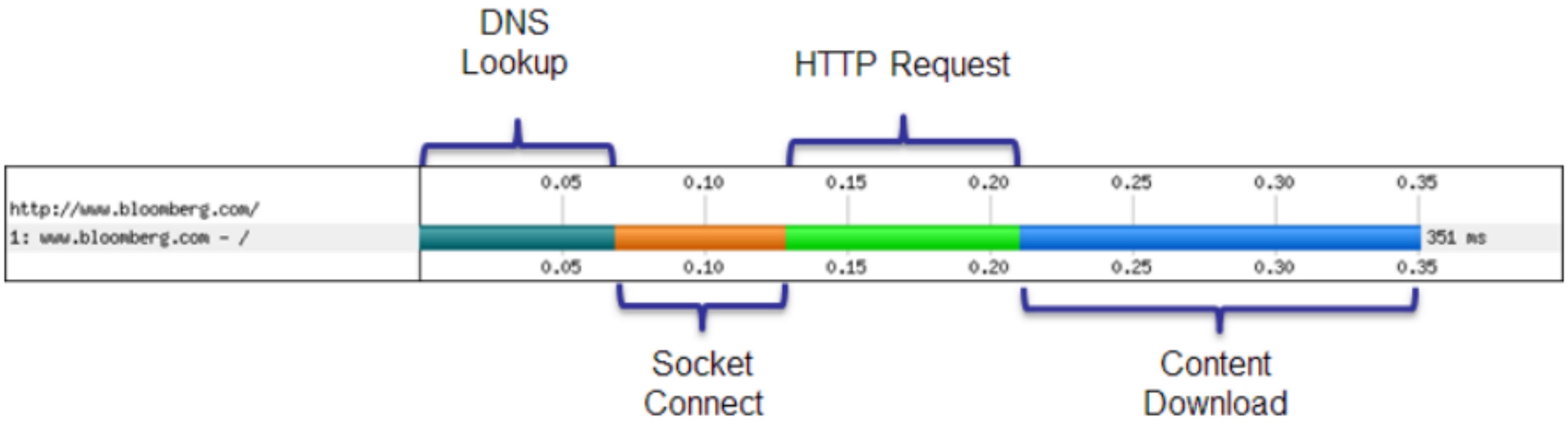
TCP Congestion Control & Avoidance...

- 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.

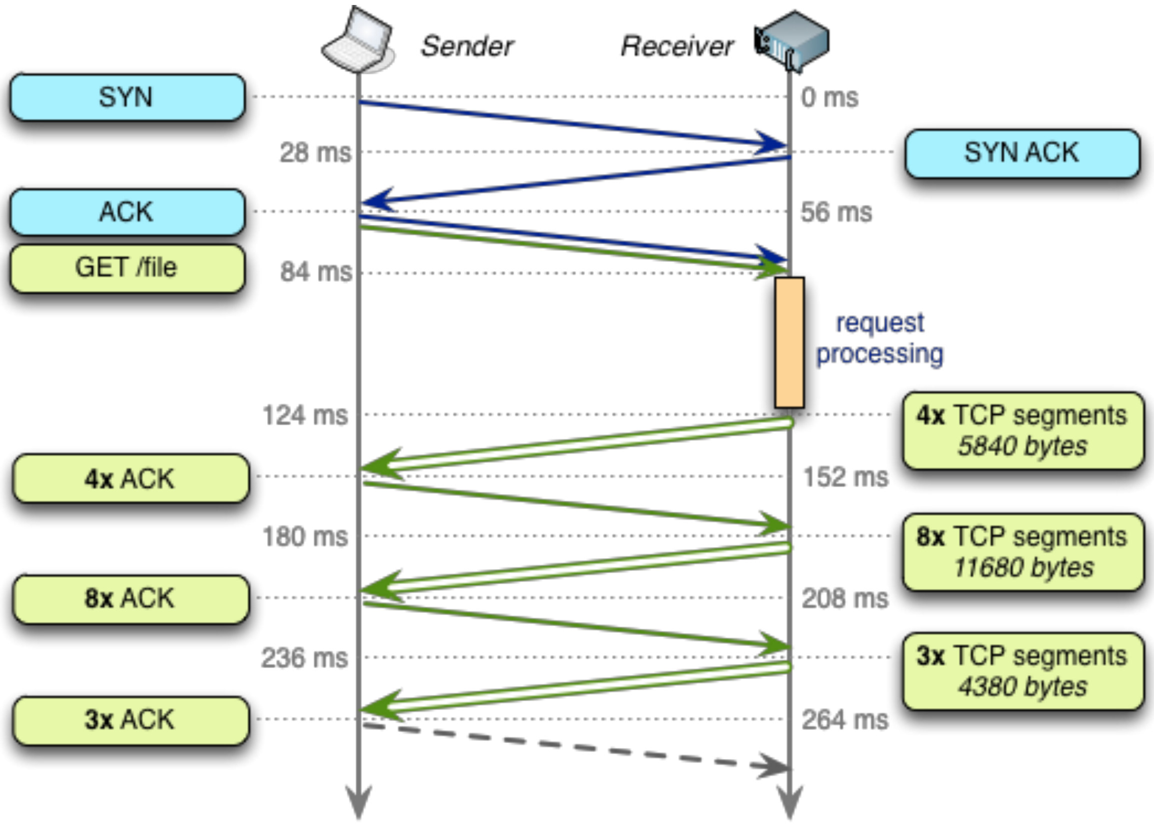
The *(short)* life of a web request



- *(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**

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



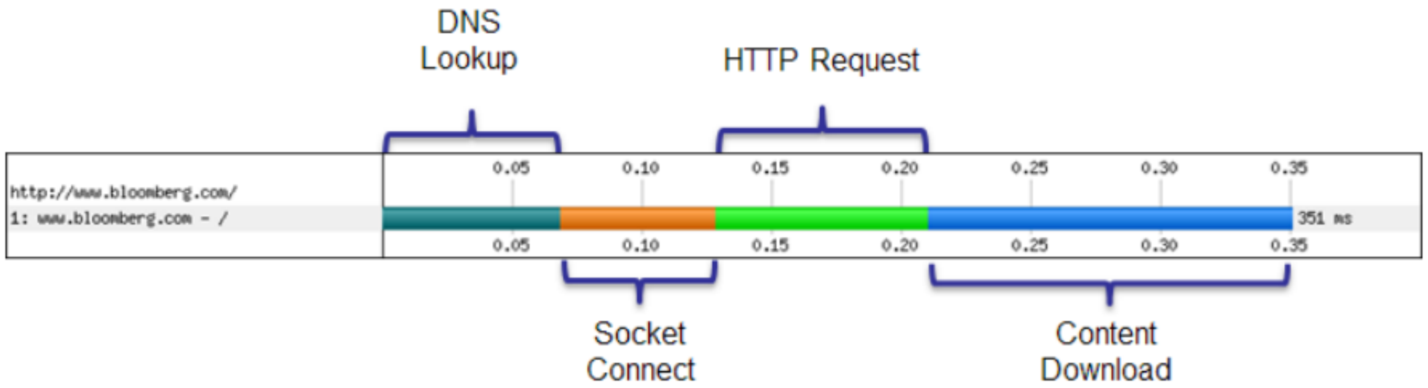
- **5 Mbps** connection
- **56 ms** roundtrip time (NYC > London)
- **40 ms** server processing time

4 roundtrips, or 264 ms!

Plus DNS and TLS roundtrips

[Congestion Avoidance and Control](#)

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



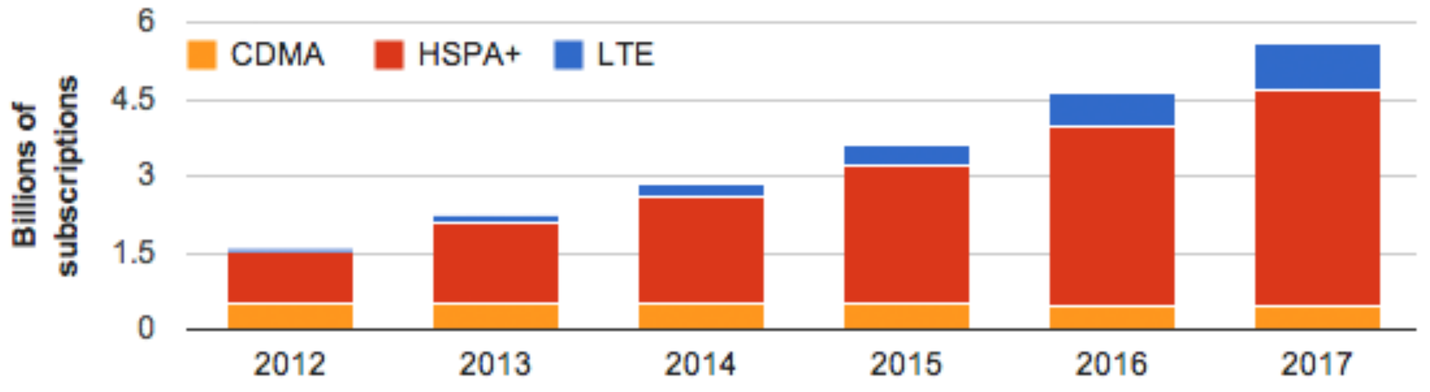
	3G *(200 ms RTT)*	**4G** *(100 ms RTT)*
Control plane	*(200-2500 ms)*	*(50-100 ms)*
DNS lookup	200 ms	100 ms
TCP Connection	200 ms	100 ms
TLS handshake (optional)	(200-400 ms)	(100-200 ms)
HTTP request	200 ms	100 ms
Total time	**800 - 4100 ms**	**400 - 900 ms**

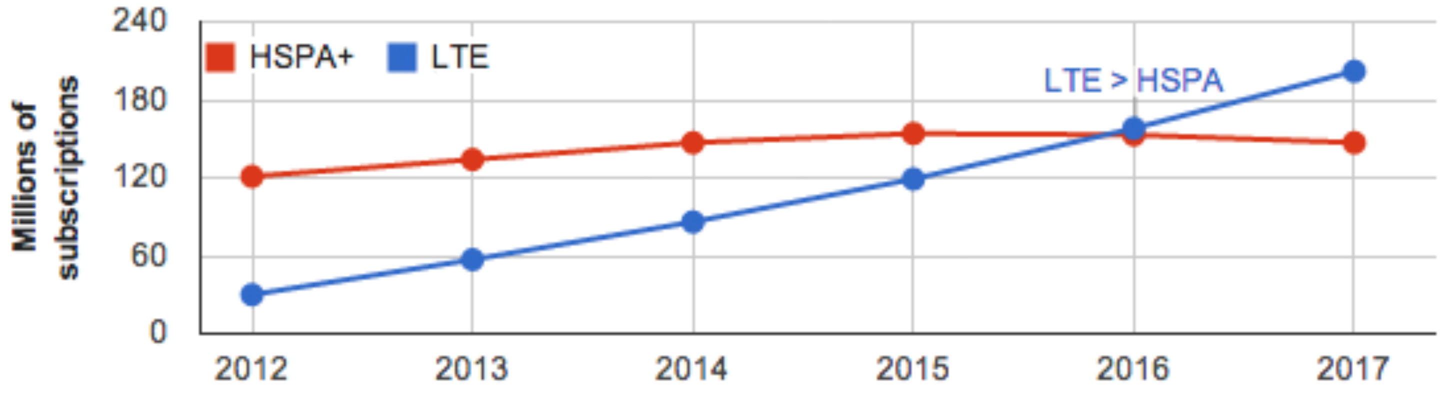
x4 (slow start)

One 20 KB HTTP request!

Anticipate network latency overhead

Not so good ***news everybody! ....***

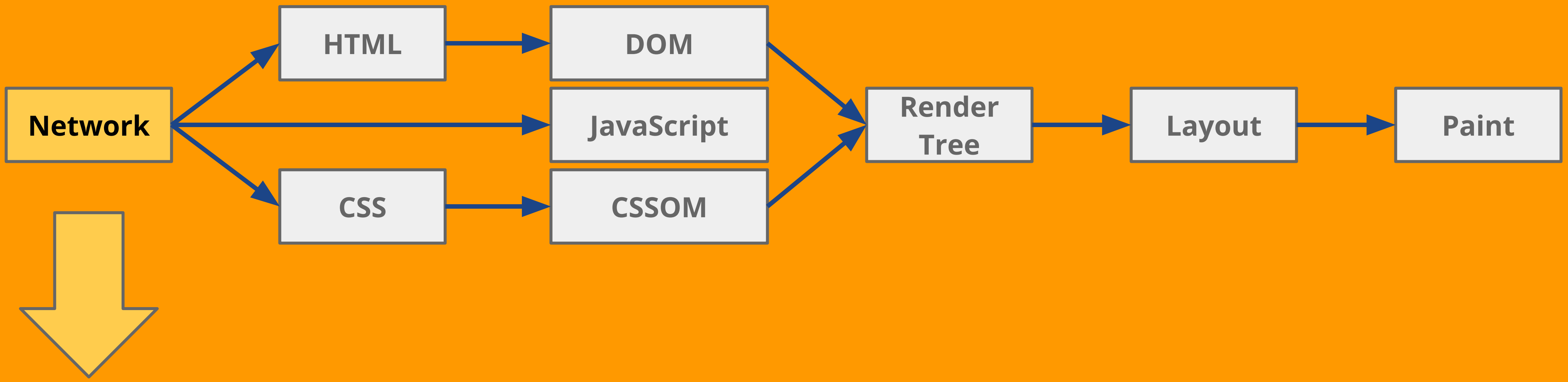




HSPA+ will be the dominant network type of the next decade!

- *Latest* HSPA+ releases are comparable to LTE in performance
- 3G networks will be with us for ***at least another decade***
- LTE adoption in US and Canada is ***way ahead*** of the world-wide trends

[4G Americas - Statistics]



Latency is the bottleneck for web performance

- Lots of small transfers
- New TCP connections are expensive
- High latency overhead on mobile networks

*... in short: **no, the network won't save us.***

Network optimization tips?

Glad you asked... :-)

TCP, TLS, mobile / wireless and HTTP best practices...

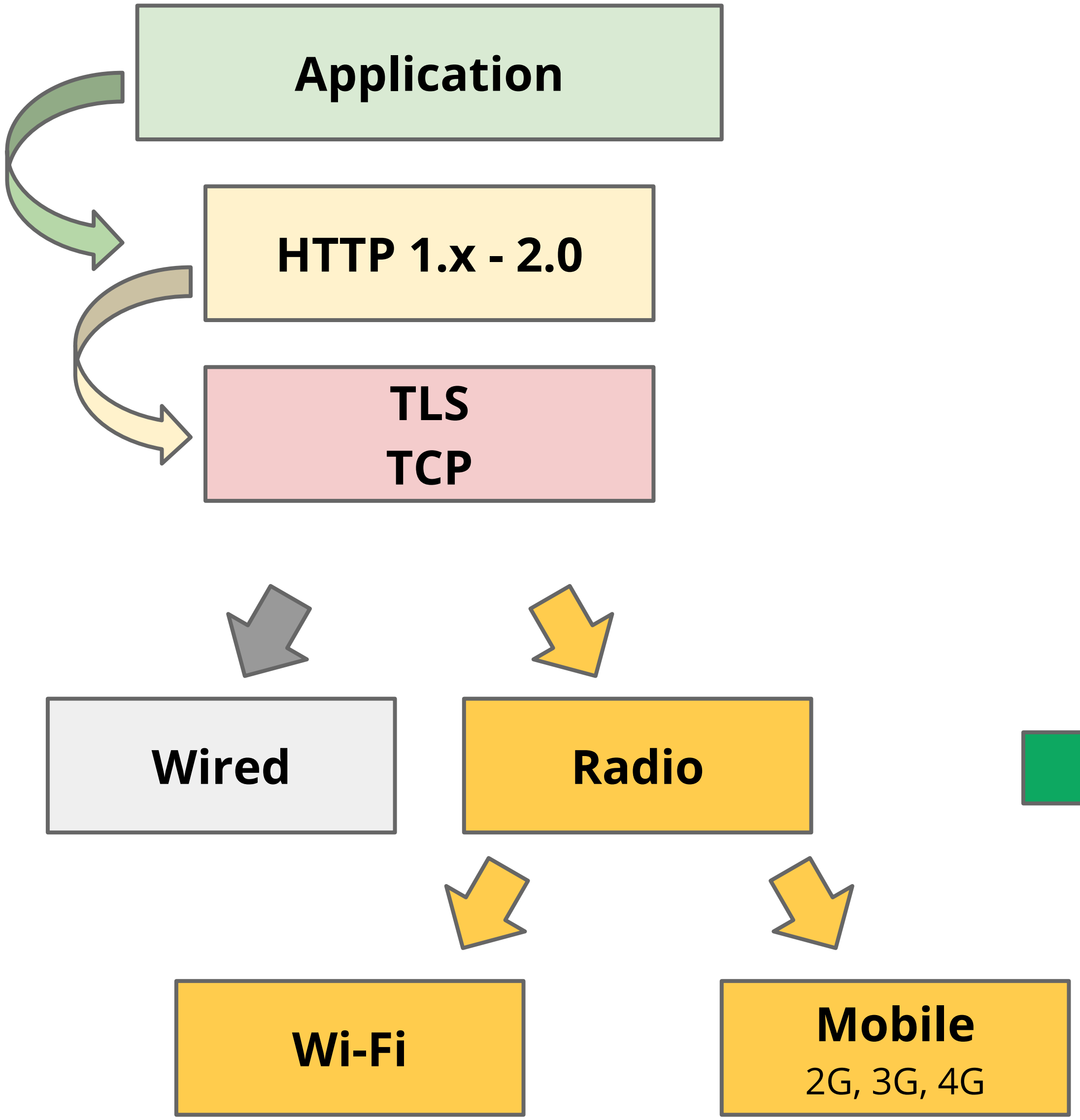
</shameless self promotion>

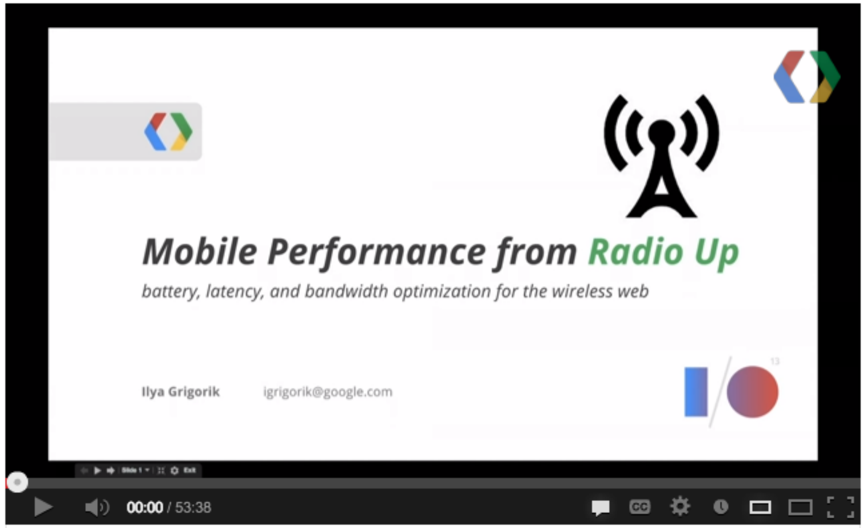
- *Optimize your **TCP** server stacks*
- *Optimize your **TLS** deployment*
- *Optimizing for **wireless** networks*
- *Optimizing for **HTTP 1.x quirks***
- *Migrating to **HTTP 2.0***
- *XHR, SSE, WebSocket, WebRTC, ...*

~~$29.99~~ **Read Online for Free**

Brought to you by Fluent Conference

http://bit.ly/fluent-hpbn

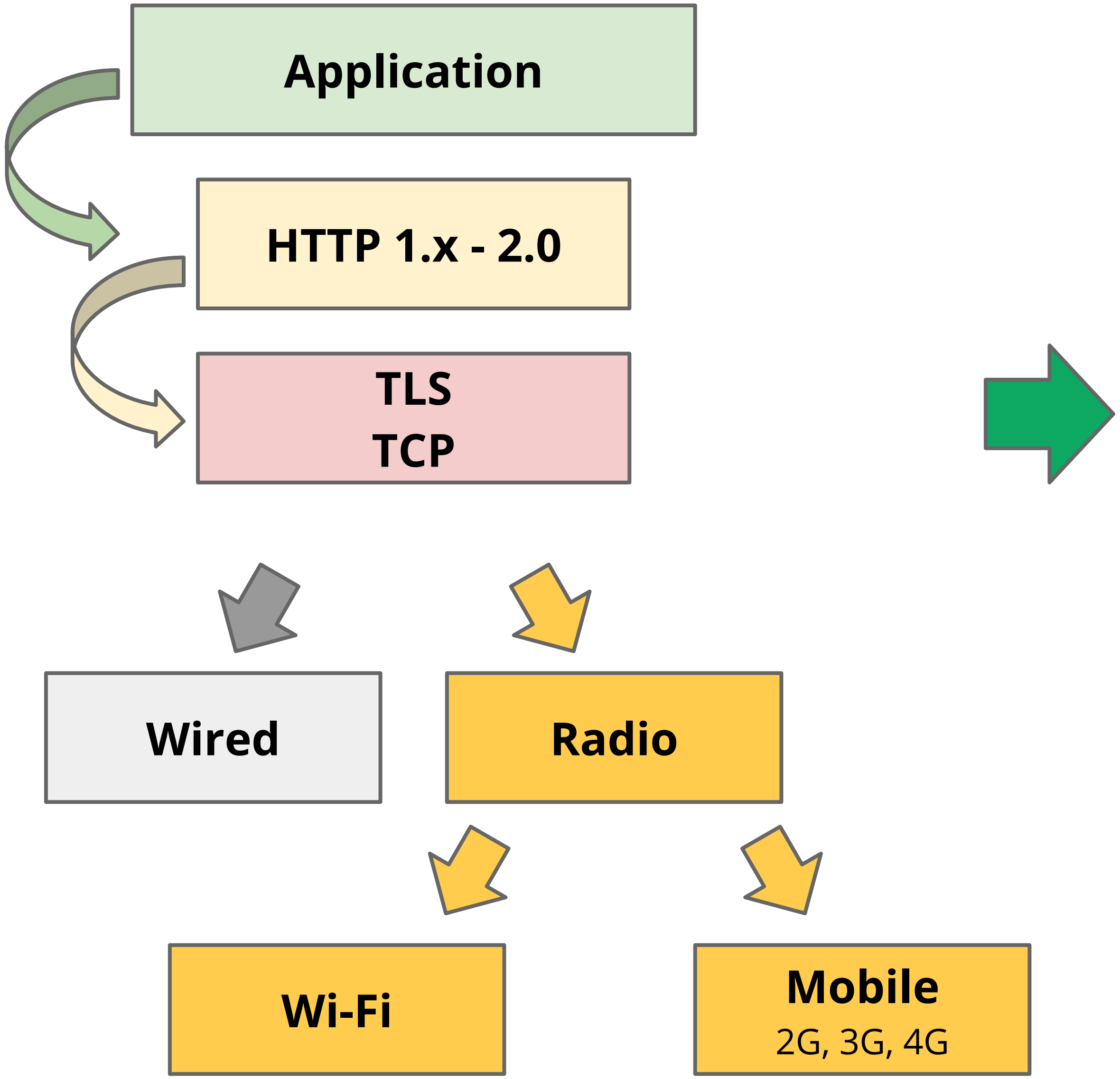




http://bit.ly/io-radioup

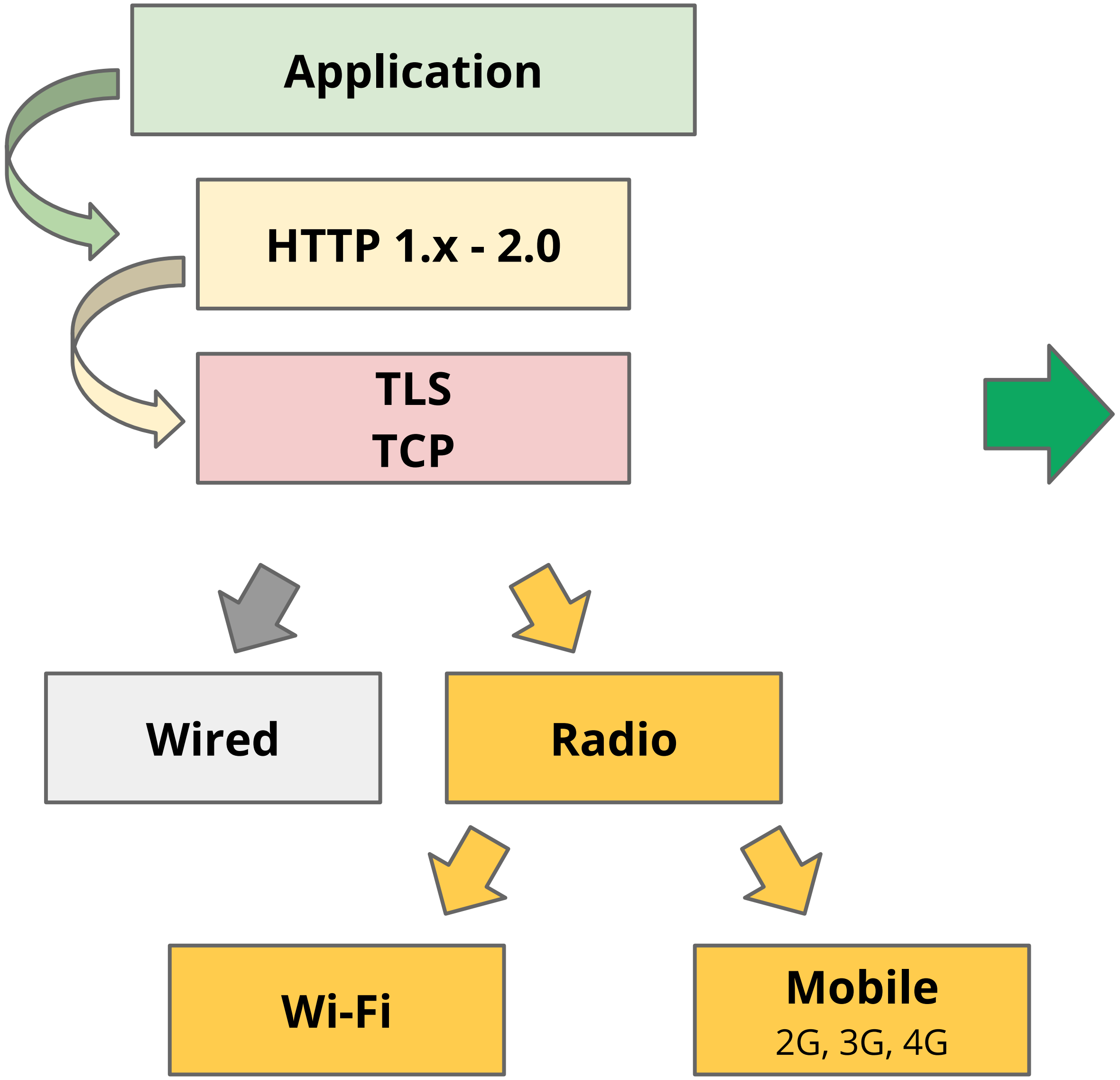
- How Wi-Fi + 3G/4G works
- RRC + battery life optimization
- Data bursting, prefetching
- Inefficiency of periodic transfers
- Intermittent connectivity
- ....

http://bit.ly/fluent-hpbn

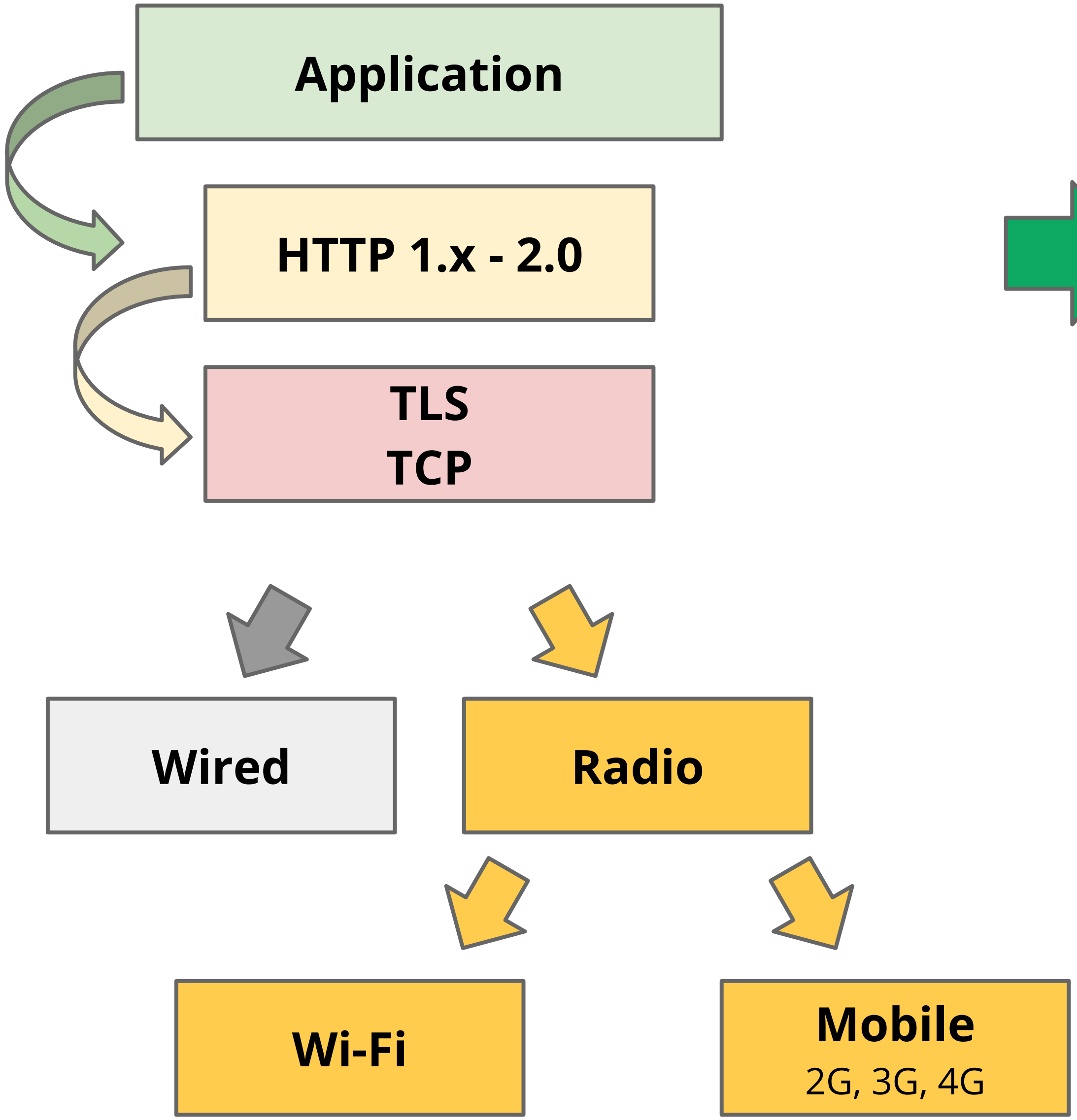


- Upgrade kernel: Linux 3.2+
- IW10 + disable slow start after idle
- TCP window scaling
- Position servers closer to the user
- Reuse established TCP connections
- Compress transferred data
- ....

http://bit.ly/fluent-hpbn

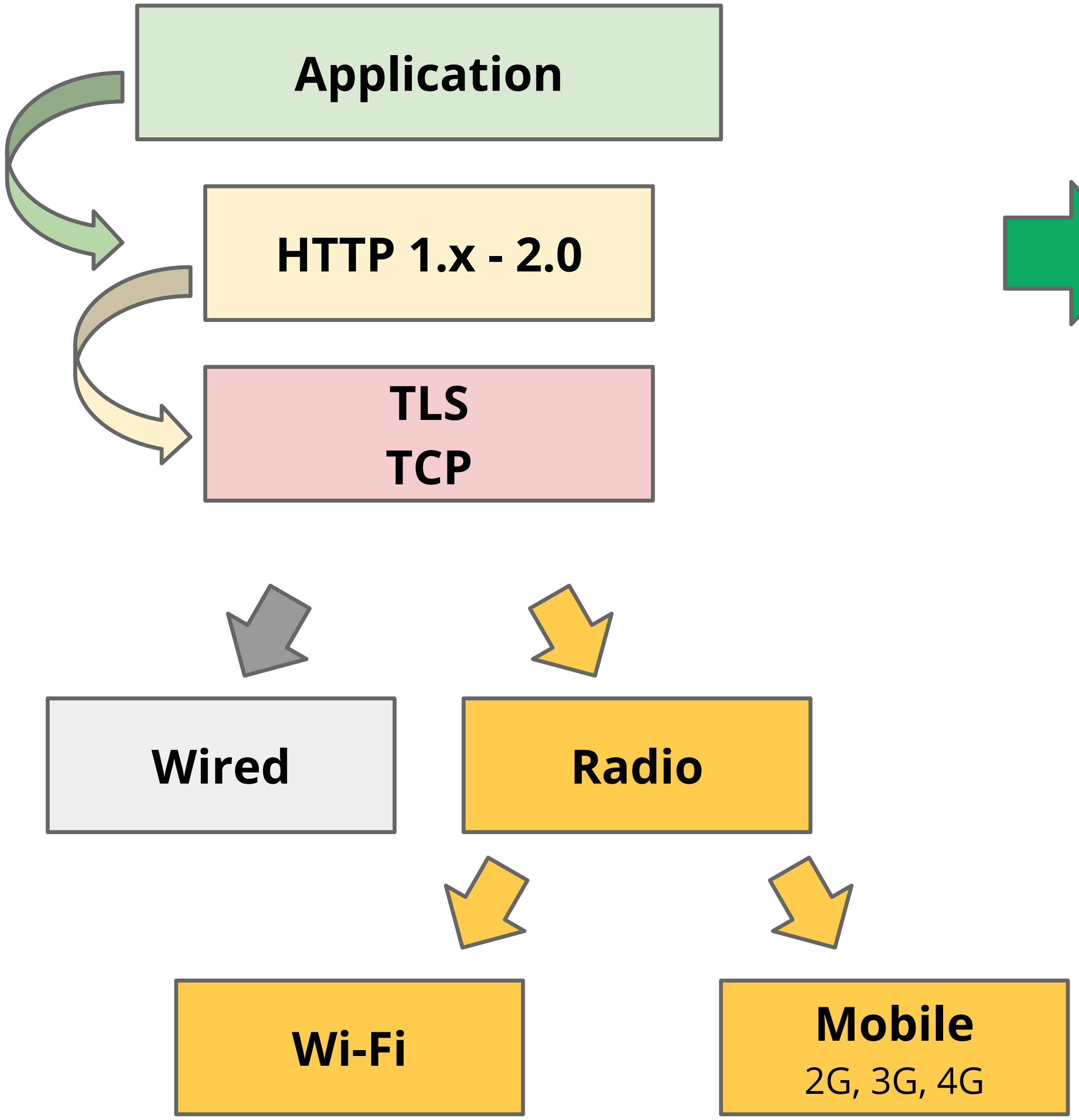


- Upgrade TLS libraries
- Use session caching / session tickets
- Early TLS termination (CDN)
- Optimize TLS record size
- Optimize certificate size
- Disable TLS compression
- Configure SNI support
- Use HTTP Strict Transport Security
- ....



HTTP 1.x hacks and best practices:

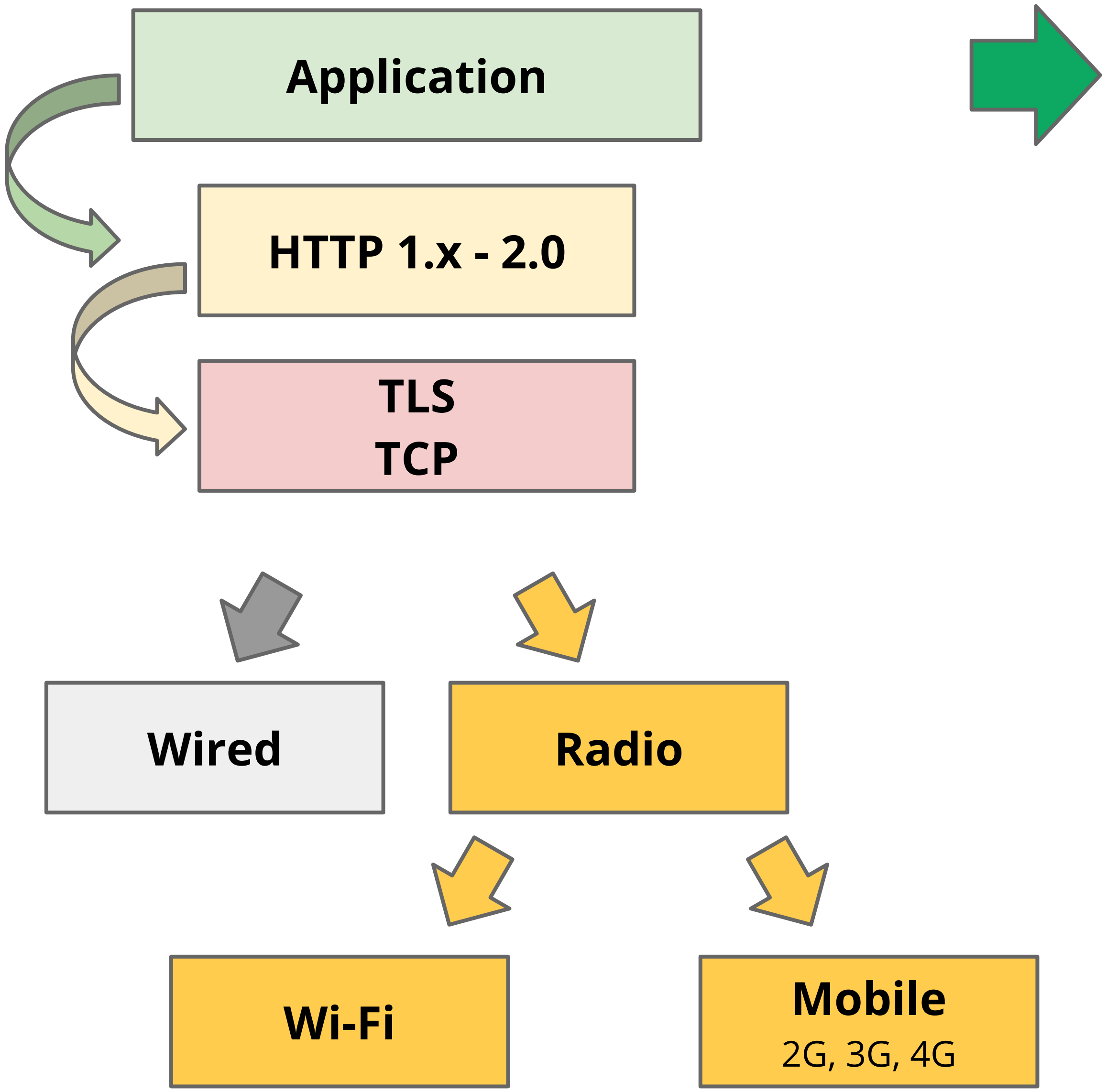
- Concatenate files (CSS, JS)
- Sprite small images
- Shard assets across origins
- Minimize protocol overhead
- Inline assets
- Compress (gzip) assets
- Cache assets!
- ....



HTTP 2.0 to the rescue!

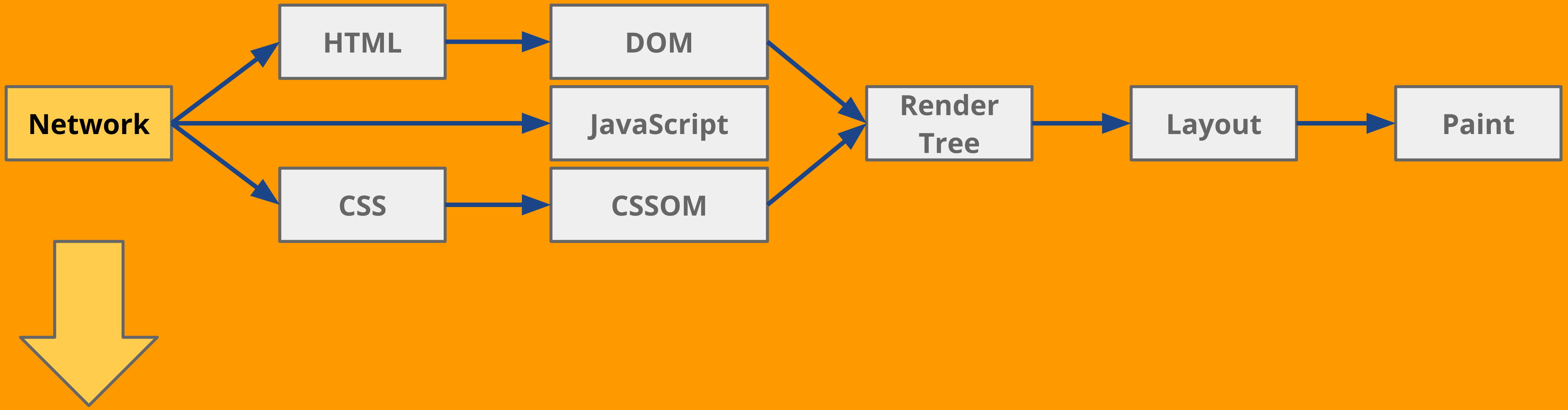
- Undo HTTP 1.x hacks... :-)
- Unshard your assets
- Leverage server push
- ....

(more on this in a second)



- XMLHttpRequest do's and don'ts
- Server-Sent Events
- WebSocket
- WebRTC
  - DataChannel - UDP in the browser!

http://bit.ly/fluent-hpbn



Foundation of your performance strategy.

Get it right!

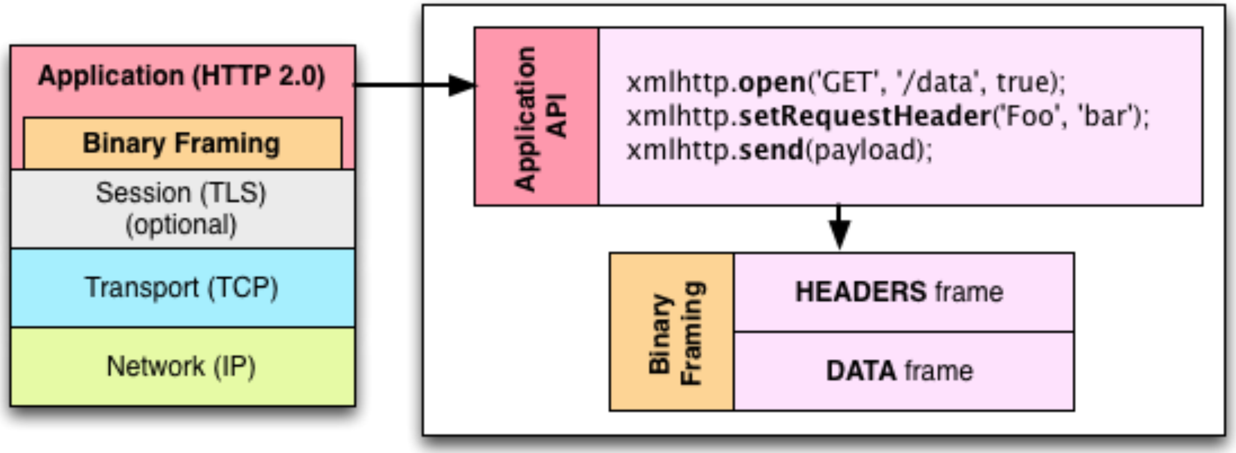
Let's (briefly) talk about **HTTP 2.0**

Will it fix all things? No, but many...

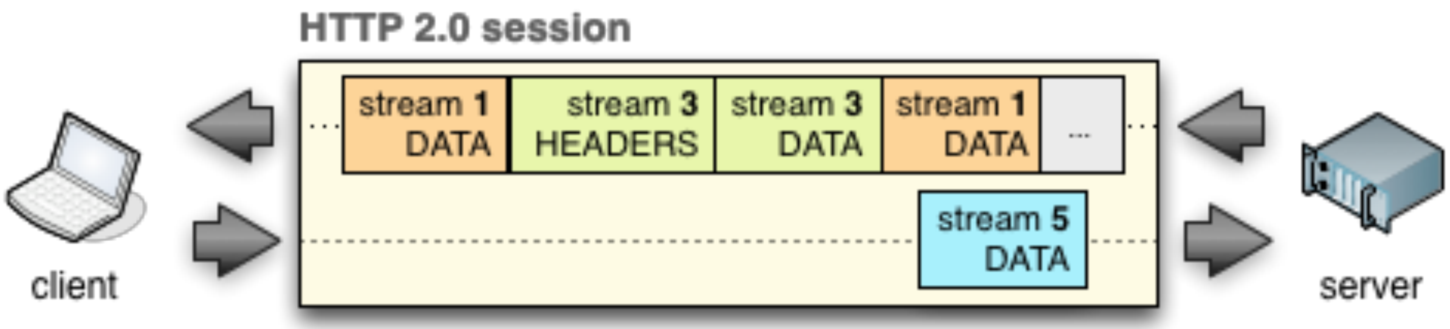
*... **we're not replacing all of HTTP** — the methods, status codes, and most of the headers you use today will be the same. Instead, **we're re-defining how it gets used "on the wire" so it's more efficient**, and so that it is more gentle to the Internet itself ....*

- Mark Nottingham

HTTP 2.0 in a nutshell...



- New binary framing
- One connection (session)
- Many parallel requests (streams)
- Header compression
- Stream prioritization
- Server push



[High performance browser networking: HTTP 2.0]

What's HTTP server push?

Premise: server can push multiple resources in response to one request

- What if the client doesn't want the resource?
  - *Client can cancel stream if it doesn't want the resource*
- Resource goes into browsers cache
  - *HTTP 2.0 server push does not have an application API (JavaScript)*

Newsflash: we are already using "server push"

- Today, we call it "inlining" (to be exact it's "forced push")
- Inlining works for unique resources, bloats pages otherwise

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
- ...

HTTP 2.0 / SPDY FAQ

- **Q: Do I need to modify my site to work with SPDY / HTTP 2.0?**
- **A:** No. But you can optimize for it.

- **Q: How do I optimize the code for my site or app?**
- **A:** "Unshard", stop worrying about silly things (like spriting, etc).

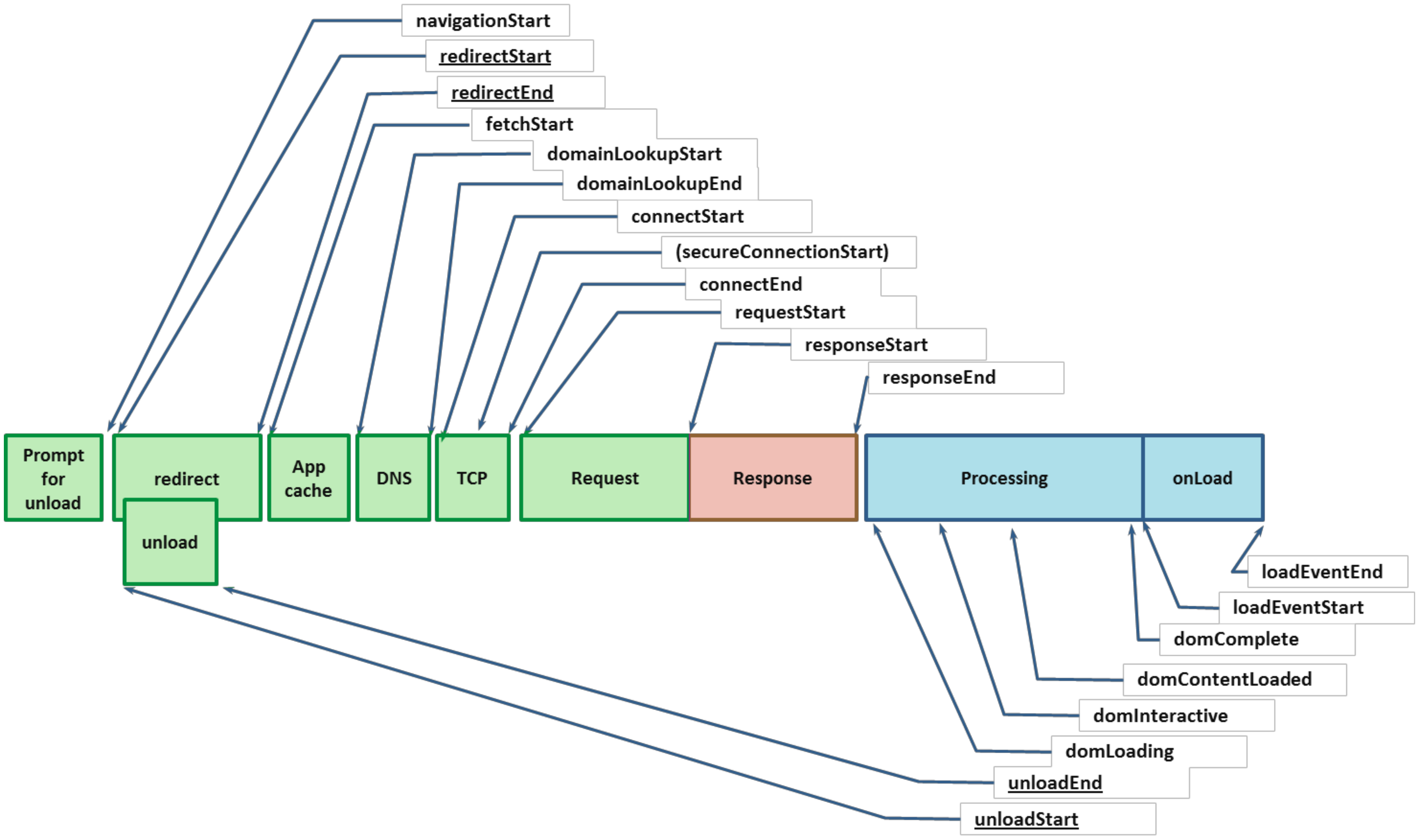
- **Q: Any server optimizations?**
- **A:** Yes!
  - CWND = 10
  - Check your SSL certificate chain (length)
  - TLS resume, terminate SSL connections closer to the user
  - Disable TCP slow start on idle

- **Q: Sounds complicated...**
- **A:** mod_spdy, nginx, GAE!

Measuring network performance

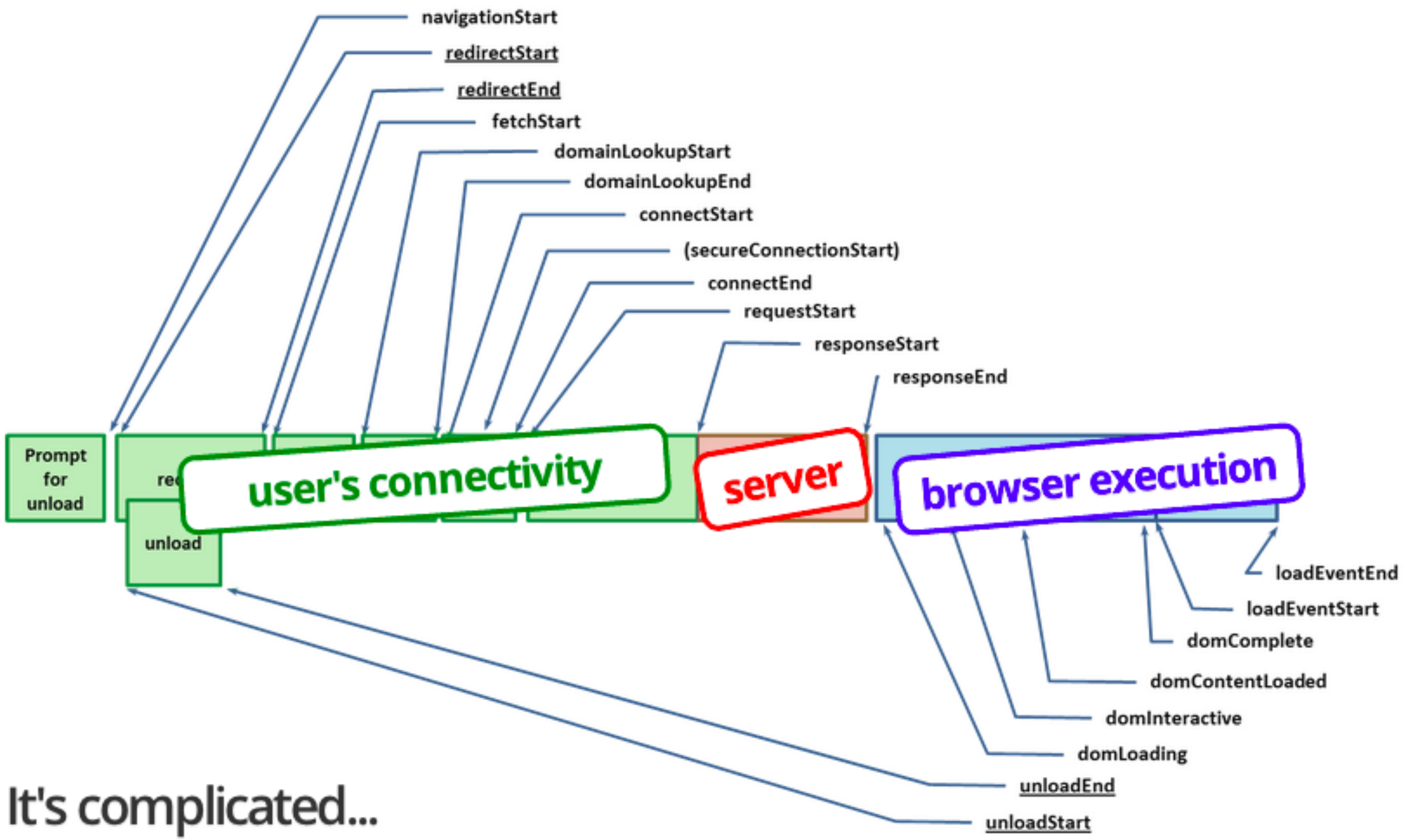
Real users, on real networks, with real devices...

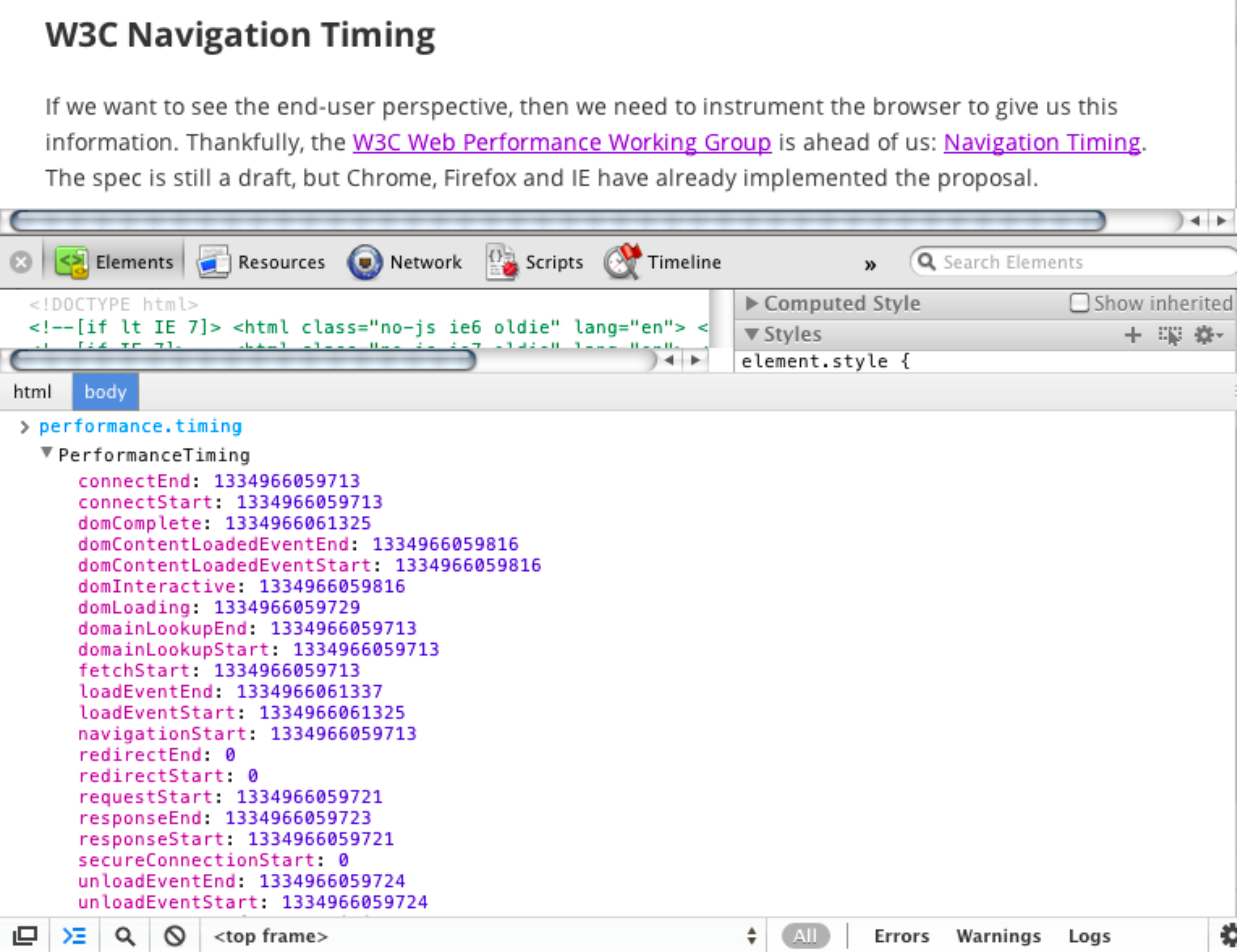
Navigation Timing (W3C)



[Navigation Timing spec](#)

Navigation Timing (W3C)





Available in...

- IE 9+
- Firefox 7+
- Chrome 6+
- Android 4.0+

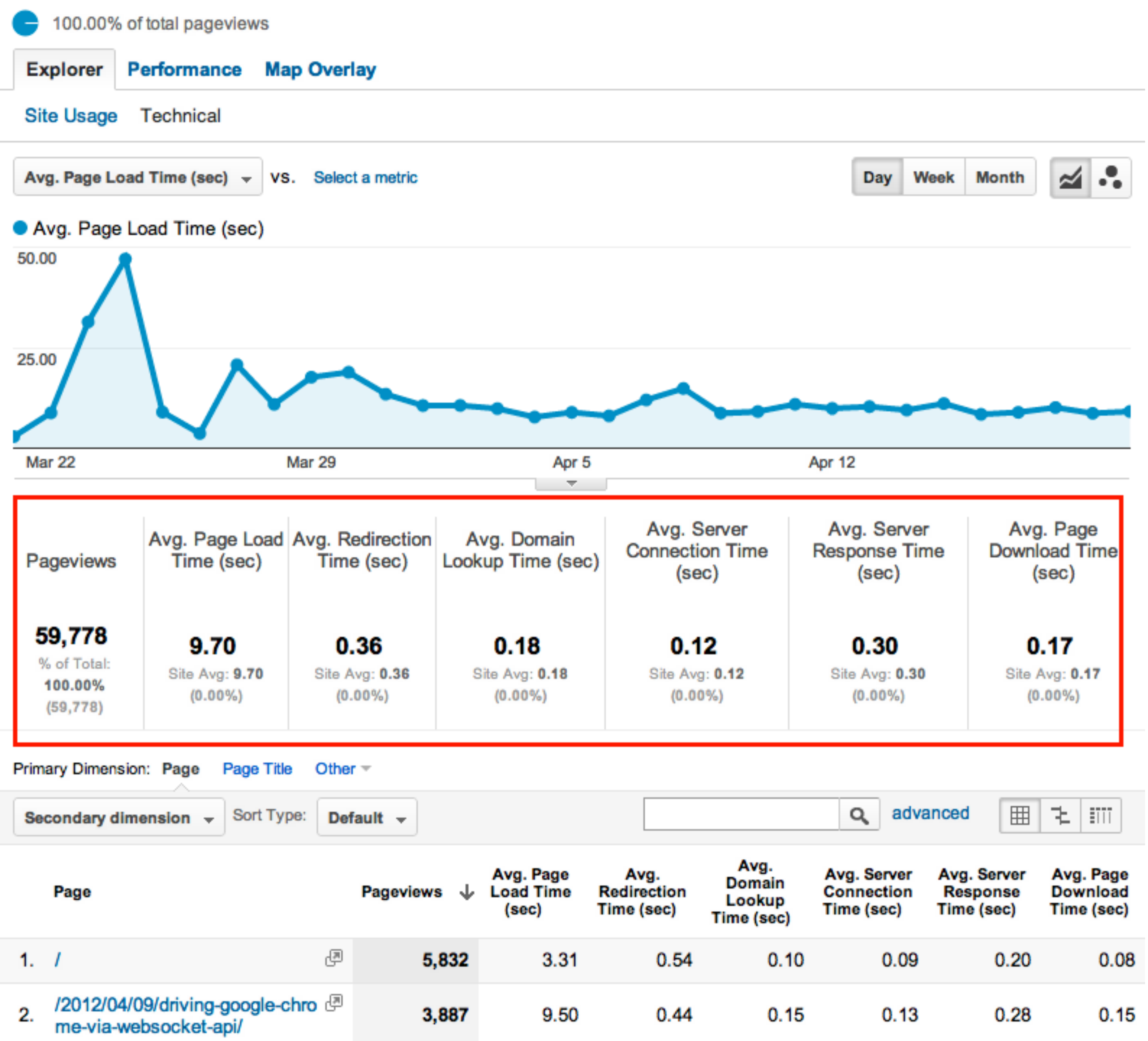
Real User Measurement (RUM) with Google Analytics

```
<script>
  _gaq.push(['_setAccount','UA-XXXX-X']);
  _gaq.push(['_setSiteSpeedSampleRate', 100]); // #protip
  _gaq.push(['_trackPageview']);
</script>
```

Google Analytics > Content > Site Speed

- Automagically collects this data for you - defaults to 1% sampling rate
- Maximum sample is 10k visits/day
- You can set custom sampling rate

You have all the power of Google Analytics! Segments, conversion metrics, ...



Performance data from **real users**, on **real networks**

Singapore - Desktop 0.59% of total pageviews

San Francisco - Desktop 3.21% of total pageviews

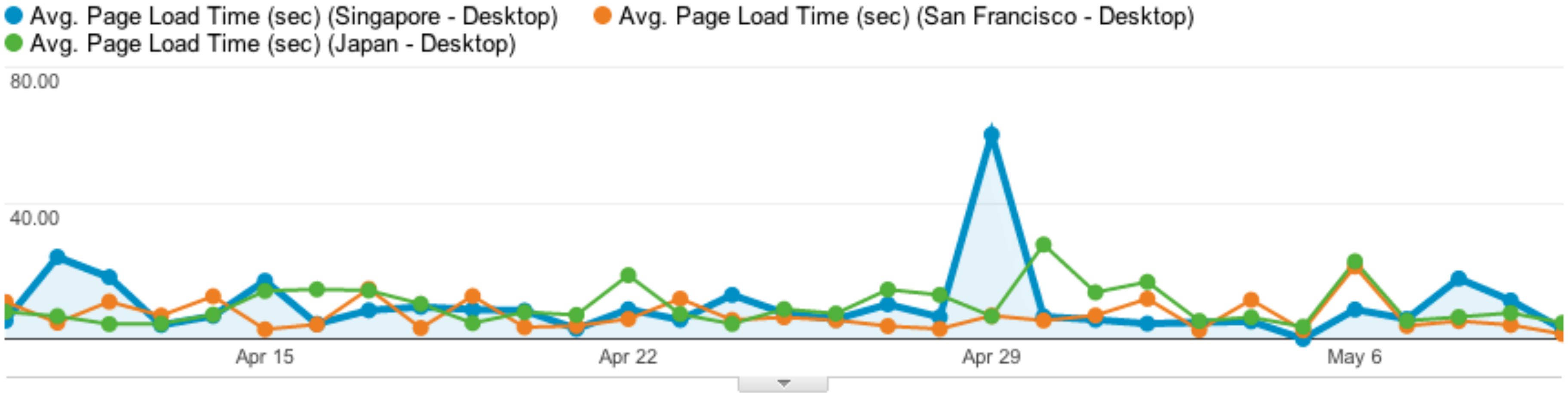
Japan - Desktop 1.66% of total pageviews

Explorer Performance Map Overlay

Site Usage Technical

Avg. Page Load Time (sec) vs. Select a metric

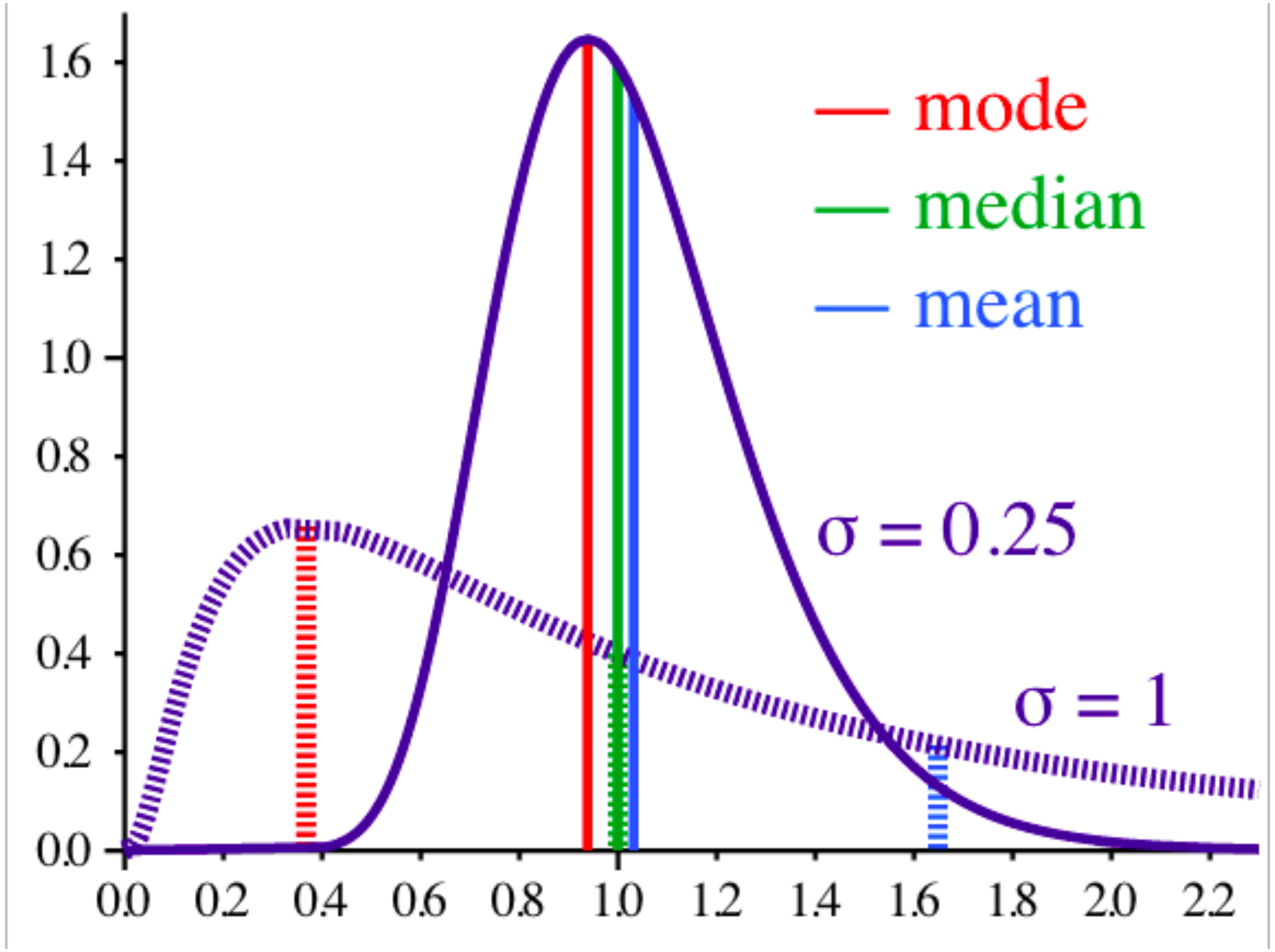
Day Week Month



	Pageviews	Avg. Page Load Time (sec)	Avg. Redirection Time (sec)	Avg. Domain Lookup Time (sec)	Avg. Server Connection Time (sec)	Avg. Server Response Time (sec)	Avg. Page Download Time (sec)
Singapore - Desktop	**347** % of Total: 0.59% (58,355)	**10.77** Site Avg: 9.63 (11.86%)	**1.89** Site Avg: 0.24 (694.56%)	**0.29** Site Avg: 0.18 (65.83%)	**0.08** Site Avg: 0.12 (-33.25%)	**0.41** Site Avg: 0.31 (33.72%)	**0.10** Site Avg: 0.20 (-49.40%)
San Francisco - Desktop	**1,873** % of Total: 3.21% (58,355)	**6.83** Site Avg: 9.63 (-29.09%)	**0.27** Site Avg: 0.24 (12.65%)	**0.10** Site Avg: 0.18 (-43.08%)	**0.05** Site Avg: 0.12 (-60.74%)	**0.20** Site Avg: 0.31 (-34.36%)	**0.14** Site Avg: 0.20 (-32.17%)

Full power of GA to **segment, filter, compare, ...**

Averages are **misleading**...



*Head into the **Technical reports** to see the histograms and distributions!*

Case study: igvita.com page load times

Dec 1, 2011 - Dec 31, 2011

Page Load Time Bucket (sec)	Page Load Sample	Percentage of total
0 - 1	22	5.35%
1 - 3	116	28.22%
3 - 7	148	36.01%
7 - 13	66	16.06%
13 - 21	22	5.35%
21 - 35	14	3.41%
35 - 60	10	2.43%
60+	13	3.16%

Jan 1, 2012 - Jan 31, 2012

Page Load Time Bucket (sec)	Page Load Sample	Percentage of total
0 - 1	83	13.61%
1 - 3	256	41.97%
3 - 7	158	25.90%
7 - 13	58	9.51%
13 - 21	14	2.30%
21 - 35	9	1.48%
35 - 60	6	0.98%
60+	26	4.26%

Content > Site Speed > Page Timings > Performance

Migrated site to new host, server stack, web layout, and using static generation. Result: noticeable shift in the user page load time distribution.

Case study: igvita.com server response times

Dec 1, 2011 - Dec 31, 2011

Server Response Time Bucket (sec)	Response Sample	Percentage of total
0 - 0.01	18	4.40%
0.01 - 0.10	33	8.07%
0.10 - 0.50	168	41.08%
0.50 - 1	22	5.38%
1 - 2	124	30.32%
2 - 5	38	9.29%
5+	6	1.47%

Jan 1, 2012 - Jan 31, 2012

Server Response Time Bucket (sec)	Response Sample	Percentage of total
0 - 0.01	188	31.92%
0.01 - 0.10	120	20.37%
0.10 - 0.50	249	42.28%
0.50 - 1	23	3.90%
1 - 2	3	0.51%
2 - 5	5	0.85%
5+	1	0.17%

Content > Site Speed > Page Timings > Performance

Bimodal response time distribution?
Theory: user cache *vs.* database cache *vs.* full recompute

1. *Measure user perceived network latency with Navigation Timing*
2. *Analyze RUM data to identify performance bottlenecks*
3. *Use GA's advanced segments (or similar solution)*
4. *Setup {daily, weekly, ...} reports*

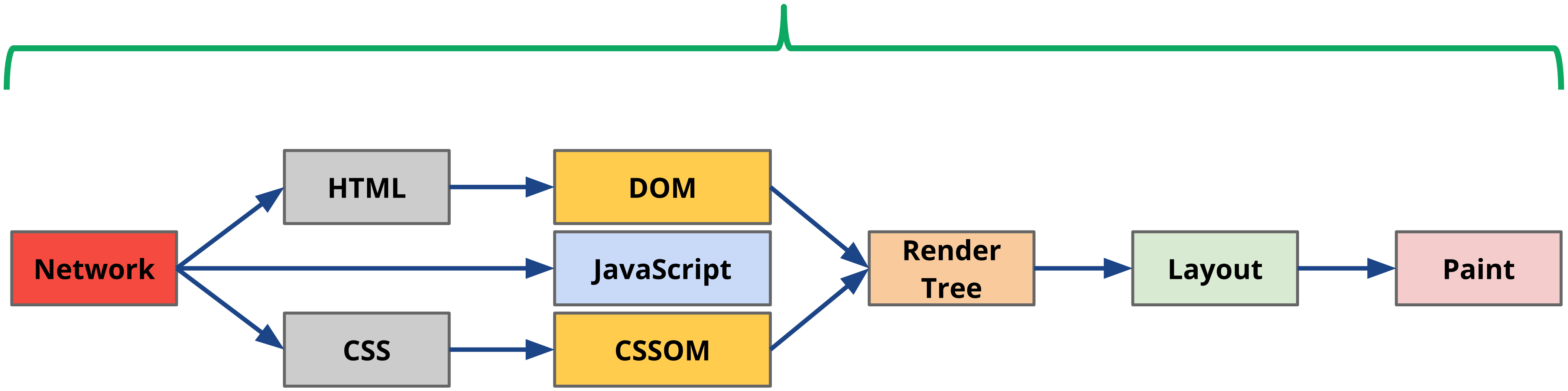
Measure, analyze, optimize, repeat...

10m break... Questions?

Twitter @igrigorik
G+ gplus.to/igrigorik
Web igvita.com

2

Critical rendering path: resource loading



What's the "critical" part?

To answer that, we need to peek inside the browser...

Let's try a simple example...

index.html

```
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

```
p    { font-weight: bold; }
span { display: none; }
```

- Simple (valid) HTML file
- External CSS stylesheet

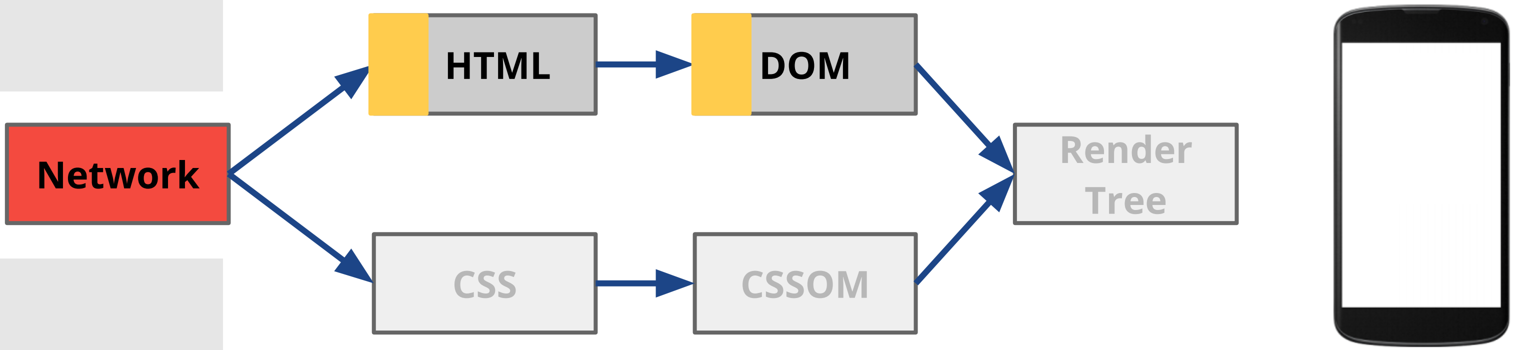
What could be simpler, right?

HTML bytes are arriving on the wire...

index.html

```
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

- first response packet with index.html bytes
- we have not discovered the CSS yet...

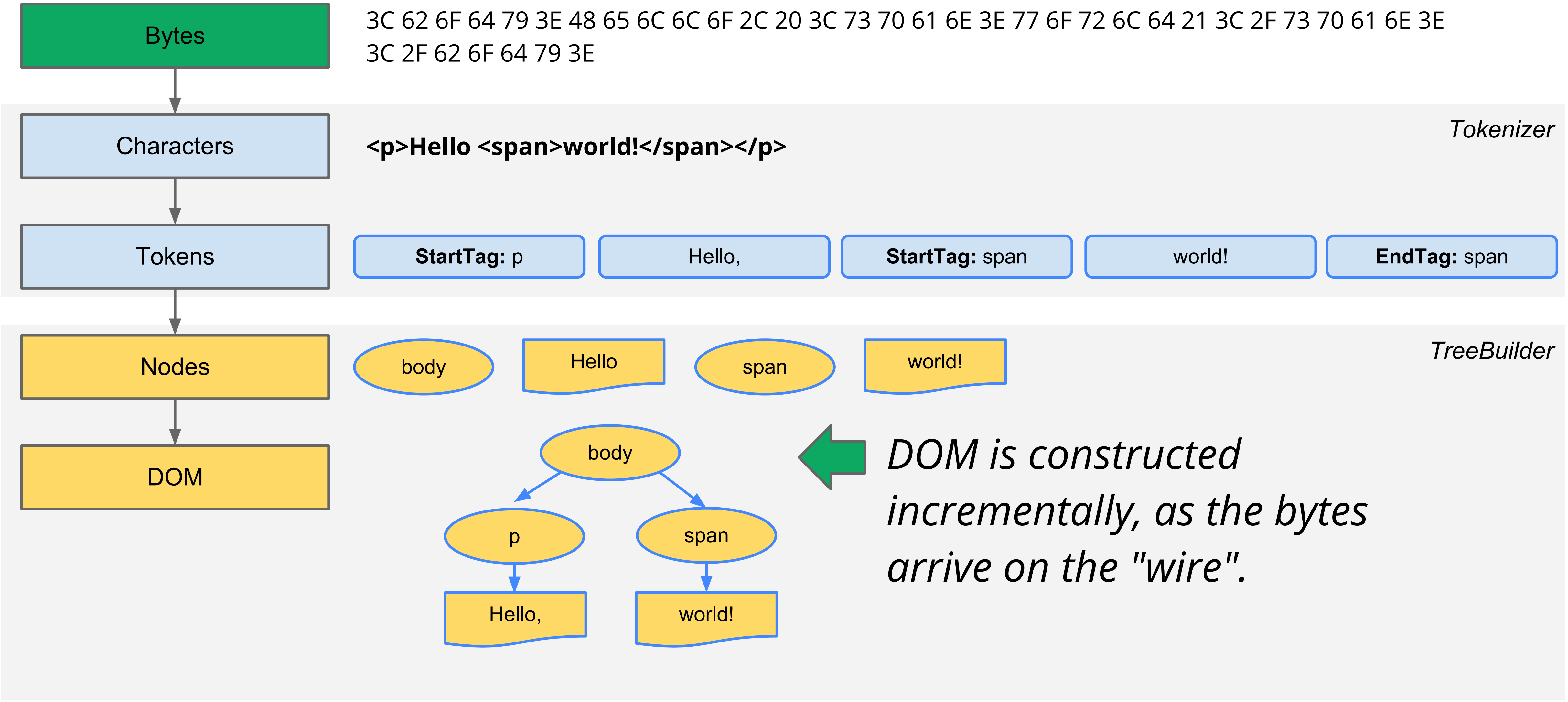


styles.css

```
p    { font-weight: bold; }
span { display: none; }
```

We're splitting packets for convenience...

The HTML5 **parser** at work...



DOM construction is complete... waiting on CSS!

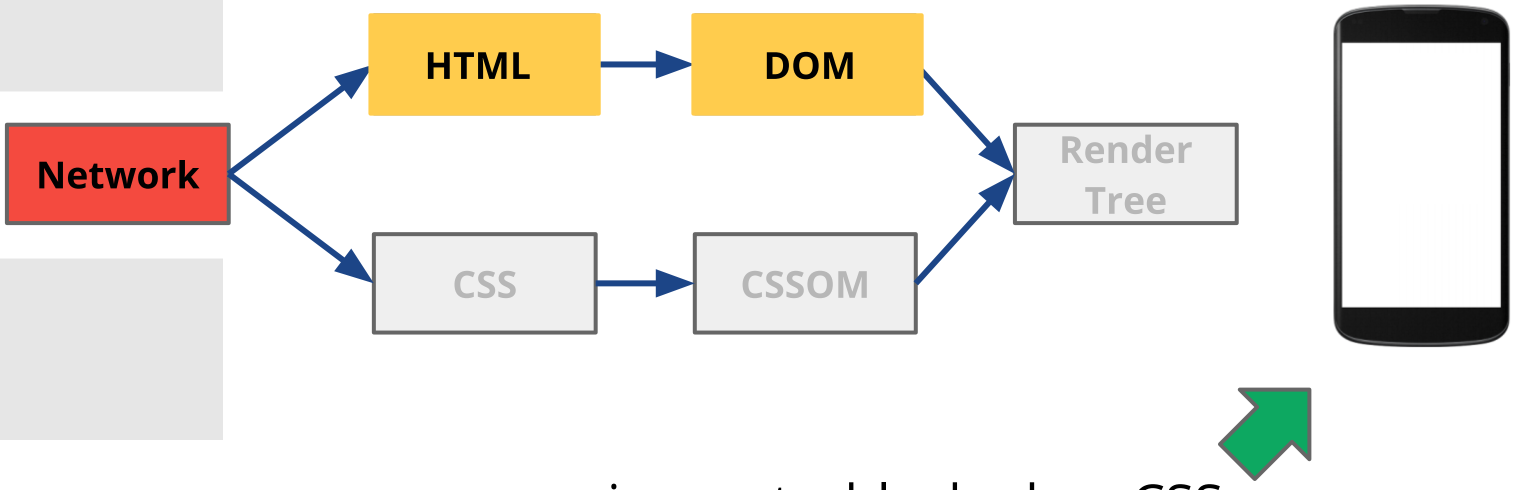
index.html

```
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

```
p    { font-weight: bold; }
span { display: none; }
```

- <link> discovered, network request sent
- DOM construction complete!



- screen is empty, blocked on CSS
    - otherwise, flash of unstyled content (FOUC)

First CSS bytes arrive... still waiting on CSS!

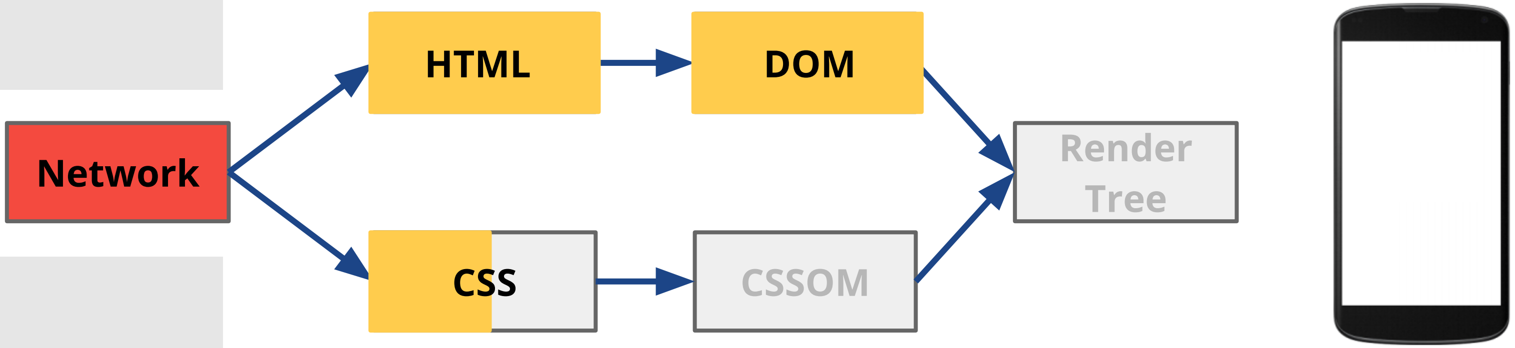
index.html

```html
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

```css
p    { font-weight: bold; }
span { display: none; }
```

- First CSS bytes arrive
- But, we must wait for the ***entire file...***



- Unlike HTML parsing, CSS is **not incremental**

Finally, we can construct the CSSOM!

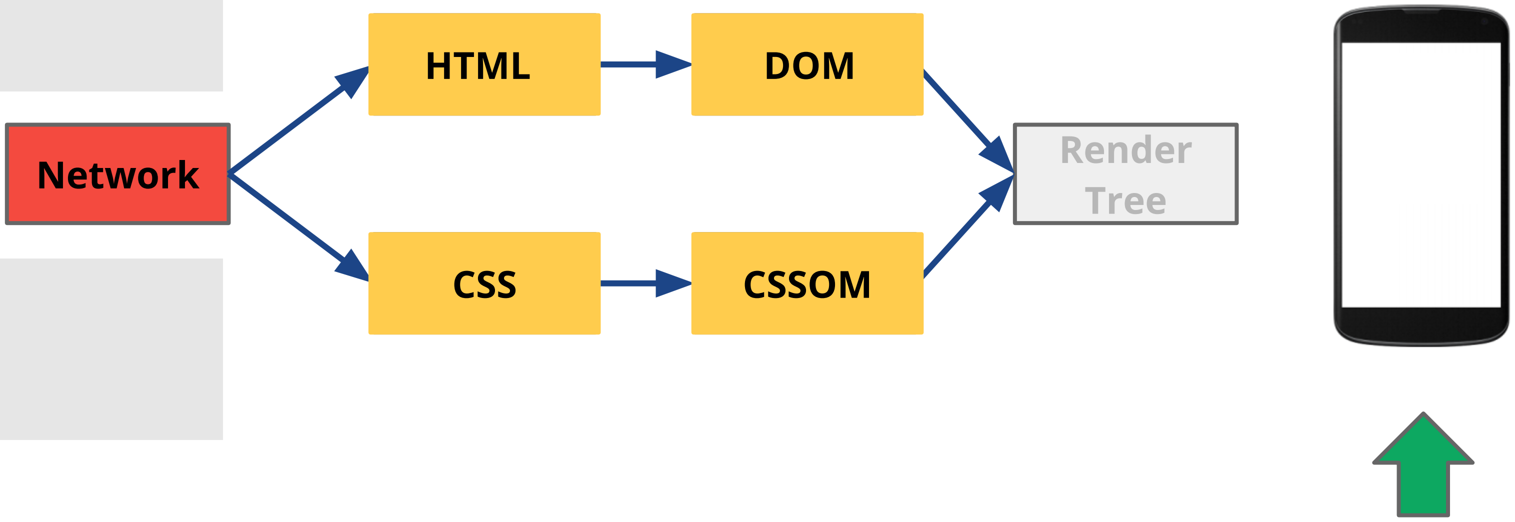
index.html

```
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

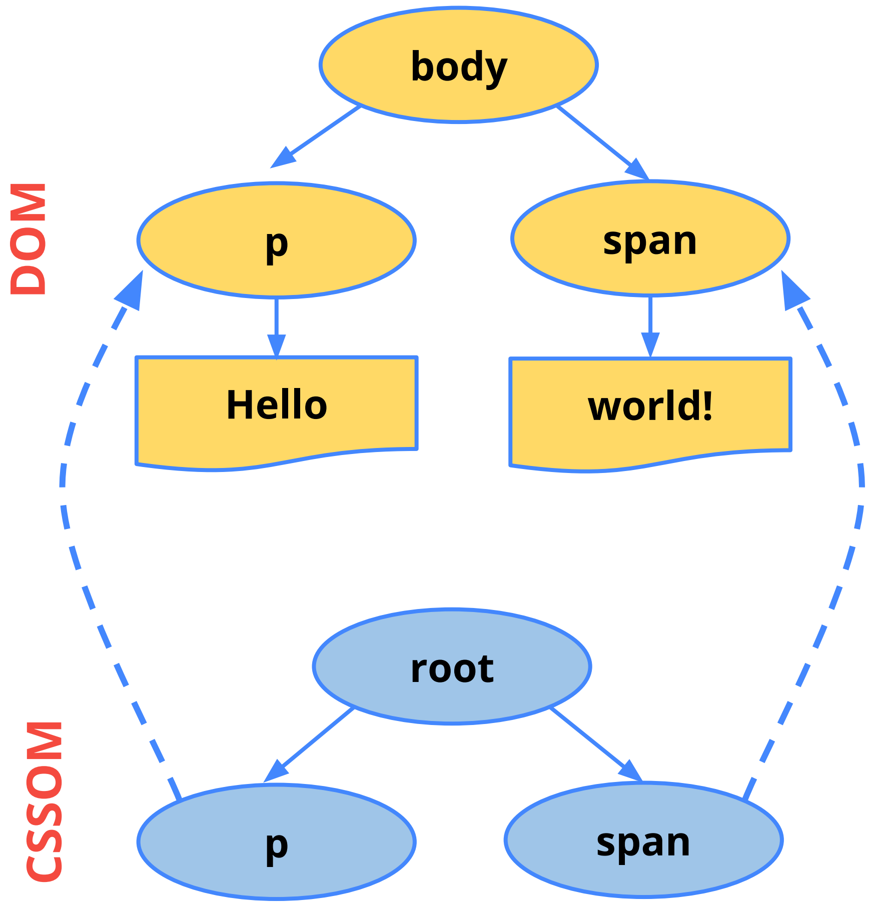
```
p    { font-weight: bold; }
span { display: none; }
```

- CSS download has finished - yay!
- We can now construct the CSSOM



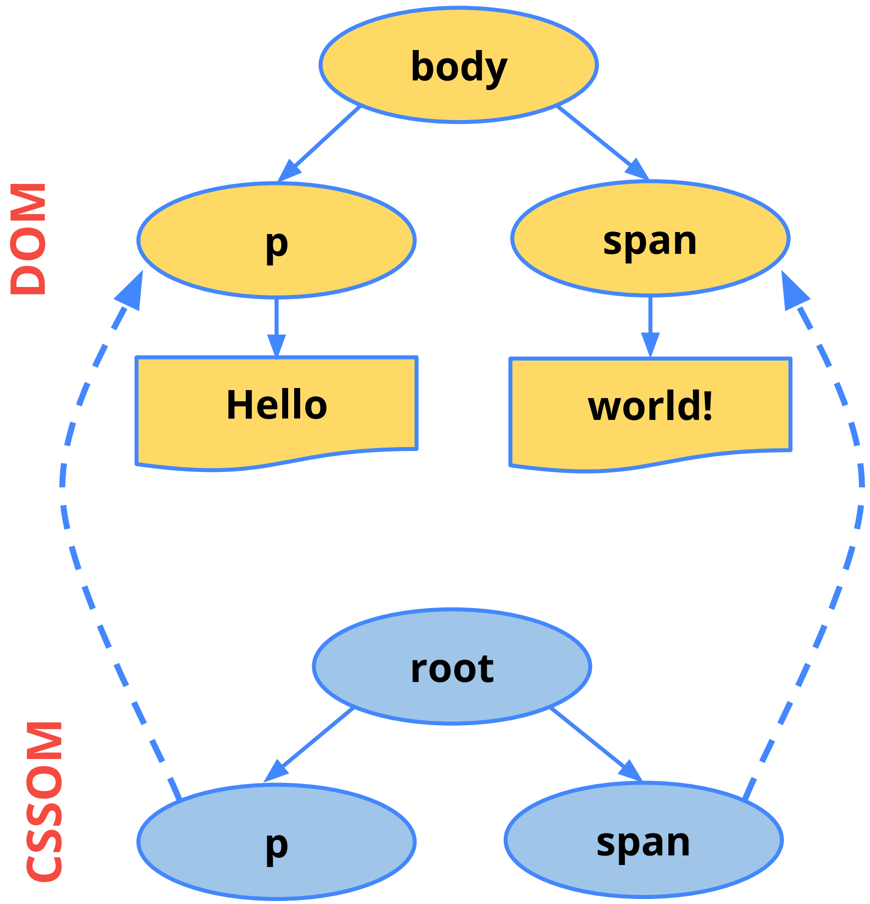
still blank :(

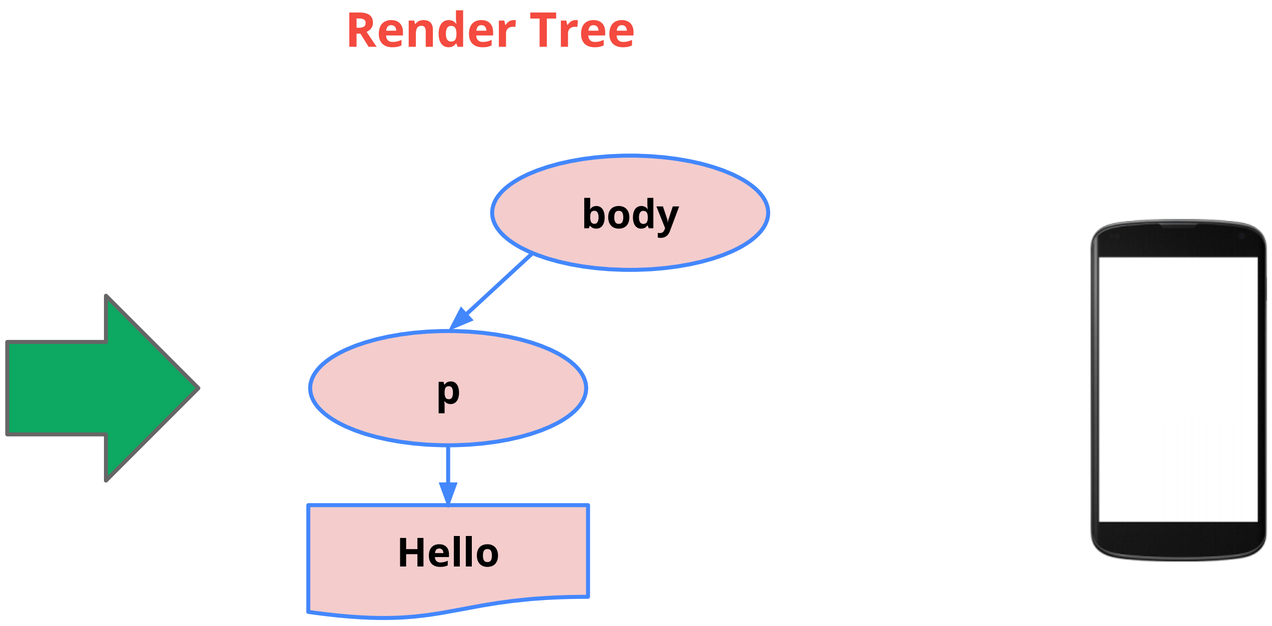
DOM + CSSOM = Render Tree(s)



- Match CSSOM to DOM nodes
- Yes, the screen is still empty....

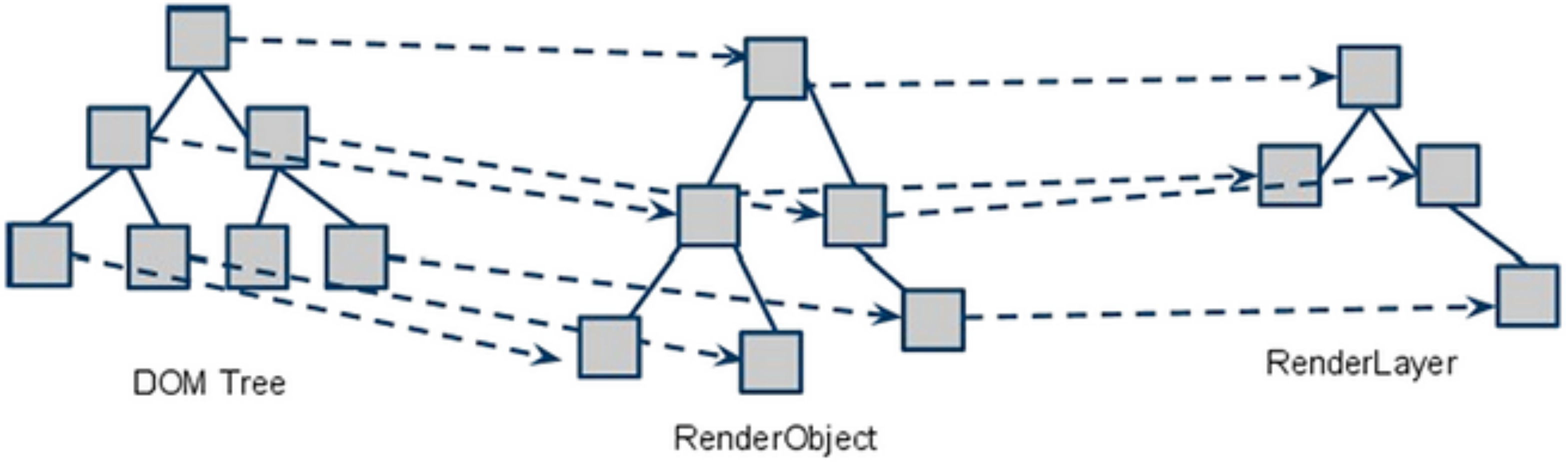
DOM + CSSOM = Render Tree(s)





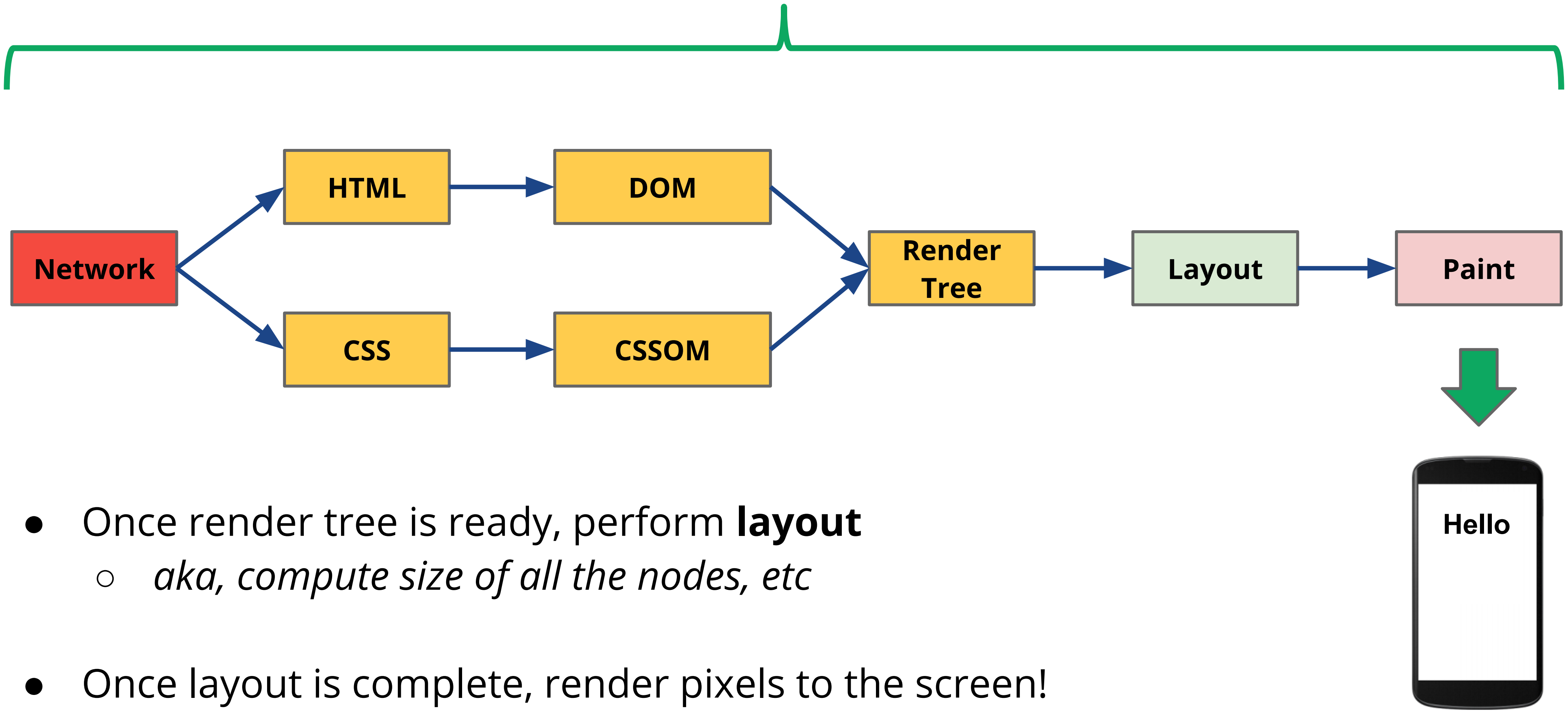
- **<span>** is not part of render tree!
  - "display: none"

DOM + CSSOM = Render*



RenderObject Tree	StyleObject Tree	RenderLayer Tree
owned by DOM tree	computed styles for all renderers	"helper" class for rendering
rendered content only	owned by RenderObject tree	used for <video>, <canvas>, ...
responsible for layout & paint	RenderObjects share RenderStyles	Some RenderLayers have GPU layers
answers DOM API measurement requests	RenderStyles share data members	...

Critical rendering path



- Once render tree is ready, perform **layout**
  - *aka, compute size of all the nodes, etc*

- Once layout is complete, render pixels to the screen!

Performance rules to keep in mind...

(1) HTML is parsed incrementally

(3) Rendering is **blocked on CSS**...

Which means...

(1) Stream the HTML response to the client

- *Don't wait to render the full HTML file - flush early, flush often.*

(2) Get CSS down to the client as fast as you can

- *Blank screen until we have the render tree ready!*

Err, wait. Did we forget something?

How about that JavaScript thing...

JavaScript... our friend and foe.

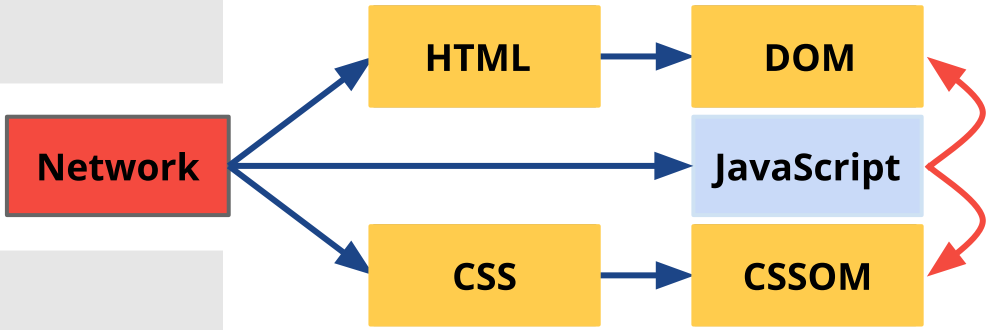
index.html

```
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<script src=application.js></script>
<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

```
p    { font-weight: bold; }
span { display: none; }
```

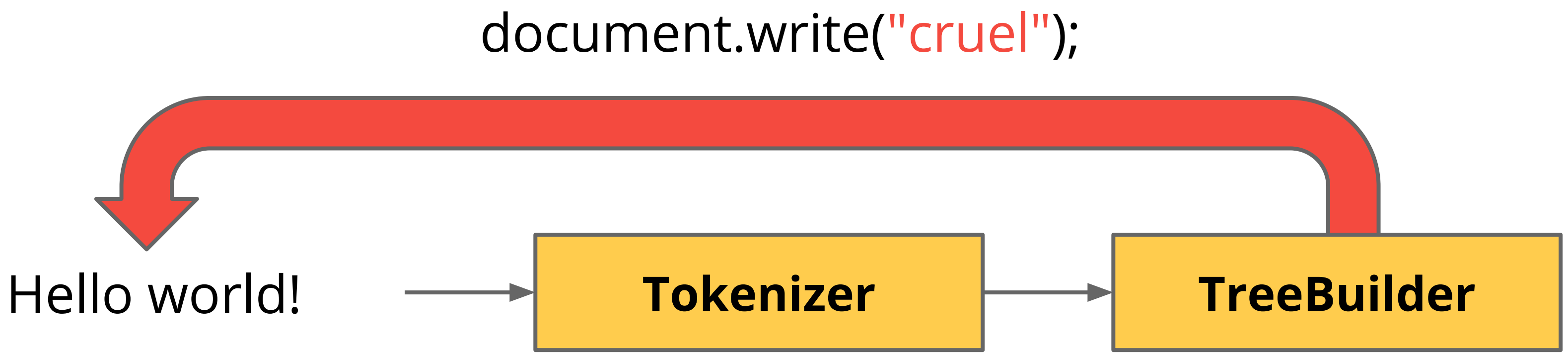
In some ways, JS is similar to CSS, except ...



elem.style.width = "500px"

JavaScript can query (and modify) DOM, CSSOM!

JavaScript can modify the DOM and CSSOM...



Script execution can change the input stream. Hence we **must wait**.

<script> could doc.write, stop the world!

- DOM construction **can't proceed** until JavaScript is fetched *
- DOM construction **can't proceed** until JavaScript is executed *

Sync scripts block the parser...

Sync script **will block** the DOM + rendering of your page:

```html
<script type="text/javascript"
        src="https://apis.google.com/js/plusone.js"></script>
```

Async script **will not block** the DOM + rendering of your page:

```html
<script type="text/javascript">
  (function() {
    var po = document.createElement('script'); po.type = 'text/javascript';
    po.async = true; po.src = 'https://apis.google.com/js/plusone.js';
    var s = document.getElementsByTagName('script')[0];
    s.parentNode.insertBefore(po, s);
  })();
</script>
```

Async all the things!

```
<script src="file-a.js"></script>
<script src="file-c.js" async></script>
```

- **regular - block on HTTP request**, parse, execute, proceed
- **async - download in background**, execute when ready

JavaScript performance pitfalls...

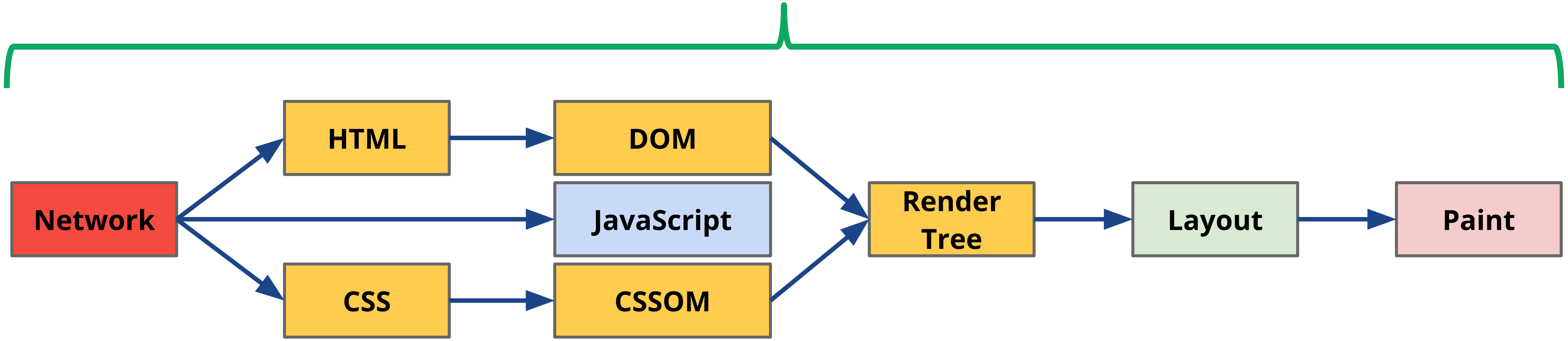
application.js

```
<script>

 var old_width = elem.style.width;
 elem.style.width = "300px";

 document.write("I'm awesome")

</script>
```

- JavaScript can **query** CSSOM
- JavaScript can **block on CSS**
- JavaScript can **modify** CSSOM

- JavaScript can **query DOM**
- JavaScript can **block DOM construction**
- JavaScript can **modify DOM**

Critical rendering path



(1) Stream the HTML to the client

- *Allows early discovery of dependent resources (e.g. CSS / JS / images)*

(2) Get CSS down to the client as fast as you can

- *Unblocks paints, removes potential JS waiting on CSS scenario*

(3) Use async scripts, avoid doc.write

- *Faster DOM construction, faster DCL and paint!*
- *Do you need scripts in your critical rendering path?*

Rendering path optimization?

Theory in practice...

*Breaking the **1000 ms** time to glass mobile barrier... **hard facts:***

1. **Majority of time is in network overhead**
   - *Especially for mobile! Refer to our earlier discussion...*
2. **Fast server processing time is a must**
   - *Ideally below 100 ms*
3. **Must allocate time for browser parsing and rendering**
   - *Reserve at least 100 ms of overhead*

Therefore...

*Breaking the **1000 ms** time to glass mobile barrier... **implications:***

1. **Inline just the required resources for above the fold**
    - *No room for extra requests... unfortunately!*
    - *Identify and inline critical CSS*
    - *Eliminate JavaScript from the critical rendering path*
2. **Defer the rest until after the above the fold is visible**
    - *Progressive enhancement...*
3. ...
4. *Profit*

```html
<html>

<head>
 <link rel="stylesheet" href="all.css">
 <script src="application.js"></script>
</head>

<body>
 <div class="main">
   Here is my content.
 </div>
 <div class="leftnav">
   Perhaps there is a left nav bar here.
 </div>
 ...
</body>
</html>
```

1. Split all.css, **inline critical** styles
2. Do you need the JS at all?
   - Progressive enhancement
   - **Inline critical** JS code
   - Defer the rest

```html
<html>
<head>

 <style>
   .main { ... }
   .leftnav { ... }
   /* ... any other styles needed for the initial render here ... */
 </style>

 <script>
   // Any script needed for initial render here.
   // Ideally, there should be no JS needed for the initial render
 </script>

</head>
<body>
 <div class="main">
   Here is my content.
 </div>
 <div class="leftnav">
   Perhaps there is a left nav bar here.
 </div>

 <script>
    function run_after_onload() {
      load('stylesheet', 'remainder.css')
      load('javascript', 'remainder.js')
    }
 </script>

</body>
</html>
```

Above the fold CSS

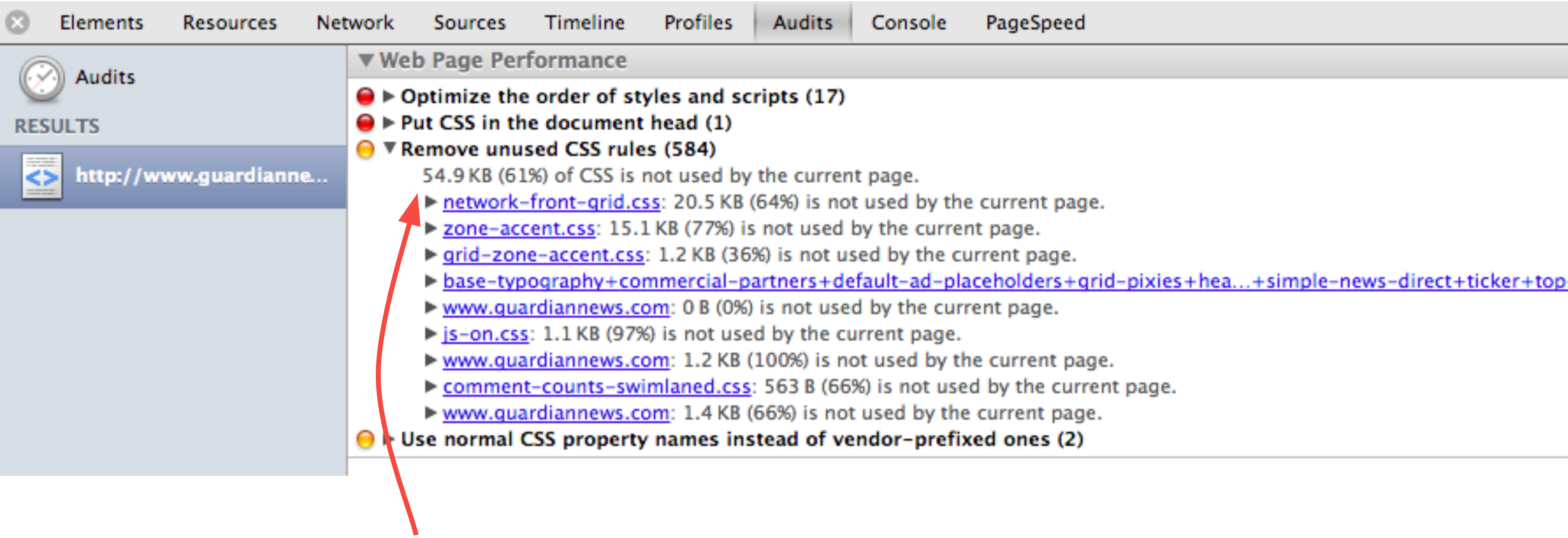
Above the fold JS
(ideally, none)

Paint the above the fold, then fill in the rest

A few tools to help you...

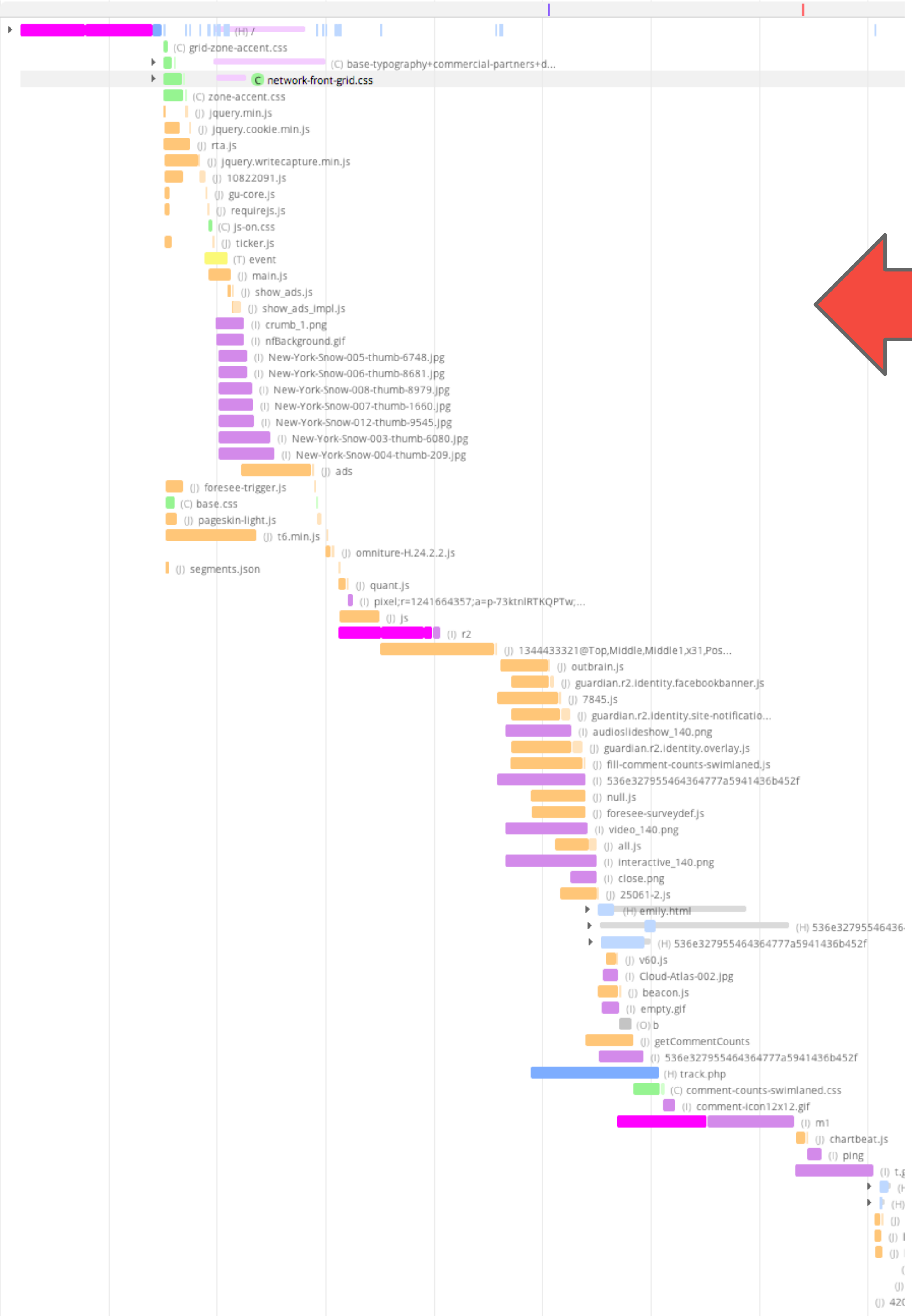
How do I find "critical CSS" and my critical rendering path?

Identify critical CSS via an Audit



DevTools > Audits > Web Page Performance

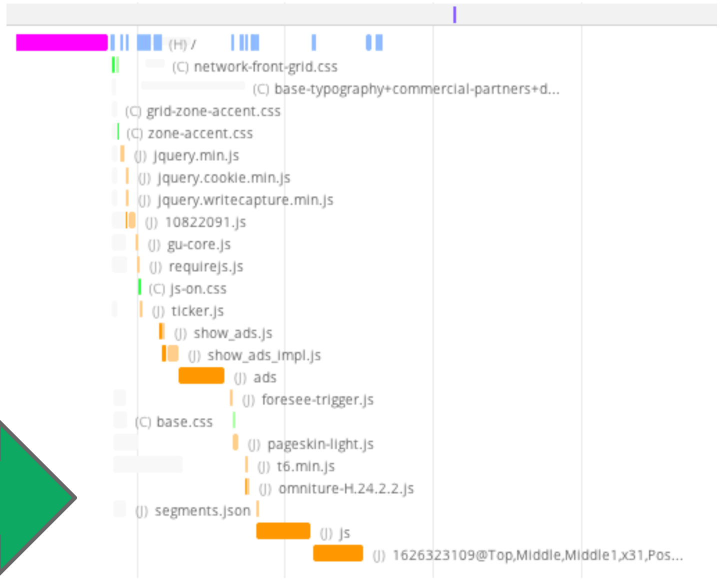
Another fun tool: http://css.benjaminbenben.com/v1?url=http://www.igvita.com/



guardian.co.uk

Full Waterfall

Critical Path



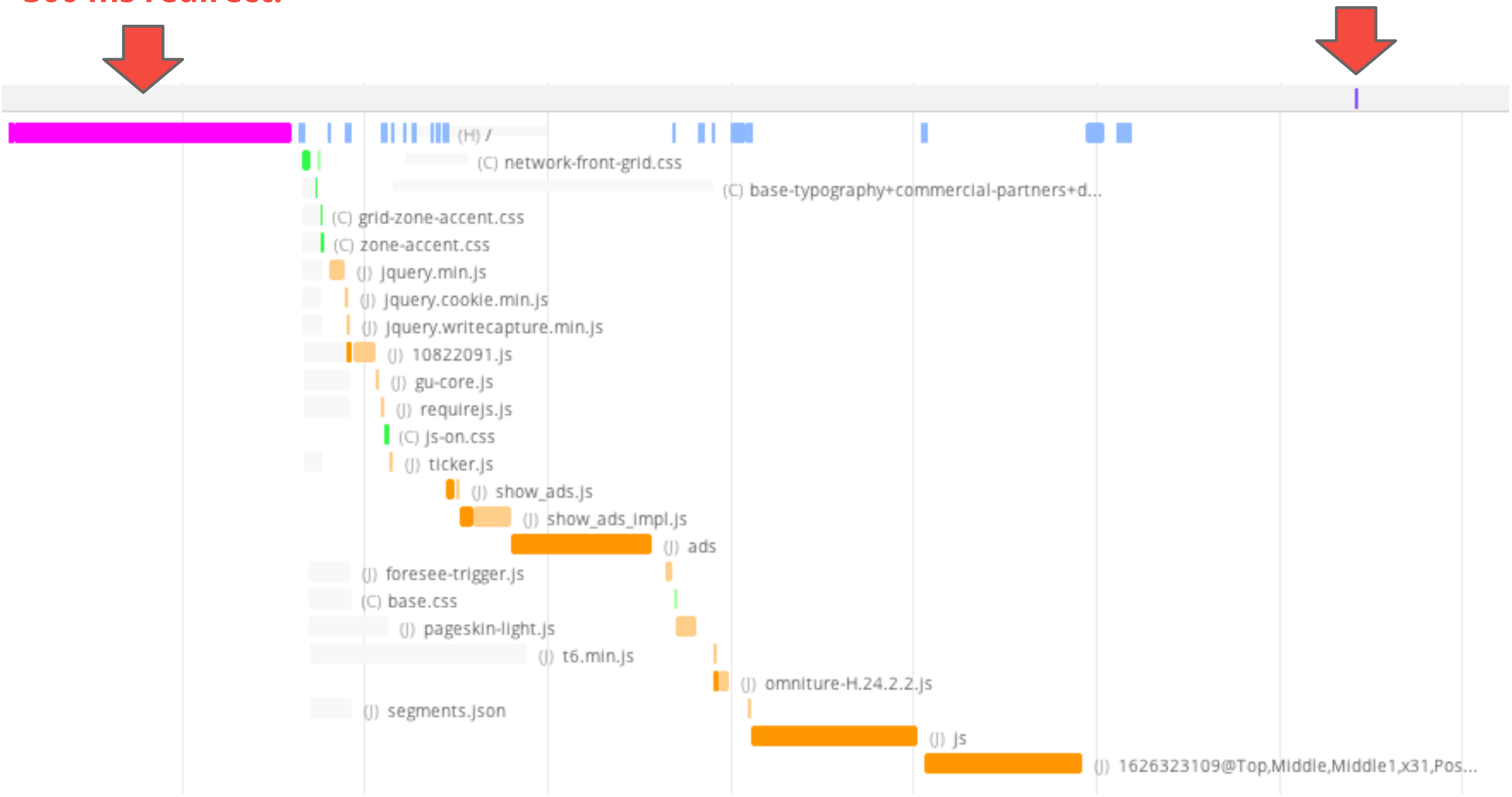
Critical Path Explorer* extracts the subtree** *of the waterfall that is in the "critical path***" of the* ***document parser and the renderer.***

(webpagetest run)

@igrigorik

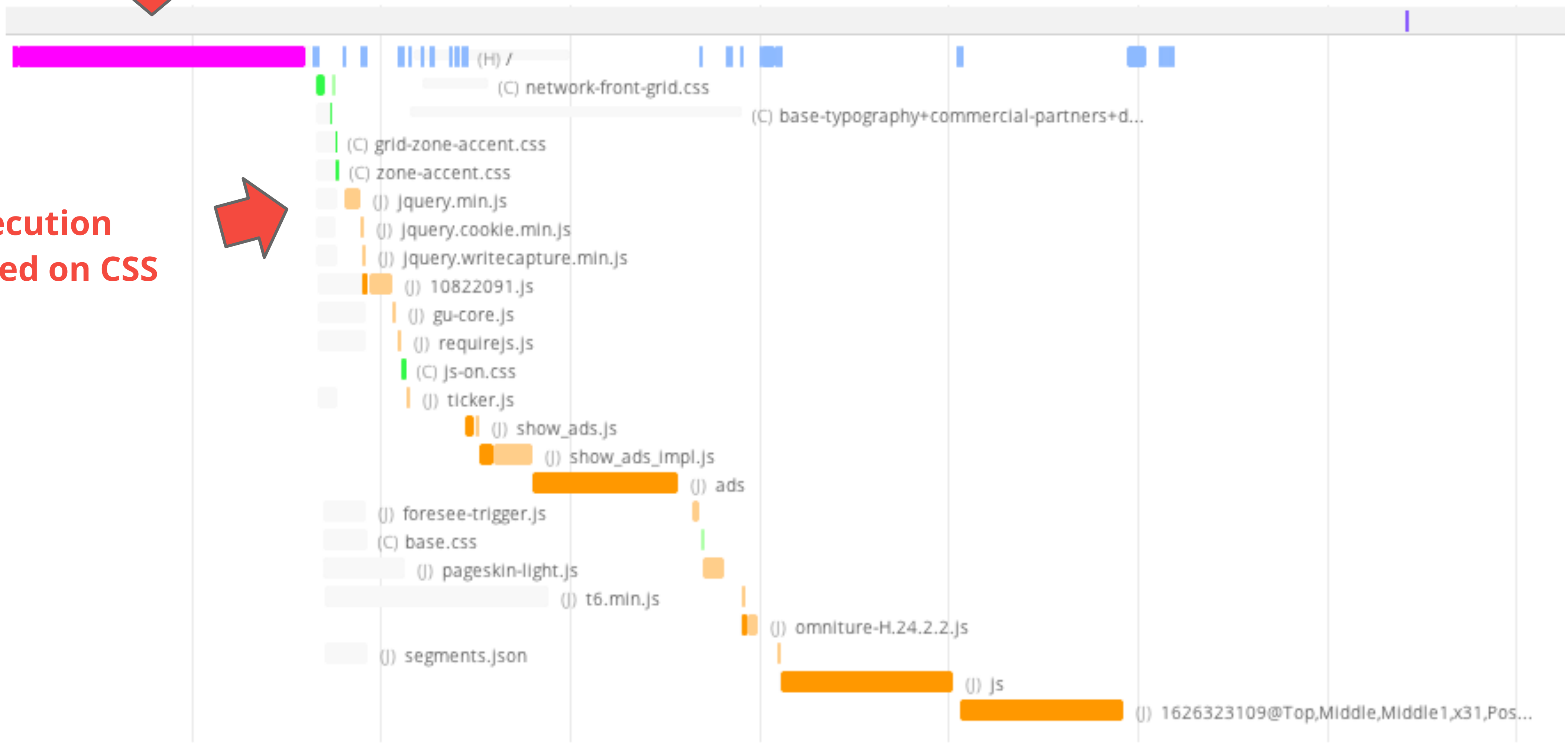
300 ms redirect!

DCL.. no defer



300 ms redirect!

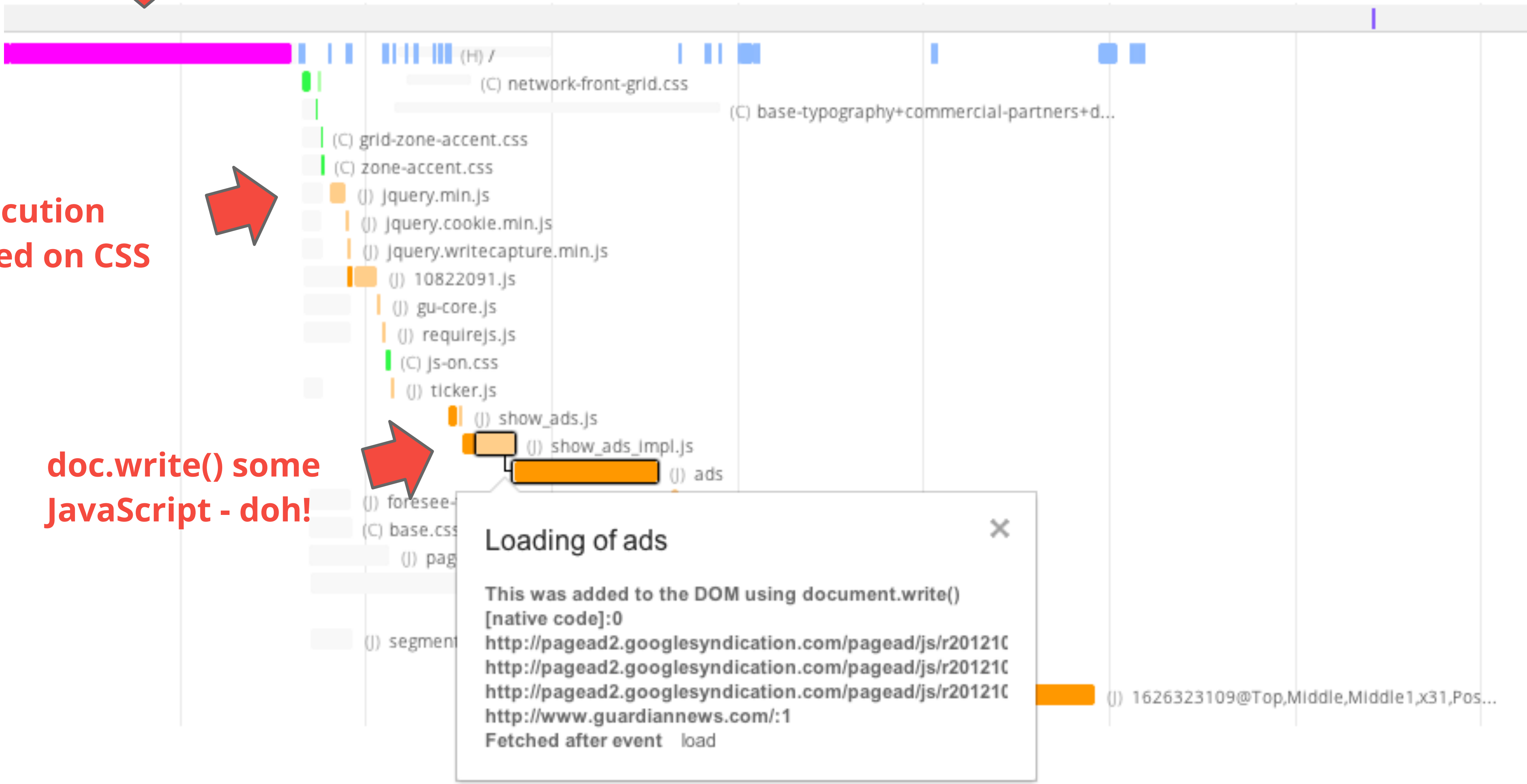
JS execution blocked on CSS



300 ms redirect!

JS execution blocked on CSS

doc.write() some JavaScript - doh!

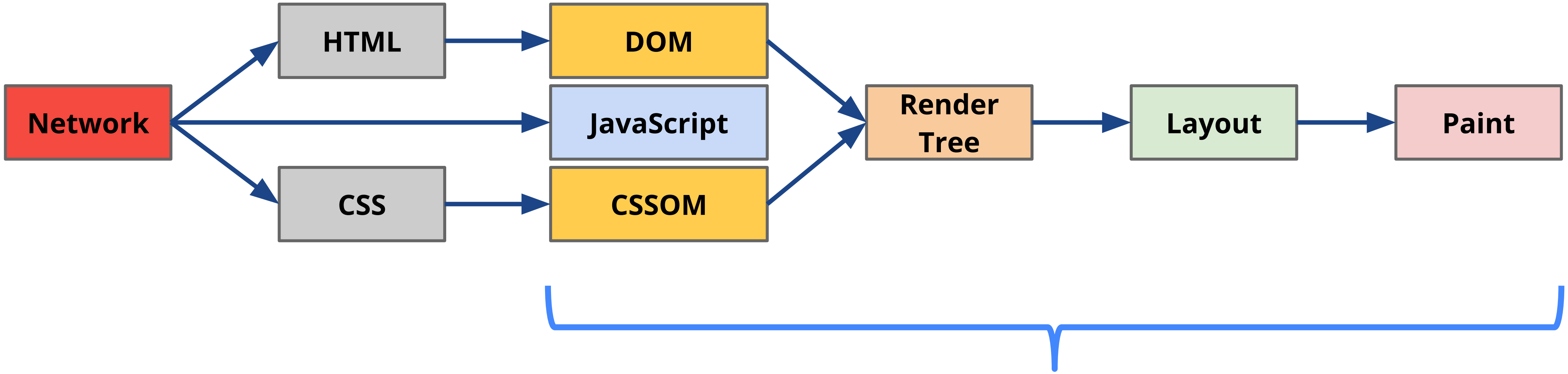



300 ms redirect!
(H) /
(C) network-front-grid.css
(C) base-typography+commercial-partners+d...
(C) grid-zone-accent.css
(C) zone-accent.css
(J) jquery.min.js
(J) jquery.cookie.min.js
(J) jquery.writecapture.min.js
(J) 10822091.js
(J) gu-core.js
(J) requirejs.js
(C) js-on.css
(J) ticker.js
(J) show_ads.js
(J) show_ads_impl.js
(J) ads
(J) foresee-trigger.js
(C) base.css
(J) pageskin-light.js
(J) t6.min.js
omniture-H.24.2.2.js
(J) segments.json
J js
(J) 1626323109@Top,Middle,Middle1,x31,Pos...
JS execution blocked on CSS
doc.write() some JavaScript - doh!
long-running JS

@igrigorik

10m break... Questions?

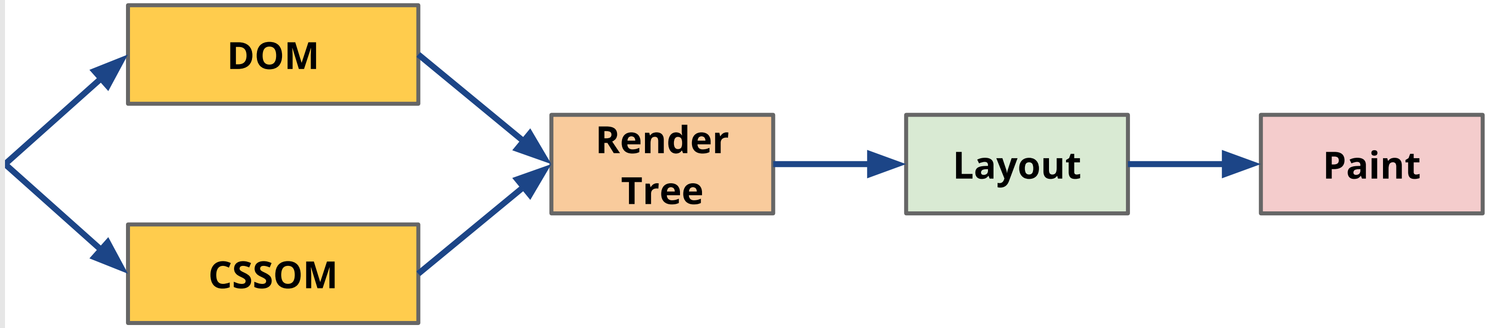
Twitter @igrigorik
G+ gplus.to/igrigorik
Web igvita.com



In-app performance: CPU + Render

Same pipeline... except running in a loop!

```
document.write("<p>I'm awesome</p>");

var old_width = elem.style.width;
elem.style.width = "300px";

 // or user input...
```



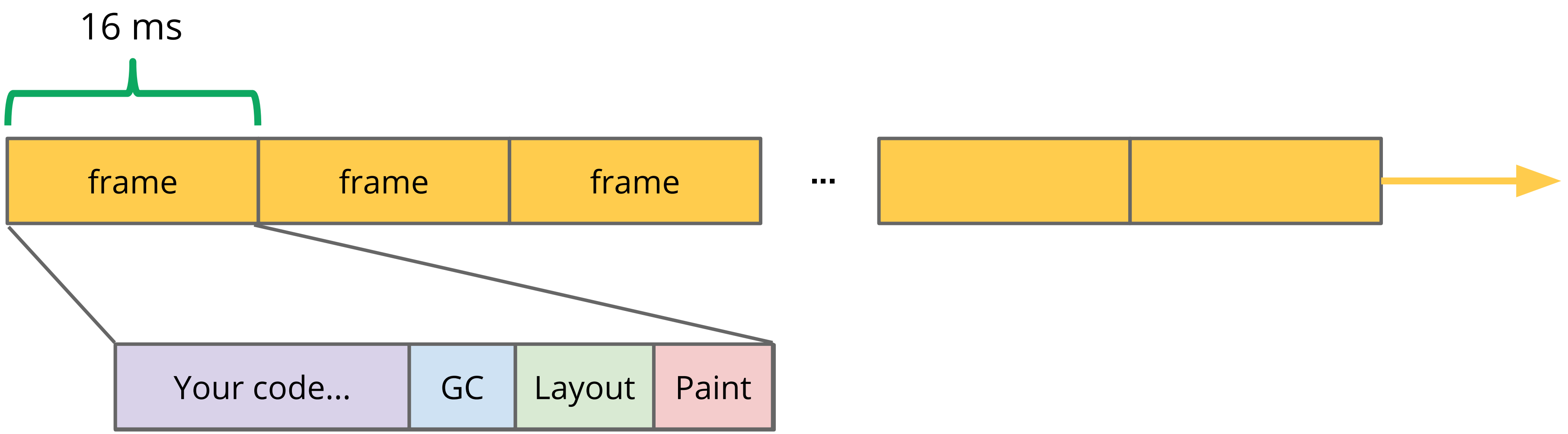
- User can trigger an update: click, scroll, etc.
- JavaScript can manipulate the DOM
- JavaScript can manipulate the CSSOM

- Which may trigger a:
  - Style recalculation
  - Layout recalculation
  - Paint update

Performance = 60 FPS.

1000 ms / 60 FPS = ***16 ms / frame***

Brief anatomy of a "frame"

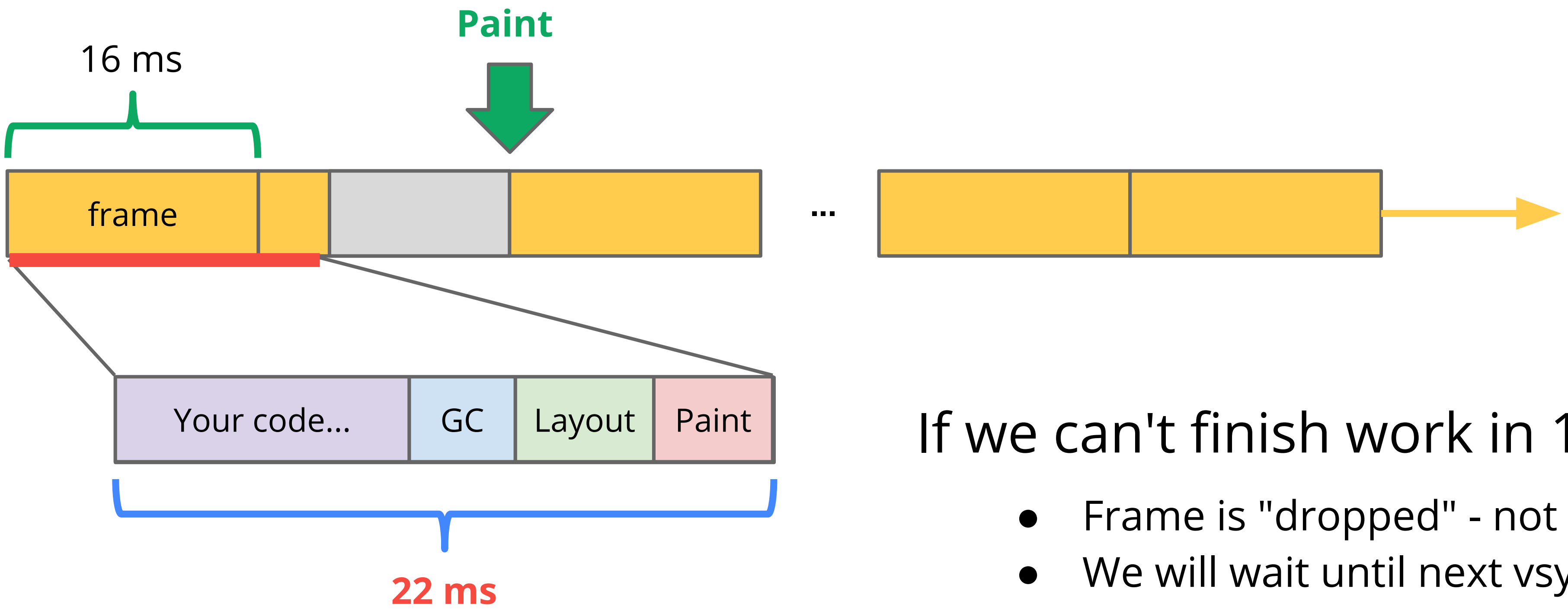


16 milliseconds is **not a lot of time**! The budget is split between:

- Application code
- Style recalculation
- Layout recalculation
- Garbage collection
- Painting

Not necessarily in this order, and we (hopefully) don't have to perform all of them on each frame!

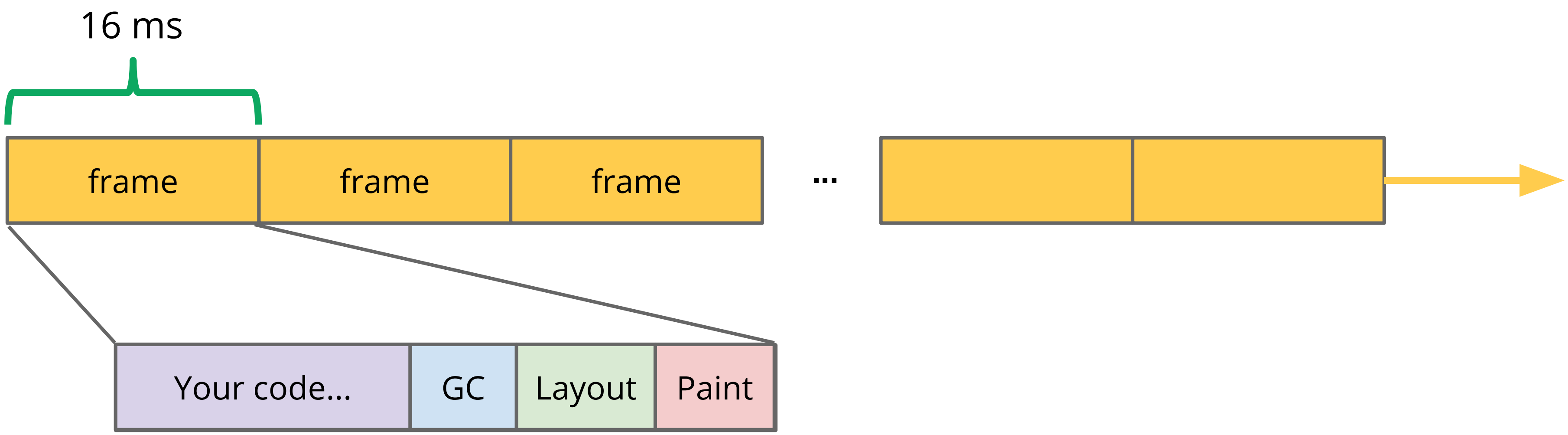
What happens if we exceed the budget?



If we can't finish work in 16 ms...

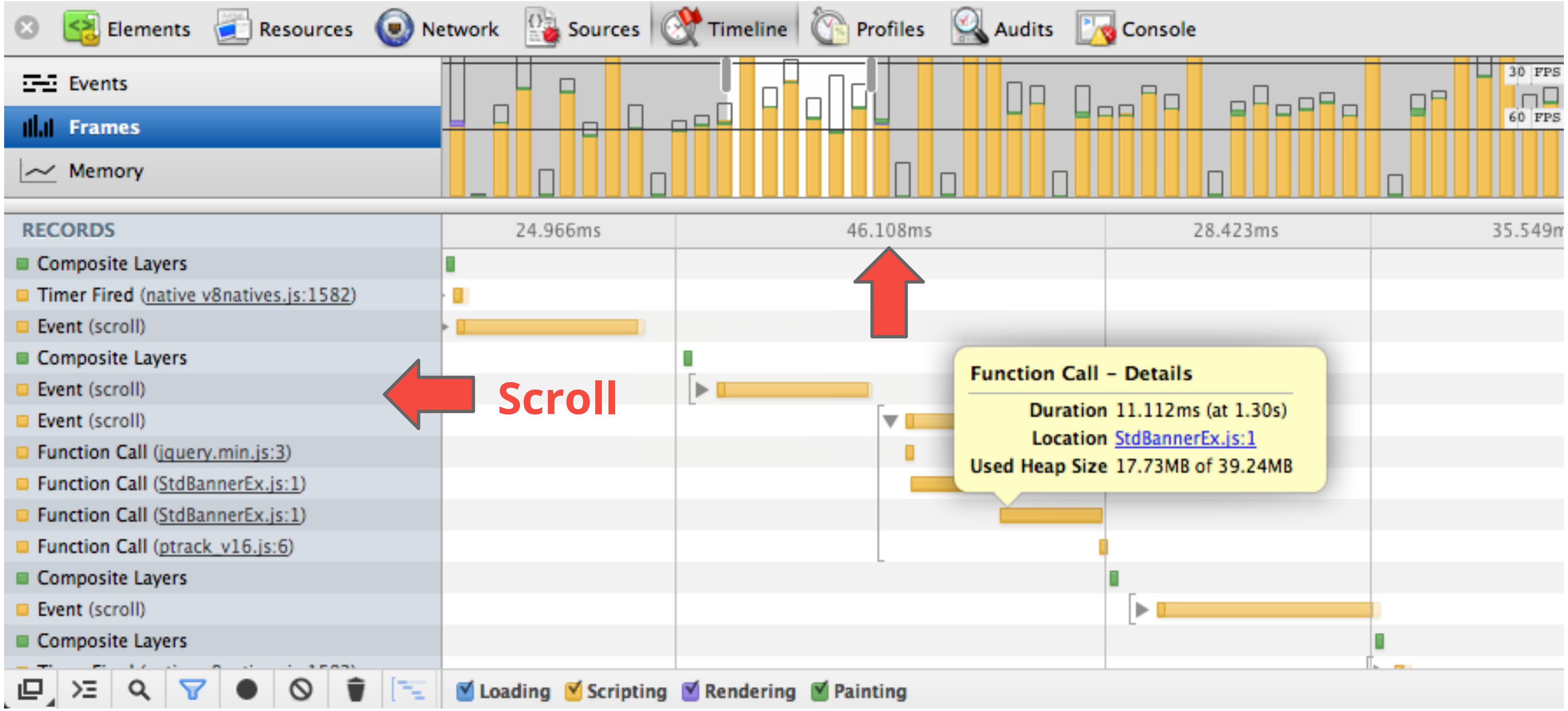
- Frame is "dropped" - not rendered
- We will wait until next vsync
- ...
- Dropped frames = **"jank"**

Jank-free axioms



- Your code must yield control in **less than 16 ms!**
  - Aim for <10ms
  - Browser needs to do extra work: GC, layout, paint
  - Don't forget that "10 ms" is not absolute (e.g. slower CPU's)
- Browser won't (can't) interrupt your code...
  - Split long-running functions
  - Aggregate events (e.g. handle scroll events once per frame)

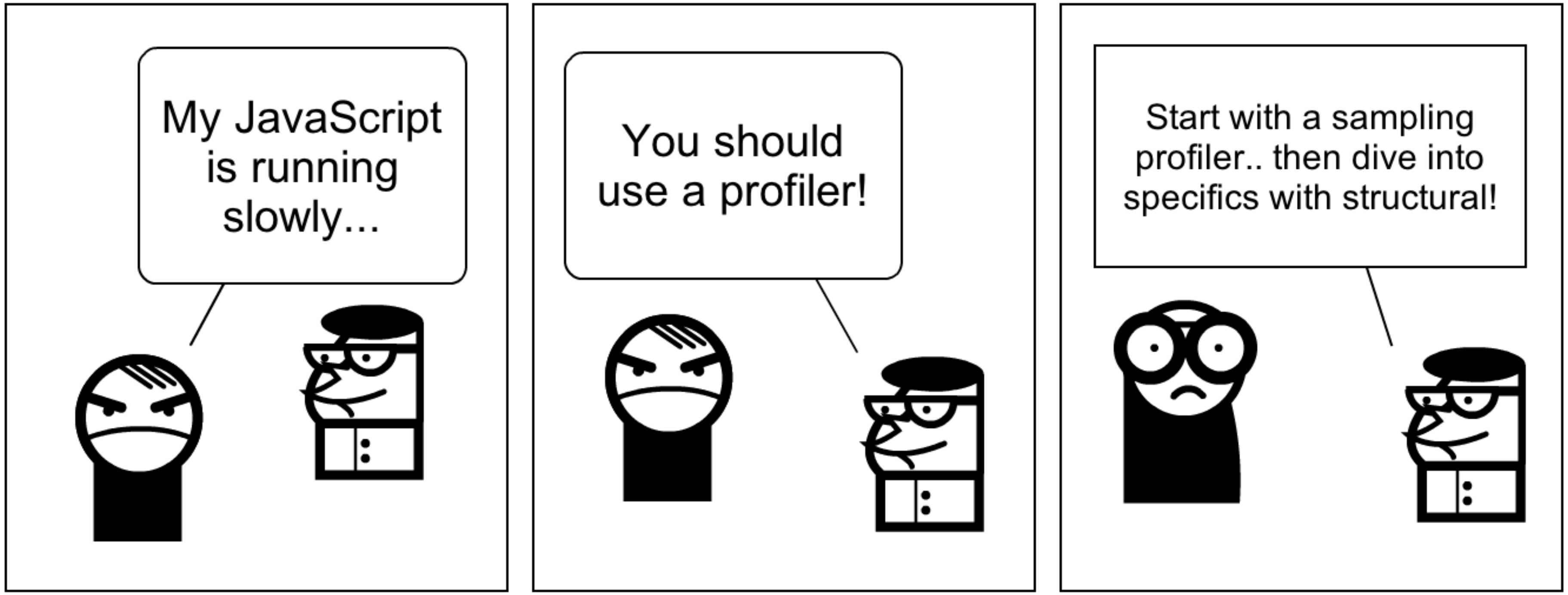
JavaScript induced jank...

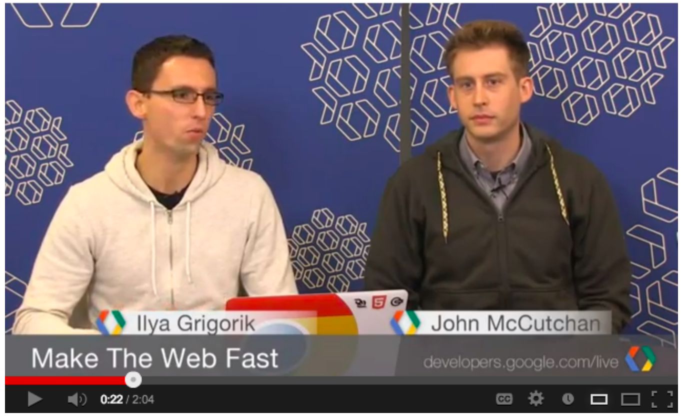


- Aggregate your scroll events and **defer** them
- Process aggregated events on **next** requestAnimationFrame callback!

Profile your JavaScript code!

10 ms is not a lot of time. What's your bottleneck?



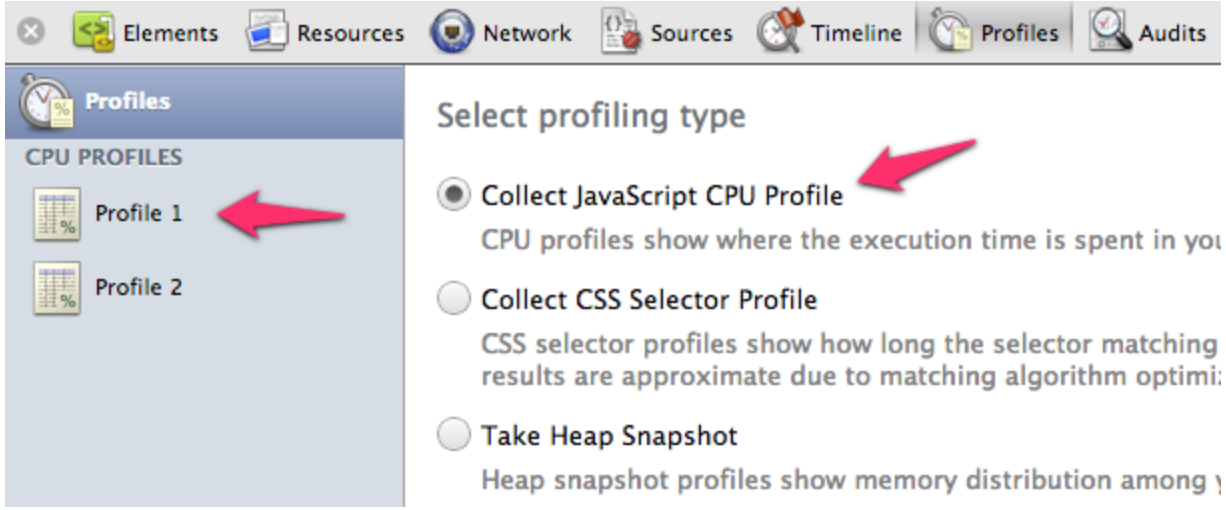


Structural and Sampling JavaScript Profiling

in Google Chrome

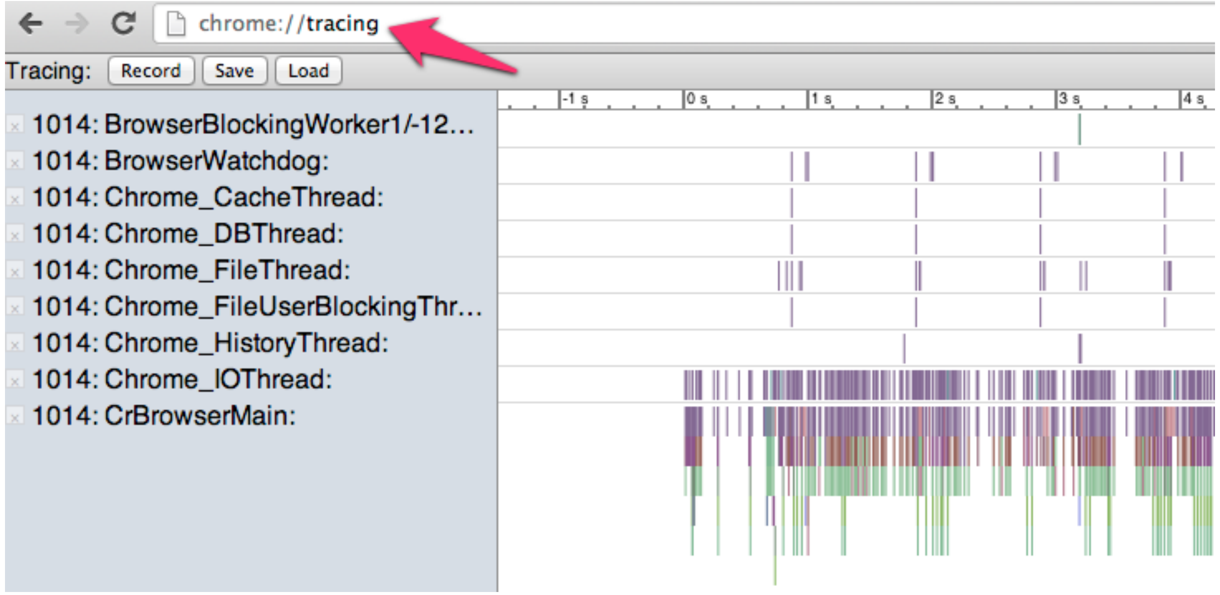
http://www.youtube.com/watch?v=nxXkquTPng8

1. **Sampling**
    a. Measures samples



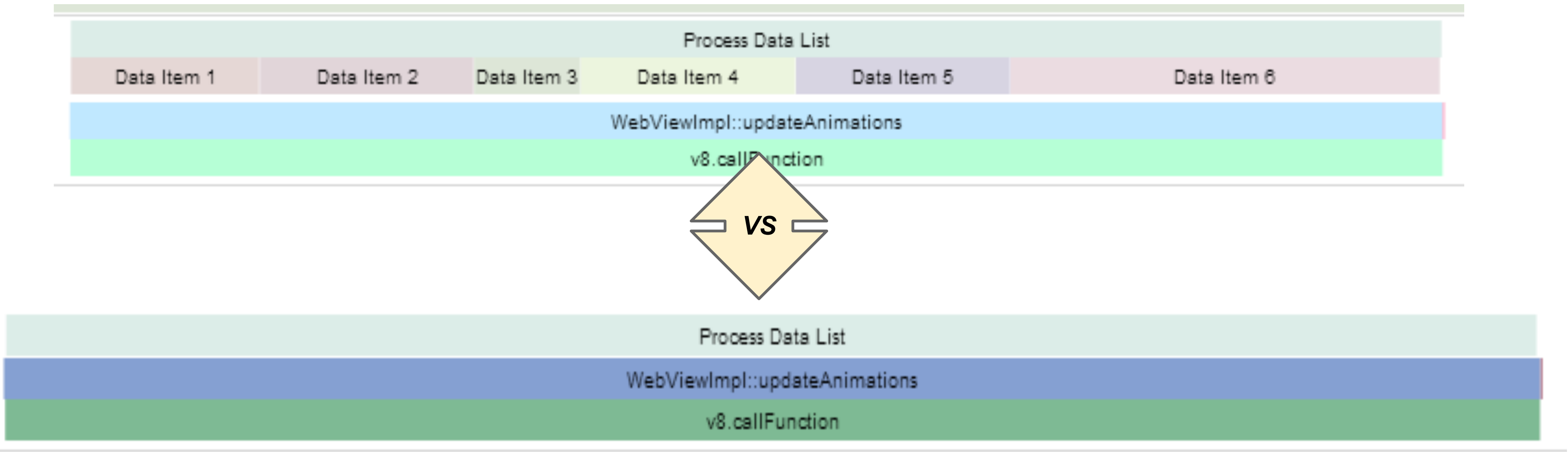
2. **Structural**
    a. Measures time
    b. aka, **instrumenting** / markers / inline

aka... **chrome://tracing**



Annotate your code for structural profiling!

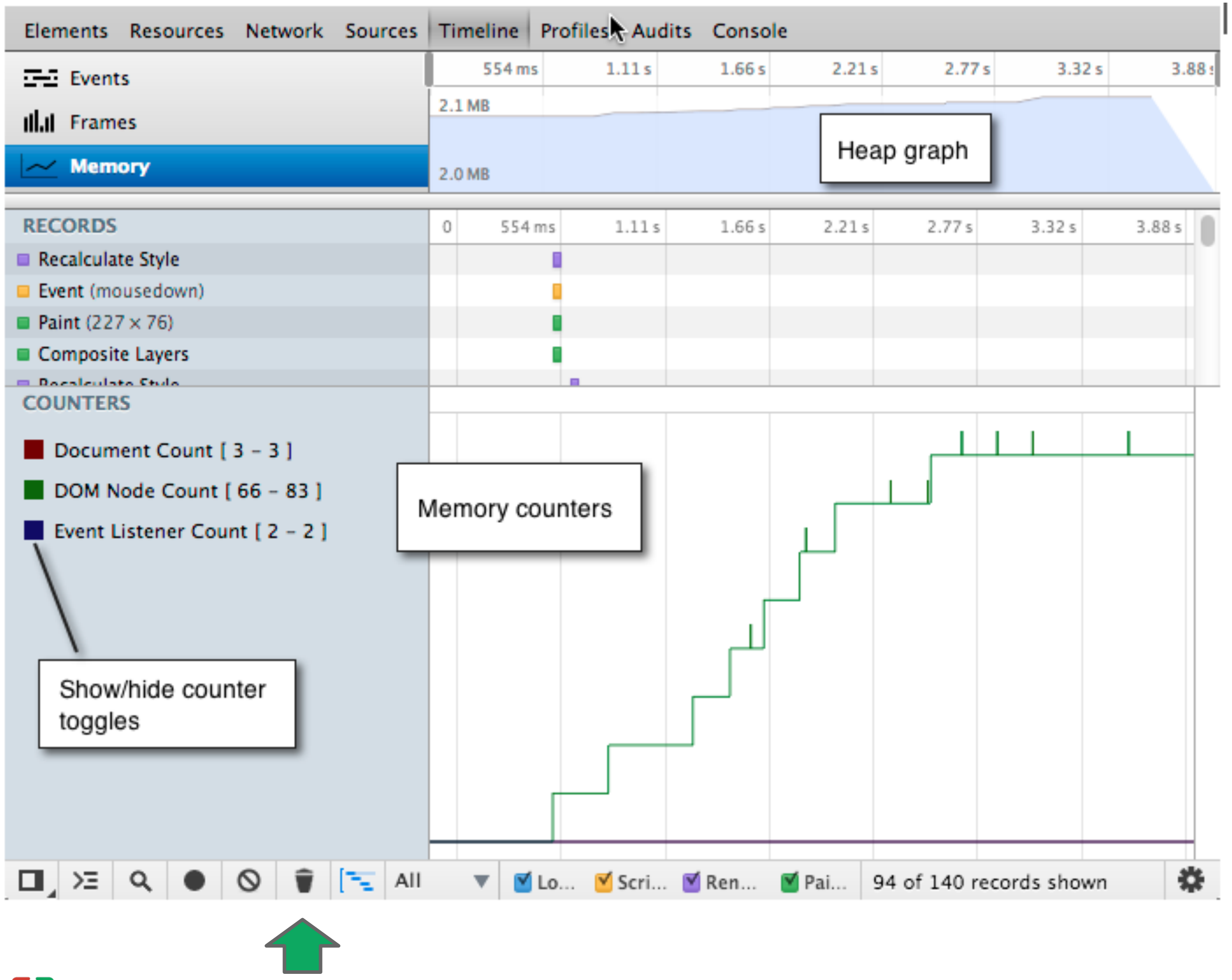
```
function A() {
  console.time("A");
  spinFor(2);    // loop for 2 ms
  B();
  console.timeEnd("A");
}
```



Garbage happens...

And that's ok. But, is GC your bottleneck? Memory leaks?

Timeline » Memory



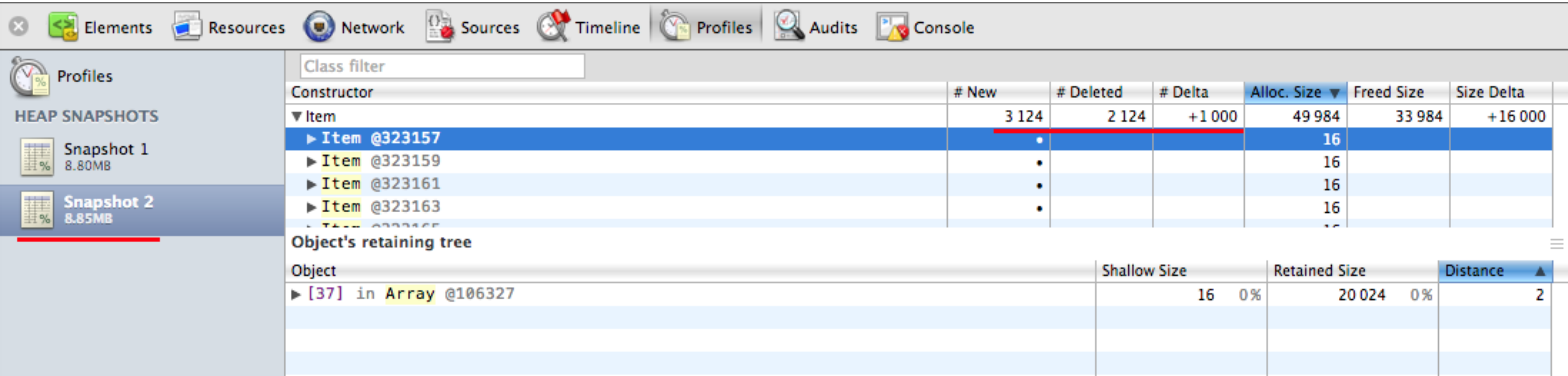
Force GC

1. CMD-E to start recording
2. Interact with the page
3. Track amount of allocate objects
4. ...
5. ***Fix leak(s)***
6. ...
7. *Profit*

Tip: use an ***Incognito*** *window when profiling code!*

Heap snapshot + comparison view

1. Snapshot, save, import heap profile
2. Use comparison view to identify potential memory leaks (demo)
3. Use summary view to identify DOM leaks (demo)



Know thy memory model

http://goo.gl/dtRl8

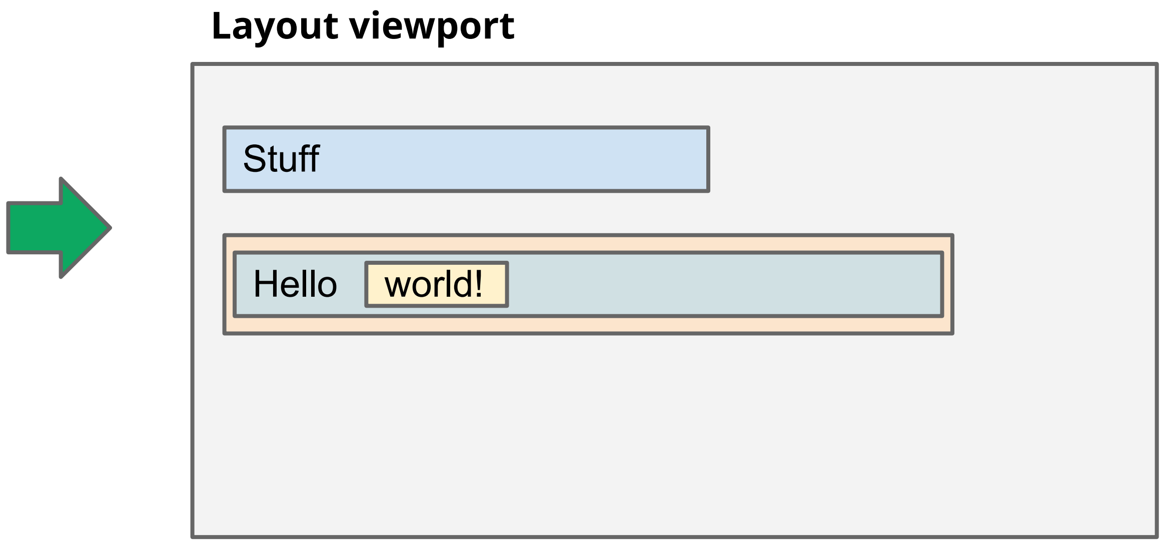
- What are memory leaks?
- Tracking down memory leaks...
- War stories from GMail team

What's a "layout" anyway?

And how do we optimize for it?

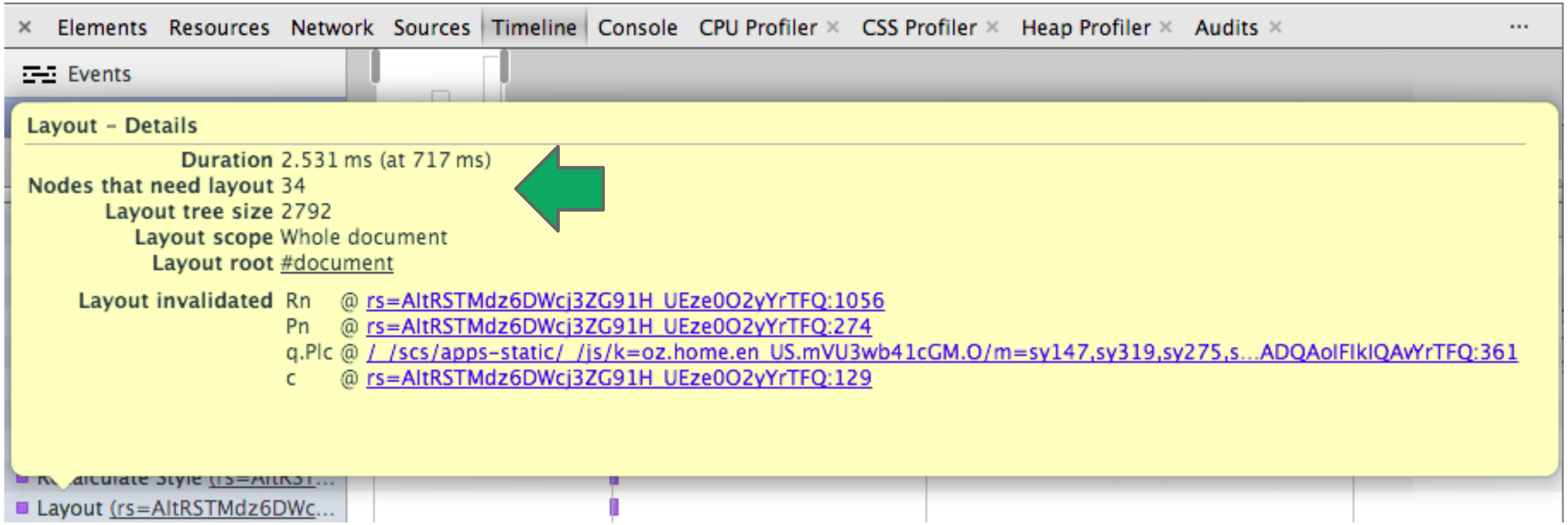
Layout: computing the width/height/position...

```
<div style="width:50%">
  Stuff
</div>

<div style="width:75%">
  <p>
    Hello <span>world!</span>
  </p>
</div>
```



- Layout phase calculates the size of each element: width, height, position
  - margins, padding, absolute and relative positions
  - propagate height based on contents of each element, etc...
- **What will happen if I resize the parent container?**
  - All elements under it (and around it, possibly) will have to be recomputed!

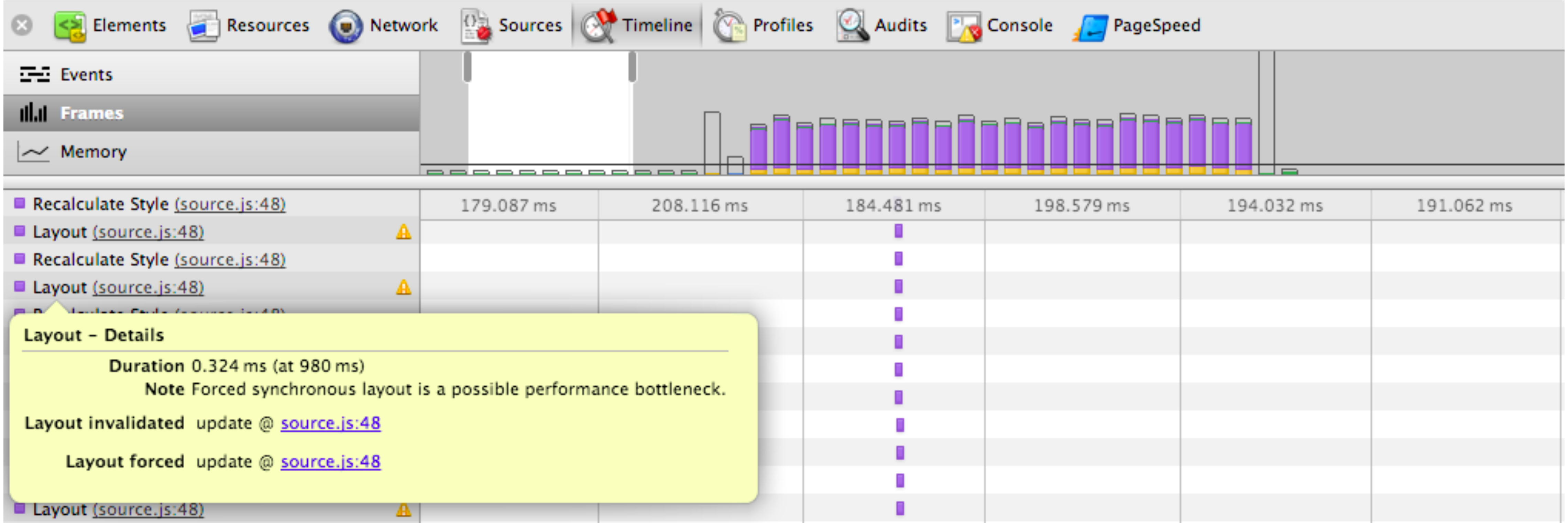
Diagnosing layout performance



- **2.5 ms** to perform triggered layout
- **34 affected nodes** (children)
  - Total DOM size: 2792 nodes

- Be careful about triggering expensive layout updates!
  - *Adding nodes, removing nodes, updating styles, … just about anything, actually. :-)*

Layout can be *very* expensive....

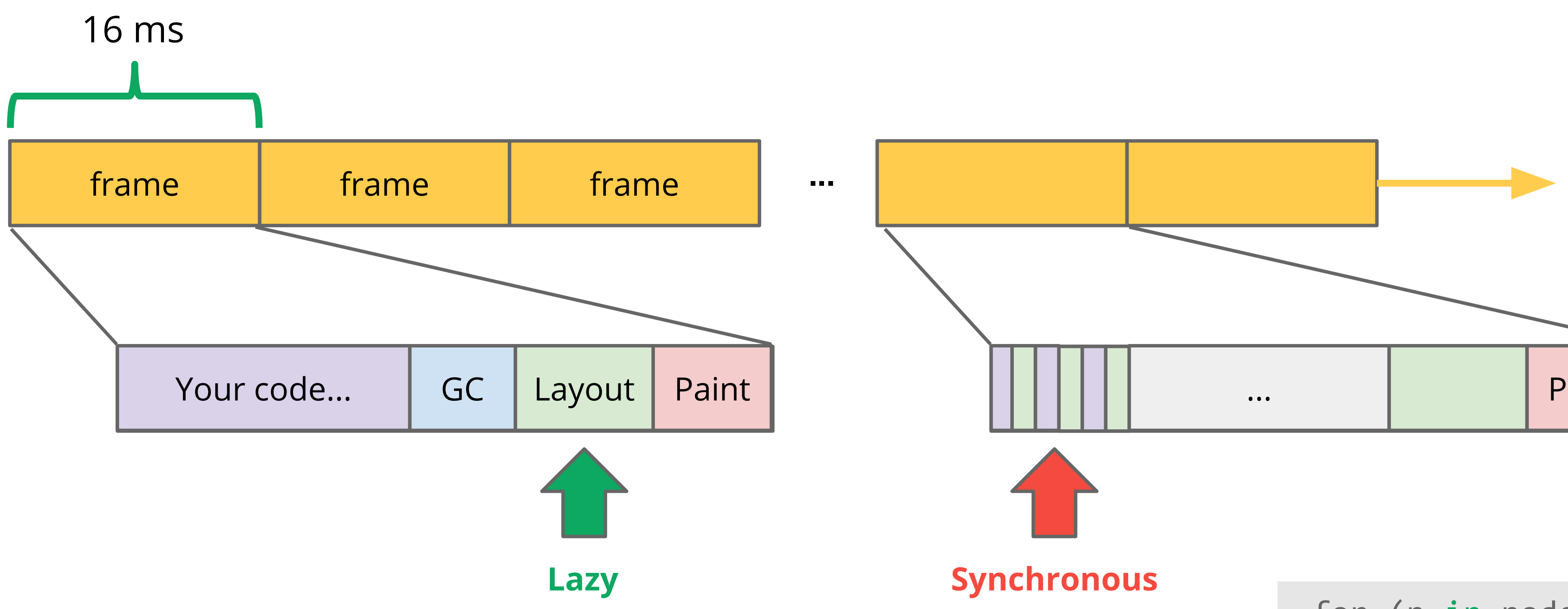


- **Style recalculation is forcing a layout update...** (hence the warning)
  - Change in size, position, etc...
- Synchronous layout? Glad you asked...

https://developers.google.com/chrome-developer-tools/docs/demos/too-much-layout/

@igrigorik

Ideally, the layout is performed only once



- DOM / CSSOM modification → **dirty tree**
  - Ideally, **recalculated once**, immediately prior to paint
- Except.. you can force a **synchronous layout!**

```
for (n in nodes) {
  n.style.left =
     n.offsetLeft + 1 + "px";
}
```

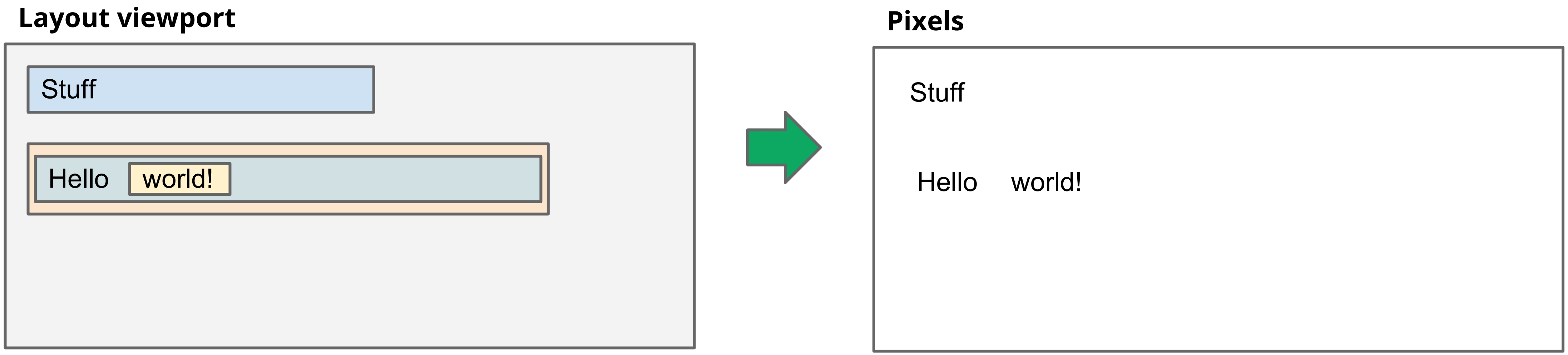
- First iteration marks tree as dirty
- Second iteration forces layout!

https://developers.google.com/chrome-developer-tools/docs/demos/too-much-layout/

OK. Let's paint some pixels!

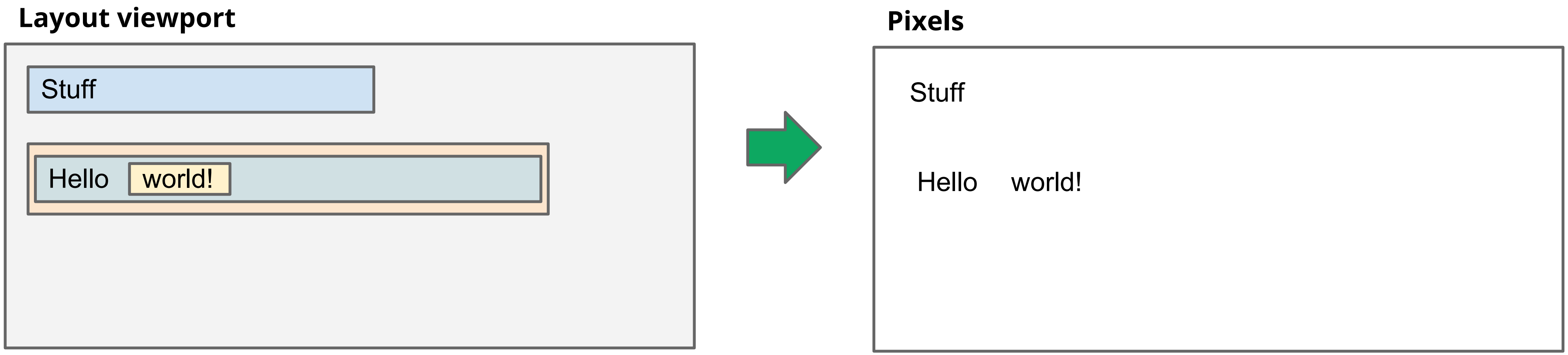
Only took us a few hours to get here...

Paint process in a nutshell



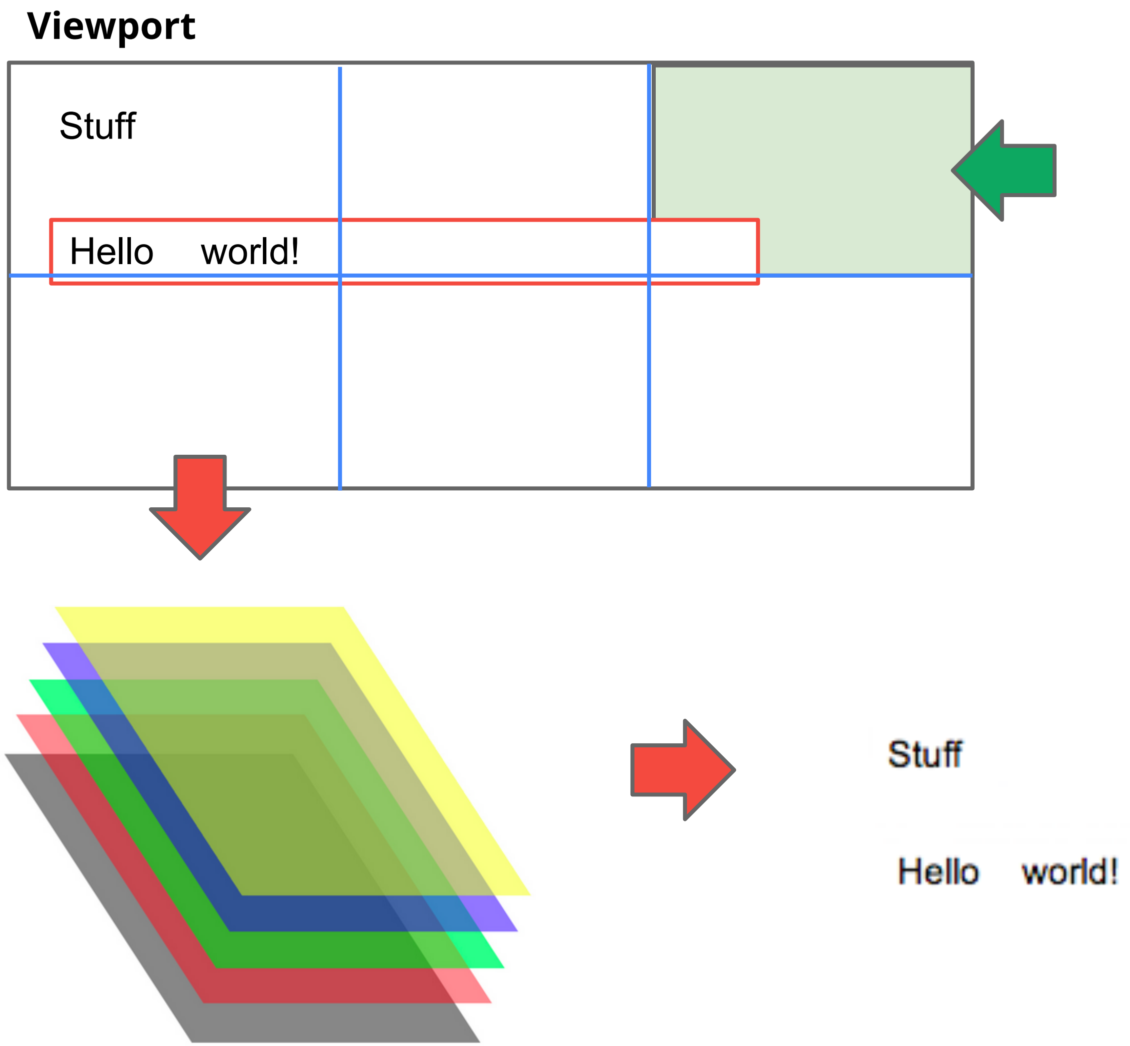
- Given layout information of all elements
    - Apply all the visual styles to each element
    - Composite all the elements and layers into a bitmap
    - Push the pixels to the screen

Paint process has variable costs based on...



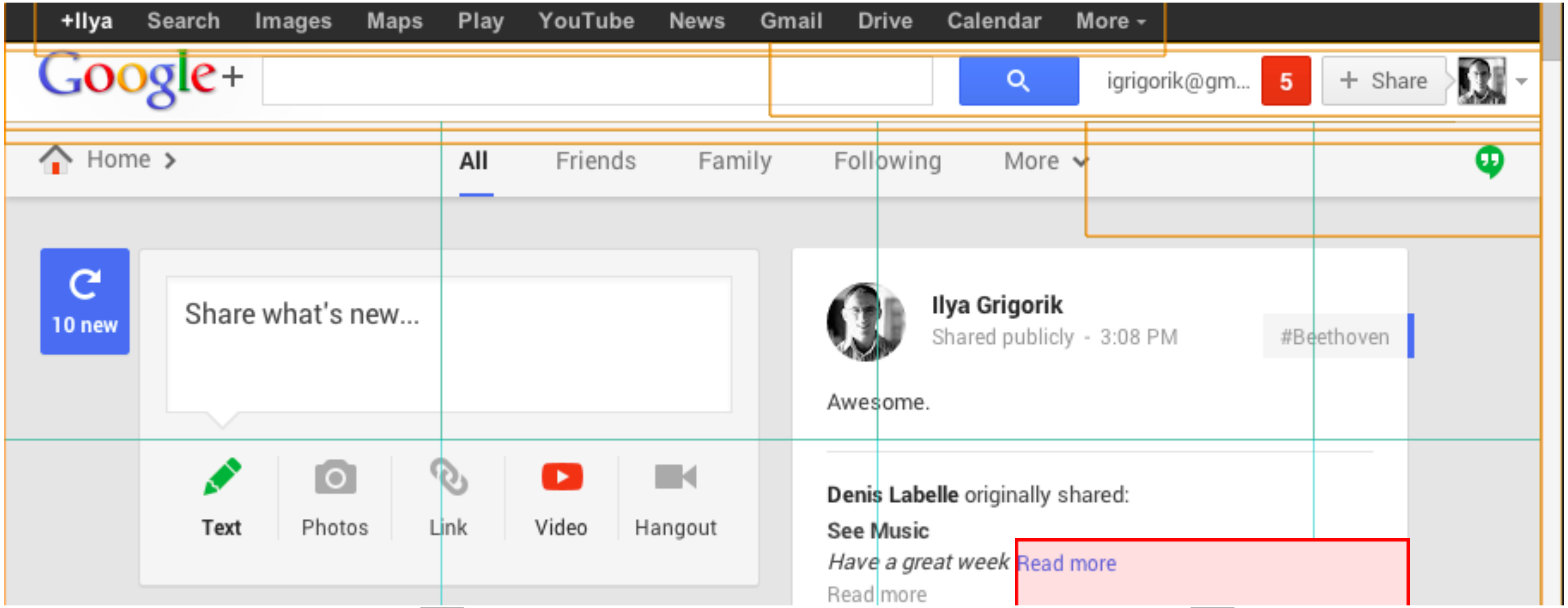
- **Total area** that needs to be (re)painted
  - *We want to update the minimal amount*
- Pixel rendering cost varies based on **applied effects**
  - *Some styles are more expensive than others!*

Rendering 101



- **Viewport is split into rectangular tiles**
  - *Each tile is rendered and cached*
- **Elements can have own layers**
  - *Allows reuse of same texture*
  - *Layers can be composited by GPU*

***Gold borders* show independent layers**



Rendering is done in rectangular tiles

Red border shows repainted area

Wait, DevTools could do THAT?

@igrigorik

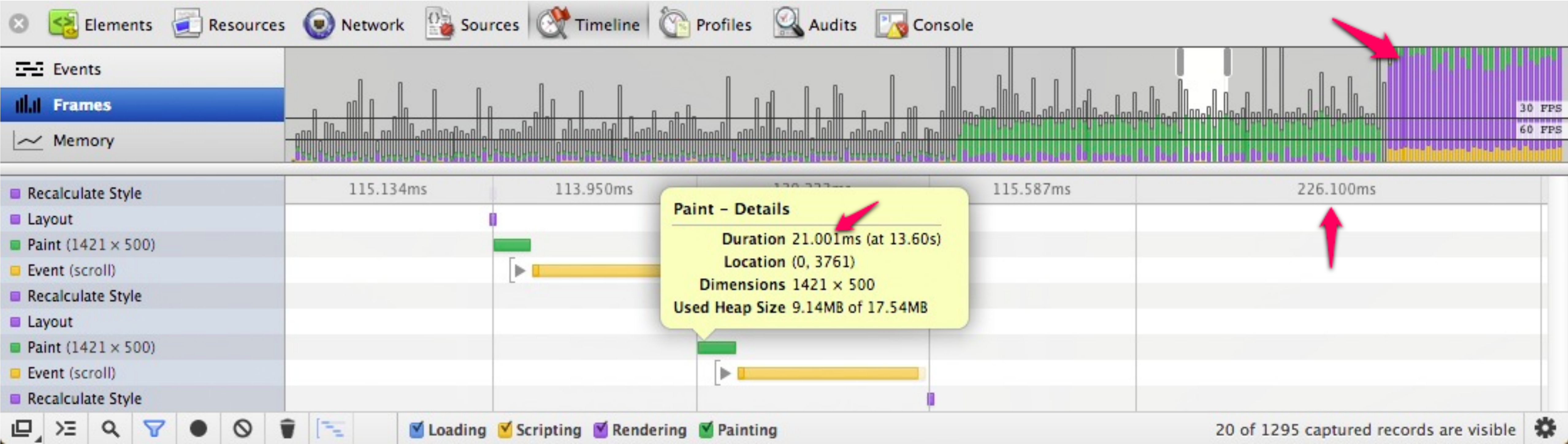
Let's diagnose us some Jank....

- ☑ Show paint rectangles
- ☑ Show composited layer borders
- ☑ Show FPS meter
- ☑ Enable continuous page repainting

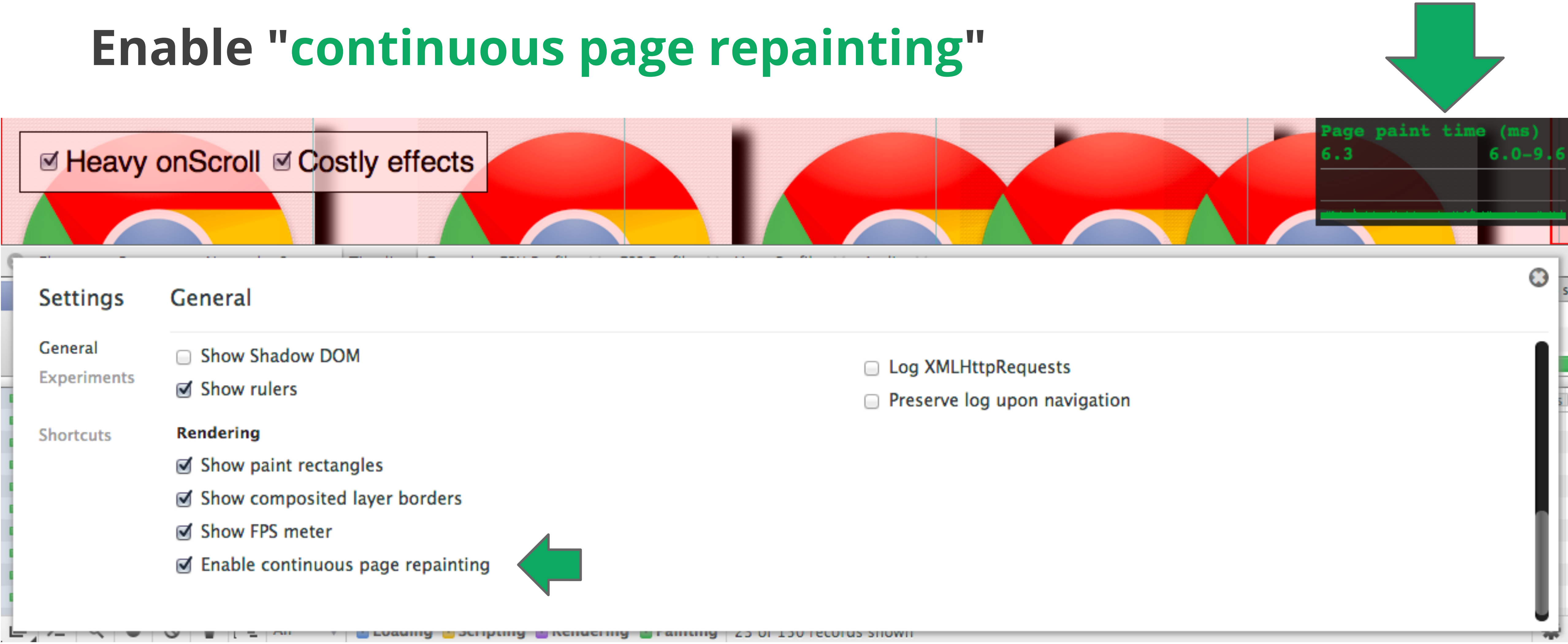
What's the source of the problem?

- Large paints?
- CPU / JavaScript bound?
- Costly CSS effects?

Let's find out... *(hint, all of the above)*



Enable "continuous page repainting"



- Force full repaint on **every frame** to help find expensive elements and effects
- In Elements tab, hit "**h**" to hide the element, and watch the paint time costs!

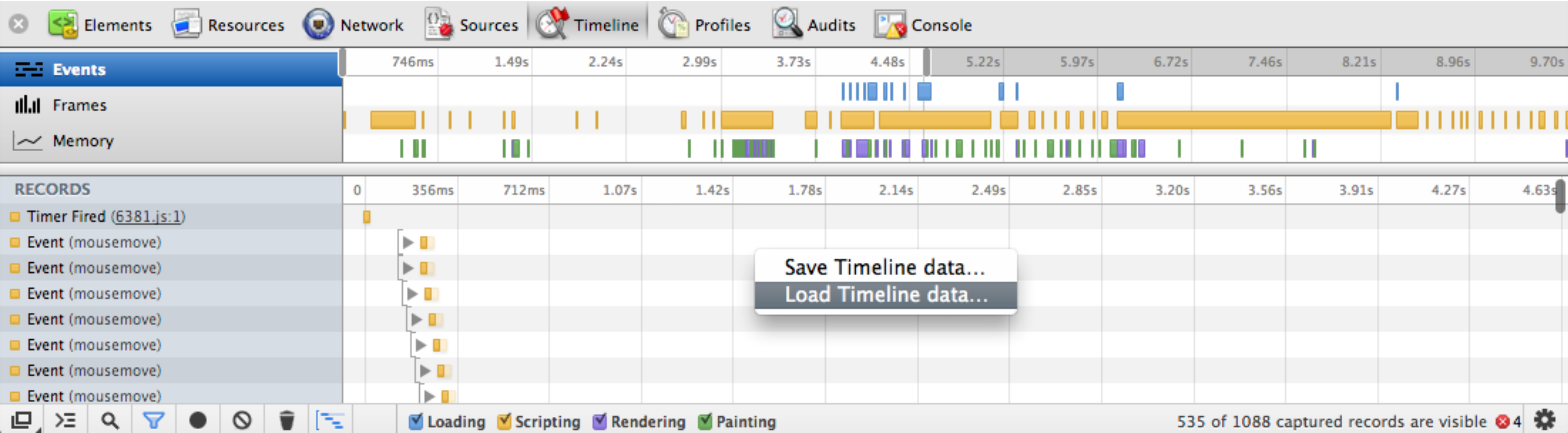
A few Chrome tips...

to make your debugging workflow more productive

Timeline trace ***or it didn't happen...***

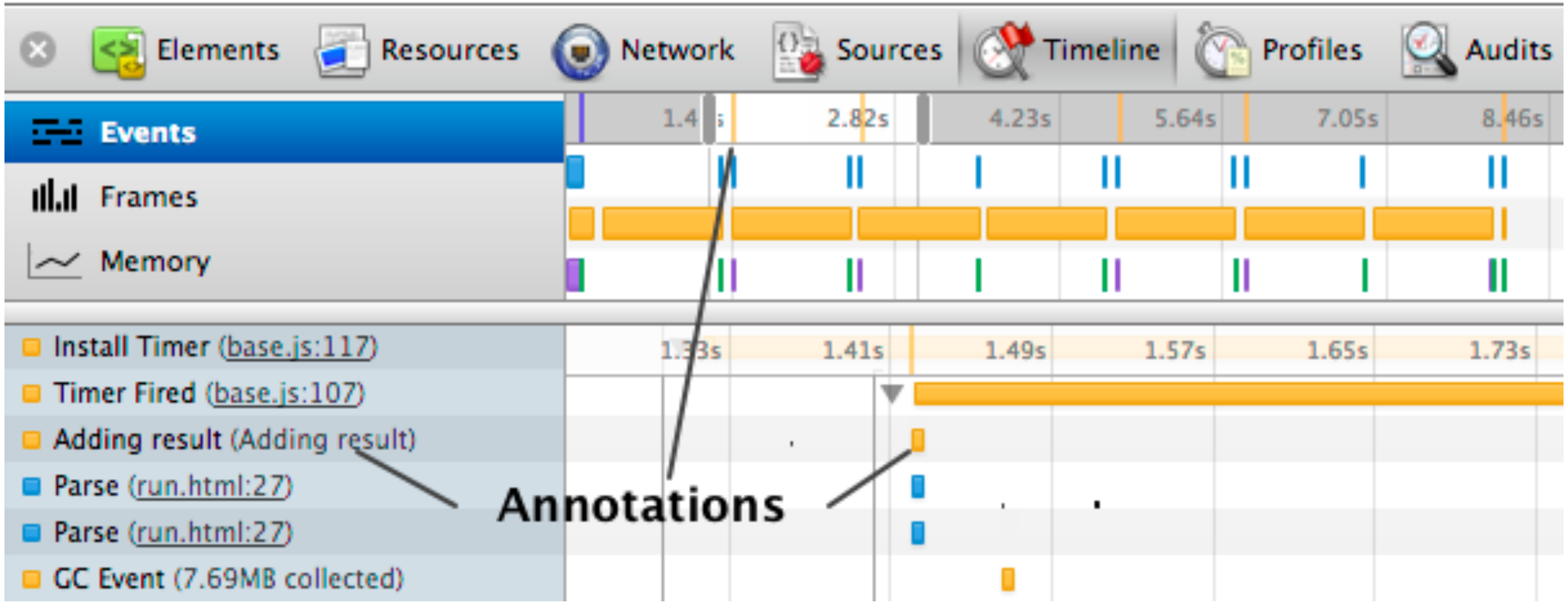
1. **Export timeline trace** (raw JSON) for bug reports, later analysis, ...
2. **Attach said trace** to bug report!
3. **Load trace** and analyze the problem - kthnx!

*Protip: **CMD-e** to start and stop recording!*



Annotate your Timeline!

```
function AddResult(name, result) {
  console.timeStamp("Adding result");
  var text = name + ': ' + result;
  results.innerHTML += (text + "<br>");
}
```



Test your **rendering performance** on mobile device!

*Connect your Android device via USB to the desktop and view and debug the code executing **on the device**, with **all the same DevTools features!***

1. *Settings > Developer Tools > **Enable USB Debugging***
2. ***chrome://inspect*** *(on Canary)*
3. ...
4. *Profit*

Wait, what about the GPU?

Won't it make rendering "super fast"?

Hardware Acceleration 101

1. The **object is painted** to a buffer (texture)
2. **Texture is uploaded** to GPU
3. Send commands to GPU: **apply op X to texture Y**

- A RenderLayer can have a GPU backing store
- Certain elements are GPU backed automatically
  - *canvas, video, CSS3 animations, ...*
- Forcing a GPU layer: *-webkit-transform:translateZ(0)*
  - *don't abuse it, it can hurt performance!*

GPU is **really fast** at **compositing**, **matrix operations** and **alpha blends.**

Hardware Acceleration 101

- Minimize CPU-GPU interactions
- Texture **uploads are not free**
  - **No upload:** position, size, opacity
  - **Texture upload:** everything else

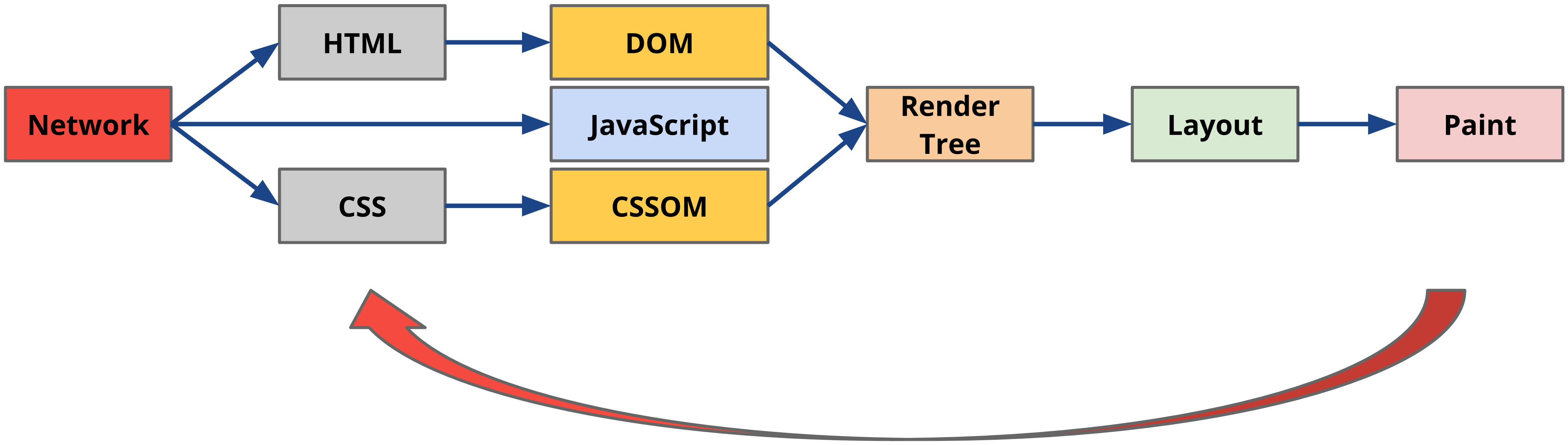
CSS3 Animations with no Javascript!

CSS3 Animations are as close to "free lunch" as you can get **

```
<style>
  .spin:hover {
    -webkit-animation: spin 2s infinite linear;
  }

  @-webkit-keyframes spin {
    0% { -webkit-transform: rotate(0deg);}
    100% { -webkit-transform: rotate(360deg);}
  }
</style>

<div class="spin" style="background-image: url(images/chrome-logo.png);"></div>
```

- Look ma, no JavaScript!
- Example: poster circle.

*** Assuming no texture reuploads and animation runs entirely on GPU...*



Done? Repeat it all over... at 60 FPS! :-)

Let's wrap it up...

I heard you like top {N} lists...

Optimize your networking stack!

- **Reduce DNS lookups**
    - **130 ms** average lookup time! And much slower on mobile..
- **Avoid redirects**
    - Often results in **new handshake** (and maybe even DNS)
- **Make fewer HTTP requests**
    - No request is faster than no request
- **Account for network latency overhead**
    - Breaking the 1000 ms mobile barrier requires careful engineering
- **Use a CDN**
    - Faster RTT = faster page loads
    - Also, terminate SSL closer to the user!

Reduce the size of your pages!

- GZIP your (text) assets
    - ~80% compression ratio for text
- Optimize images, pick optimal format
    - ~60% of total size of an average page!
- Add an Expires header
    - No request is faster than no request
- Add ETags
    - Conditional checks to avoid fetching **duplicate content**

Optimize the critical rendering path!

- **Stream the HTML to the client**
  - Allows the document parser to discover resources early
- **Place stylesheets at the top**
  - Rendered, and potentially DOM construction, is blocked on CSS!
- **Load scripts asynchronously, whenever possible**
  - Eliminate JavaScript from the critical rendering path
- **Inline / push critical CSS and JavaScript**
  - Eliminate extra network roundtrips from critical rendering path

Eliminate jank and memory leaks!

- **Performance == 60 FPS**
  - 16.6 ms budget per frame
  - Shared budget for your code, GC, layout, and painting
  - Use frames view to hunt down and eliminate jank
- **Profile and optimize your code**
  - Profile your JavaScript code
  - Profile the cost of layout and rendering!
  - Minimize CPU > GPU interaction
- **Eliminate JS and DOM memory leaks**
  - Monitor and diff heap usage to identify memory leaks
- **Test on mobile devices**
  - Emulators won't show you true performance on the device

Performance is a discipline.

Yes, this stuff is hard... let's not pretend otherwise.

zomg, we made it.

Feedback & Slides @ **bit.ly/fluent-perfshop**

Twitter @igrigorik

G+ gplus.to/igrigorik

Web igvita.com