Building Faster Mobile Websites

de the nuts and bolts of hitting the 1000 millisecond "time to glass" target ...

+Ilya Grigorik @igrigorik
Make The Web Faster, Google

Video of the talk: http://bit.ly/12GFKDE
What's the impact of slow sites?

Lower conversions and engagement, higher bounce rates...
The cost of delay increases over time and persists.
Delays under half a second impact business metrics.
"Speed matters" is not just lip service.
### Server Delays Experiment

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

- Means no statistically significant change

- Strong negative impacts
- Roughly linear changes with increasing delay
- Time to Click changed by roughly double the delay
How speed affects bounce rate

\[ y = 0.6517x + 33.682 \]

\[ R^2 = 0.91103 \]

Every second = 0.65 increase in bounce rate
So, how are we doing today?

Okay, I get it, speed matters... but, are we there yet?
## Usability Engineering 101

<table>
<thead>
<tr>
<th>Delay</th>
<th>User reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 100 ms</td>
<td>Instant</td>
</tr>
<tr>
<td>100 - 300 ms</td>
<td><em>Feels sluggish</em></td>
</tr>
<tr>
<td>300 - 1000 ms</td>
<td>Machine is working...</td>
</tr>
<tr>
<td>1 s+</td>
<td>Mental context switch</td>
</tr>
<tr>
<td>10 s+</td>
<td>I'll come back later...</td>
</tr>
</tbody>
</table>

*Stay under 250 ms to feel "fast".*

*Stay under 1000 ms to keep users attention.*
How Fast Are Websites Around The World? - Google Analytics Blog

**Desktop**
- Median: ~2.7s
- Mean: ~6.9s

**Mobile** *
- Median: ~4.8s
- Mean: ~10.2s

* optimistic
### Content Type

<table>
<thead>
<tr>
<th>Content Type</th>
<th>Avg # of Requests</th>
<th>Avg size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>6</td>
<td>39 kB</td>
</tr>
<tr>
<td>Images</td>
<td>39</td>
<td>490 kB</td>
</tr>
<tr>
<td>Javascript</td>
<td>10</td>
<td>142 kB</td>
</tr>
<tr>
<td>CSS</td>
<td>3</td>
<td>27 kB</td>
</tr>
</tbody>
</table>

**Total Transfer Size & Total Requests**

- **Total Transfer Size (kB)**
- **Total Requests**

*HTTP Archive - Mobile Trends (Feb, 2013)*
For many, mobile is the one and only internet device!

<table>
<thead>
<tr>
<th>Country</th>
<th>Mobile-only users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Egypt</td>
<td>70%</td>
</tr>
<tr>
<td>India</td>
<td>59%</td>
</tr>
<tr>
<td>South Africa</td>
<td>57%</td>
</tr>
<tr>
<td>Indonesia</td>
<td>44%</td>
</tr>
<tr>
<td>United States</td>
<td>25%</td>
</tr>
</tbody>
</table>

*onDevice Research*
The network will save us!

1000 ms is plenty of time.. 4G will fix everything! Right, right?

* Nope.
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.
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."

<table>
<thead>
<tr>
<th></th>
<th>3G</th>
<th>4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint</td>
<td>400 ms</td>
<td>150 ms</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>150 - 400 ms</td>
<td>100 - 200 ms</td>
</tr>
</tbody>
</table>
Mobile design constraint: **Battery life**

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

<table>
<thead>
<tr>
<th></th>
<th>LTE</th>
<th>HSPA+</th>
<th>3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle to connected latency</td>
<td>&lt; 100 ms</td>
<td>&lt; 100 ms</td>
<td>&lt; 2.5 s</td>
</tr>
<tr>
<td>user-plane one-way latency</td>
<td>&lt; 5 ms</td>
<td>&lt; 10 ms</td>
<td>&lt; 50 ms</td>
</tr>
</tbody>
</table>

same process happens for incoming data, just reverse steps 1 and 2
LTE power state transitions (AT&T)

- **Idle to Active**: 260 ms control-plane latency
- **Dormant to Active**: <50 ms control-plane latency (spec)
- **Timeout driven** state transitions back to idle
  - 100 ms, 100 ms, 10 s > Idle
- Similar state machine for 3G devices
  - Except CP latencies are much higher

https://github.com/attdevsupport/ARO/blob/master/ARODataAnalyzer/src/lte.conf
3G power state transitions (AT&T)

- Radio cycles between 3 states
  - **Idle**
  - Low TX power (**FACH**)
  - High TX power (**DCH**)

![Diagram of 3G power state transitions](image)
I just wanted to make a fast mobile app.....
1. **Latency variability can be very high on mobile networks**

2. **4G networks will improve latency, but...**
   a. We still have a long way to go until everyone is on 4G - a decade!
   b. And 3G is definitely not going away anytime soon
   c. Ergo, latency and variability in latency is a problem

3. **What can we do about it?**
   a. Re-use connections
   b. Download resources in bulk, avoid waking up the radio
   c. Compress resources
   d. Cache
How do we render the page?

we're getting bytes off the wire... and then what?
Life of a web-page in the browser...

1. Fetch resources from the network
2. Parse, tokenize, construct the DOM
   a. Run scripts...
3. Output to the screen

Diagram:
- Network
  - Resource Loader
    - HTML Parser
      - CSS
      - DOM
      - Script
        - Render Tree
          - Graphics Context
The HTML5 parser at work...

Bytes

Characters

Tokens

Nodes

DOM

3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E

Tokenizer

<body>Hello, <span>world!</span></body>

StartTag: body

Hello,

StartTag: span

world!

EndTag: span

TreeBuilder

DOM is constructed incrementally, as the bytes arrive on the "wire".
Deciphering the Critical Rendering Path

- HTML > Document Object Model - incremental parsing
- CSS > CSS Object Model
- Rendering is blocked on CSSOM and DOM
The HTML5 parser at work...

HTMLDocumentParser begins parsing the received data ...

Stop. Dispatch request for application.js. Wait...
Scripts can block the document parser...

```javascript
document.write("<script>");
...

Tokenizer  DOM TreeBuilder

JavaScript can **block** DOM construction.

Script execution can change the input stream.
Hence we **must wait for script to execute**.
Sync scripts block the parser...

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

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

Async script **will not block** the 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>
```
(2) Javascript can query CSS, which means...

JavaScript can **block on CSS**.

DOM construction can be blocked on Javascript, which can be blocked on CSS

- *ex: asking for computed style, but stylesheet is not yet ready...*

*At least CSS can't query javascript.. phew!*
(3) Rendering is blocked on CSS...

CSS must be fetched & parsed before Render tree can be painted.

Otherwise, the user will see "flash of unstyled content" + reflow and repaint when CSS is ready.

At least CSS can't query javascript.. phew!
Performance rules to keep in mind...

(1) JavaScript can block the DOM construction
(2) JavaScript can block on CSS
(3) Rendering is blocked on CSS...

Which means...

(1) Get CSS down to the client as fast as you can
   ○ Unblocks paints, removes potential JS waiting on CSS scenario
(2) If you can, use async scripts + avoid doc.write at all costs
   ○ Faster DOM construction, faster DCL and paint!
   ○ Do you need scripts in your critical rendering path?
Let's put it all together now

network, browser rendering pipeline, and the rest...
Navigation Timing (W3C)

Graphical representation of navigation timing events.

- navigationStart
- redirectStart
- redirectEnd
- fetchStart
- domainLookupStart
- domainLookupEnd
- connectStart
- (secureConnectionStart)
- connectEnd
- requestStart
- responseStart
- responseEnd
- unloadEnd
- loadEventStart
- loadEventEnd
- domLoading
- domLoadingComplete
- domInteractive
- domContentLoaded
Navigation Timing (W3C)

user's connectivity

server

browser execution

- navigationStart
- redirectStart
- redirectEnd
- fetchStart
- domainLookupStart
- domainLookupEnd
- connectStart
- (secureConnectionStart)
- connectEnd
- requestStart
- responseStart
- responseEnd
- unloadEnd
- unloadStart
- loadEventEnd
- loadEventStart
- domComplete
- domContentLoaded
- domInteractive
- domLoading
W3C Navigation Timing

If we want to see the end-user perspective, then we need to instrument the browser to give us this information. Thankfully, the W3C Web Performance Working Group is ahead of us: Navigation Timing. The spec is still a draft, but Chrome, Firefox and IE have already implemented the proposal.

Available in...

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

caniuse.com/nav-timing

@igrigorik
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
The (short) life of our 1000 ms budget

<table>
<thead>
<tr>
<th></th>
<th>3G</th>
<th>4G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(200 ms RTT)</td>
<td>(80 ms RTT)</td>
</tr>
<tr>
<td>Control plane</td>
<td>(200-2500 ms)</td>
<td>(50-100 ms)</td>
</tr>
<tr>
<td>DNS lookup</td>
<td>200 ms</td>
<td>80 ms</td>
</tr>
<tr>
<td>TCP Connection</td>
<td>200 ms</td>
<td>80 ms</td>
</tr>
<tr>
<td>TLS handshake</td>
<td>(200-400 ms)</td>
<td>(80-160 ms)</td>
</tr>
<tr>
<td>HTTP request</td>
<td>200 ms</td>
<td>80 ms</td>
</tr>
<tr>
<td>Leftover budget</td>
<td>0-400 ms</td>
<td>500-760 ms</td>
</tr>
</tbody>
</table>

Network overhead of one HTTP request!
Our mobile apps and pages are not single HTTP requests... are they?

But, perhaps they {could, should} be?
<table>
<thead>
<tr>
<th>Leftover budget</th>
<th>3G (200 ms RTT)</th>
<th>4G (80 ms RTT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-400 ms</td>
<td>500-760 ms</td>
<td></td>
</tr>
</tbody>
</table>

~400 ms of budget left for...

- **Server processing time**
  - what is your server processing time?

- **Client-rendering**
  - what does it take to render a page?

Reserve 100 ms for layout, rendering

200 ms

JavaScript execution and an extra request if we're lucky!
Breaking the **1000 ms** time to glass mobile barrier... **hard facts:**

1. **Majority of time is in network overhead**
   - *Leftover budget is ~400 ms on average*

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**
A simple example in action...

network, browser rendering pipeline, and the rest...
Here is my content.

Perhaps there is a left nav bar here.

1. Split **all.css**, inline AFT styles
2. Do you need the JS at all?
   - Progressive enhancement
   - Inline AFT JS code
   - Defer the rest
<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
Critical Path Explorer extracts the subtree of the waterfall that is in the "critical path" of the document parser and the renderer.

(automation for the win!)
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!

Loading of ads

This was added to the DOM using document.write()
[native code]:0
http://pagead2.googlesyndication.com/pagead/js/201211
http://pagead2.googlesyndication.com/pagead/js/201211
http://pagead2.googlesyndication.com/pagead/js/201211
http://www.guardiannews.com/1
Fetched after event load
300 ms redirect!

JS execution blocked on CSS

doc.write() some JavaScript - doh!

long-running JS
One request. Inline. Defer the rest.

It's not as crazy, or as hard as it sounds: *investigate your critical rendering path.*
Thanks! Questions?

- 1000 ms total budget
  - 600 ms in network overhead
  - 400 ms for server processing and browser rendering
    - aim for <100 ms server response
    - reserve 100 ms for browser rendering

- To beat 1000 ms time to glass barrier
  - Inline critical CSS (no room for other requests)
  - Eliminate JavaScript from critical rendering path

Slides @ bit.ly/mobile-barrier
Video @ bit.ly/12GFKDE

+Ilya Grigorik - igrigorik@google.com - @igrigorik

bit.ly/browser-networking