Is the Web HTTP/2 yet?

Jeremy Blackburn and Matteo Varvello

K. Schomp, D. Naylor, A. Finamore, K. Papagiannaki
Outline

• HTTP History
• HTTP/2 Background
• What, why and how?
• HTTP/2 Adoption
• HTTP/2 Content Analysis
• HTTP/2 Performance Analysis
History

- Hypertext, HTTP, World Wide Web
- HTTP functions as a request–response protocol in the client–server computing model
History

• TCP connection on port 80
• ASCII protocol
  – Simply telnet (1969) for debugging
• Only one method: GET
  – GET an HTML page from a server
• RFC1945 describes the “common usage’” of HTTP/1.0, not a formal standard
• Introduce POST and HEAD
• Content-Encoding header
  – end-to-end content-coding(s) used for a message
• Simple caching
  – server may mark a response using the Expires header
  – cache can include If-Modified-Since header to check cache validity
History

• IETF Draft Standard
• Add methods: OPTIONS, PUT, DELETE, TRACE and CONNECT
  – HTTPS through HTTP proxy
• Pipelining
  – Multiple request over a single (TCP) connection
• New header options
  – Range requests
  – Hop-by-hop headers
• Clarify and add flexibility to caching operations
• Upgrade request header
  – Ease deployment of future protocols
History

Breakdown by Protocol Version (Announced Support)

Click and drag in the plot area to zoom in

Number of Domains

- **H2 Draft 14**

<table>
<thead>
<tr>
<th>Year</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1965</td>
<td>HTTP/0.9</td>
</tr>
<tr>
<td>1991</td>
<td>HTTP/1.0</td>
</tr>
<tr>
<td>1996</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>1999</td>
<td>HTTPS</td>
</tr>
<tr>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>2014-2015</td>
<td>H2-14</td>
</tr>
</tbody>
</table>

Highcharts.com
History

Breakdown by Protocol Version (Announced Support)

Click and drag in the plot area to zoom in

Number of Domains

0k 10k 20k 30k 40k 50k 60k 70k 80k

Jan '15 May '15 Sep '15 Jan '16

H2 Draft 14  H2 Draft 15  H2 Draft 16  H2 Draft 17

HTTP/0.9  HTTP/1.0  HTTP/1.1  HTTPS

History

Breakdown by Protocol Version (Announced Support)
Click and drag in the plot area to zoom in

Number of Domains

HTTP/0.9
HTTP/1.0
HTTP/1.1
HTTPS
H2 Draft 14
H2 Draft 15
H2 Draft 16
H2 Draft 17
H2

1965
1991
1996
1999
2000
2014-2015
2015
Outline

• HTTP History
• HTTP/2 Background
• What, why and how?
• HTTP/2 Adoption
• HTTP/2 Content Analysis
• HTTP/2 Performance Analysis
Binary Protocol

- H2 is binary rather than ASCII
- Pros:
  - efficient parsing
  - lighter network footprint
  - less prone to security issues due to unsanitized input strings
Header Compression

• HTTP is state-less, thus repetitive
  – Each object needs to carry header information
  – HTTP/2 is state-less as well

• HTTP/2 reduces redundant information shared between objects using compression
  – HPACK7: solve BREACH and CRIME vulnerabilities with SPDY (gzip)
Multiplexing

• Head of line blocking – Any early request for a large object can delay all subsequent pipelined requests
  – Clients open several TCP connections in parallel (6 in Chrome and 15 in Firefox)
  – Overhead of multiple TCP connections (TCP state, TLS handshake and setup)

• H2 opens a single TCP connection to a domain where it *multiplexes* requests and responses (streams)
Priority

• A client can specify which streams are most important
• It allows the client to load important objects (e.g., CSS and JavaScript) earlier
Push

- It allows the server to push objects before the client requests them
  - Server knows what the client will ask in the future
HTTP/2 Background – Summary

• *Binary protocol* – efficient parsing, light network footprint, less prune to security issues due to unsanitized input strings

• *Header compression* – reduces redundant information shared between objects

• *Multiplexing* – opens a single TCP connection to a domain where it *multiplexes* requests and responses

• *Priority* – allows the client to load important objects (e.g., CSS and JavaScript) earlier

• *Server push* – allows the server to push objects before the client requests them
Outline

• History
• HTTP/2 Background
• What, Why and How?
• HTTP/2 Adoption
• HTTP/2 Content Analysis
• HTTP/2 Performance Analysis
What?

• Monitored HTTP/2 (H2) adoption
  – Alexa top 1 million* websites on a daily basis
• Content and performance analysis
  – Subset of website “truly” supporting H2
• Built a website
  – http://isthewebhttp2yet.com/
Why?

• Who is behind H2?
  – Unique chance to understand how a new protocol is adopted in the modern Web

• Is H2 bringing any performance benefit?

• Did web content change with H2 adoption?
  – Are H1 hack (e.g., inlining and domain sharding) still used?

• Is there any interest in H2-in-the-clear (H2C)?
How?

- Alexa Top 1 Million
- Sites that **Announce** HTTP/2 Support
- Sites that **Actually** Support HTTP/2
- Usage & Performance Statistics
Phase 1

• Challenges are scale and speed
  – Alexa’s top 1M, daily

• We build prober
  – Bash tool for ALPN/NPN negotiations
  – Run on PlanetLab instrumented by a “tracker”
Phase 2

• Small(er) scale (for now)
  – Fraction of sites announcing H2 support
• We build **h2-lite**
  – a tiny H2 client that attempts to download the root object of a website using H2
  – H2 library implemented in Node.js
• Run from Telefonica (Spain) and CMU (USA)
Phase 3

• Small scale (for now)
  – Fraction of sites with certified H2 support
• Require full machine control: result reproducibility, 3/4g dongle, etc.
• We build **chrome-loader**
  – Loads pages using Chrome and H1/H2 alternatively for N repetitions
  – HAR retrieval via Chrome developer interface
  – Error/crash management
• It runs from Telefonica (Barcelona, Spain), Case (Cleveland, USA) and CMU (Pittsburgh, USA)
Limitations

• **Location** – H2 adoption can be impacted by location, e.g., Facebook
  – Phase 1 tasks are randomly assigned by location (for scalability)

• **Limited vantage points** – Phase-3 only runs from a handful of locations

• **Scalability** – already facing this issue with phase 3
  – Increase vantage points
  – Rethink/optimize (see lazy strategy in the paper)

• **PLT == onLoad()** – though this metric is well-known and largely used, it is questionable in the modern web
Outline

• History
• HTTP/2 Background
• What, Why and How?
• HTTP/2 Adoption
• HTTP/2 Content Analysis
• HTTP/2 Performance Analysis
Adoption

HTTP/2 Support
In the top 1 million Alexa sites

- Announced Support: 186,310 sites announce support for HTTP/2.
- Partial Support: 117,898 sites partially support HTTP/2.
- True Support: 81,247 sites truly support HTTP/2.
Adoption

http://isthewebhttp2yet.com/measurements/adooption.html
Outline

• History
• HTTP/2 Background
• What, Why and How?
• HTTP/2 Adoption
• HTTP/2 Content Analysis
• HTTP/2 Performance Analysis
Content Analysis – Marcoscopic
Content Analysis -- Marcoscopic

No significant change over time
Content Analysis – Marcoscopic

No significant change over time

More complex websites adopt H2 over time
Content Analysis – Microscopic

50th: 18 dom
Content Analysis – Microscopic

50th: 3 obj
Outline

• History
• HTTP/2 Background
• What, Why and How?
• HTTP/2 Adoption
• HTTP/2 Content Analysis
• HTTP/2 Performance Analysis
TCP Analysis

On average, H2 requires half as many TCP connections as H1.
TCP Analysis

On average, H2 requires half as many TCP connections as H1.

Few sites can be served via a single TCP connection.
TCP Analysis

- On average, H2 requires half as many TCP connections as H1.
- As high as 100 TCP connections.
- Few sites can be served via a single TCP connection.

Graph showing the cumulative distribution function (CDF) of the average number of TCP connections. The colors indicate different thresholds: $th \geq 0.0$, $th \geq 0.4$, and HTTP/1.
20-40% of sites with \( \geq 80\% \) of H2 objects are served via a single TCP connection.
Performance Analysis

• No standardized definition of “page load time” (PLT)
  – the time from when the user enters a URL in the browser to when the page is displayed
• We approximate PLT as the time from when Chrome starts fetching the page to the firing of the JavaScript “onLoad” event
  – occurs once the page’s embedded resources have been downloaded, but possibly before all objects loaded via scripts are downloaded
Performance Analysis

The diagram shows the cumulative distribution function (CDF) of H2 speedup in milliseconds (ms) for different locations:
- Pittsburgh – fiber
- Cleveland – fiber
- Barcelona – fiber
- Barcelona – 4G
- Barcelona – 3G

The x-axis represents the H2 speedup in ms, ranging from -3000 to 3000 ms, while the y-axis represents the CDF ranging from 0 to 1.
Performance Analysis
Performance Analysis

25% of the websites “suffer” due to h2
Performance Analysis

25% of the websites “suffer” due to h2

More benefits on more challenging network access (3G)
Performance Analysis – Controlled

More benefits as loss rate increases and available bdw decreases
The Web is Poorly Defined

- Many challenges to measuring performance
- Everyone measures Page Load Time (PLT)

- PLT?!?!?!!?
  - “onload” event in browser

- “onload” event?!?!?!
  - “The onload event occurs when an object has been loaded.” – W3Schools
Web Browser Events – Quick Info

• A web page is built from a “DOM” → “Document Object Model”
  – Root Object → The web page
    • <html>…</html>
  – Child objects → The “content”
    • <body><div>Hello World!</div></body>

• Browser fires and receives events from/to different nodes in the DOM
“onload”

• “The load event fires at the end of the document loading process. At this point, all of the objects in the document are in the DOM, and all the images, scripts, links and sub-frames have finished loading.” – Mozilla

• OK?!?!?!?
The Problem...

• onload wrt PLT is only for the $root\ object$
• Many websites these days do things after the onload event has fired...

• WHEN IS A PAGE ACTUALLY LOADED?!?!?!
Pathological Example...

```html
<html>
  <head>
    <script type="text/javascript">
      function doMoreStuff() {
        var img = document.createElement("img");
        img.src = "http://i.imgur.com/S2HfRQz.jpg";
        var divElement = document.getElementById("stuff");
        divElement.appendChild(img);
      }
      function doStuff() {
        var divElement = document.getElementById("stuff");
        var spanElement = document.createElement("span");
        spanElement.textContent = "onload fired; waiting 3 seconds to load image!";
        divElement.appendChild(spanElement);
        setTimeout(doMoreStuff, 3000);
      }
    </script>
  </head>
  <body onload="doStuff();">
    <div id="stuff" name="stuff"></div>
  </body>
</html>
```
Houston, We Have A Problem...

• Yes, the onload event indicates that “all resources are loaded”

• No, the onload event does not mean that a page is “done” as far as humans are concerned!
SpeedIndex for PLT

• Speed Index
  – Uses “visual completeness”
  – Makes a video of the page and checks frame-by-frame
  – Calculates area under the “visually completed” curve

\[
SpeedIndex = \int_{0}^{\text{end}} 1 - \frac{VC}{100}
\]

  – end \(\rightarrow\) end time in ms
  – VC = % visually complete

• Not really used in literature that we can tell..
Still... What About HUMANS

• Human perception is tricky...
  – We can only see at around 30 frames per second
  – But, that doesn’t mean we can’t tell when something is slower!

• Example: Speed up a page by 34%
  – 2 seconds to load → 1.28 seconds to load
  – Does it actually matter?! Is it perceivably faster?!
eyeorg.net

• Our work in progress to answer how humans perceive web page load time

• Basic idea: we crowdsource human eyeballs and have them measure PLT in a variety of ways

• Start off with: “Which is faster?”
What Do We Need To Build?

1. Crawler
2. HAR Capturer
3. Video Encoder
4. Frontend
5. Backend
Crawler

- Similar to the HTTP/2 adoption crawler
- Set of bash scripts that triggers crawls
- Controls all other operations
HAR Capturer

• HTTP Archive format
• www.softwareishard.com/blog/har-12-spec/
• *From browser’s perspective*
• JSON
• Lots of info!
  – Timing
  – Connections
  – Protocols
  – Etc.
How to Capture a HAR?

• Not that simple...

• Chrome allows you to export a HAR
  – Via developer tools
  – Can’t really automate though! Requires clicking...

• Chrome Remote Debugging Protocol!
  – Allows us to hook into Chrome and capture various events
Chrome Remote Debugging Protocol

• Uses *WebSockets* to push events to a listener
  – Allows full-duplex comms over an HTTP TCP conn
  – *NOT* HTTP → Handshake treated as an upgrade

1. Launch chrome with special flag
   – “--remote-debugging-port=9222”

2. Connect to Chrome

3. Listen for events
   – All different types of events...
Remote Debugging Protocol Messages

• JSON

• Lots of info

• Many message types...
What Events Do We Care About?

- **Page.domContentEventFired**
  - The “DOMContentLoaded” event
    - Even less well specified than “onload”
- **Page.loadEventFired**
  - “onload”
- **Network.requestWillBeSent**
  - Browser will make a network request
- **Network.dataReceived**
  - The browser got some data from the network
- **Network.responseReceived**
  - A full HTTP response is available
- **Network.loadingFinished**
  - HTTP request has finished loading (whatever that means)
- **Network.loadingFailed**
  - A network request failed to load
Our HAR Capturer (bkPLTHAR)

• Ruby
  – About 550 LOC

• Uses EventMachine
  – Gives us an event loop

• Uses Faye WebSocket library
  – Let’s us talk to Chrome
module BKPLTHar

LOGGER = Logger.new STDERR

# we need to ensure that we get all the responses
# and some stuff might come to us after onload event is received
PAGE_FINISHED_DELAY = 1.0

CLEAN_UP_SCRIPT = 'chrome.benchmarking.clearCache();' +
  'chrome.benchmarking.clearHostResolverCache();' +
  'chrome.benchmarking.clearPredictorCache();' +
  'chrome.benchmarking.clearSupplemental();'

# check to see if the chrome debugging port is available
def self.debug_port_listening?(host: "localhost", port: 9222)
  begin
    TCPSocket.new(host, port).close
    true
  rescue Errno::ECONNREFUSED
    false
  end
end

def self.start_capture(debug_host: null, out: nil, debug_port: 9222, url_to_load: nil, post_on_load_delay: 0)
  raise "MUST PROVIDE A VALID URL TO NAVIGATE TO" if url_to_load.nil?
  # TODO Add a timeout to this
  until debug_port_listening?
    LOGGER.debug "Waiting for Chrome to become available..."
  sleep 0.1
  end
  LOGGER.debug "Chrome is available."
  har = BKPLTHar::Har::new
  har.log.pages << BKPLTHar::Har::Page.new
  har.log.pages.first.timeout = post_on_load_delay
  har.run do
    debug_url = "#{debug_host}:#{debug_port}/json/new"
    conn = EM::HttpClient.new(debug_url).get
    conn.callback do
      resp = JSON.parse(conn.response)
      ws_url = resp["webSocketDebuggerUrl"]
      ws_tab_url = resp["url"]
      ws = Paye::WebSocket::Client.new(ws_url)
      ws.onopen = lambda do [event]
        ws.send JSON.dump(id: 1, method: "Page.enable")
        ws.send JSON.dump(id: 2, method: "Network.enable")
        ws.send JSON.dump(id: 3, method: "Network.setCacheddisabled", params:
          {cacheDisabled: true})
        ws.send JSON.dump(id: 4, method: "Runtime.evaluate", params:
          {expression: CLEAN_UP_SCRIPT })
        ws.send JSON.dump(id: 5, method: "Page.navigate", params: {
          url: url_to_load
        })
      end
    end
    # Listen for events...
    ws.onmessage = lambda do [event]
      event_data = JSON.parse(event_data)
      har.log.pages.first.process_message event_data
      # Are we done loading?
      if har.log.pages.first.loaded?
        EventMachine.add_timer(post_on_load_delay + PAGE_FINISHED_DELAY) do |
          f
            f.write JSON.pretty_generate(har.to_h)
          end
      end
      # Stop event and exit
      EM.stop
      File.open(out, "w") { |f|
        f.puts "OnLoad: #{har.log.pages.first.page_timings.on_load}"
      }
      exit
    end
  end # EM.run
end

# # def start_capture
module BKPLTHar

LOGGER = Logger.new STDERR

# we need to ensure that we get all the responses
# and some stuff might come to us after onload event is received
PAGE_FINISHED_DELAY = 1.0

CLEAN_UP_SCRIPT = 'chrome.benchmarking.clearCache();' +
  'chrome.benchmarking.clearHostResolverCache();' +
  'chrome.benchmarking.clearPredictorCache();' +
  'chrome.benchmarking.closeConnections();'

# Check to see if the chrome debugging port is available
def self.debug_port_listening?(host: "localhost", port: 9222)
begins
  TCPSocket.new(host, port).close
true
rescue Errno::ECONNREFUSED
false
ends

def self.start_capture(debug_host: "http://localhost", out: nil
deploy_host: 9222, url_to_load: nil, post_on_load_delay: 0)
raise "MUST PROVIDE A VALID URL TO NAVIGATE TO" if url_to_load.nil?
# wait until chrome becomes available
# TODO Add a timeout to this
while debug_port_listening?
  LOGGER.debug "Waiting for Chrome to become available...
  sleep 0.1
end

LOGGER.debug "Chrome is available."
har = BKPLTHar::Har::new
har.log.pages << BKPLTHar::Har::Page.new
har.log.pages.first.timeout = post_on_load_delay

EM.run do
  debug_url = "#{debug_host}:#{debug_port}/json/new"
  conn = EM::HttpRequest.new(debug_url).get
  conn.callback do
    resp = JSON.parse(conn.response)
    ws_tab_url = resp["url"]
  end
  ws = Payr::WebSocket::Client.new(ws_url)
  ws.onopen = lambda do [event]
    setup our debugging environment
    ws.send JSON.dump(id: 1, method: "Page.enable")
    ws.send JSON.dump(id: 2, method: "Network.enable")
    ws.send JSON.dump(id: 3, method: "Network.setCacheddisabled", params: (cache: disabled: true))
    ws.send JSON.dump(id: 4, method: "Runtime.evaluate", params: (expression: CLEAN_UP_SCRIPT))
  end

  # navigate to the page we want to page
  ws.send JSON.dump(
    id: 5,
    method: "Page.navigate",
    params: {
      url: url_to_load
    }
  )

  # Listen for events...
  ws.onmessage = lambda do [event]
    event_data = JSON.parse(event_data)
    # Pass the event on to the event processor
    har.log.pages.first.process_message event_data

    # Are we done loading?
    if har.log.pages.first.loaded?
      EventMachine.add_timer(post_on_load_delay + PAGE_FINISHED_DELAY) do |f|
        f.write JSON.pretty_generate(har.to_h)
      end
    # Stop event and exit
    EM.stop
    puts "OnLoad: #{har.log.pages.first.page_timings.on_load}"
    exit
  end
end # EM.run

end # def start capture
def self.debug_port_listening?(host: "localhost", port: 9222)
begin
  TCPSocket.new(host, port).close
  true
rescue Errno::ECONNREFUSED
  false
end
end

raise "MUST PROVIDE A VALID URL TO NAVIGATE TO" if url_to_load.nil?
# wait until chrome becomes available
# TODO Add a timeout to this
until debug_port_listening?
  LOGGER.debug "Waiting for Chrome to become available..."
  sleep 0.1
end

LOGGER.debug "Chrome is available."
har = BKPLTHar::Har::Har.new
har.log.pages << BKPLTHar::Har::Page.new
har.log.pages.first.timeout = post_on_load_delay
EM.run do
  debug_url = "#{debug_host}:#{debug_port}/json/new"
  conn = EM::HttpRequest.new(debug_url).get
  conn.callback do
    resp = JSON.parse(conn.response)
    ws_url = resp["webSocketDebuggerUrl"]
    ws_tab_url = resp["url"]
    ws = Paye::WebSocket::Client.new(ws_url)
    ws.onopen = lambda do |event|
      # Setup up our debugging environment
      ws.send JSON.dump(id: 1, method: "Page.enable")
      ws.send JSON.dump(id: 2, method: "Network.enable")
      ws.send JSON.dump(id: 3, method: "Network.setCachedDisabled", params:
        {cacheDisabled: true})
      ws.send JSON.dump(id: 4, method: "Runtime.evaluate", params:
        {expression: CLEAN_UP_SCRIPT})
    end
    # navigate to the page we want to capture info
    ws.send JSON.dump(id: 5,
      method: "Page.navigate",
      params: {
        url: url_to_load
      })
  end
  # Listen for events...
  ws.onmessage = lambda do |event|
    event_data = JSON.parse(event.data)
    # Pass the event on to the event processor
    har.log.pages.first.process_message event_data
    # Are we done loading?
    if har.log.pages.first.loaded?
      EventMachine.add_timer(post_on_load_delay + PAGE_FINISHED_DELAY) do
        f = File.open(out, "w")
        f.write JSON.pretty_generate(har.to_h)
        f.close
      end
    end
    # Stop event and exit
    EM.stop
    puts "OnLoad: #{har.log.pages.first.page_timings.on_load}"
    exit
  end
end
end # def start_capture
module BKPLTHar

LOGGER = Logger.new STDERR

# we need to ensure that we get all the responses
# and some stuff might come to us after onload event is received
PAGE_FINISHED_DELAY = 1.0

CLEAN_UP_SCRIPT = 'chrome.benchmarking.clearCache();
' +
'chrome.benchmarking.clearHostResolverCache();
' +
'chrome.benchmarking.clearPredictorCache();
' +
'chrome.benchmarking.closeConnections();

# Check to see if the chrome debugging port is available
def self.debug_port_listening?(host: "localhost", port: 9222)
begin
  TCPSocket.new(host, port).close
  true
rescue Errno::ECONNREFUSED
  false
end
end

def self.start_capture(debug_host: "http://localhost", out: nil
  debug_port = 9222, url_to_load: nil, post_on_load_delay: 0)
raise "MUST PROVIDE A VALID URL TO NAVIGATE TO" if url_to_load.nil?

# wait until chrome becomes available
# TODO Add a timeout to this
until debug_port_listening?
  LOGGER.debug "Waiting for Chrome to become available...
  sleep 0.1
end

LOGGER.debug "Chrome is available."
har = BKPLTHar::Har::new
har.log.pages << BKPLTHar::Har::Page.new
har.log.pages.first.timeout = post_on_load_delay
EM.run do
  debug_url = "#{debug_host}:#{debug_port}/json/new"
  conn = EM::HttpRequest.new(debug_url).get
  conn.callback do
    resp = JSON.parse(conn.response)
    ws_url = resp["WebSocketDebuggerUrl"]
    ws_tab_url = resp["url"]
    ws = Paye::WebSocket::Client.new(ws_url)
    ws.onopen = lambda do [event|
      # Set up our debugging environment
      ws.send JSON.dump({id: 1, method: "Page.enable"})
      ws.send JSON.dump({id: 2, method: "Network.enable"})
      ws.send JSON.dump({id: 3, method: "Network.setCached_disabled", params: {cacheDisabled: true}})
      ws.send JSON.dump({id: 4, method: "Runtime.evaluate", params: {expression: CLEAN_UP_SCRIPT}})
    
    # navigate to the page we want to capture for
    ws.send JSON.dump(
      id: 5,
      method: "Page.navigate",
      params: {url: url_to_load}
    )
  end

  listen for events...
  ws.onmessage = lambda do [event|
    event_data = JSON.parse(event.data)
    # Pass the event on to the event processor
    har.log.pages.first.process_message event_data
  end

  # Are we done loading?
  if har.log.pages.first.loaded?
    EventMachine.add_timer(post_on_load_delay + PAGE_FINISHED_DELAY) do |
      File.open(out, "w") do |f|
        f.write JSON.pretty_generate(har.to_h)
      end
    end
    # Stop event and exit
    EM.stop
    puts "OnLoad: #{har.log.pages.first.page_timings.on_load}"
    exit
  end
end
end
Capturing Start Of Network Load

```ruby
when "Network.requestWillBeSent"
  # if the page has already loaded (i.e., onload event fired) we don't
  # want to start tracking new requests
  return if self.loaded? && !(timeout > 0)

  # the first request id we get here is the actual url that we are
  # supposed to be visiting.
  # subsequent ids will be due to embedded resources
  if (@main_request_id.nil? && msg_params["initiator"]["type"] == "other")
    LOGGER.debug "GOT MAIN REQUEST STUFF"
    @main_request_id = request_id
    @id = @main_request_id
    @main_request_started_at = msg_params["timestamp"] * 1_000
    @started_date_time = Time.at(msg_params["timestamp"]).utc.iso8601(3)
    @title = msg_params["request"]["url"]
    @main_request_wall_time_started_at = msg_params["wallTime"]
  end

  # we are loading some embedded object here...
  entry = Entry.new
  entry.page_ref = @main_request_id
  entry.request_message = msg
  entry.response_message = nil
  entry.response_length = 0
  entry.response_finished_time = nil
  entry.response_body = nil
  entry.encoded_response_length = 0
  entry.response_body_is_base_64 = nil
  entry.timings = Timings.new
  @objects[request_id] = entry
  @pending_requests[request_id] = true
```
Capturing Responses

```ruby
when "Network.responseReceived"
  unless @objects[request_id].nil?
    entry = @objects[request_id]
    entry.response_message = msg

    unless msg_params["response"]['response'] == nil?
      timing = msg_params['response']['response']

      dns_start = timing['dnsStart']
      dns_end = timing['dnsEnd']
      entry.timings.dns = time_delta(dns_start, dns_end)

      send_start = timing['sendStart']
      send_end = timing['sendEnd']
      entry.timings.send = time_delta(send_start, send_end)

      ssl_start = timing['sslStart']
      ssl_end = timing['sslEnd']
      entry.timings.ssl = time_delta(ssl_start, ssl_end)

      recieve_headers_end = timing['receiveHeadersEnd']
      entry.timings.wait = time_delta(send_end, recieve_headers_end)

      connect_start = timing['connectStart']
      connect_end = timing['connectEnd']
      entry.timings.connect = msg_params['response']["connectionReused"] ? -1 : time_delta(connect_start, connect_end)

      entry.started_date_time = Time.at(timing['requestTime']).utc.iso8601(3)
  else
    a_url = msg['params']["response"]["url"]
  end
end
```
Gotchas...

1. Chrome gives us weird timestamps...
   – NOT a UNIX timestamp
   – Appears to be “number of seconds since 00:00:00 today”
   – We discovered an undocumented field that appears in one message that gives us a UNIX timestamp (very weird)

2. Protocol a resource is loaded with is not exposed
   – Oh wait, it is exposed via partially documented field...

3. We might receive a Network.loadEventFired before we receive all the Network.responseReceived events
   – This one took awhile to figure out...
   – Solution: add a bit of a delay after onload to keep capturing
   – Ugly, but best we have right now.
Capturing Video

• How do we use a headless browser to capture video?!
• Virtual Frame Buffer (xvfb)
  – Does actually render stuff, but not displayed anywhere
• Use ffmpeg to record the rendering
  – We tried a variety of methods, e.g., actually copying the framebuffer file
  – ffmpeg does things in memory
  – “no” latency → “guaranteed/stable” framerate
  – We encode to WebM format
Storing Things

• We are going to build a web application...
• Need to access the data to show to users
• Could use flat files I guess...
  – Would suck... Random access is awful here!

Database Time!!!!!
PostgreSQL

• “World’s most advanced open source RDBMS”
• We have a lot of JSON data (HARs)
  – Doesn’t map well to typical RDBMS rows/columns
• Postgres 9.2+ offers us a JSON column type
  – 9.4+ gives us JSONB (a binary version)
  – Perfect for us!
eyeorg.net - Frontend

• Jekyll
  – Toolkit for building static websites
  – Gives us templating, themes, etc.

• Frontend communicates with backend via AJAX
  – Hits a web API that returns JSON
eyeorg.net - Backend

• Ruby on Rails
  – Major functionality

• Grape API DSL
  – DSL to build web APIs
  – Frontend will access
### experiment: Operations about experiments

**GET**  /api/v1/experiment/random_h1_h2.json

Get a random H1 v H2 A/B experiment.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection_strategy</td>
<td>closest_to_median</td>
<td></td>
<td>query</td>
<td>string</td>
</tr>
</tbody>
</table>

**POST**  /api/v1/experiment/random_h1_h2.json

Post results for a random H1 H2 A/B experiment.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(required)</td>
<td>The a for the experiment</td>
<td>form</td>
<td>integer</td>
</tr>
<tr>
<td>b</td>
<td>(required)</td>
<td>The b for the experiment</td>
<td>form</td>
<td>integer</td>
</tr>
<tr>
<td>details[picked]</td>
<td>(required)</td>
<td></td>
<td>form</td>
<td>integer</td>
</tr>
</tbody>
</table>

### session: Operations about sessions

**GET**  /api/v1/session.json

Get a session (see if you are logged in)

**POST**  /api/v1/session.json

Create a new session

### respondent: Operations about respondents

**POST**  /api/v1/respondent/random_h1_h2_respondant.json

Create a new respondent

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gender</td>
<td>(required)</td>
<td></td>
<td>form</td>
<td>integer</td>
</tr>
<tr>
<td>age</td>
<td>(required)</td>
<td></td>
<td>form</td>
<td>integer</td>
</tr>
<tr>
<td>current_country</td>
<td>(required)</td>
<td></td>
<td>form</td>
<td>string</td>
</tr>
<tr>
<td>origin_country</td>
<td>(required)</td>
<td></td>
<td>form</td>
<td>string</td>
</tr>
</tbody>
</table>

**GET**  /api/v1/respondent/random_h1_h2_respondant.json

Get the results for a respondent
Putting it all together...

Which version is faster?
Select the version of the page that seems ready to use first by clicking a video. Choose “No Difference” if they appear to be ready at the same time.
Demo
Preliminary Results

• 43 total respondents

• Matteo Demo’d at PAM ‘16
  – We got ~30 or so respondents
  – Not bad!

• Also had some people from our lab use it
  – Another ~10 respondents
How Were Protocols Distributed A/B?

We are looking for a 50/50 split
We got ~54/46 split...
Which Was Faster?

Which is faster?

H1

H2

No Difference

count

Which is faster?
How Do Humans Compare to onload?

[Bar chart showing percentage "Correct" for different onload delta (ms) intervals.
- Y-axis: % "Correct"
- X-axis: onload delta (ms)
- Data points include: 0, 110.75, 1119.99, 212.565, 2719.19, 333.039, 4125.29, 436.877, 624.916, 767.361, 954.173]
Lessons Learned So Far

1. Some people viewed it as a *competition*
   - “The goal is to pick the site that is H2!”
   - This wasn’t what we expected...

2. Some people used the “slow motion” feature
   - Kind of gives them “super powers”
   - Still trying to decide if we should remove this

3. Gotta be careful with how we load things
   - Need to make sure that we sync things
   - Possible solution: make a *single* video
Next Steps

• Deploy to crowdsourcing platform
  – Mechanical Turk
  – Crowdflower
  – …

• Build second experiments where users select when a page has “finished loading”
  – Compare human derived PLT to SpeedIndex and “onload” timing
The eyeorg.net Vision

• A *distributed, crowdsourced* platform for measuring PLT with a variety of metrics

• You build an image
  – Has your custom code (e.g., you modified chrome)
  – Has some configuration
    • URLs to crawl
    • How to choose which videos to display
    • Whether to clear cache
    • Network configuration (delay, TCP stack, etc...)
    • …

• Push image to eyeorg.net → get results
Who Did This Work?

Check out “Is The Web HTTP/2 Yet?” from PAM 2016 and isthewebhttp2yet.com

Check out eyeorg.net and take a test!
Thank You For Your Time!

ANY QUESTIONS?