# Rules of Thumb for HTTP/2 Push

*Tom Bergan, Simon Pelchat, Michael Buettner*
*{tombergan, spelchat, buettner}@chromium.org*
*Last Updated: 2016/08/03*

HTTP/2 has a new feature called *server push* that promises to improve page load times. The idea: rather than waiting for the client to send a request, the server preemptively pushes a resource that it predicts the client will request soon afterwards. For example, if the server sends the client an HTML document, the server can reasonably predict that the client will also request subresources linked from that HTML document, such as JS and CSS files.

More broadly, we can build a *fetch dependency graph* for a page. This graph has an edge from A to B if resource A reveals the need to fetch resource B. For example, given that doc.html imports a.js and a.js import b.js via document.write, there is an edge from doc.html -> a.js and another edge from a.js -> b.js. Each time a client requests a.js, the server can proactively *push* b.js along with any or all of the other descendants of a.js in the fetch dependency graph.

Unfortunately, server push does not always improve page load performance. It is not always obvious why this is so. Further, indiscriminate use of server push can actually make page load times *worse*. This document compiles lessons we learned while experimenting with server push. Many of these lessons will be obvious and common-sense, at least in retrospect; others may not be so obvious.

To summarize, we recommend the following:

1. ***Push just enough resources to fill idle network time, and no more.***
   Server push eliminates idle network time between the server sending one response to the client and waiting for the next request. During this idle time, the server can fill the network with as many bytes as are allowed by the current TCP congestion window (cwnd). Pushing more than cwnd bytes has no additional benefit and can actually *hurt* performance, for reasons described below. A corollary is that TCP slow start makes server push less effective on cold connections than on warm connections.

2. ***Push resources in evaluation-dependence order.***
   The browser evaluates each subresource in a specific order. Pushing resources in the wrong order delays evaluation, which can delay discovery of hidden resources that are requested dynamically, which can make page load time *worse*. For this reason, it may be better to avoid pushing resources when their evaluation order is not precisely known, especially when those resources cumulatively exceed the current cwnd (recall rule #1).

3. ***Consider using special strategies to track the client-side cache.***
   Push wastes bandwidth when the client has already cached the resource being pushed. Further, pushing more than cwnd bytes makes page load time *worse* if the extra bytes pushed are already cached (again, recall rule #1). Unfortunately, there is no perfect, well-tested method to avoid pushing cached resources. We mention a few recent proposals.

4. ***Use the right cookies when pushing resources that vary by cookie.***
   Unlike the above issues, this is a correctness problem, not a performance problem. This is particularly important when a pushed resource depends on path-scoped cookies that may not have been sent with the mainframe request that initiated the push.

5. ***Use server push to fill the initial cwnd and consider using preload links to reveal the remaining critical or hidden resources.***
   In cases where a resource should not be pushed due to the above rules, a preload link may make more sense. Preload links cannot eliminate idle network time as well as server push. However, preload links are cache- and cookie-aware, and further, they can be applied to cross-domain or third-party resources, while a server can only push resources for which it is authoritative.

**Table of contents:**


Intended Audience and Scope
Experimental Methodology
Rule #1: Push Enough to Fill Idle Network Time, and No More
    A Simple Example
    Slow Start Means Slow Push
    How Resource Evaluation Speed Affects Idle Time
    Generalizing to Multiple RTTs
    Implications for Developers
Rule #2: Push Resources in the Right Order
    Push and HTTP/2 Priorities
    Bad Interactions Between HTTP/2 Priorities and Bufferbloat
    Don’t Let Pushed Responses Delay the Main Response
    Implications for Developers
Rule #3: Don’t Push Resources the Client has Already Cached
Rule #4: Use the Right Cookies
    What Happens if the Server Uses the Wrong Request Headers?
Rule #5: Consider Preload Instead of Push
Case Studies on Actual Pages
    Straw Man Comparison

# Intended Audience and Scope

The intended audience is HTTP/2 server implementers, who want to support server push in their servers, and web developers, who think their pages may be benefit from server push. We assume the reader is already familiar with basic web concepts a high-level, particularly HTTP/2 and HTML, and also basic networking concepts like BDP, congestion control, and head-of-line blocking. Ilya Grigorik's *High Performance Browser Networking* gives a great overview of these topics.

Our examples focus on using server push to improve page load performance in web browsers. This is not the only case where server push is useful. Other cases include: responses to user actions on already-loaded web pages, or responses to RPC requests where the RPC transport layer uses HTTP/2. Our rules of thumb should generalize to these other cases as well, although rules #3-#5 apply only when the client is a web browser.

When we discuss response ordering, particularly in rule #2, we assume responses should be ordered as atomic units. This matches HTTP/2's notion of stream dependence, where each response corresponds with a single HTTP/2 stream. Advanced servers may want to carefully interleave pushed responses with the main response to micro-optimize resource delivery. For example, a server may "inline" a JS file by pushing it just after sending the corresponding <script> tag from the main HTML. This effectively splits the HTML into two parts. If you treat these two parts as separate "responses", our rules of thumb should apply directly. However, the implementation may be tricky. We do not discuss this idea further.

# Experimental Methodology

Unless otherwise stated, all experiments used the following setup:

- Pages were served from a version of WebPageReplay (WPR) implemented in Go, with support for custom HTTP/2 push policies for each replay.
- The server ran in a GCE instance with 2 vcpus and 5GB memory.
- The client was Chrome version 51 running on a Moto G.
- The network was simulated using WebPageTest.

Unless otherwise stated, all measurements are medians over at least 25 runs, where the number of runs is high enough that the median is stable. We consider a median stable if the inter-quartile range is at most 1% of the median. Concretely, this means $(p(75)-p(25))/p(50) \leq 1\%$, where $p(x)$ is the xth percentile. All page loads used a cold browser cache.

We configured WebPageTest to simulate 2G and 3G networks using the parameters shown below. We disabled simulated packet loss to minimize variability. After setting the simulation parameters, we measured the actual throughput and RTT for our setup. Throughput was computed from the time to download a 1MB file. RTT was computed from the TCP connect time. Measured results are shown below:

| **Network** | **Throughput (kbps)** | | **RTT (ms)** | | **Bandwidth-Delay Product (measured)** |
|---|---|---|---|---|---|
| | **Measured** | **Expected** | **Measured** | **Expected** | |
| 3G | 1360 | 1800 | 209 | 200 | 35 KB |
| 2G | 113 | 120 | 1420 | 1400 | 20 KB |

# Rule #1: Push Enough to Fill Idle Network Time, and No More

Push helps when there is idle network time. If there is no idle time, then push cannot deliver resources any faster than the browser can request them. The amount of data that can be pushed during idle time depends on the network's bandwidth-delay product (BDP, see above table), the TCP congestion window (cwnd), bufferbloat, and the amount of idle time itself, which depends on how quickly the browser evaluates the page. We explore the effect of these parameters on server push. We make the following recommendation:

> *Push enough resources to fill idle network time, but no more.*

## A Simple Example

To illustrate, we start with a simple HTML page that loads a synchronous Javascript file in the <head> section. We assume the server can generate the JS and HTML responses instantly. The server will push the JS file after sending the HTML. Our expected improvement from push depends on the size of the HTML file relative to the BDP:

- If size(HTML) ≤ BDP/2, the page should load about 1 RTT faster.
- If size(HTML) ∈ (BDP/2, BDP), the page should load < 1 RTT faster.
- If size(HTML) ≥ BDP, the page will not load faster.

Suppose size(HTML) ≤ BDP/2 and push is disabled. By the time the client receives the first few bytes of HTML, the entire HTML document is already on the wire. The client can send the JS request instantly (since the JS link is in the <head> of the document), but it must wait 1 RTT to get the first byte of JS. Push can eliminate this 1 RTT penalty by sending the JS file immediately after the HTML.

However, if size(HTML) ≥ BDP, then even with server push enabled, the server cannot put the JS file onto the wire any more quickly because it takes at least one full RTT to put the entire HTML document on the wire. By this point, the client's request for the JS has already arrived.[1]

To demonstrate this effect, we use a simple HTML page that imports a single Javascript file in the <head> section and contains random text in the <body>. The page is loaded once both files have been downloaded. We request this page with push disabled, then again with push enabled. The JS file is pushed immediately after the HTML is sent.

---

[1] We send the entire HTML followed by the JS for demonstration. A clever server may want to interleave the JS file (and perhaps other resources) with the <head> section of the HTML to truly minimize the latency of JS evaluation. In this case, it may makes sense to push the JS even if size(HTML) ≥ BDP, however our high-level point still holds -- there is no benefit to pushing more than the BDP.

The table below shows results. The HTML document size is varied as shown in the left column; the JS file was a constant 10KB in all cases. The *Diff* column gives the percentage improvement from push, if any (rounded to the nearest percent).

| HTML Size | Load Times for Cold Connections (ms) | | | | | |
|---|---|---|---|---|---|---|
| | 3G (BDP = 35KB) | | | 2G (BDP = 20KB) | | |
| | No Push | With Push | Diff | No Push | With Push | Diff |
| 5 KB | 1231 | 1097 | +11% | 7240 | 6721 | +7% |
| 20 KB | 1233 | 1175 | +5% | 7760 | 7781 | - |
| 40 KB | 1344 | 1349 | - | 9136 | 9164 | - |
| 80 KB | 1532 | 1537 | - | 11813 | 11832 | - |
| 200 KB | 2120 | 2134 | - | 19930 | 19953 | - |

We can estimate what the above table *should* look like given our basic model of push. First, we should see no improvements for push when size(HTML) ≥ BDP. This holds true. Second, the following formula predicts the *Diff* column in the above table:

```
Tnopush = rtt + max(rtt, size(HTML)/bw) + size(JS)/bw
Tpush   = rtt + size(HTML)/bw + size(JS)/bw
Diff    = 1 - Tpush/Tnopush
```

For the cases where size(HTML) ≤ BDP, we get:

- 2G, size(HTML)=5KB, Expected Diff = +29.5%
- 3G, size(HTML)=5KB, Expected Diff = +37.6%
- 3G, size(HTML)=20KB, Expected Diff = +18.9%

These predictions are much higher than the measurements shown in the above table. Why?

## Slow Start Means Slow Push

BDP is actually limited by the server's congestion window (cwnd), which may not reflect the actual network bandwidth. In fact, the above experiments used cold connections, which means cwnd is artificially low due to TCP slow start. Hence, the inequalities from the beginning of this section should actually read:

- If size(HTML) ≤ cwnd/2, we expect the page should load about 1 RTT faster.
- If size(HTML) ∈ (cwnd/2, cwnd), the page should load < 1 RTT faster.
- If size(HTML) ≥ cwnd, the page should not load faster.

For warm connections, cwnd should approximately match the network's actual BDP. For cold connections, cwnd can be lower, effectively lowering the amount of data we can push during idle network times. Unfortunately, many page loads occur on cold TCP connections -- we expect this will limit the benefits of server push in practice.

Modern TCP implementations use an initial cwnd of about 14KB. After completing a TCP handshake and sending the first 14KB, the server must wait one RTT for an ACK, after which cwnd doubles and the server can send another 28KB.[2] We wrote a simple simulation in Python to estimate the expected performance of push on cold connections after TCP Slow Start is accounted for.

Our simulation results are given below. These numbers match the measured numbers in the above table much more closely:

- 2G, size(HTML)=5KB, Expected Diff = +11.3%
- 3G, size(HTML)=5KB, Expected Diff = +13.6%
- 3G, size(HTML)=20KB, Expected Diff = 0%

To verify that push performs better over warm connections, we reran the above experiments using warm connections. We created a warm connection by first loading a 100KB HTML page, where the bottom of that HTML contains Javascript code that redirects to the actual HTML page. Our page load timer starts when the browser makes this redirect request. Results are shown in the table below:

| **HTML Size** | **Load Times for Warm Connections (ms)** | | | | | |
|---|---|---|---|---|---|---|
| | **3G (BDP = 35KB)** | | | **2G (BDP = 20KB)** | | |
| | **No Push** | **With Push** | **Diff** | **No Push** | **With Push** | **Diff** |
| 5 KB | 746 | 462 | +38% | 3876 | 2691 | +30% |
| 20 KB | 1378 | 1161 | +16% | 3884 | 3763 | +3% |
| 40 KB | 3319 | 3237 | +2% | 5143 | 5166 | - |
| 80 KB | 5885 | 5891 | - | 7899 | 7910 | - |
| 200 KB | 14142 | 14150 | - | 16226 | 16228 | - |

These numbers match the above formulas for *Diff, Tpush,* and *Tnopush* much more closely.

[2] This explanation is approximate -- ACKs can be pipelined, meaning cwnd can increase slightly more quickly in practice, but the high-level effect is the same. Note that our simulation uses a more precise model that accounts for pipelined ACKs.

## How Resource Evaluation Speed Affects Idle Time

If we modify the above example to import the JS file from the *end* of the HTML, instead of <head>, then the browser won't request the JS file until *after* the HTML file has been entirely downloaded. There is now exactly one RTT's worth of network idle time regardless of the HTML size. Still, our basic rule has not changed -- there is no benefit to pushing more resources than necessary to occupy the entire idle time.

To illustrate, we load a simple HTML page that imports two JS files (1.js and 2.js) just before the closing </body> tag. We push both 1.js and 2.js after sending the HTML. We vary size(HTML) and size(1.js) but fix size(2.js) at a constant 10KB. Note that for any fixed size(1.js), size(HTML) does not affect whether any benefits are seen from push. However, size(1.js) matters -- we see no additional benefit from pushing 2.js once size(1.js) exceeds cwnd (20KB):

| **size(HTML)** | **size(1.js)** | **2G Load Times (ms)<br>Warm Connection, BDP = 20KB** | | |
|---|---|---|---|---|
| | | **Push (1.js only)** | **Push (1.js, 2.js)** | **Diff** |
| 5 KB | 5 KB | 4125 | 2995 | +27% |
| 20 KB | 5 KB | 5204 | 4067 | +21% |
| 40 KB | 5 KB | 6685 | 5461 | +18% |
| 40 KB | 20 KB | 6681 | 6580 | - |
| 40 KB | 40 KB | 8079 | 8089 | - |

The above example is not very practical -- servers will likely push resources from the <head>, not the tail, since the <head> section likely contains critically important resources. However, the example is a good model for the following scenario:

Suppose a page uses many resources, one of which is a script that inserts images into the page. Perhaps the script and images are not important enough to push with the HTML, but the server would like to push the images when the script is eventually requested. Browsers typically do not evaluate scripts incrementally like HTML -- instead, the *entire* script file must be downloaded before it is evaluated. Further, evaluating a script can introduce significant extra delays when the script is slow or the device is low-powered. Hence, in this case, the idle

network time includes one RTT plus the time to evaluate the script. We recommend that the server should push just enough images to fill this idle time, and no more.[3]

## Generalizing to Multiple RTTs

So far we have seen cases where the idle network time covers one RTT. Now we show how our rule generalizes when idle time spans multiple RTTs. Consider an HTML page that imports a JS file (1.js) just before </body>, where 1.js imports two more JS files (2.js and 3.js) via calls to document.write. Without push, this page load has about two RTTs worth of idle network time:

1. Between receiving the last byte of HTML and the first byte of 1.js.
2. Between receiving the last byte of 1.js and the first byte of 2.js.

Push can eliminate both of these RTTs. Let's suppose the connection is warm. We expect a performance improvement from pushing *both* 1.js and 2.js with the HTML. These two pushes eliminate the two RTTs above, respectively. It may be beneficial to push 3.js as well, but this depends on whether 2.js is large enough to occupy the entire second RTT:

- If size(2.js) < BDP, we expect an improvement from pushing both 2.js and 3.js
- If size(2.js) > BDP, there is no additional improvement from pushing 3.js.

For completeness, we demonstrate this with an experiment:

| **size(2.js)** | **2G Load Times (ms)<br>Warm Connection, BDP = 20KB<br>HTML, 1.js, and 3.js are all 10KB** | | |
|---|---|---|---|
| | **Push (1.js, 2.js only)** | **Push (1.js, 2.js, 3.js)** | **Diff** |
| 5 KB | 5419 | 4206 | +22% |
| 20 KB | 5491 | 5303 | +3% |
| 40 KB | 6813 | 6786 | - |

## Implications for Developers

Server developers should be aware of the connection's current cwnd and should take this into account when deciding how much data to push. On Linux, the current cwnd can be obtained through `getsockopt(fd, SOL_TCP, TCP_INFO)`, which returns a tcp_info struct that

[3] We actually recommend using preload links in this scenario when possible. Starting a push in the middle of a page load introduces other challenges since that push can now steal bandwidth from other in-flight requests. See rules #2 and #5.

contains the send congestion window size (also see this article). The default initial cwnd can be changed with the global `initcwnd` setting (see this article). Note that Linux lowers cwnd and reverts to slow-start if keep-alives are enabled and the TCP connection has become idle. This means even "warm" connections may behave like cold connections. This behavior can be disabled by setting `tcp_slow_start_after_idle=0` (see the tcp manpage).

Server developers should also be aware that the kernel-reported cwnd may be inaccurate in two respects. First, the true cwnd may be artificially limited by the client's receive window. In Linux, the effective cwnd is actually the minimum of `tcpi_snd_cwnd` and `tcpi_rcv_space` (in the tcp_info struct). Second, cwnd may over-estimate the true BDP due to bufferbloat. We have more to say about bufferbloat in the next section, where we discuss rule #2.

Web developers should be aware of typical BDP sizes and how those sizes compare to the resources they want to push. Below is a table estimating BDP for a variety of connection types:

| Connection Type | RTT (ms) | Bandwidth (kbps) | BDP |
|---|---|---|---|
| 2G (as defined above) | 1420 | 113 | 20 KB |
| 3G (as defined above) | 209 | 1360 | 35 KB |
| 4G LTE (sprint) | 150 | 4500 | 85 KB |
| 4G LTE (verizon) | 150 | 8500 | 160 KB |
| 25 Mbps cable | 50 | 25000 | 156 KB |
| 1 Mbps satellite | 638 | 1000 | 80 KB |

A network sees the largest improvement from push if it has high RTT (maximizing idle time to remove) and high bandwidth (maximizing data pushed during idle times). Unfortunately, such networks don't really exist. The closest is probably satellite internet, which is becoming increasingly rare.

The httparchive has stats for a large corpus of web pages. Suppose we served all of these web pages using server push, where a set of "critical" resources are pushed along with each HTML document. Based on stats from httparchive and our data above, we can make a few tentative observations:

- We expect no improvement from push on cold 2G connections when the HTML document is larger than 5KB. This covers 80% of HTML documents.

- The situation is better for warm 2G connections, where we expect small improvements for HTML documents up to 20KB. This covers 85% of HTML documents.

- We expect more relative improvement for faster connections than for slow connections, since faster connections generally have a higher BDP. It is unfortunate that this trend is not reversed.

# Rule #2: Push Resources in the Right Order

Web pages are evaluated in a well-defined order. For example, synchronous scripts are executed sequentially in the order they appear in the HTML file. When a server decides to push one or more resources, it must schedule those resources on the wire in some order. Sending resources in the wrong order will delay client-side evaluation, which can cause slower page loads for a number of reasons, but most importantly:

- Rendering is delayed if a non-render-blocking resource is sent on the wire before a render-blocking resource.

- Evaluation of one resource may reveal the need to fetch another resource, such as when a synchronous script injects a reference to another script via document.write. Delay of the first script delays the request for the second script. Note that it may not be possible to push the second script since the URL may be determined dynamically.

To demonstrate this effect, we load the following web page:

```
<html>
  <head>
    <script src=”1.js”></script>
    <script src=”3.js”></script>
    <script src=”4.js”></script>
  </head>
  <body>... 10 KB of junk …</body>
</html>
```

The first script (1.js) loads another script (2.js) via a document.write call at the end of the script. The other scripts (3.js, 4.js) are no-ops. The first script (1.js) is 5KB. All other files are 10KB. We experimented with different push strategies to measure how push order affects load time. Using a 2G connection, there is 10KB left in the BDP after sending the HTML file. This means we expect a performance improvement when we push a single JS file. However, when we push multiple JS files, the expected improvement depends on the files we choose to push and the order we push them in.

Results are shown in the table below. Some of these push orders seem contrary to common sense. Obviously a server would not push 3.js and 4.js when 1.js is clearly more important! However, a server may be tempted into this scenario if it cannot push 1.js for some reason, for example, when 1.js is a dynamic URL, or when 1.js is hosted on a different domain (perhaps a CDN or third-party site). The point of this experiment is to demonstrate the danger of this temptation.

| Push Order | 2G Load Times (ms)<br>Warm Connection, BDP = 20KB | |
|---|---|---|
| | Load Time | Diff to No Push |
| No Push | 3593 | - |
| 1.js | 2740 | +23% |
| 3.js | 2789 | +22% |
| 1.js, 2.js | 2796 | +22% |
| 3.js, 1.js | 3465 | - |
| 3.js, 4.js | 4179 | **-16%** |

These results are most easily explained by timelines. Below is the timeline for “No Push”. The first row shows the passage of time in terms of RTTs, the second row shows when requests are made, and the third row shows when responses are downloaded:

| 1 RTT | 2 RTT | | 3 RTT | | | 4 RTT | 5 RTT |
|---|---|---|---|---|---|---|---|
| HTML | 1.js,3.js,4.js | | | 2.js | | | |
| (idle) | HTML | (idle) | 1.js | 3.js | 4.js | 2.js | |

Up to 10KB can be pushed during the idle time in the second RTT. This explains why the first three push examples are faster. Below is the timeline for the fourth example (“Push Order = 3.js, 1.js”). This is not any faster than No Push because 1.js is not received any sooner:

| 1 RTT | 2 RTT | | 3 RTT | | | 4 RTT | 5 RTT |
|---|---|---|---|---|---|---|---|
| HTML | 4.js | | | 2.js | | | |
| (idle) | HTML | 3.js | 1.js | 4.js | (idle) | 2.js | |

The final example performs *worse* than No Push because it orders 1.js after both 3.js and 4.js. This delays the request of 2.js, adding extra idle time into the page load.

| 1 RTT | 2 RTT | | 3 RTT | | 4 RTT | | 5 RTT |
|---|---|---|---|---|---|---|---|
| HTML | 1.js | | | | 2.js | | |
| (idle) | HTML | 3.js | 4.js | 1.js | (idle) | 2.js | |

## Push and HTTP/2 Priorities

HTTP/2 encodes request priorities using a weighted dependence tree. By default, pushed streams depend on the request that initiated the push (see Section 5.3.5). We suggest using an *exclusive dependence* for pushed streams.

To illustrate, suppose the client makes requests for A and B, where B depends on A and the server decides to push C and D when A is requested. The server should not decide to push C and D in the first place unless C+D are needed immediately after A. Otherwise, there is a risk that concurrent requests (like B) are actually more important, in which case pushing C+D can steal bandwidth from those more important requests. Further, if C is more important than D, then those resources should be ordered relative to each other. Exclusive dependencies are the simplest way to express these orderings:

*default dependence tree:* A ⟶ {B, C, D}
*using exclusive dependencies:* A ⟶ C ⟶ D ⟶ B

One way to implement this is for the server to update its HTTP/2 priority tree, then send PRIORITY frames to the client that make A the exclusive parent of C and C the exclusive parent of D. This is an attractive implementation because the server can continue using the HTTP/2 priority tree to order requests C, D, and B.

However, this is fundamentally racey: if both ends (client and server) update the priority tree concurrently, it can easily become out-of-sync. For this reason, we advocate *not* mutating the priority tree on the server. Instead, the server should continue using the default priority tree, but should make a note in its tree that siblings {C,D,B} should be prioritized in that order. If the client later sends a PRIORITY frame to reorder one of these streams, that new priority should take precedence.

## Bad Interactions Between HTTP/2 Priorities and Bufferbloat

A known problem with HTTP/2 priorities is that you cannot reprioritize data that has already been buffered in the kernel. If the kernel's send buffer is considerably larger than the TCP cwnd, the server may write data to the kernel "too soon", resulting in suboptimal scheduling. This blog post has a great discussion. For cellular networks, the problem is even worse due to bufferbloat in ISPs -- data can be buffered in cellular networks in far excess of the link's true BDP, and once data has been pushed into network buffers, it cannot be reprioritized.

What does this mean for server push? It turns out that server push is one possible work-around for the above problem. Suppose the client makes N concurrent requests. The first request is for a JS file that will load a render-blocking CSS file. After evaluating the JS, the client will request the CSS file with high priority (as of version 51, Chrome moves the CSS file to the top of the

priority tree). In theory the server should respond with that CSS file immediately. However, if the kernel and network buffers are already full, those buffers must be drained before the CSS file can be received. In bufferbloat scenarios, this can require multiple RTTs.

Suppose the server instead pushes the CSS file with the JS file. This avoids the above prioritization problem entirely because the CSS will be sent immediately after the JS, without being preempted by a long sequence of buffered responses. Note that we do *not* consider server push the best solution for this prioritization problem -- we mention it only because we've spent time puzzling over waterfalls, wondering why we were seeing more improvement from server push than expected, only to realize that server push was masking the true problem (bufferbloat). It is better to avoid bufferbloat directly. To fix excessive kernel-level buffering, the server should take control using options like TCP_NOTSENT_LOWAT (also see this paper). Another solution is to switch from TCP to QUIC. Unlike TCP, QUIC is implemented entirely in user-space -- this allows QUIC to use a priority queue for its send buffer, effectively eliminating head-of-line blocking due to large kernel send buffers (although this behavior is not currently standardized).

We revisit this problem at the end of this document as we have seen it happen in practice (see the "artofliving" example in "Case Studies on Actual Pages").

## Don't Let Pushed Responses Delay the Main Response

We encountered an interesting ordering problem in the following scenario:

1. The client sends an HTML request to a frontend server (e.g., an edge proxy), which forwards the request to a backend server.
2. The backend server responds with response headers for the HTML. This includes a special header "X-Push: foo.js".
3. On receiving that header, the frontend server sends the client a PUSH_PROMISE for foo.js and creates an internal request to handle foo.js.
4. This JS file is cached locally in the frontend server, so that response is served instantly.
5. The HTML response body finally arrives at the frontend server from the backend server.
6. The frontend has already written the JS file into the outgoing kernel socket, which means that the JS file will arrive at the client *before* the HTML. This delays parsing of the HTML, which delays requests for other subresources that were not pushed.

We can deterministically reproduce the above race by ensuring there is enough latency between the frontend and backend. Why don't HTTP/2 stream dependencies solve this problem? The HTTP/2 spec is clear that priorities are only a suggestion. In fact, it can be a *good* idea to start pushing the JS file before the HTML response is ready. Suppose it takes a long time for the backend to generate the HTML response body. If the frontend follows HTTP/2 priorities strictly, the network will stay idle until the HTML is finally received. It is a good idea is

to start pushing the JS file during this idle time, even though that push is technically out-of-order.

The problem comes when we introduce bufferbloat, either in the kernel or the network. Now, a careless frontend may write a very large JS file into the kernel's send buffer. If the HTML arrives before that send buffer is flushed, the HTML will be delayed, which can hurt page load performance. This is the scenario we ran into above. The moral of this story is that out-of-order pushes can be useful when higher-priority responses are not yet ready, but this benefit can be completely reversed if the low-priority responses get buffered ahead of the high-priority responses, due to bufferbloat.

It is tempting to believe that the above race triggers only when responses are generated by slow backends. Alas, that is not the case. We encountered this same problem in our simple replay server that serves all responses from an in-memory cache. Briefly, the problem is randomized thread scheduling -- if a server has one main response and N pushes in flight, the main response handler will be scheduled after N/2 push handlers on average if scheduling is random. If the pushes occupy exactly one BDP, this delays the main response by ½ RTT. On actual pages, we observed a performance degradation of up to 10% (see the "bollywoodhungama" example in "Case Studies on Actual Pages"). Once again, the solution is to avoid buffering low-priority responses in front of higher-priority responses, either by using QUIC, by following priorities strictly, or by throttling writes for low-priority responses.

## Implications for Developers

Developers should understand how their page will be evaluated and should arrange to have resources pushed in evaluation order. An optimal strategy is to build a topological sort of the page evaluation graph, label each resource by its size, then push the first N resources from the topological order, where the cumulative size of the HTML and the first N-1 resources is less than cwnd, while the Nth resource brings the cumulative size just above cwnd.

This optimal strategy is not always possible. For example, sometimes a very "critical" resource is either computed dynamically by a script (meaning it cannot be pushed) or the resource is hosted on a third-party domain (recall that a server cannot push resources for which the server is not authoritative). When the optimal strategy cannot be used, it is always safe to push any N resources where the cumulative size of the HTML and those N resources is less than cwnd -- this may not result in a performance improvement, but at minimum it should avoid degrading performance relative to "no push".

It is risky to start a push in the middle of a page load (i.e., from a response other than the initial HTML response) because there may already be higher-priority requests in flight. Developers should push resources in this situation only if the resources are truly needed immediately.

Developers should be aware of the bufferbloat problem as it may help explain unintuitive performance results. Servers should respect HTTP/2 priorities as much as possible, particularly when the priorities are intended to order the main HTML response relative to pushed responses. Servers should consider switching from TCP to QUIC, as QUIC can eliminate kernel-level head-of-line blocking.

# Rule #3: Don’t Push Resources the Client has Already Cached

A challenge arises due to client caches. It wastes bandwidth to push resources that the client has already cached. This is particularly bad for data-conscious users.

Suppose the server pushes a resource that the client has already cached. Recall rule #1: if the server’s push fits in existing idle time, then at worst this wastes bandwidth without degrading performance. However, if the server’s push exceeds idle time, it is easy to see how pushing unneeded bytes can delay needed bytes, thus degrading performance.

If a server had perfect knowledge, it would simply skip any resource the client has already cached. However, it is difficult to know the state of the client’s cache. A few techniques have been proposed. The most promising are:

1. Each time the client requests a resource from a server, the server updates a cookie that contains a Bloom filter summarizing all resources the client has requested from that server. The client sends this cookie with each request. The server inspects the cookie when deciding what to push and does not push any resources the client has requested recently. (This approach has been implemented in H20.)

2. A variation on the prior approach is to use a separate cookie for each resource. The server can then use cookie expiration times to model cache expiration times, at the cost of more cookies. (We read this idea somewhere but cannot remember the reference.) Note that using too many cookies may run afoul of browser cookie limits.

3. An obvious extension is for the client to describe its cache directly via a header, perhaps using a Bloom filter. This only works for specific client/server pairs that understand the new header. Or, in cases where service workers control the cache, it may be possible to implement this directly in a service worker without any browser changes. An IETF draft has been written for a new HTTP/2 frame to communicate the client’s cache state to the server.

4. If the server predicts the client has a cached version of a resource, it can preemptively push a 304 to the client. This removes the latency penalty from revalidation of stale resources. Unfortunately, current browsers don’t respect 304 pushes in this way (see discussion in this blog post).

Other proposed techniques are less promising:

5. If the page request includes a conditional header, like If-None-Match, then the client has requested the page previously and has likely cached all cacheable subresources needed by the page, assuming the contents of the page have changed only slightly. This idea works only for repeat visits to the same page, not visits across a site where many pages share a standard set of resources. (Idea proposed in this mailing list thread.)

6. The server can include If-Match in its PUSH_PROMISE to name the version of the resource being pushed. If the client has already cached that version, the client can close the stream with RST_STREAM. This idea seems impractical in most cases because it relies on the client to cancel unneeded pushes. The delay between the server sending a PUSH_PROMISE and receiving the client's cancellation is 1 RTT, at which point the server will likely have written the entire pushed response on the wire. However, this idea may work for resources that are very large relative to the network's BDP. (Idea proposed in this mailing list thread.)

An evaluation of the above approaches is meaningful only if the evaluation is done at scale. We are not aware of any evaluation that has been done so far. We leave such an evaluation for future work.

# Rule #4: Use the Right Cookies

Each server push event is a response to a pseudo request that is generated by the server. The server serializes these pseudo requests into PUSH_PROMISE frames that are sent to the client, where each pseudo request corresponds to a hypothetical future request that the client is expected to make shortly. The server must predict the headers the client will use for that future request. An incorrect prediction can lead to the server pushing the wrong response should the response depend on specific header values.

One solution is to not push any response with a non-empty Vary header. However, this is overly restrictive. For example, it's reasonable to expect that all requests from the same client will contain the same User-Agent header.

Special care is needed when handling cookies. Suppose a server receives a request for page A and wants to push resource B with the response. The server knows the cookies the client sent when requesting A. However:

- *Which of A's cookies are applicable to B?* If any of A's cookies are path-scoped, some cookies may not apply to B. If the server is authoritative on multiple hosts or domains, then it is possible that none of A's cookies are applicable to B.

- *Is the set of cookies sent with request A the complete set of cookies needed to handle B?* Similarly to the prior question: the client may have cookies applicable to B but not A. Those cookies are not sent in the request for A, so they are not known when the push is started.

- *If the response for A includes Set-Cookie, should those cookies be forwarded to B?* The server may need to generate the response headers for A before it can generate the push for B, in case it needs to copy any newly-set cookies into the pseudo request for B.

Answers to these questions are fundamentally application-specific and server-specific.

## What Happens if the Server Uses the Wrong Request Headers?

If the server generates a push response using the wrong cookies, or more generally, using the wrong pseudo-request headers, the HTTP/2 spec does not explicitly say what should happen. Current browsers match pushed responses to client requests using the URL only, without matching the Vary header in the pushed response with headers from the client request, although some developers consider this behavior wrong (see discussion here and here).

We suggest the following rule:

> *Servers should not push a resource unless: (1) the resource does not vary on any request headers, or (2) the server is confident that it knows the correct value for all headers the resource varies on.*

In the current state, where browsers do not validate headers when matching pushed responses to requests, breaking the above rule can result in broken pages. Further, it would be a bad idea to violate this rule even if browsers correctly validated request headers -- each time the server pushes a response that violates the above rule, it risks pushing a response that will not match any client requests, in which case the push is useless and wastes bandwidth.

# Rule #5: Consider Preload Instead of Push

The Link: rel=preload header is a way to tell the browser that a resource will be needed for the current document. This lets the browser request resources before they are discovered through evaluation of the page. The HTML will be received immediately after the preload header. Therefore, unlike for push, there is little value in preloading resources that will be discovered quickly by the browser's preload scanner.

Unlike push, where requests are initiated by the server, preloaded requests are initiated by the browser using the browser's standard request flow. This sidesteps issues regarding caching and cookies (see rules #3 and #4). Further, cross-domain resources can be preloaded but not pushed. However, issues around scheduling and prioritization of requests remain: just like push, it is possible that preload requests will take bandwidth from more important requests. In other words, rule #2 applies to preload too.[4]

Preload has a single drawback compared to push: there is an extra RTT during which the server has to wait for the browser's requests. Note that the cwnd rule (rule #1) does not apply to preload. In fact, we see push and preload as complementary:

- Push can be used to fill the initial cwnd after downloading the page's HTML, which preload cannot do. If the pushed resources plus the HTML are smaller or equal to one cwnd, there is no worry about the pushed resources interfering with the other requests scheduled (rule #1).

- Push can be used to let the browser know of new resources that were discovered by the server when serving requests other than HTML requests (e.g.. the server could scan a CSS file for @imports statements). In this scenario, it's likely that the connection is already being used to serve concurrent requests, meaning the network may not be idle. We recommend pushing in this scenario only if it is certain that no higher-priority requests are in flight. Otherwise, push can degrade performance by mis-ordering requests (rule #2). If push is used in this scenario, there is no reason to continue pushing resources beyond one cwnd (rule #1). It can be difficult to ensure that push will help in this scenario. We suggest using preload instead.

- Preload should be used to inform the browser of resources it cannot learn of easily from scanning the HTML. Since preload requests are likely to interfere with non-preload requests, it is essential to make sure that preloaded resources are critical and that downloading them early actually improves key metrics.

[4] Moreover, the preload spec has not yet finalized the semantics for how preload requests should be prioritized relative to other requests. See the discussion here.

# Case Studies on Actual Pages

We ran experiments on a few arbitrarily-selected HTTP web pages. We don't claim this set of pages is representative-- these are intended to be case studies, not large-scale conclusions. Our target metric is SpeedIndex as computed by WPT. Each page was served from our replay server running on GCE to minimize variability from run-to-run. Our replay server doesn't support DNS hijacking, so we configured Chrome to use the replay server as an HTTP-to-HTTP/2 proxy server. This setup has several differences compared to a real-world deployment:

- HTTPS requests are not served through the replay server. This limits us to mostly-HTTP pages. Note that there are very few entirely-HTTP pages since ads are commonly served over HTTPS. This means some pages experienced replay nondeterminism due to nondeterministic ad selection, though this effect should be averaged away given enough runs.

- Chrome opens a single H2 connection to the replay server since it's configured to use the replay server as a proxy. In a real deployment, Chrome would need to open a new H2 connection for each authority (e.g., for each host).

- Developers will typically have control of how the web page is constructed and may change its structure to better take advantage of push.

All page loads used a cold browser cache and a warm TCP connection to the replay server. Since our server does not dynamically generate responses (all responses are copied directly from the replay archive), we can ignore the cookie problem mentioned in rule #4.

To prepare the experiments, we ran each page through a program that computes the fetch dependency graph (as defined at the beginning of this document) along with the evaluation order of CSS and JS files need by the page. We discarded dynamic URLs from the fetch graph, where dynamic URLs were identified from two consecutive loads of a given page: if a URL is fetched in one load but not the other, it is dynamic.

After constructing fetch graphs, we loaded each page in four configurations to evaluate the best practices suggested by our rules of thumb:

- **NoOpt**
  Push and preload are both disabled.

- **Push**
  When resource R is requested, we lookup the immediate children of R in the fetch graph, sort those children in evaluation order, then push the first N resources from this order

such that the cumulative size of those N resources just exceeds the BDP. We ignore cross-origin children of R since servers are not allowed to push resources for which they are not authoritative.

- **Preload**
  We inject the Link:rel=preload header into the HTML response to preload all resources that are *hidden*, meaning they cannot be learned from parsing the HTML. Further, we restrict to resources that are *critical*, meaning they are required to render above-the-fold content. The *critical* restriction ensures that preloaded URLs are all high-priority -- this is necessary since there is currently no good way to specify relative priorities for preloaded requests (see the footnote in the discussion of rule #5).

- **PushAndPreload**
  Combines Push with Preload. Following our suggestion in rule #5, we apply Push to the main HTML request only -- we rely on Preload to make the browser aware of other resources hidden further down the fetch graph.

The list of resources to push or preload was generated automatically based on the dependency graph. Results follow. All experiments were done on the same simulated 3G network described earlier. Results may vary on different networks (recall the discussion at the end of rule #1). We measure SpeedIndex for each configuration, with the relative improvement over NoOpt shown in parentheses. We consider all changes under 3% within the margin of error (they are rounded down to 0%).

| **Page** | **Median SpeedIndex (ms) over 50 runs<br>3G, Warm Connection, BDP = 35KB** | | | |
|---|---|---|---|---|
| | **NoOpt** | **Push<br>(Diff to NoOpt)** | **Preload<br>(Diff to NoOpt)** | **PushAndPreload<br>(Diff to NoOpt)** |
| artofliving | 7557 | 7744 (0%) | 6269 (+17%) | 6231 (+17%) |
| t24mobile | 2319 | 2097 (+9%) | 2048 (+11%) | 2067 (+10%) |
| bleacherreport | 10671 | 10455 (0%) | 10609 (0%) | 10655 (0%) |
| bollywoodhungama | 2809 | 2421 (+13%) | 2816 (0%) | 2455 (+12%) |
| vguard | 5712 | 5394 (+5%) | 5716 (0%) | 5708 (0%) |
| pornhub | 4031 | 4009 (0%) | 3550 (+11%) | 3537 (+12%) |

Throughout this doc we have described pathological cases where server push can hurt performance. We designed our “rules of thumb” to avoid these cases. The table above suggests we have succeeded -- there are no cases where Push hurts performance, although it doesn’t always help. Next, we comment on each page individually:

## artofliving

This page sees no improvement from Push. This page uses a *lot* of bandwidth, nearly 2MB, from over 200 total requests. Much of this bandwidth is needed to render above-the-fold content (e.g., there are 37 CSS files imported from the <head> of the HTML). Without push, the network is idle for only about ½ RTT after the initial HTML is downloaded. It then takes over 50 RTTs until the page is downloaded. This means we expect at most a 1% improvement from Push, which is within our measurement error.

However, we do see a significant improvement from Preload. We preload just two CSS files that are imported by a render-blocking script. Both of these CSS files are tiny (<1KB). How can preloading two tiny CSS files improve SpeedIndex by nearly 20%? Digging into the waterfalls, we noticed the following:

- For Preload, the two CSS files were requested at the beginning of the pageload and each request completed in under 500ms. This is expected.
- For NoOpt, the two CSS files were requested in the middle of the pageload (expected) but it took 3s for each request to complete (unexpected!).

It should not take 3s to download each CSS file in the NoOpt case. The CSS files are important, because they are imported by a render-blocking script, so Chrome moves them to the top of the HTTP/2 priority tree. We verified that our replay server respected this priority -- as soon as it received requests for those CSS files, it wrote the responses to the kernel immediately. Why did it take 3s to download these responses? After more digging, we realized the TCP cwnd and the kernel send buffer had both exploded by the time those CSS files were requested:

- Near the beginning of the pageload, cwnd was around 25 packets (35KB at 1500 bytes/packet) and the kernel send buffer was 85KB.
- When the CSS files were requested, cwnd had increased to 50 packets (75KB) and the kernel send buffer was 435KB.

This means the kernel had buffered 400KB that must be flushed before the CSS files could be sent (not counting 35KB already on the wire). With BDP=35KB, it takes 12 RTTs to flush this buffer, or about 2.5s. Yikes! This is a classic bufferbloat scenario. Not only is the kernel's buffer too big, but it also turns out that WPT's traffic shaper has a 100-packet buffer. (WPT's buffer is not a bug -- it is intended to emulate the bloated buffers found in most ISP networks.)

This is a case where Preload "fixes" bufferbloat by making important requests early, before bufferbloat has taken effect (recall the discussion in rule #2). We believe Push would have the same effect had we decided to push those CSS files, although we have not tested this to verify.

## t24mobile

This page sees a 10% improvement from both Push and Preload. Push sends one file, style.css, which is the first stylesheet referenced in the HTML. Note that style.css is not preloaded because it is mentioned in the static HTML, hence there is no reason to preload it. However, Preload loads a CSS from fonts.googleapis.com, which happens to be loaded by style.css. It seems the key is downloading that fonts file quickly. Push does this by making style.css available immediately for parsing. Preload does this by requesting the fonts file concurrently with style.css. Both approaches work equally well.

## bleacherreport

Like the artofliving page, this page is very large (over 3MB of assets). Further, the main HTML file is 41KB, which exceeds our BDP of 35KB, which means there is no idle time to fill after the main HTML response. There is some idle time later in the page load while ads and other analytics requests are loading. However, these requests have little impact on SpeedIndex because the page is essentially fully loaded above-the-fold by this point. Further, although there are five cases where we can push ads assets (mostly we can push ads script “bar.js” when it is loaded by ads script “foo.js”), the majority of ads requests are HTTPS, which our replay server doesn’t support, and/or cross-domain, which we cannot push.

We noticed this same two-phase pattern for most pages: The first phase loads the main HTML and the majority of the above-the-fold content. This phase is largely bandwidth-limited with relatively little idle network time. (Pages with large HTML files have no idle time, while pages with small HTML files have some idle time.) The second phase loads ads and analytics requests. This phase is largely latency-limited with many redirects and other chains of dependencies. Unfortunately, many hops in these dependence chains are cross-origin, meaning they cannot be optimized with server push.

## bollywoodhungama

This is an example where the main HTML file is small (9KB), most CSS and JS files in the <head> section are also small, and none of those resources are cross-origin. We are able to fit the main HTML plus six pushed resources within the first BDP. Basically, this is the textbook case for Push. Hence, we see a 10%+ improvement when using Push.

Interestingly, the first time we ran this experiment, performance was 10% *worse* with Push rather than 10% better. After digging, we realized the pushed resources were being sent on the wire before the main HTML response. This delayed parsing of the HTML, which delayed other requests, which ultimately delayed rendering. This is the resource ordering problem described by rule #2.

To explain how this happened, we need to explain how our replay server is implemented. Our server is written in Go, which uses a “goroutine” (i.e., thread) per request. Each request handler generates HTTP responses by calling “Write” methods on a ResponseWriter object. A special “Push” method generates a PUSH_PROMISE frame and spawns a concurrent request handler to serve the pushed response. The main HTML handler makes a sequence of “Push” calls to initiate pushes, followed by “Write” calls to generate the response. Note that “Push” must happen before “Write”, otherwise there is a race where the client receives and parses the HTML (and sends requests) before it receives any PUSH_PROMISE frames. This race is actually documented in the HTTP/2 spec. Hence, by the time the main HTML handler gets to the first “Write” call, the push handlers are already running concurrently.

Our server respects HTTP/2 priorities, which means it should prioritize the HTML before the pushes. However, our response scheduler has a dilemma -- if a push handlers calls “Write” before the HTML handler calls “Write”, should the scheduler wait for the HTML handler, or should it eagerly send the data it has already received from the push handler? Our original implementation sent data eagerly. Unfortunately, multiple push handlers frequently won the race with the HTML handler, causing pushes to be sent before the HTML. After realizing this, we changed our implementation to strictly adhere to priorities. Although this implementation has proved sufficient for our experiments so far, we are not convinced it is the right implementation in the general case.

## vguard

This page sees a small improvement from Push but none from Preload or PushAndPreload. The HTML page is much larger than BDP (70KB) so we do not push any resources with the HTML. The only pushes happen in the middle of the pageload: with two CSS files, we push a few small images that are referenced by those files. All of these images appear above-the-fold. We were unable to determine exactly why Push performs faster than NoOpt for this page, though we believe this is another bufferbloat scenario similar to the artofliving example above (except in this case, bufferbloat is less severe since the page is smaller).

For Preload, the page has no *hidden* and *critical* resources, so nothing is preloaded. For PushAndPreload, since the HTML is larger than BDP, nothing is pushed. Hence, both of these configurations are equivalent to NoOpt.

## pornhub

This page sees improvements from Preload but not Push. The page loads all of its assets from a CDN, which effectively disables push since those assets are cross-origin. Further, the HTML page is 49KB, which exceeds our BDP of 35KB, so even if we could push cross-origin resources, there is no idle time that push can fill. The Preload configuration preloads a single file, a render-blocking CSS from https://fonts.googleapis.com/, which is loaded by a script.

## Straw Man Comparison

Lastly, we show results from a straw man comparison that uses two new configurations:

- **PushStaticWithHTML**
  This is intended to emulate a naive developer who wants to "push all the resources!" (they are not aware of rule #1). In this configuration, we push all same-origin resources that are statically defined in the main HTML. We follow rule #2: resources are pushed in evaluation order. We also assume the service has been at least partially redesigned for HTTP/2 -- specifically, we assume that first-party resources are delivered from the same host (no CDNs), which considerably increases the number of pushable resources for some pages. Third-party resources are still ineligible for push.

  Note that we push resources on first-party CDNs, but not on third-party CDNs. We made this distinction by inspecting the URLs manually. For example, given "www.foo.com", "cdn.foo.com" is a first-party CDN while "cdn.bootstrap.com" is a third-party CDN.

- **PushStaticWithHTMLOneBDP**
  This is like PushStaticWithHTML, except that it respects rule #1 and sends no more than 1 BDP worth of resources.

Results follow. Note that the numbers for NoOpt are slightly different from those in the above table because the numbers below were computed in a separate run and with different replay archives (meaning the pages may have changed):

| Page | First-Party CDN? | Median SpeedIndex (ms) over 50 runs 3G, Warm Connection, BDP = 35KB | | |
|---|---|---|---|---|
| | | NoOpt | PushStaticWithHTML OneBDP (Diff to NoOpt) | PushStaticWithHTML (Diff to NoOpt) |
| artofliving | Yes | 7720 | 7740 (0%) | 9872 (-27%) |
| t24mobile | No | 2286 | 2078 (+9%) | 2962 (-29%) |
| bleacherreport | Yes | 13487 | - | 13311 (0%) |
| bollywoodhungama | Yes | 5479 | 4857 (+11%) | 5058 (+7%) |
| vguard | No | 5687 | - | 7176 (-26%) |
| pornhub | Yes | 4538 | - | 4623 (0%) |

Note that PushStaticWithHTMLOneBDP is equivalent to NoOpt for all pages where the main HTML is larger than BDP (which disables push, due to rule #1). These cases are marked with a dash.

PushStaticWithHTML always pushes at least one resource, often more than ten. For two pages, it performs no better than NoOpt. Both of these pages have a main HTML that is larger than BDP, so we expect no improvement from push. For three pages, PushStaticWithHTML is worse than NoOpt. We believe this happens due to ordering problems caused by pushing too many requests, particularly for larger pages (like artofliving) where bufferbloat can have a significant impact. For the bollywoodhungama page, PushStaticWithHTML performs better than NoOpt. This is the page we called a “textbook case for push” in the prior section. However, it performs no better than PushStaticWithHTMLOneBDP -- it actually performs very slightly worse -- demonstrating once again that there is no benefit to pushing more than is needed to fill idle network time.