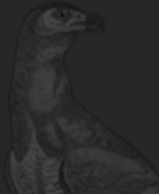




Optimizing Application Delivery

HTTP, CHAPTER 13



High-performance browser networking relies on a host of networking technologies (Figure 13-1), and the overall performance of our applications is the sum total of each of their parts.

We cannot control the network weather between the client and server, nor the client hardware or the configuration of their device, but the rest is in our hands: TCP and TLS optimizations on the server, and dozens of application optimizations to account for the peculiarities of the different physical layers, versions of HTTP protocol in use, as well as general application best practices. Granted, getting it all right is not an easy task, but it is a rewarding one! Let's pull it all together.

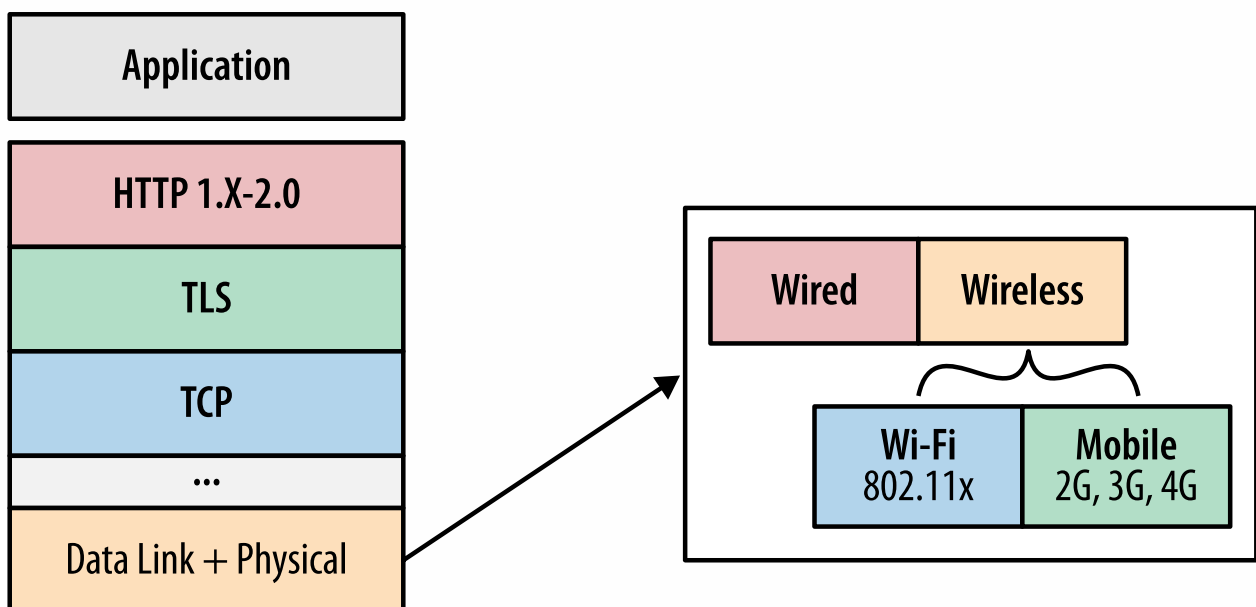


Figure 13-1. Optimization layers for web application delivery

Optimizing Physical and Transport Layers



The physical properties of the communication channel set hard performance limits on every application: speed of light and distance between client and server dictate the propagation latency, and the choice of medium (wired vs. wireless) determines the processing, transmission, queuing, and other delays incurred by each data packet. In fact, the performance of most web applications is limited by latency, not bandwidth, and while bandwidth speeds will continue to increase, unfortunately the same can't be said for latency:

- [The Many Components of Latency](#)
- [Delivering Higher Bandwidth and Lower Latencies](#)



As a result, while we cannot make the bits travel any faster, it is crucial that we apply all the possible optimizations at the transport and application layers to eliminate unnecessary roundtrips, requests, and minimize the distance traveled by each packet—i.e., position the servers closer to the client.

Every application can benefit from optimizing for the unique properties of the physical layer in wireless networks, where latencies are high, and bandwidth is always at a premium. At the API layer, the differences between the wired and wireless networks are entirely transparent, but ignoring them is a recipe for poor performance. Simple optimizations in how and when we schedule resource downloads, beacons, and the rest can translate to significant impact on the experienced latency, battery life, and overall user experience of our applications:

- [Optimizing for WiFi Networks](#)
- [Optimizing for Mobile Networks](#)

Moving up the stack from the physical layer, we must ensure that each and every server is configured to use the latest TCP and TLS best practices. Optimizing the underlying protocols ensures that each client can get the best performance—high throughput and low latency—when communicating with the server:

- [Optimizing for TCP](#)
- [Optimizing for TLS](#)

Finally, we arrive at the application layer. By all accounts and measures, HTTP is an incredibly successful protocol. After all, it is the common language between billions of clients and servers, enabling the modern Web. However, it is also an imperfect protocol, which means that we must take special care in how we architect our applications:

- We must work around the limitations of HTTP/1.x.
- We must leverage new performance capabilities of HTTP/2.
- We must be vigilant about applying the evergreen performance best practices.

Note

The secret to a successful web performance strategy is simple: invest into monitoring and measurement tools to identify problems and regressions (see [Synthetic and Real-User Performance Measurement](#)), link business goals to performance metrics, and optimize from there—i.e., treat performance as a feature.

Evergreen Performance Best Practices





applications should always seek to eliminate or reduce unnecessary network latency and minimize the number of transferred bytes. These two simple rules are the foundation for all of the evergreen performance best practices:

Reduce DNS lookups

Every hostname resolution requires a network roundtrip, imposing latency on the request and blocking the request while the lookup is in progress.

Reuse TCP connections

Leverage connection keepalive whenever possible to eliminate the TCP handshake and slow-start latency overhead; see [Slow-Start](#).

Minimize number of HTTP redirects

HTTP redirects impose high latency overhead—e.g., a single redirect to a different origin can result in DNS, TCP, TLS, and request-response roundtrips that can add hundreds to thousands of milliseconds of delay. The optimal number of redirects is zero.

Reduce roundtrip times

Locating servers closer to the user improves protocol performance by reducing roundtrip times (e.g., faster TCP and TLS handshakes), and improves the transfer throughput of static and dynamic content; see [Uncached Origin Fetch](#).

Eliminate unnecessary resources

No request is faster than a request not made. Be vigilant about auditing and removing unnecessary resources.

By this point, all of these recommendations should require no explanation: latency is the bottleneck, and the fastest byte is a byte not sent. However, HTTP provides some additional mechanisms, such as caching and compression, as well as its set of version-specific performance quirks:

Cache resources on the client

Application resources should be cached to avoid re-requesting the same bytes each time the resources are required.

Compress assets during transfer

Application resources should be transferred with the minimum number of bytes: always apply the best compression method for each transferred asset.

Eliminate unnecessary request bytes

Reducing the transferred HTTP header data (e.g., HTTP cookies) can save entire roundtrips of network latency.

Parallelize request and response processing

Request and response queuing latency, both on the client and server, often goes unnoticed, but contributes significant and unnecessary latency delays.

Apply protocol-specific optimizations



access domain, and more by contrast, HTTP/2 performs best when a single connection is used, and HTTP/1.x specific optimizations are removed.

Each of these warrants closer examination. Let's dive in.

Cache Resources on the Client



The fastest network request is a request not made. Maintaining a cache of previously downloaded data allows the client to use a local copy of the resource, thereby eliminating the request. For resources delivered over HTTP, make sure the appropriate cache headers are in place:

- `Cache-Control` header can specify the cache lifetime (`max-age`) of the resource.
- `Last-Modified` and `ETag` headers provide validation mechanisms.

Whenever possible, you should specify an explicit cache lifetime for each resource, which allows the client to use a local copy, instead of re-requesting the same object all the time. Similarly, specify a validation mechanism to allow the client to check if the expired resource has been updated: if the resource has not changed, we can eliminate the data transfer.

Finally, note that you need to specify both the cache lifetime and the validation method! A common mistake is to provide only one of the two, which results in either redundant transfers of resources that have not changed (i.e., missing validation), or redundant validation checks each time the resource is used (i.e., missing or unnecessarily short cache lifetime).

Note

For hands-on advice on optimizing your caching strategy, see the ["HTTP caching" section on Google's Web Fundamentals](#) .

Web Caching on Smartphones: Ideal vs. Reality



Caching of HTTP resources has been one of the top performance optimizations ever since the very early versions of the HTTP protocol. However, while seemingly everyone is aware of its benefits, real-world studies continue to discover that it is nonetheless an often-omitted optimization! A recent joint study by AT&T Labs Research and University of Michigan reports:



Our findings suggest that redundant transfers contribute 18% and 20% of the total HTTP traffic volume in the two datasets. Also they are responsible for 17% of the bytes, 7% of the radio energy consumption, 6% of the signaling load, and 9% of the radio resource utilization of all cellular data traffic in the second dataset. Most of such redundant transfers are caused by the smartphone web caching implementation that does not

*Web Caching on Smartphones, MobiSys 2012*

Is your application fetching unnecessary resources over and over again? As evidence shows, that's not a rhetorical question. Double-check your application and, even better, add some tests to catch any regressions in the future.

Compress Transferred Data




Leveraging a local cache allows the client to avoid fetching duplicate content on each request. However, if and when the resource must be fetched, either because it has expired, it is new, or it cannot be cached, then it should be transferred with the minimum number of bytes. Always apply the best compression method for each asset.

The size of text-based assets, such as HTML, CSS, and JavaScript, can be reduced by 60%–80% on average when compressed with Gzip. Images, on the other hand, require more nuanced consideration:

- Images often carry a lot of metadata that can be stripped—e.g., EXIF.
- Images should be sized to their display width to minimize transferred bytes.
- Images can be compressed with different lossy and lossless formats.

Images account for over half of the transferred bytes of an average page, which makes them a high-value optimization target: the simple choice of an optimal image format can yield dramatically improved compression ratios; lossy compression methods can reduce transfer sizes by orders of magnitude; sizing the image to its display width will reduce both the transfer and memory footprints (see [Calculating Image Memory Requirements](#)) on the client. Invest into tools and automation to optimize image delivery on your site.

Note

For hands-on advice on reducing the transfer size of text, image, webfont, and other resources, see the ["Optimizing Content Efficiency" section on Google's Web Fundamentals](#) .

Eliminate Unnecessary Request Bytes



HTTP is a stateless protocol, which means that the server is not required to retain any information about the client between different requests. However, many applications require state for session management, personalization, analytics, and more. To enable this functionality, the HTTP State Management Mechanism (RFC 2965) extension allows any website to associate



then automatically appended onto every request to the origin within the `Cookie` header.

The standard does not specify a maximum limit on the size of a cookie, but in practice most browsers enforce a 4 KB limit. However, the standard also allows the site to associate many cookies per origin. As a result, it is possible to associate tens to hundreds of kilobytes of arbitrary metadata, split across multiple cookies, for each origin!

Needless to say, this can have significant performance implications for your application. Associated cookie data is automatically sent by the browser on each request, which, in the worst case can add entire roundtrips of network latency by exceeding the initial TCP congestion window, regardless of whether HTTP/1.x or HTTP/2 is used:

- In HTTP/1.x, all HTTP headers, including cookies, are transferred uncompressed on each request.
- In HTTP/2, headers are compressed with HPACK, but at a minimum the cookie value is transferred on the first request, which will affect the performance of your initial page load.

Cookie size should be monitored judiciously: transfer the minimum amount of required data, such as a secure session token, and leverage a shared session cache on the server to look up other metadata. And even better, eliminate cookies entirely wherever possible—chances are, you do not need client-specific metadata when requesting static assets, such as images, scripts, and stylesheets.

Note

When using HTTP/1.x, a common best practice is to designate a dedicated "cookie-free" origin, which can be used to deliver responses that do not need client-specific optimization.

Parallelize Request and Response Processing



To achieve the fastest response times within your application, all resource requests should be dispatched as soon as possible. However, another important point to consider is how these requests will be processed on the server. After all, if all of our requests are then serially queued by the server, then we are once again incurring unnecessary latency. Here's how to get the best performance:

- Reuse TCP connections by optimizing connection keepalive timeouts.
- Use multiple HTTP/1.1 connections where necessary for parallel downloads.
- Upgrade to HTTP/2 to enable multiplexing and best performance.
- Allocate sufficient server resources to process requests in parallel.

Without connection keepalive, a new TCP connection is required for each HTTP request, which incurs significant overhead due to the TCP handshake and slow-start. Make sure to identify and



prematurely. With that in place, and to get the best performance, use HTTP/2 to allow the client and server to reuse the same connection for all requests. If HTTP/2 is not an option, use multiple TCP connections to achieve request parallelism with HTTP/1.x.

Identifying the sources of unnecessary client and server latency is both an art and science: examine the client resource waterfall (see [Analyzing the Resource Waterfall](#)), as well as your server logs. Common pitfalls often include the following:

- Blocking resources on the client forcing delayed resource fetches; see [DOM, CSSOM, and JavaScript](#).
- Underprovisioned proxy and load balancer capacity, forcing delayed delivery of the requests (queuing latency) to the application servers.
- Underprovisioned servers, forcing slow execution and other processing delays.

Optimizing Resource Loading in the Browser



The browser will automatically determine the optimal loading order for each resource in the document, and we can both assist and hinder the browser in this process:

- We can provide hints to assist the browser; see [Browser Optimization](#).
- We can hinder by hiding resources from the browser.

Modern browsers are designed to scan the contents of HTML and CSS files as efficiently and as soon as possible. However, the document parser is also frequently blocked while waiting for a script or other blocking resources to download before it can proceed. During this time, the browser uses a "preload scanner," which speculatively looks ahead in the source for resource downloads that could be dispatched early to reduce overall latency.

Note that the use of the preload scanner is a speculative optimization, and it is used only when the document parser is blocked. However, in practice, it yields significant benefits: based on experimental data with Google Chrome, it offers a ~20% improvement in page loading times and rendering speeds!

Unfortunately, these optimizations do not apply for resources that are scheduled via JavaScript; the preload scanner cannot speculatively execute scripts. As a result, moving resource scheduling logic into scripts may offer the benefit of more granular control to the application, but in doing so, it will hide the resource from the preload scanner, a trade-off that warrants close examination.

Optimizing for HTTP/1.x



The order in which we optimize HTTP/1.x deployments is important: configure servers to deliver the best possible TCP and TLS performance, and then carefully review and apply mobile and evergreen application best practices: measure, iterate.



the application, evaluate whether the application can benefit from applying HTTP/1.x specific optimizations (read, *protocol workarounds*):

Leverage HTTP pipelining

If your application controls both the client and the server, then pipelining can help eliminate unnecessary network latency; see [HTTP Pipelining](#).

Apply domain sharding

If your application performance is limited by the default six connections per origin limit, consider splitting resources across multiple origins; see [Domain Sharding](#).

Bundle resources to reduce HTTP requests

Techniques such as concatenation and spriting can both help minimize the protocol overhead and deliver pipelining-like performance benefits; see [Concatenation and Spriting](#).

Inline small resources

Consider embedding small resources directly into the parent document to minimize the number of requests; see [Resource Inlining](#).

Pipelining has limited support, and each of the remaining optimizations comes with its set of benefits and trade-offs. In fact, it is often overlooked that each of these techniques can hurt performance when applied aggressively, or incorrectly; review [HTTP/1.X](#) for an in-depth discussion.

Note

HTTP/2 eliminates the need for all of the above HTTP/1.x workarounds, making our applications both simpler and more performant. Which is to say, the best optimization for HTTP/1.x is to deploy HTTP/2.

Optimizing for HTTP/2



HTTP/2 enables more efficient use of network resources and reduced latency by enabling request and response multiplexing, header compression, prioritization, and more—see [Design and Technical Goals](#). Getting the best performance out of HTTP/2, especially in light of the one-connection-per-origin model, requires a well-tuned server network stack. Review [Optimizing for TCP](#) and [Optimizing for TLS](#) for an in-depth discussion and optimization checklists.

Next up—surprise—apply the evergreen application best practices: send fewer bytes, eliminate requests, and adapt resource scheduling for wireless networks. Reducing the amount of data transferred and eliminating unnecessary network latency are the best optimizations for any application, web or native, regardless of the version or type of the application and transport protocols in use.



With HTTP/2 we are no longer constrained by limited parallelism. Requests are cheap, and both requests and responses can be multiplexed efficiently. These workarounds are no longer necessary and omitting them can improve performance.

Eliminate Domain Sharding



HTTP/2 achieves the best performance by multiplexing requests over the same TCP connection, which enables effective request and response prioritization, flow control, and header compression. As a result, the optimal number of connections is exactly one and domain sharding is an anti-pattern.

HTTP/2 also provides a TLS connection-coalescing mechanism that allows the client to coalesce requests from different origins and dispatch them over the same connection when the following conditions are satisfied:

- The origins are covered by the same TLS certificate—e.g., a wildcard certificate, or a certificate with matching "Subject Alternative Names."
- The origins resolve to the same server IP address.

For example, if `example.com` provides a wildcard TLS certificate that is valid for all of its subdomains (i.e., `*.example.com`) and references an asset on `static.example.com` that resolves to the same server IP address as `example.com`, then the HTTP/2 client is allowed to reuse the same TCP connection to fetch resources from `example.com` and `static.example.com`.

An interesting side effect of HTTP/2 connection coalescing is that it enables an HTTP/1.x friendly deployment model: some assets can be served from alternate origins, which enables higher parallelism for HTTP/1 clients, and if those same origins satisfy the above criteria then the HTTP/2 clients can coalesce requests and reuse the same connection. Alternatively, the application can be more hands-on and inspect the negotiated protocol and deliver alternate resources for each client: with sharded asset references for HTTP/1.x clients and with same-origin asset references for HTTP/2 clients.

Depending on the architecture of your application you may be able to rely on connection coalescing, you may need to serve alternate markup, or you may use both techniques as necessary to provide the optimal HTTP/1.x and HTTP/2 experience. Alternatively, you might consider focusing on optimizing HTTP/2 performance only; the client adoption is growing rapidly, and the extra complexity of optimizing for both protocols may be unnecessary.

Note

Due to third-party dependencies it may not be possible to fetch all the resources via the same TCP connection—that's OK. Seek to minimize the number of origins regardless of the protocol



Minimize Concatenation and Image Spriting



Bundling multiple assets into a single response was a critical optimization for HTTP/1.x where limited parallelism and high protocol overhead typically outweighed all other concerns—see [Concatenation and Spriting](#). However, with HTTP/2, multiplexing is no longer an issue, and header compression dramatically reduces the metadata overhead of each HTTP request. As a result, we need to reconsider the use of concatenation and spriting in light of its new pros and cons:

- Bundled resources may result in unnecessary data transfers: the user might not need all the assets on a particular page, or at all.
- Bundled resources may result in expensive cache invalidations: a single updated byte in one component forces a full fetch of the entire bundle.
- Bundled resources may delay execution: many content-types cannot be processed and applied until the entire response is transferred.
- Bundled resources may require additional infrastructure at build or delivery time to generate the associated bundle.
- Bundled resources may provide better compression if the resources contain similar content.

In practice, while HTTP/1.x provides the mechanisms for granular cache management of each resource, the limited parallelism forced us to bundle resources together. The latency penalty of delayed fetches outweighed the costs of decreased effectiveness of caching, more frequent and more expensive invalidations, and delayed execution.

HTTP/2 removes this unfortunate trade-off by providing support for request and response multiplexing, which means that we can now optimize our applications by delivering more granular resources: each resource can have an optimized caching policy (expiry time and revalidation token) and be individually updated without invalidating other resources in the bundle. In short, HTTP/2 enables our applications to make better use of the HTTP cache.

That said, HTTP/2 does not eliminate the utility of concatenation and spriting entirely. A few additional considerations to keep in mind:

- Files that contain similar data may achieve better compression when bundled.
- Each resource request carries some overhead, both when reading from cache (I/O requests), and from the network (I/O requests, on-the-wire metadata, and server processing).

There is no single optimal strategy for all applications: delivering a single large bundle is unlikely to yield best results, and issuing hundreds of requests for small resources may not be the optimal



access patterns, and other criteria. To get the best results, gather measurement data for your own application and optimize accordingly.

Eliminate Roundtrips with Server Push



Server push is a powerful new feature of HTTP/2 that enables the server to send multiple responses for a single client request. That said, recall that the use of resource inlining (e.g., embedding an image into an HTML document via a data URI) is, in fact, a form of application-layer server push. As such, while this is not an entirely new capability for web developers, the use of HTTP/2 server push offers many performance benefits over inlining: pushed resources can be cached individually, reused across pages, canceled by the client, and more—see [Server Push](#).

With HTTP/2 there is no longer a reason to inline resources just because they are small; we're no longer constrained by the lack of parallelism and request overhead is very low. As a result, server push acts as a latency optimization that removes a full request-response roundtrip between the client and server—e.g., if, after sending a particular response, we know that the client will always come back and request a specific subresource, we can eliminate the roundtrip by pushing the subresource to the client.

Note

If the client does not support, or disables the use of server push, it will initiate the request for the same resource on its own—i.e., server push is a safe and transparent latency optimization.

Critical resources that block page construction and rendering (see [DOM](#), [CSSOM](#), and [JavaScript](#)) are prime candidates for the use of server push, as they are often known or can be specified upfront. Eliminating a full roundtrip from the critical path can yield savings of tens to hundreds of milliseconds, especially for users on mobile networks where latencies are often both high and highly variable.

- Server push, as its name indicates, is initiated by the server. However, the client can control how and where it is used by indicating to the server the maximum number of pushed streams that can be initiated in parallel by the server, as well as the amount of data that can be sent on each stream before it is acknowledged by the client. This allows the client to limit, or outright disable, the use of server push—e.g., if the user is on an expensive network and wants to minimize the number of transferred bytes, they may be willing to disable the latency optimization in favor of explicit control over what is fetched.
- Server push is subject to same-origin restrictions; the server initiating the push must be authoritative for the content and is not allowed to push arbitrary third-party content to the client. Consolidate your resources under the same origin (i.e., eliminate domain sharding) to enable more opportunities to leverage server push.



browser-initiated requests—i.e., they can be cached and reused across multiple pages and navigations! Leverage this to avoid having to duplicate the same content across different pages and navigations.

Note that even the most naive server push strategy that opts to push assets regardless of their caching policy is, in effect, equivalent to inlining: the resource is duplicated on each page and transferred each time the parent resource is requested. However, even there, server push offers important performance benefits: the pushed response can be prioritized more effectively, it affords more control to the client, and it provides an upgrade path towards implementing much smarter strategies that leverage caching and other mechanisms that can eliminate redundant transfers. In short, if your application is using inlining, then you should consider replacing it with server push.

Automating Performance Optimization via Server Push



How does the server determine which resources should be delivered via server push? The HTTP/2 standard does not specify any particular algorithm, and the server is free to implement custom strategies for each application.

For example, server-side application code can specify which resources should be pushed and when. This strategy requires explicit configuration but provides full control to the application developer. Alternatively, the server can learn the associated resources based on observed traffic patterns (e.g., by observing Referrer headers) and automatically initiate server push for related resources; use some mechanism to track or guess client's cache state and initiate push for missing resources; and so on.

Server push enables many new and previously not possible optimization opportunities. Check the documentation of your HTTP/2 server for how to enable, configure, and deploy the use of server push for your application.

Test HTTP/2 Server Quality



A naive implementation of an HTTP/2 server, or proxy, may "speak" the protocol, but without well implemented support for features such as flow control and request prioritization, it can easily yield less than optimal performance. For example, it might saturate the user's bandwidth by sending large low priority resources (such as images), while the browser is blocked from rendering the page until it receives higher priority resources (such as HTML, CSS, or JavaScript).

With HTTP/2 the client places a lot of trust on the server. To get the best performance, an HTTP/2 client has to be "optimistic": it annotates requests with priority information (see [Stream Prioritization](#)) and dispatches them to the server as soon as possible; it relies on the server to use the communicated dependencies and weights to optimize delivery of each response. A well-



additional and critical responsibilities that, previously, were out of scope.

Do your due diligence when testing and deploying your HTTP/2 infrastructure. Common benchmarks measuring server throughput and requests per second do not capture these new requirements and may not be representative of the actual experience as seen by your users when loading your application.

Optimizing Response Delivery with Request Prioritization

§

The purpose of request prioritization is to allow the client to express how it would prefer the server to deliver responses when there is limited capacity—e.g., the server may be ready to send multiple responses, but due to limited bandwidth it should prioritize sending some resources ahead of others.

- What if the server disregards all priority information?
- Should higher-priority streams always take precedence?
- Are there cases where different priority streams should be interleaved?

If the server disregards all priority information, then it runs the risk of causing unnecessary processing delays for the client—e.g., block the browser from rendering the page by sending images ahead of more critical CSS and JavaScript resources. However, delivering streams in a strict dependency order can also yield suboptimal performance as it may reintroduce the head-of-line blocking problem where a high priority but slow response may unnecessarily block delivery of other resources. As a result, a well-implemented server should give precedence to high priority streams, but it should also interleave lower priority streams if all higher priority streams are blocked.

[« Back to the Table of Contents](#)

Copyright © 2013 [Ilya Grigorik](#). Published by O'Reilly Media, Inc. Licensed under [CC BY-NC-ND 4.0](#).