Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 10, 2014

M. Piatek
W. Chan
Google

January 6, 2014

HTTP/2 Stream Dependencies
draft-chan-http2-stream-dependencies-00

Abstract

The existing HTTP/2 prioritization scheme relies purely on integer values to indicate priorities. This simple scheme misses critical support for priority grouping, and does not support other features like resource ordering. This draft proposes using stream dependencies to solve the lack of priority grouping, as well as provide other features.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 10, 2014.

Copyright Notice

Table of Contents

1. Introduction

This document proposes changes to HTTP/2 to support stream dependencies. During a pageload, the server uses dependencies to improve performance by allocating bandwidth capacity to the most important resource transfers first.

The remainder of this document describes the motivation for dependencies, protocol changes to support them, and examples of how those mechanisms can be used by the browser. We conclude with a discussion of the client and server policies afforded by expressing dependency information in HTTP/2.

(Note that flow control is the subject of a separate document and is out of scope here.)

2. Motivation

Dependencies allow an HTTP/2 server to allocate bandwidth capacity efficiently in several common use-cases:

Specifying an ordering of resource transfers
: Sharing bandwidth between resources transfer often degrades performance, e.g., when transferring two Javascript resources that cannot be executed until transfer is complete, or two video chunks that will be played back-to-back. In these circumstances, the browser may wish to specify an ordering --- HTML before script1.js before script2.js, for example, or video_chunk1 before video_chunk2.

Reacting to document parsing
: Because the browser's document parser blocks while waiting for script and style resource transfers to complete, many resource requests will be issued by simply scanning the tokenized HTML. (For more background, see [PRELOADSCANNER])

  As the document parser proceeds, it may learn of higher priority resources. For example, if a script a.js uses document.write to embed another script, b.js, the transfer of b.js should preempt other in-flight resource transfers since the receipt of b.js blocks page layout. Similarly, image transfers that will be styled with display: none should be deferred to prioritize visible content.

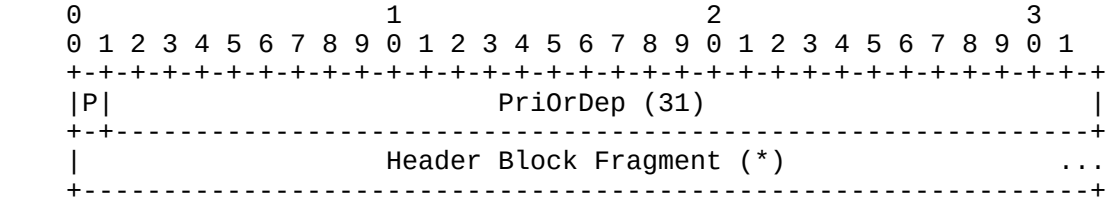
Reacting to user behavior
: In the case of HTTP/2 proxies, a single TCP connection may multiplex several sites in several tabs. Changing tabs may reorder the relative importance of outstanding streams, e.g., concurrent AJAX requests or page loads. Similarly, a proxy server may coalesce streams to a common origin onto a single connection. As the set of outstanding requests and users changes, the relative importance of each user's streams may change as well.

Server Push
: Server push can improve performance by eliminating round trips, but it may degrade performance if a pushed stream preempts a more important transfer. For example, a Javascript transfer may block layout and be high priority, or it may be a low-priority async request. Dependencies provide a hint to the server about the relative importance of pushed resources.

3. Protocol Changes

Dependencies are expressed using the existing optional priority field the HEADERS frames and in PRIORITY frames. To ensure clients and servers have consistent view of active streams, we propose the FIN_ACK frame. The section concludes with a set of invariants that clients and servers must maintain when using these frames.

3.1. HEADERS frame



HEADERS Frame Payload

The HEADERS frame defines the following flags:

ORDERED (0x10): Bit 5 being set indicates that the dependency specified by PriOrDep is ordered. If this flag is unset, any dependency is treated as unordered.

Here, the 4 octets previously used by the unused bit and 31 bit Priority field in the HEADERS frame are reinterpreted. The unused bit is now known as the P bit, and the 31 bit Priority field is now

PriOrDep.

P: A bit indicating whether the following PriOrDep bits specify a priority (P = 1) or a stream ID (P = 0) on which this new stream depends.

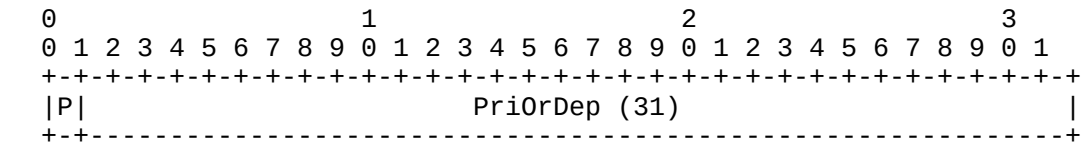
PriOrDep: Depending on the value of P, either the priority of the new stream or a stream ID on which this new stream depends.

The structure and semantics of the Header Block Fragment are unchanged.

P is exclusive; a stream may be assigned a priority or a parent dependency upon creation, but not both. If P = 0 and PriOrDep indicates a dependency; the value MUST correspond to an active stream.

Server push streams are assigned a priority or dependency id at the discretion of the server.

3.2. PRIORITY frame



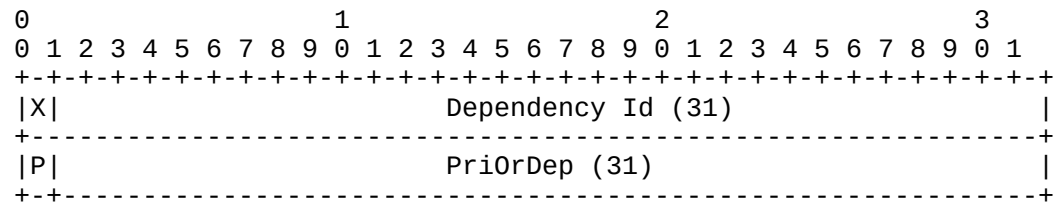
PRIORITY Frame Payload

The PRIORITY frame defines the following flags:

ORDERED (0x10): Bit 5 being set indicates that the dependency specified by PriOrDep is ordered. If this flag is unset, any dependency is treated as unordered.

As in HEADERS, the Priority field is changed to be a P/PriOrDep field indicating an update to the 31 bit Dependency Id specified in the header. We relabel the typical Stream Id here as Dependency Id to distinguish it as a referent.

To support batched updates of dependencies, an optional list of DependencyPriOrDep pairs with identical semantics may follow. The number of such pairs is determined by examining the frame length.



DependencyPriOrDep

3.3. END_STREAM_ACK frame

The END_STREAM_ACK frame has no payload. It is sent by a client to a server after receiving a frame with the END_STREAM flag set. The frame is used to ensure a consistent set of active streams between the client and the server. Consistency is required to maintain the protocol invariants described below.

4. Protocol invariants and definitions

Each stream has at most one dependency An update to a stream's dependent stream id replaces any existing dependency for the claimant. Specifying multiple dependency ids for a single stream in a PRIORITY frame is a protocol error.

Each stream is depended on by at most one stream An update to a stream's dependent stream id replaces any existing dependency on the target. Repeating a single dependency id in a PRIORITY frame is a protocol error.

Each dependency has a type: ordered or unordered Ordered dependencies indicate a sequential transfer preference with respect to the dependent stream id. Unordered dependencies indicate a concurrent transfer preference for the range of the dependency list with unordered dependency links.

For example, where <- indicates an ordered dependency and - indicates an unordered dependency

a.htm <- a.js <- 1.png - 2.png

indicates that a.html should preempt a.js which itself should preempt 1.png and 2.png, each of which should transfer concurrently, sharing capacity.

All frames with the END_STREAM flag set MUST be explicitly acknowledged by clients To ensure that the client and server have an identical view of active stream ids when specifying dependencies, we require that clients explicitly acknowledge frames with the END_STREAM flag set by sending END_STREAM_ACK. Servers MUST retain dependency relationships for a stream until its END_STREAM_ACK is received (or the session is closed). Explicit acknowledgements obviate timeouts for garbage collecting dependency state and enable clients and servers to have a consistent view of dependency relationships.

A dependency id MUST correspond to an active stream id An active stream id is one for which the client has not yet sent an END_STREAM_ACK frame. It is a protocol error to name a stream id as a dependency that is not active.

If a server receives an END_STREAM_ACK for a stream X on which another stream Y depends, it SHOULD update the dependency pointer for Y to reflect the removal of X. The rules for updating dependencies are:

1. If X does not depend on another stream id, Y inherits the priority of X.

2. If X does depend on another stream id W, Y inherits the dependency pointer from X to W.

For example, for dependencies

a.htm <- a.js <- 1.png - 2.png

where the server receives an END_STREAM_ACK for 1.png, the resulting dependencies would be

a.htm <- a.js <- 2.png

Of course, clients may reconfigure dependencies using whatever policy they wish by sending an explicit PRIORITY frame for stream Y before the END_STREAM_ACK for stream X.

Updating dependencies when overwriting values is analogous to list insertion. If stream Y depends on X and a HEADERS or PRIORITY frame is received indicating a dependency on X for stream Z, Z replaces Y as X’s dependent, and Y’s dependency is updated to Z with the same ordering as it had to X. For example, if

a.htm - 1.png

and the server receives a HEADERS frame for a.js with an ordered dependency on a.htm, the result is

a.htm <- a.js - 1.png

5. Examples

The combination of dependencies and priorities suffices to express serialized as well as concurrent transfer schedules. But, how should the browser choose dependencies and priorities when making requests? This question is best answered quantitatively. As a starting point, we consider the following policy in our examples:

1. Resource dependencies reflect parser-blocking order. Non-streaming resources are serialized; i.e., non-async scripts and styling.

2. Progressive resources (e.g., images) are transferred concurrently and configured to depend on parser-blocking resource transfers.

3. To ensure that the speculative parser can maintain enough in-flight requests to fill the pipe between the client and server, page HTML does not depend on other streams. (Although, a background tab should have lower priority.)

Concretely, suppose a HTTP/2 connection is multiplexing multiple tabs from a user connected to a HTTP/2 proxy, with parent pointers and priorities as shown below. (P6, for example, indicates a priority of 6.)

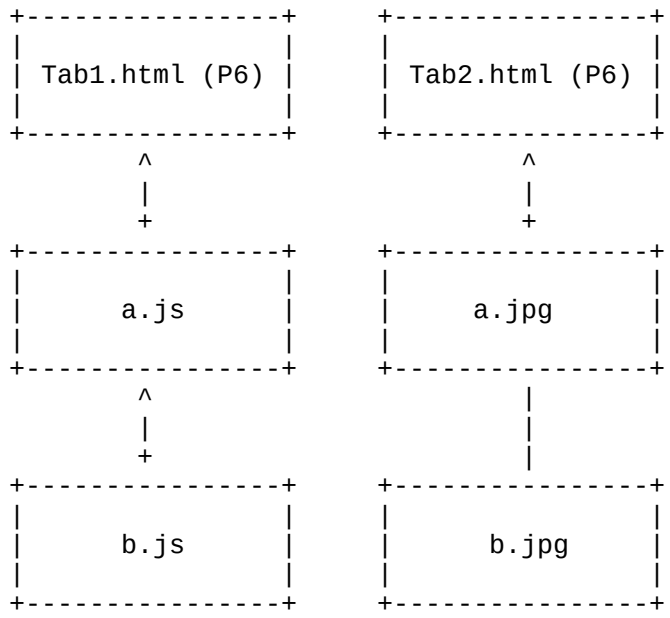


Figure 1: Multiple Tab Example

To color in this example, suppose that Tab 1 is the foreground tab, loading in parallel with Tab 2 in the background. Thus, its relatively higher weight. a.js and b.js are scripts required for the first tab and should be transferred serially (as scripts are executed in the order they are declared in the document, and are not parsed until transfer completes.) Thus, a.js depends on b.js depends on tab1.htm. In the background tab, two image transfers share capacity as both can be rendered progressively. Thus, the dependency between b.jpg and a.jpg is unordered, indicating that writes for the tab2.html stream should be scheduled first, but capacity may be shared between the streams for a.jpg and b.jpg.

When scheduling transfers, we consider a server that treats dependencies conceptually as lists. Recall that streams depend on and are depended on by at most one other stream. These can be treated as predecessor and successor ids. Stream writes are scheduled in two steps: 1) choosing a dependency list with at least one stream ready to write and 2) then selecting the stream to write by traversing the list. (An implementation might maintain ready queues of streams for efficiency, but we consider a simplified setting for clarity.)

Because the streams associated with the transfers of tab1 and tab2 have priorities rather than dependencies, they are always scheduled

before any dependent streams. But, bandwidth allocation between dependency lists remains proportional as defined by the relative priority of tab1 and tab2. For example, if the transfer of tab2.htm is in progress and tab1.htm (now complete) is ready and selected by the scheduler, a.js will be scheduled before tab2.htm completes. This process proceeds until all transfers in a list have completed.

5.1. Specifying an ordering of resource transfers and reacting to document parsing

We illustrate the need for both serial dependencies, concurrency, and reprioritization in these cases with a simple example.

Suppose site.com has index.htm:

```
<html>
<body>
<script src="a.js"></script>
<img src="a.jpg" width="100" height="100"/>
<img src="b.jpg" width="100" height="100"/>
</body>
```

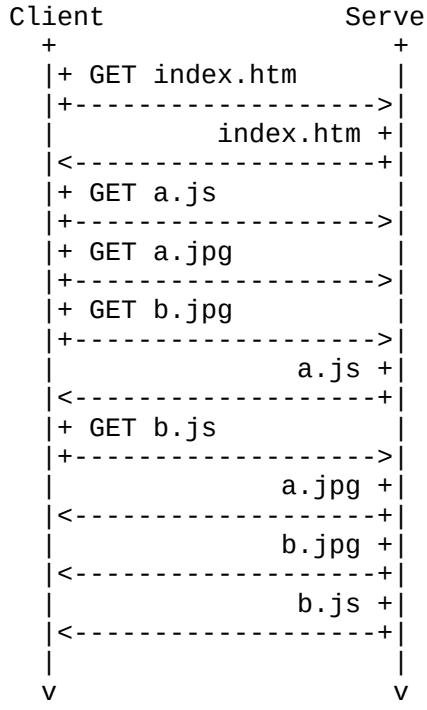
with a.js:

```
document.write('<script src="b.js"></script>');
```

and b.js:

```
document.write('<div>blocker</div>');
```

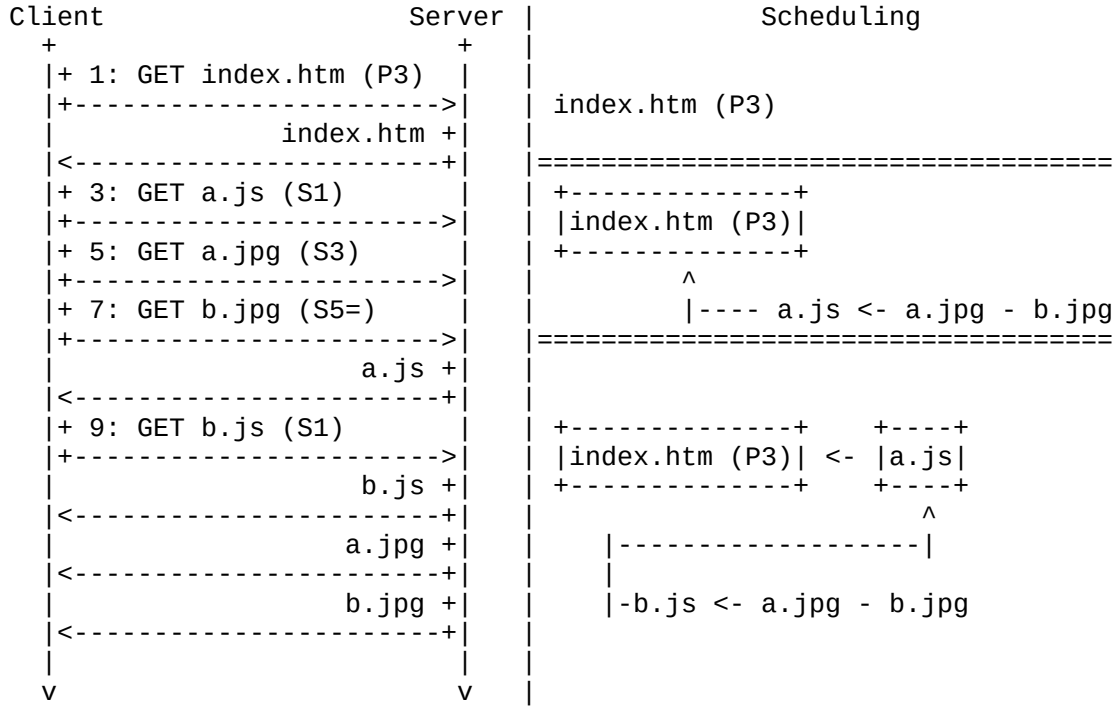
How would this example page be transferred today? As the main HTML is received and parsed, a request for a.js will be issued and block the document parser. As the remaining HTML streams in, the speculative parser will issue requests for a.jpg and b.jpg in quick succession. Once a.js is received and executed, a request for b.js will be issued, which again blocks parsing until received. Visually:



This transfer schedule is suboptimal. Page rendering will complete only when once b.js has completed, but receiving b.js is slowed by competition for bandwidth capacity for a.jpg and b.jpg, which do not block rendering.

Ideally, the order resources are transferred would reflect the document parse order with bandwidth sharing only for progressive resources. More specifically, we want to receive: 1) index.htm, 2) a.js, and 3) b.js sequentially. After those critical transfers have completed, a.jpg and b.jpg should be transferred concurrently since they may be displayed progressively.

Folding in the protocol mechanisms described above:



In the figure, each resource request corresponds to a new HTTP/2 stream with the form ID: request (PriOrDep). In more detail:

- The HEADERS for the index.htm request indicates a default priority (3) and a stream id of 1.

- The document parser is blocked once the external script a.js is parsed. At this point, the speculative parser looks ahead and creates new streams for a.jpg and b.jpg in parse order. a.jpg and b.jpg can be progressively rendered, so their transfer is concurrent (a.jpg has an ordered dependency on a.js, and b.jpg has an unordered dependency on a.jpg).

- Once a.js completes, the document parser continues by executing a.js and inserting b.js via document.write(), again blocking document parsing on the receipt of b.js. At this point, b.js should preempt all other transfers since it’s a non-streaming resource that is blocking page rendering. To this end, the client creates the b.js stream which depends on a.js (or, equivalently, index.htm).

This transfer schedule improves performance by serializing the transfer of resources on the critical path. The browser can ensure

that resources needed immediately do not compete for bandwidth capacity with less important transfers. The pipe remains full, as a queue of requests is maintained in the dependency list, filling any idle capacity with useful data. Where we cannot make an informed scheduling decision, we hedge our bets with concurrent transfers by hinting that they are unordered and letting the server decide what makes the most sense --- as in the case of two above the fold images that can be rendered progressively.

5.2. Servicing multiple tabs/users over a single HTTP/2 connection

As an illustration of this case, recall the example (Figure 1) from our straw-man design.

Suppose concurrent tabs are loading with the dependencies shown. When a user changes tabs, the browser sends a PRIORITY frame updating the stream associated with tab2.htm to, say, priority 8. (A batched message might also reduce the priority of tab1.htm to weight 3.) Because bandwidth is allocated among streams with priorities before considering their dependents, increasing the priority of tab2.htm effectively shifts capacity for all resource transfers depending on tab1.htm to tab2.htm.

5.3. Server Push

Push streams are assigned a priority or dependency at the discretion of the server. Typically, the Promised-Stream-ID would depend on the stream id carrying the PUSH_PROMISE frame. As information about resources needed for parsing is learned, the browser may update the dependency relationship by sending a PRIORITY message.

6. Policy Considerations

Both priorities and stream dependencies are advisory hints. Browsers may adopt sophisticated policies or leave dependencies entirely unspecified. Similarly, servers may incorporate dependency hints into very sophisticated schedulers or ignore them entirely. The protocol mechanisms for encoding dependencies are designed to be simple. But, these mechanisms afford a very flexible set of policies depending on how browsers and servers use them. This section expands on several policy considerations.

6.1. Assigning and updating dependencies

In our examples, we consider a browser that configures dependencies to reflect parser-blocking order for resources, updated as parsing continues. We expect this to improve performance, but browsers are

free to deviate from this policy, and there may be good reasons to do so. For example, if the parser-blocking order is highly dynamic (e.g., in response to many JS events), the overhead of updating dependencies may not be worth the cost, particularly for small transfers. A sophisticated client may base dependency update decisions on content-length and/or RTT, restricting updates to only those streams likely to benefit from it. Quantitative implementation experience is needed to determine how best to assign and update dependencies.

6.2. Server scheduling

A conformant server should respect the semantics of priorities and dependencies in its scheduling policy. Priorities indicate a preference for weighted scheduling (e.g., using a lottery scheduler [LOTTERYSCHEDULING]) among top-level streams; i.e., those created with a priority and not a dependency. Capacity should be shared among a sequence of streams with unordered dependencies.

Server scheduling should reflect guidance from dependencies, but it need need not be strict. If all streams in a dependency tree have data available to write at the server, writes should be serviced first for top-level streams, then ordered dependents, with sharing among unordered streams. But, depedents that are ready to write should not starve to enforce a scheduling dependency. In other words, scheduling dependencies should not lead servers to waste capacity. If data is not available to continue writing the top-level stream, for example, a dependent ready to write should do so.

Finally, we point out that servers may improve performance even if clients do not provide dependency information or priorities. For example, an intelligent server may inspect the content type of resources to make informed prioritization decisions on its own without client guidance. (However, respecting client-provided hints when available is likely to improve performance, as clients have detailed knowledge of parser dependencies.)

6.3. Garbage collecting dependency information

HTTP/2 implementations must take care to protect themselves from the use of dependencies as a DoS vector. The protocol provides wide flexibility in this regard; servers are free to drop dependency or priority data at any time without sacrificing correctness.

Typically, we envision servers will drop dependency information along with other stream state when an END_STREAM_ACK frame is received or the session is closed.

7. Security Considerations

TODO

8. Informative References

[LOTTERYSCHEDULING]
Waldspurger, C. and W. Weihl, "Lottery scheduling: flexible proportional-share resource management", 1994, <http://dl.acm.org/citation.cfm?id=1267639>.

[PRELOADSCANNER]
Gentilcore, T., "The WebKit PreloadScanner", 2011, <http://gent.ilcore.com/2011/01/webkit-preloadscanner.html>.

Appendix A. Acknowledgements

This document resulted from discussions amongst the SPDY team at Google. The authors merely took that discussion and edited this document. The individuals who contributed to those discussions include, but are not limited to: Roberto Peon, Hasan Khalil, Ryan Hamilton, Jim Roskind, Bryan McQuade, Chris Bentzel, Ilya Grigorik.

Authors' Addresses

Michael Piatek
Google

Email: piatek@chromium.org

William Chan
Google

Email: willchan@chromium.org