Proposal for Stream Dependencies in SPDY

Draft 1
Last Updated: 26 October 2012

This document proposes changes to the SPDY protocol 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 SPDY.

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

Motivation

In SPDY today, each stream has a *priority* (0–7) chosen by the client upon stream creation. Push streams are an exception. The server policy today is to assume push streams are all low priority. Push or pull, the priority of a stream cannot be changed once created.

Priorities provide hints to the server about which streams are most important to the client, but they are poorly suited to several common use-cases.

- *Specifying an ordering of resource transfers*
  Sharing bandwidth between resource transfers may degrade performance as measured by page-load time, 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 before image.png, for example, or video_chunk1 before video_chunk2 and so on. (Moreover, changing the priority of the HTML transfer itself may benefit performance; e.g., a large blocking script will be interpreted and executed more quickly if it does not compete for bandwidth capacity with a large HTML transfer.)

  With a small number of fixed priorities, the browser is simply unable to express an ordering over many resource transfers, and with a large number of priorities, reordering is costly.

- *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 speculative. (For more background, see Tony Gentilcore's excellent summary of Chrome's implementation of speculative parsing, the preload scanner.) These requests may need to be preempted as the document parser learns 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, as the receipt of `b.js` blocks page layout. As another example, consider images styled with `display: none`; once such styling is discovered during parsing, associated image transfers should be deferred to prioritize visible content.

- *Reacting to user behavior*
  Suppose a SPDY proxy is servicing multiple users. In this case, many tabs (and their associated streams) are multiplexed over the same SPDY connection. Fixed priorities (i.e., unchanging over the lifetime of a stream) preclude reacting to user behavior; e.g., a user may switch among concurrently loading tabs.

- *Server push*
  No single fixed priority is appropriate for server push. A stream pushing a large image, for example, should have lower priority than JS/CSS. But, when pushing JS/CSS that the browser needs, those stream should have high priority.

In sum, for many common scenarios, fixed priorities are not sufficient to optimize the allocation of bandwidth among competing requests.

Protocol changes

To address the limitations of priorities, we propose expressing *dependencies among streams*. Dependencies improve matters in two main ways:

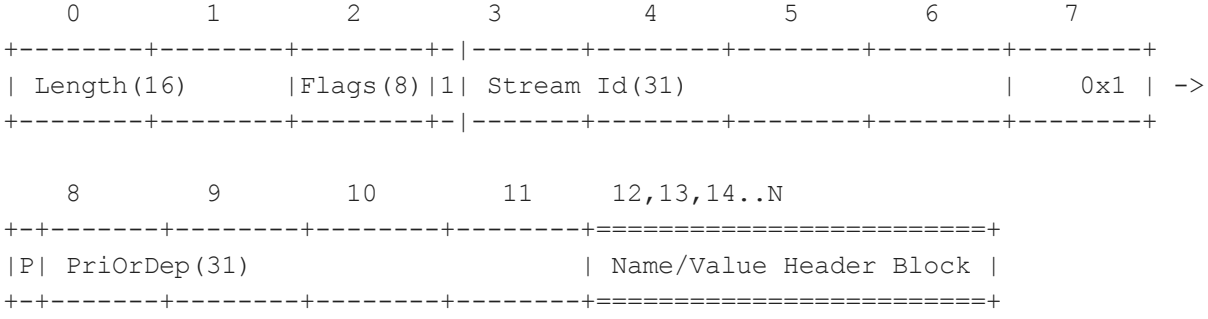
1. **Dependencies more accurately reflect the constraints of the browser.**
   Rendering a page is a streaming process that naturally leads to a series of dependencies among resource transfers. For example, a script may block HTML parsing, and a final layout may depend on an external stylesheet.

2. **Dependencies can be updated efficiently.**
   The relative importance of streams may change as Javascript executes or a user changes tabs, for example. Dependencies allow the browser to express these changes compactly. If a user changes tabs, for example, the browser may simply signal a change in priority of the tab's dependency root, thereby reducing (or increasing) the bandwidth allocated to all dependent transfers.

Dependencies are expressed in two ways: 1) a new dependency field in the `SYN_STREAM` message, and 2) a new `REPRI` message that updates the dependency pointers of existing streams. To allow servers to advertise their support for scheduling transfers based on dependencies, we propose a new `SETTINGS` id/value pair. We describe the layout and semantics of each in turn.

Note that these protocol changes are defined in terms of the latest version of the SPDY draft specification.

SYN_STREAM:



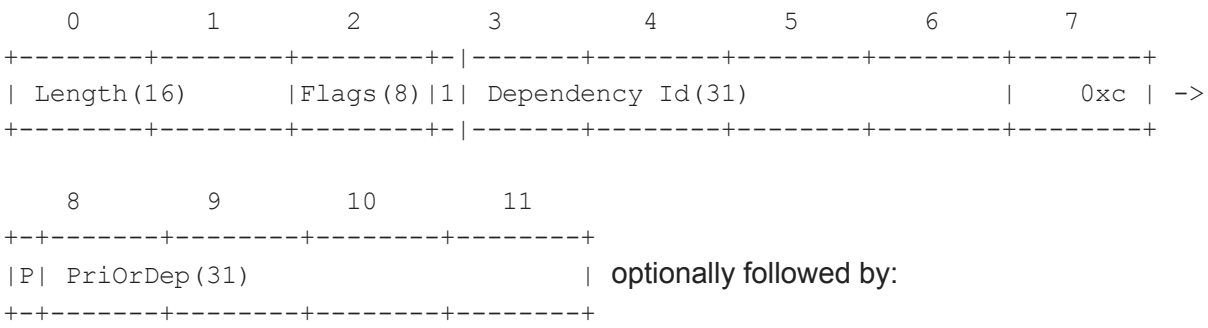
Here, the first 8 bytes are the standard control frame header (§2.2.2). A new 32 bit field replaces the existing `SYN_STREAM` priority bits (§2.6.1) with:

- `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 Name/Value header block (§2.6.11) are unchanged.

`P` is exclusive; a stream may be assigned a priority or a parent dependency upon creation, but not both. There are no constraints of the value of `PriOrDep`; any 31 bit value is valid. Thus, a stream may refer to a dependency identifier that does not correspond to any current or previous stream ID. This is a deliberate design choice that increases flexibility for clients when structuring dependencies, a topic we expand upon in the policies section.

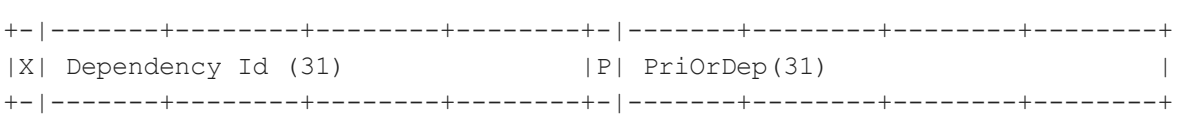
Server push streams are assigned an initial parent at the discretion of the server. A conformant implementation SHOULD create a dependency on the push stream's associated-to-stream-id (§3.3.1).

REPRI:



optionally followed by:

`DependencyPriOrDep` pairs, where a `DependencyPriOrDep` pair is:



As in `SYN_STREAM`, the control frame header is standard, followed by a `P/PriOrDep` label indicating an update to the 31 bit `Dependency Id` specified in the header. We relabel the typical `Stream Id` here as `Dependency Id` since a dependency need not correspond to an actual stream. (Recall that any 31 bit value is a valid dependency identifier.)

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. number-of-pairs = ((length - 12) / 8). (12 required bytes, 8 bytes from len(`stream_id`) + len(`PriOrDep`))

We expect most streams to have at most a single dependency, but this is not a protocol requirement. (Later, we describe scenarios where multiple parents may improve efficiency.) If a stream is referenced more than once in a single frame, this indicates multiple parents. A server implementation which does not support multiple parents MUST use the last referenced parent. Clients which send multiple parents thus SHOULD put the most important parent last.

SETTINGS:
Recall that dependencies and priorities are advisory. While servers must accept the messages, they are not required to incorporate them into scheduling decisions. A client may benefit from knowing a server's level of support; e.g., a client may specify priorities only if it knows a server will ignore dependencies. To communicate this, we propose a new `SETTINGS` ID/value pair (§2.6.4),

- ID 9 - `SETTINGS_MAX_CONCURRENT_DEPENDENCY_SCHEDULING_NODES` allows the server to indicate resource limits for dependency scheduling, e.g., to limit memory consumption. A value of 0 indicates that the server does not support dependency scheduling. (We expect most implementations will select a value greater than or equal to `MAX_CONCURRENT_STREAMS`.)

- ID 10 - `SETTINGS_DEPENDENCY_SCHEDULING_NODE_TIMEOUT` indicates how long the server will maintain dependency nodes after creation. The value is an interval in milliseconds. This allows the client to estimate if previously created dependency relationships are still available for reference at the server. (We expect conformant implementations to maintain dependencies for at least as long as associated streams are active, although this is not a correctness requirement.)

Both of these values are advisory. Servers need not abide by their stated values and clients may disregard them. Conformant clients should respect the concurrency limit, but servers must be robust to a client that exceeds it. Similarly, servers may drop dependency information at any time regardless of previous statements made in `SETTINGS`. This is intended to provide flexibility for service policies; e.g., a server may reduce the timeout in response to memory pressure or abandon dependency scheduling entirely.

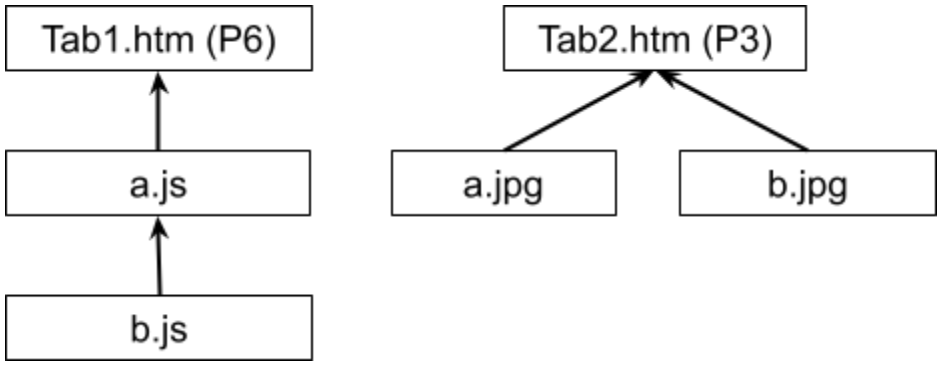
Examples / use-cases revisited

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

1. Resource dependencies are (re)configured to reflect parser-blocking order. The transfer of non-streaming resources is always serialized; i.e., non-async scripts and styling.
2. Resources that can be progressively rendered (e.g., images) are transferred concurrently and (re)configured to depend on parser-blocking resource transfers.
3. To ensure that the speculative parser can maintain enough in-flight requests to fill pipe between the client and server, page HTML is always a top-level dependency, although it may have lower priority than a resource transfer currently blocking document parsing.

When scheduling transfers, we consider a server that allocates bandwidth *hierarchically* within dependency trees and *splits equally* among streams with the same parent.

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



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. Both image transfers have the same parent and hence transfer concurrently.

Because the streams associated with the transfers of tab1 and tab2 have no parent, they are always scheduled before any lower level in their trees. But, bandwidth allocation among trees remains proportional as defined by the relative priority of roots. For example, if the transfer of `tab2.htm` is in progress and `tab1.htm` (now complete) is selected, `a.js` will be scheduled before `tab2.htm` completes. This process proceeds until all transfers in a tree have completed.

As a practical matter, the timeout for pruning nodes in a tree should be selected to allow transfers to complete and to allow clients to name currently completed parents when defining transfer dependencies. Concretely, on a high delay path, a small HTML transfer may be flushed entirely by the server before the client receives any data and begins making dependent requests for resources embedded in the page.

With these client and server policies in mind, we revisit the motivating use-cases described above in greater detail.

- *Specifying an ordering of resource transfers*
- *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"/>
<link rel="stylesheet" type="text/css" href="style.css">
</body>
```

with a.js:

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

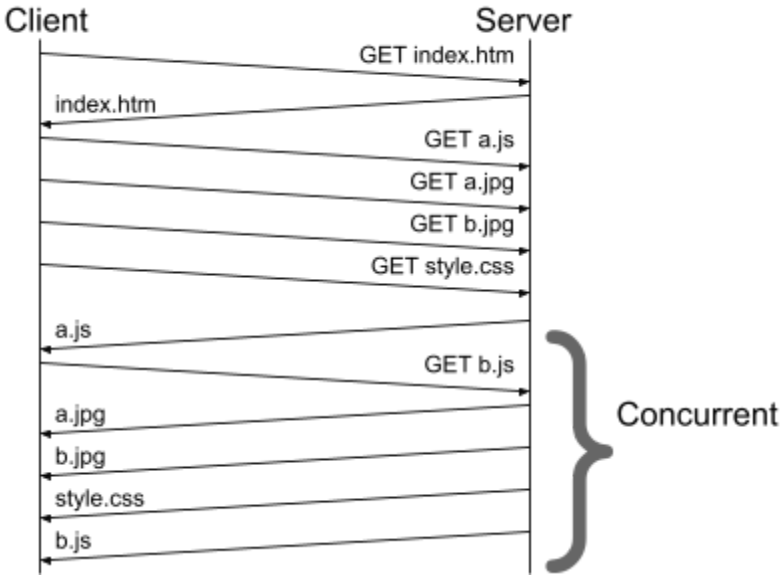
b.js:

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

and style.css:

```
        div {
          border: 1px solid #000;
        }
```

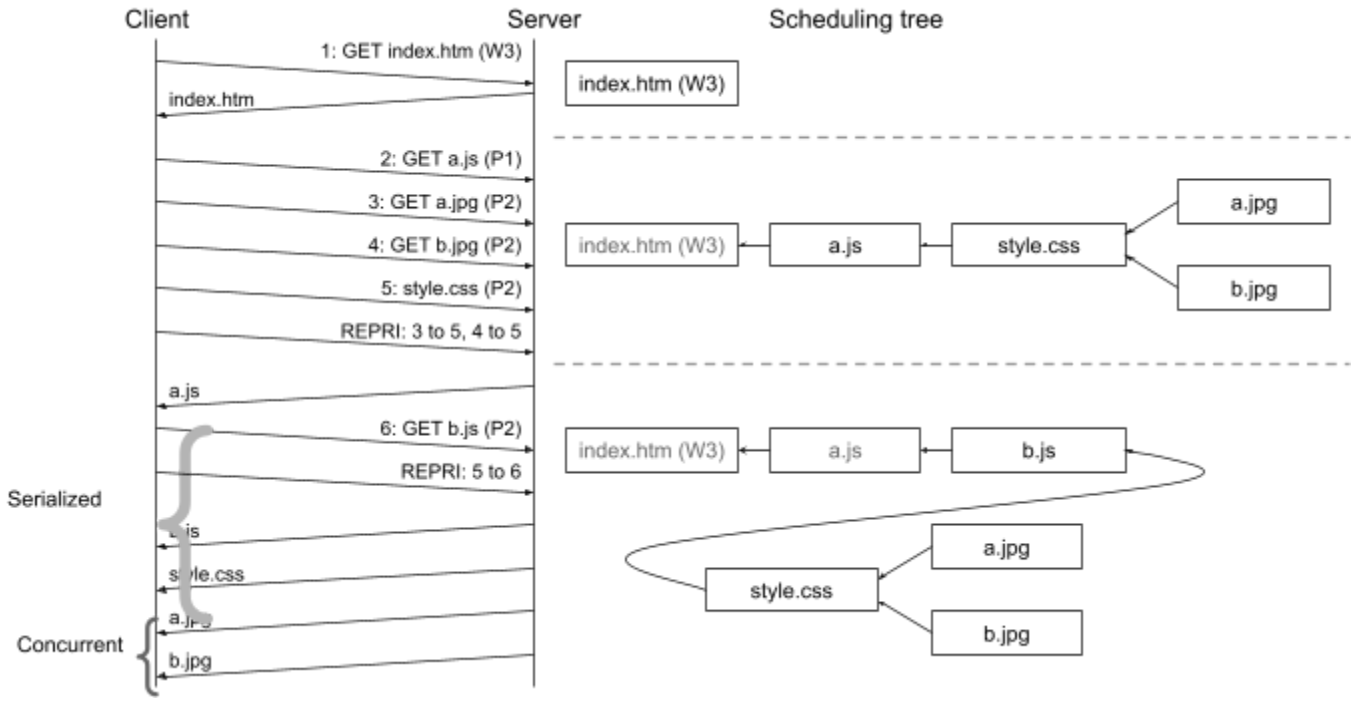
How would this example page be transferred today? As the main HTML is received and parsed, a request for a.js will be issued and block the document parser. As the remaining HTML streams in, the speculative parser will issue requests for a.jpg, b.jpg, and style.css in quick succession. Once a.js is received 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 style.css and b.js have completed, but receiving each of those critical resources is slowed by competition for bandwidth capacity with bulk data that's not on the critical path (a.jpg and b.jpg).

What we would like is *serialized transfer that reflects the document parse order* with *concurrency for nonblocking, streaming resources.* More specifically, we want to receive: 1) index.html, 2) a.js, 3) b.js, and 4) style.css serialized (i.e., with no deliberate sharing of capacity among the ordered transfers). After those critical transfers have completed, a.jpg and b.jpg should be transferred concurrently (as they may be displayed progressively.)

Folding in the protocol mechanisms described above:



In the figure, each resource request corresponds to a new SPDY stream with the form ID: reqest (`PriOrDep`). In more detail:

- The `SYN_STREAM` for the index.htm request has a parent indicating 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, b.jpg in parse order. a.jpg and b.jpg can be progressively rendered, so their transfer is concurrent (same parent, 2, corresponding to a.js).

- When the parser encounters style.css, back-to-back control messages are sent to create the stream and update the dependencies of the image transfers. Since stylesheets block rendering and cannot be streamed, the image transfers are updated to depend on style.css (P5).

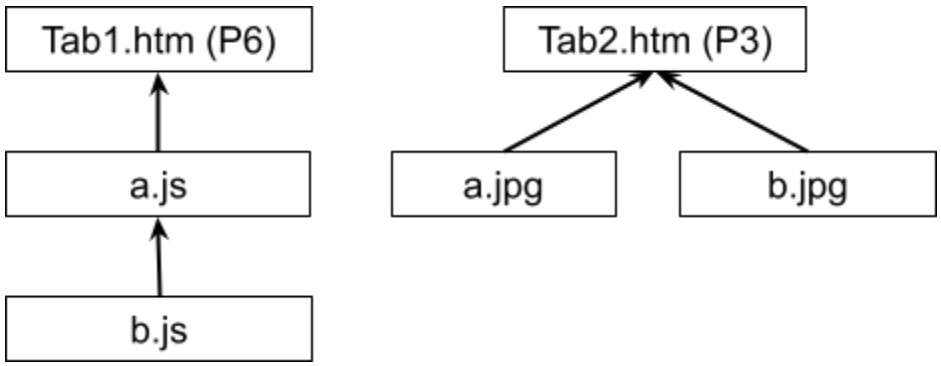
- Once a.js completes, the document parser continues, executing a.js and inserting b.js via `document.write()`, again blocking document parsing on the receipt of b.js. At this point, b.js should preempt all other transfers since it's a non-streaming resource that is blocking page rendering. To this end, the client creates the b.js stream with a.js as its parent (or, equivalently, index.htm). Batched with this `SYN_STREAM` is another `REPRI` message rewiring style.css to depend on b.js. This serializes the transfers (modulo the delay associated with message propagation and any transfer buffering delay at the server).

This transfer schedule may significantly improve 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. Yet, the pipe remains full, as a queue of requests is maintained in the scheduling tree ready to fill 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 peers 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.

Note that this sort of explicit scheduler hinting is not possible in HTTP today. Requests, once issued, cannot be reprioritized or reordered on a single connection. This results in suboptimal transfer schedules given the limitations of HTML lookahead scanning. Yet, lookahead is essential for ensuring the concurrency necessary to keep the client <-> server pipe full. While the browser might serialize transfers itself, the many small transfers typical of pageloads would significantly limit utilization. With ordering and reprioritization in SPDY, browsers can jointly optimize both the transfer pipeline *and* resource priority as desired, rather than being forced to accept poor utilization or poor transfer schedules.

- *Servicing multiple tabs/users over a single SPDY session*
As an illustration of this case, recall the example from our straw-man design:



Suppose concurrent tabs are loading with a scheduling forest as shown. When a user changes tabs, the browser simple sends a `REPRI` for 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 allocation decisions are made tree-by-tree and level-by-level, increasing the priority of tab2.htm effectively shifts capacity for *all* resource transfers depending on tab1.htm to tab2.htm.

- *Server push*
As in client `SYN_STREAM` messages, server push messages indicate the priority and dependencies of a resource *as chosen by the server*. Much like the client, the server is free to adopt prioritization policies to improve performance, e.g., by prioritizing pushes of styles over images. But, as in our example above, the browser may update the server's choices as information about resources needed for parsing is learned. (Again, expressed via `REPRI` messages.)

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.

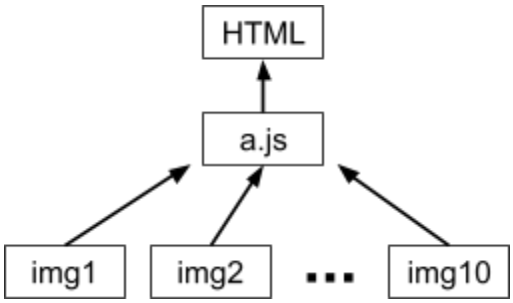
Assigning and updating dependencies.
Updates and overhead

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 will be helpful here.

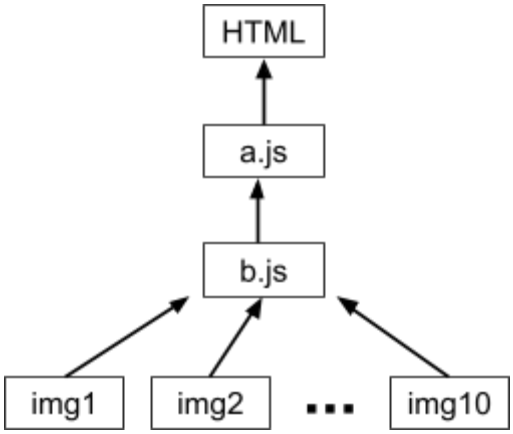
The overhead of updating dependencies depends in part on the existing structure of dependencies. In some scenarios, it may be more efficient to introduce placeholder nodes to improve the efficiency of common update operations. For example, consider a variant of our earlier example page:

```
<html>
<body>
<script src="a.js"></script> <!-- containing: document.write('<script src="b.js"></script>'); -->
<img src="1.jpg" width="100" height="100"/>
<img src="2.jpg" width="100" height="100"/>
<img src="3.jpg" width="100" height="100"/>
...
<img src="10.jpg" width="100" height="100"/>
</body>
```

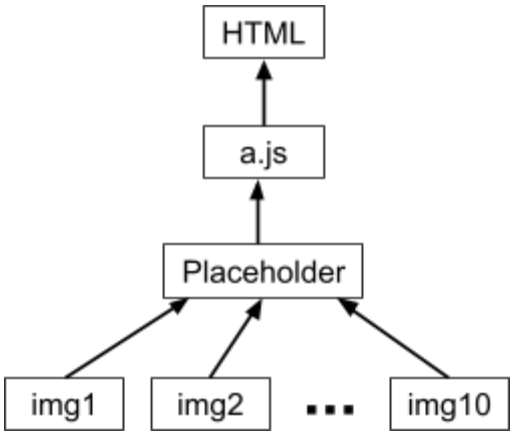
In this example, the speculative parser might create 10 streams depending on the JS transfer; i.e.,



But, once a.js is executed, the transfer of b.js should preempt all image transfers; i.e.,



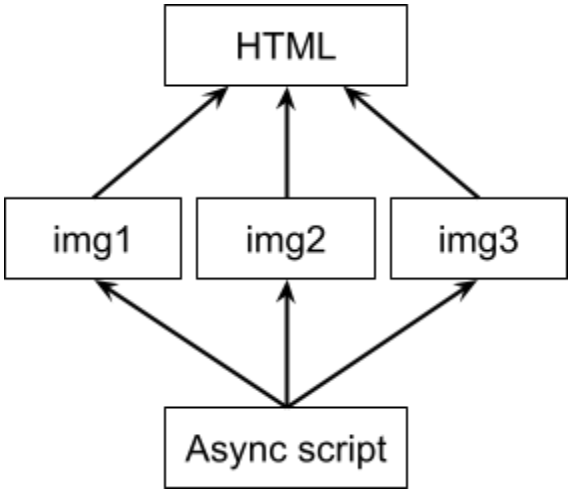
Transitioning between these dependency structures requires sending `REPRI` messages for each image. Because updating the dependencies of images is common, a client might create all image streams with a placeholder dependency, yielding an initial configuration of:



With such an initial configuration, updating the dependencies of the images to the stream associated with b.js can be accomplished with a single `REPRI` message updating the placeholder.

Multiple parents

All of our examples have considered nodes with a single parent. But, single parent trees cannot express some transfer schedules. For example, an asynchronous script at the end of HTML is non-blocking and does not block interaction or pageload. It should be loaded after all visible resources (e.g., images) have completed. A possible dependency graph is:



Here, any of img1, img2, or img3 should be prioritized before the asynchronous script, a schedule that cannot be expressed with single parent pointers.

Reacting to server capabilities

Clients should not specify dependencies to servers that do not support it (as indicated by `SETTINGS_MAX_CONCURRENT_DEPENDENCY_SCHEDULING_NODES` in the `SETTINGS` frame). Rather, an intelligent client may fall back to specifying priorities only, thereby improving performance relative to specifying dependencies that will be ignored.

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) among root nodes (i.e., those created with a priority and not a parent). Interior nodes with identical parents are weighted equally.

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 root nodes, then children, then grandchildren, and so on. But, children 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 root, for example, a child 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.)

Garbage collecting dependency information.
SPDY 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.

Otherwise, we envision servers choosing a timeout value for dependency nodes that is large enough to cover the likely time period during which a client may reference a node; e.g., page load time + rtt. Alternatively, a large fixed number of dependencies may be maintained per-session with LRU eviction. Either of these policies is likely to sharply limit the number of missed dependency references. On the client side, browsers should take care to manage dependencies according to server policies, e.g., by creating new dependency structures rather than referring to those that have likely timed out and been garbage collected.

While missed references may be rare, they are unavoidable in an asynchronous system with timeouts. (This is why we require that servers accept as correct any dependency id.) In cases where a reference is made to an unknown node at the server, the server may create the referenced id as a new root for future reference or ignore the dependency entirely (i.e., treat the new stream as a root).