PUBLIC

Implement Cache-Control: stale-while-revalidate

Adam Rice <ricea@chromium.org>

Objective

Implement the Cache-Control stale-while-revalidate directive to permit servers to reduce latency.

Background

See "PRD - stale-while-revalidate" for detailed background and justification.

Proposed Design

The feature will be implemented in several stages to minimise risk.

Stage 1: Resource-Freshness header

A server which supplies the Cache-Control: stale-while-revalidate directive with a non-zero value will receive a `Resource-Freshness` header on future requests for the same resource (as long as it remains in cache). This will only be sent with revalidation requests, ie. requests that include an `If-Modified-Since` or `If-None-Match` header.

The `Resource-Freshness` header contains three pieces of information:

1. max-age: If the cached response contains a Cache-Control: max-age directive, this is that value. Otherwise it is the max-age that Chrome calculated based on the value of other headers, in seconds.
2. stale-while-revalidate: The stale-while-revalidate directive value from the original response (seconds).
3. age: How old Chrome thinks the resource is, in seconds.

This will permit web services to experiment with stale-while-revalidate and calculate optimal values for max-age and stale-while-revalidate directives.

The extra header will be added to the request in `HttpCache::Transaction::ConditionalizeRequest()`. The age and max-age have already been calculated once to determine whether the cached entity is fresh, but currently their are not cached anywhere. For efficiency it might be a good idea to cache these values instead of recalculating them to send the header.

Negative values of the `stale-while-revalidate` directive will be ignored.

Stage 1.5: Resource-Freshness header for conditional requests from Blink

If a resource is in the Blink cache, then Blink will issue a conditional request for it. These are sent to the origin server unchanged. They will also need a `Resource-Freshness` header added.

Stage 2: Experimental implementation

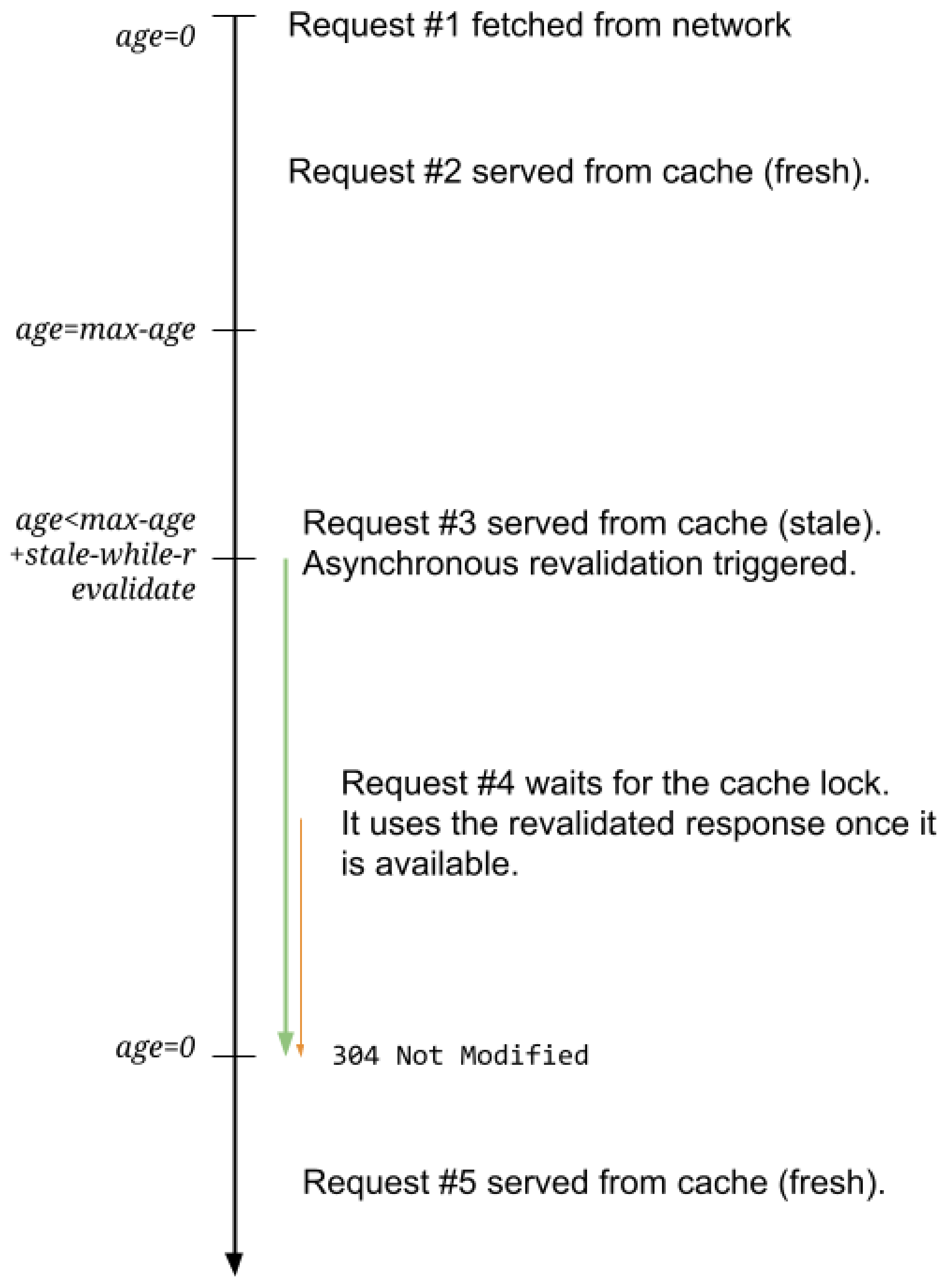
Requirements

- For every request which utilises the cache, check whether the stale-while-revalidate directive is present and applicable. "Applicable" just means that `max-age + stale-while-revalidate < age`. If applicable, the cached response should be returned, and an asynchronous revalidation should be triggered.
- Currently no specific opt-out for this behaviour is envisioned. Normal mechanisms to bypass the cache or force revalidation will prevent stale content from being used.
- To avoid confusion about what are the "real" response headers, the response as shown in devtools or `XMLHTTPRequest.getAllResponseHeaders()` will *not* have a `Warning:` or `Age:` header (unless an upstream proxy or server actually provided one).
- Request modifications such as URL rewrites or request header additions must be applied exactly once. In practice this means that only the original request will be passed through extension hooks; the copy of the request that is issued asynchronously will not.
- The asynchronous revalidation request must be issued at the IDLE priority.
- Load flags `VALIDATE_CACHE`, `BYPASS_CACHE`, `PREFERRING_CACHE`, `ONLY_FROM_CACHE` and `DISABLE_CACHE` should disable the stale-while-revalidate logic.
- The originator of the asynchronous revalidation request is the cache, not the renderer. The renderer is assumed to have sufficient privileges to make the request, since it reached the browser cache, however the cache may lack access to credentials needed to perform the request, in particular SSL client certificates. In other words, the cache won't fetch anything that the renderer would not have fetched anyway, but it might fail to fetch resources that the renderer would have fetched successfully.
- The asynchronous revalidation request will not pop up UI in response to requests for authentication.
- If the request fails in a way that would require user interaction to resolve, the cached entry should be marked as requiring revalidation on the next load so that the necessary user interaction can be performed. This should be rare. We could add a UMA histogram to track these cases. The cases I intend to handle in this way are
  - SSL client certificate required.
  - SSL certificate error manual override required.

PUBLIC

- The asynchronous revalidation request should cache redirects, but not follow them. This is because extension hooks don’t run, so we don’t know whether any redirect would be blocked by an extension.
- The logic to determine whether to use stale-while-revalidate and issue an asynchronous request will be run on every request, so it should be lightweight.
- Only GET and HEAD requests are in scope. POST requests are problematic because the UploadDataStream object will not outlive the original URLRequest and we have no way to copy it.
- We don’t want the asynchronous revalidation request to consume bandwidth that would otherwise have been used for a synchronous request required to render the page. It is probably impossible to do this perfectly, so we should make a best effort.
- We should avoid causing additional radio wakeups on mobile. We should also attempt to minimise the extra time that the radio is kept awake to service the asynchronous request.
- Chrome’s cache does not cache the following headers:
    - Set-Cookie and Set-Cookie2. These headers are parsed by URLRequestHttpJob, and so attempts to set cookies on asynchronous responses will fail.
    - Strict-Transport-Security and Public-Key-Pins. These are also processed by URLRequestHttpJob, and so will be lost if returned in an asynchronous response. Normally these would already have been seen when first connecting to the host, but it is conceivable that if they were added when all resources used from the host were already cached; as long as those resources were only fetched asynchronously the security settings would never get applied.
    - WWW-Authenticate and Proxy-Authenticate. Losing these is not a big problem; 403 and 407 responses are not usually cacheable anyway, and even if it was the authentication would have to be completed synchronously.
    - Connection-level headers. These are only relevant to the lower levels anyway, so losing them is not a problem.
- The feature should be disabled by default, behind a flag, to avoid wide adoption while we experiment with it. Assuming the experiment is a success, an implementation that supports cookies and the Strict-Transport-Security and Public-Key-Pins headers will be needed.
- Apart from the time it is issued, and the lower priority, the asynchronous revalidation request should be identical to what the synchronous request would have been. In particular, it should not require additional server resources to process as this would discourage server operators from using the feature.

Timeline


age=0
Request #1 fetched from network
Request #2 served from cache (fresh).
age=max-age
age<max-age +stale-while-revalidate
Request #3 served from cache (stale). Asynchronous revalidation triggered.
Request #4 waits for the cache lock. It uses the revalidated response once it is available.
age=0
304 Not Modified
Request #5 served from cache (fresh).

Because of the caching semantics of HTTP, the “age” of a resource is not necessarily 0 when we receive it from the network, however this does not significantly change the semantics.

Additional requests for the same resource will have to wait for the cache lock in order to proceed. In effect this means that they will wait for the asynchronous revalidation to complete.

There is a small time window between the synchronous request being served the stale resource from the cache, and the asynchronous revalidation being created. The cache will track what resources have asynchronous validations in progress and ignore a second attempt to create an asynchronous revalidation to a resource which already has one in progress.

Implementation

- Add a field trial "`StaleWhileRevalidate`" to experiment with turning the feature on for a fraction of users. The feature will be disabled by default.
- Add a flag `--enable-stale-while-revalidate` to control the feature from the command-line.
- Add the feature to `about:flags` so that developers can experiment with it easily.
- Modify `HttpResponseHeaders::RequiresValidation()` to return an enum with one of the values `NONE`, `ASYNCHRONOUS` or `SYNCHRONOUS` instead of the existing bool return value. Return `ASYNCHRONOUS` when `age > max-age` and `age <= max-age + stale-while-revalidate`.
- `content::AppCacheUpdateJob()` also uses the `HRH::RequiresValidation()` method but won't do asynchronous revalidation. Modify it to treat `ASYNCHRONOUS` and `SYNCHRONOUS` the same.
- Modify `HttpCache::Transaction::RequiresValidation()` to return `NONE`, `ASYNCHRONOUS` or `SYNCHRONOUS` instead of a boolean. Where it currently returns `true`, it should return `SYNCHRONOUS` and where it currently return `false` it should return `NONE`. It will return `ASYNCHRONOUS` if it called `HRH::RequiresValidation()` and that was the value returned.
- `HC::Transaction::BeginCacheValidation()` will be modified to treat `ASYNCHRONOUS` the same as `NONE` when the feature is enabled, except that before doing anything else it will call `TriggerAsyncValidation()`. When the feature is disabled, it will behave the same as `SYNCHRONOUS`, effectively ignoring stale-while-revalidate (except that the `Resource-Freshness` header will still be triggered).
- The new method `TriggerAsyncValidation()` will post a task to the `HttpCache` object to call `PerformAsyncValidation()` on the next iteration of the message loop (the `HC::Transaction` object may already have been deleted by then).
- `HttpCache` will have a new method called `PerformAsyncValidation()` which will create a new `IDLE`-priority `HC::Transaction`, store the new transaction in a map, and start it running.
- If an existing `AsyncValidation` for the same resource is found, the new one will be discarded.
- `HttpCache::AsyncValidation` will create a special `ProxyHeadersSendCallback` which doesn't depend on the `URLRequest` object being alive. This allows asynchronous revalidations to pass through the bandwidth reduction proxy.
- A new nested class, `HttpCache::AsyncValidation` will handle creating the transaction and reading back its results.
- A new load flag, `LOAD_ASYNC_REVALIDATION will be created.` This will force `HC::Transaction` to perform a synchronous validation. This cannot be done via existing load flag `LOAD_VALIDATE_CACHE` because it has the side-effect of adding a `Cache-Control: max-age=0` header to the request.

PUBLIC

- Since `HC::Transaction` already stores its results to the cache, `HttpCache::AsyncValidation` can just `Read()` and discard the response.
- A new UMA histogram, `HttpCache.AsyncValidationDuration`, will record the time spent requesting stale resources asynchronously.
- The `HttpCache` destructor will delete any asynchronous revalidations that are still in progress.
- It is not safe to re-use the `BeforeNetworkStartCallback` set by `URLRequestHttpJob` because it has a pointer to the `HttpJob` and the `HttpJob` is highly likely to be deleted. This callback is not currently used for anything so at the moment not setting it is harmless.

Rejected Alternatives

- Instead of changing the existing `RequiresValidation()` method to return an enum, create a new HttpResponseHeaders`::RequiredValidationType()` method. This would avoid the need to change `content::AppCacheUpdateJob()`.
- Factoring cache-related code out of `HttpResponseHeaders` into a dedicated class. The new class could then cache the values returned by the header lookups, reducing the number of passes made over the headers. This would take considerably more time and might be controversial.
- Returning a magic error code like `ERR_NEED_ASYNC_REVALIDATE` back to `URLRequestHttpJob` to trigger an asynchronous revalidation. This is attractive because `URLRequestHttpJob` has access to the `URLRequestContext` object, however we still need to attach the asynchronous job to an object with a longer lifetime (ie. the `HttpCache` object). There is a danger that the magic error code could leak out of the intended scope. Also if we can constrain the logic to `HttpCache` and `HttpCache::Transaction` it permits better encapsulation and reduces the impact of the change.
- Adding a delay before starting the asynchronous revalidation to give more synchronous requests a chance to run first. For transports which implement prioritisation at the protocol level (ie. SPDY/HTTP2/QUIC), and competing resources from the same host, this could only make performance worse. When the competing resources come from a different origin, or the protocol has no prioritisation (HTTP/1.1), the situation is more complicated. We might need to wait, but we don't know how much, and if we wait too long we will waste battery life.
- Sending asynchronous requests back up the stack to content::ResourceScheduler to be scheduled. This would permit the asynchronous requests to be queued behind synchronous requests that haven't been passed to net/ yet. This would probably deliver the best behaviour that we can get with the current architecture, at an considerable cost in implementation complexity. We would need to ensure that stale-while-revalidate behaviour was only used with embedders that were capable.

Code Locations

    - net/http
    - chrome/ (flag stuff)

  - build/ios (flag stuff)
  - tools/metrics/histograms/histograms.xml

Stage 3: Initial Evaluation

A field trial on dev users will enable us to measure the perceived latency improvement resulting from the feature. Based on this, we can decide whether it is worth investing the time to implement the feature better.

What success looks like:

- **Total success**
  - A statistically significant improvement in time to first paint. This would immediately justify proceeding with a better implementation. We would still need to revalidate the result against the stable channel once the new implementation was complete, as the behaviour of users on the dev and stable channels is known to differ.
- **Partial success**
  - An improvement in time to first paint, but without statistically significance. This could mean that the experiment was too small, or just nothing at all. By itself, it would not justify further investment.
  - An improvement in simulated load times, based on data gathered during the experiment. Depending on the degree of the improvement and the realism of the simulation, this might or might not justify further investment.
  - Interest from other browser vendors or server operators. This would bias towards further investment.
- **Bad success**
  - A statistically significant regression in time to first paint. We would need to invest time in investigating the cause. This investigation might ultimately lead to some insights that would improve page load times, but probably not with stale-while-revalidate.

What failure looks like:

- **Total failure**
  - No improvement in time to first paint, no potential for improvement seen based on data gathered during the experiment.
  - Chronic website misbehaviour caused by the feature.
- **Partial failure**
  - Browser instability due to the implementation.
  - Experiment aborted due to crashes.
  - Conditional requests from the Blink cache interfere with the results (there is a separate experiment planned to find out how often conditional requests from the Blink cache bypass the Chrome cache, and thus gauge the risk).

Stage 3.5: Revert Initial Implementation

If the initial evaluation justifies further investment, this means removing the code from net::HttpCache to perform asynchronous revalidation, and disabling asynchronous revalidation in HttpCache::Transaction while retaining the code which determines when we might use it.

If further investment is not justified, we just revert everything implemented in Stage 2.

Ideally these steps should be performed before the branch for M39, but as long as the functionality is disabled, it is not too harmful for it to be included in the stable release.

Stage 4: Better implementation

The implementation added in Stage 2 has serious deficiencies:

1. [Critical] Cookies, Strict-Transport-Security and Public-Key-Pins headers are lost on async responses.
2. [Bad] stale-while-revalidate logic is not applied when Blink's cache sends a conditional request.
3. [Suboptimal] Async requests compete for bandwidth with synchronous resources in many cases.
4. [Meh] The asynchronous revalidation requests are invisible to extensions.

Sketch of better implementation

- A new load flag indicates that a request is eligible for stale-while-revalidate treatment and the requestor is capable of performing asynchronous requests. This flag is set on all normal requests from blink.
- A new flag in HttpResponseInfo indicates that the response is stale and asynchronous revalidation needs to be performed.
- content::ResourceDispatcherHostImpl would recognise the flag and trigger an asynchronous request. A content::DetachableResourceHandler would be used to ensure that the request proceeded even if the renderer went away.
- Another new load flag would indicate an asynchronous revalidation request. Among other things this would enable content::ResourceScheduler to schedule it after synchronous requests, and extension APIs to treat the request specially.
- Blink's cache has an implementation closely matching the one in net::HttpCache; in the stale-while-revalidate case it sends a notification that it re-used the cache resource, and then sends a request with the "async revalidation" load flag set. The behaviour in content/ and net/ is then the same, except that the response data is actually sent back to the renderer (assuming it is still alive).

Code Locations

- net/http/
- content/
- chrome/

- src/third-party/WebKit/Source/core/fetch/

Privacy Impact

The `Resource-Freshness` header could be used to fingerprint users.

Existing cache-based fingerprinting techniques (eg. supplying a unique `Last-Modified` header) already exist. Workarounds for existing cache-based fingerprinting techniques will also be effective with the `Resource-Freshness` header.

Incognito Mode and separate User Profiles

In Chrome, Incognito Mode and different User Profiles already use separate caches to protect against information leaks, so no extra changes are required.

Security Impact

The value of the stale-while-revalidate directive in the requests can be controlled by the server. However, it is converted to an 64-bit integer internally, so it is not possible to inject arbitrary data. It is only returned for the same entity, so cross-domain or cross-protocol attacks should not be possible.

A man-in-the-middle attacker can use stale-while-revalidate to cause their malicious content to be used one more time after the attack has ended. This appears strictly weaker than a similar attack using max-age, where the malicious content will be re-used any number of times.

Revision History

- 2014-07-14: First version
- 2014-07-15: Clarified that “Chrome-Freshness” name is temporary. Simplified the “Privacy Impact” section.
- 2014-07-18: Added note about stricter parsing of stale-while-revalidate compared to max-age.
- 2014-07-22: Requirements for initial implementation added. First public version.
- 2014-07-28: Changed the temporary name to “Chromium-Resource-Freshness”. Removed requirement to obey cookie settings. Require non-zero value to stale-while-revalidate to receive the information header.

- 2014-08-07: Add implementation details for stage 2.
- 2014-08-09: `LOWEST` priority is not lowest priority. Use `IDLE` priority instead. Added the competing requirements of not stealing bandwidth from synchronous requests and not keeping the radio alive unnecessarily. Change the return type of `RequiresValidation()` rather than adding a new method.
- 2014-08-11: The initial implementation is only optimal if we are using a SPDY proxy. Discuss the alternatives in the "alternatives considered" section.
- 2014-08-13: Note that `LOAD_VALIDATE_CACHE` must be added to the load flags of the asynchronous transaction.
- 2014-08-19: "Chromium-Resource-Freshness" has been changed to "Resource-Freshness" in the code. Update the design doc to match. Add steps 1.5 and 2.5 for conditional requests from Blink.
- 2014-08-21: Note the problem with losing cookie and transport security headers.
- 2014-08-28: The feature is off by default, controlled by a flag and field trial. Noted changes that happened during implementation. Re-organised the document to reflect that we need to rewrite everything on success.
- 2014-09-1: Add a requirement that the asynchronous request be the same as what the synchronous request would have been. Add criteria for determining success or failure, and plan to revert initial implementation.
- 2014-09-22: Updated to reflect implementation changes that happened during review.