Cross-origin Prefetch + SplitCache

Author: Dominic Farolino <domfarolino@google.com> <dom@chromium.org>
Last updated: **September 23rd, 2019 (Post-TPAC)**
Status: Done [PUBLIC]

Chromium bug: https://crbug.com/939317
Spec bug(s)/PRs:

- https://github.com/w3c/resource-hints/issues/82
- https://github.com/whatwg/html/pull/4115
- https://github.com/whatwg/fetch/pull/881

Related documents:

- kinuko@'s saner caching model for prefetch
- horo@'s initial exploration
- kinuko@'s early project doc
- domfarolino@'s single-key eligible thoughts/doc
- Potassium's HTTP Cache Threat Model
- kinuko@'s speculative loading doc

Table of Contents

Motivation/Background

Currently, many websites such as Google Search make use of prefetching cross-origin resources to speed up future navigations. This works because prefetched resources are stored in the HTTP cache, which is globally shared among all origins. This global sharing has introduced side-channel attacks where one origin can detect whether another has loaded a resource via a timing-attack on the HTTP cache.

The HTTP Cache Partitioning Explainer doc describes this in more detail, as well as the proposed mitigation of double-keying the cache, a project called SplitCache. This effectively **disallows** resources loaded and stored into the HTTP cache by one origin to be served from the cache when a different origin requests them. Consequently, this breaks the main use-case for cross-origin prefetch. This document covers some proposals to retain the current behavior of cross-origin prefetch for navigations in the midst of SplitCache, while also discussing various privacy and compatibility implications.

Use-cases

We use the `as` attribute on the <link> element as a signal to distinguish prefetching main resources (as=document) and prefetching subresources (as=anythingElse). With this distinction, there are several use-cases for <link rel=prefetch> to consider:

1. Prefetched cross-origin main resources reused on cross-origin navigations
2. Prefetched cross-origin subresources reused on same-origin pages
3. Prefetched cross-origin main resources reused as same-origin subresources (<iframes>)
4. Prefetched cross-origin subresources reused on cross-origin navigations to those subresources
5. Prefetched cross-origin subresources reused on cross-origin pages as subresources
    a. There are no plans to support this, due to privacy & tracking concerns

The first 4/5 would be good to support, the first 2/5 being the highest priority.

Proposals

This section contains several proposals for the work this project requires. The first outlines changes that need to prefetch request in Blink from a privacy perspective. The next four are candidates for making cross-origin prefetches able to be matched from HTTP cache on future navigations. The last section covers proposed changes to the HTTP cache to support matching cross-origin prefetch resources.

Prefetch Request Changes

Spec issue Resource-Hints#82 indicates that to support cross-origin prefetch in a double-key cached world, prefetch requests should have the following properties:

- Redirect mode: manual/error
- Credentials mode: same-origin
- Mode: kNoCors
- Service-workers mode: none
- Referrer policy: no-referrer
- Origin: client origin
    - yhirano@ included this point just for clarity

In Blink, most of these properties are determined in PreloadHelper::PrefetchIfNeeded(). The ResourceRequest's *mode_*, *credentials_mode_*, and *referrer_policy_* are all influenced by attributes that are taken into account here. We'll want to ensure that these properties are instead unaffected by attributes (i.e., prefetch requests should never be sent with a `Referer` header, regardless of the `referrerpolicy` attribute). In order to introduce these changes, we should gate them behind flags that we'll enable at around the same time that kSplitCacheByNetworkIsolationKey gets enabled by default.

These changes to the request properties could affect existing behavior, and are not yet spec'd. We should experiment with these changes and discuss data at TPAC 2019. For now, I've published:

- Intent-to-Implement
- Chromestatus entry
- Questions to clarify proposed spec changes
- Request for TAG review

If a developer supplies values for any of these attributes, and we do not honor them, what should we do:

- Fail the request + display a console warning indicating why the request failed
    - +yhirano indicates this is the "safest" approach (preventing uncredentialed responses from matching with credentialed requests)
- Display a console warning indicating some attributes are not being considered, and continue with the request as usual

Major CLs

- Introduce PrefetchRedirectError flag
- Prefetch redirect histogram
- PrefetchPrivacyChanges flag + no `Referer` header

❌Single-key-eligible Prefetches

One option to support all use-cases described above would be to mark prefetched resources as single-key-eligible. In other words, the resource would not be subject to the exclusive single-origin partitioning that SplitCache naturally introduces. Instead, it would support matching via a classic single-key. A deeper description of this concept can be found in the Single-Key-Eligible proposal doc + slides.

While this solution would satisfy all of our needs without implementing an independent prefetch cache, it essentially requires a refactoring of the HTTP cache such that it is less like a hash table whose keys/values are:

{<top_frame_origin, RequestURL>**:** Resource}

...to instead resemble:

{<RequestURL>**:** [Resource, AllowedOriginsList, SingleKeyEligible=true|false]}

The AllowedOriginsList is a list of origins that can reuse this resource (similar to the partitioning key). If an origin wants to reuse the resource associated with <RequestURL> but does not appear in the AllowedOriginsList, it can only do so if both of the following are satisfied:

- The cache entry's SingleKeyEligible flag is *true*
- The request is for a top-level navigation

This naturally introduces a form of deduplication that the current HTTP cache does not do, which is a change that should probably considered separately. Due to the significant changes required for this solution, it is not currently an option, though theoretically it satisfies the above use-cases.

❌Separate Prefetch Cache

Another option is to store prefetch resources in a separate single-keyed cache, independent of the traditional HTTP cache. This is the approach Apple is looking into. Long-term it might be worth investing into, but the significant effort required in the midst of a simpler solution that gets us most of what we want (below) puts it on the backburner.

More information on the origins of this proposal can be found in Kinuko's early project doc. This solution would also satisfy all of the above use-cases.

❌On-the-fly URLLoaderFactory Creation

A cross-origin prefetched main resource can only be reused on navigation if it is placed in the cache partition corresponding to its origin (not the parition corresponding to its requestor). One way to make this happen is to fetch these resources with a new URLLoaderFactory whose NetworkIsolationKey is modified for the cross-origin prefetch.

Summary: When the renderer sees a cross-origin prefetch, instead of routing it to its ChildURLLoaderFactoryBundle::prefetch_url_loader_factory_ like usual, it creates a new Remote<URLLoaderFactory> and passes the corresponding PendingReceiver to the browser. The message instructs the browser to create a new URLLoaderFactory whose NetworkIsolationKey is the cross-origin itself. The browser creates the factory, registering it with the PrefetchURLLoaderService. The renderer can immediately use this remote factory to fetch the prefetch. We'll want to reuse this factory for preload-on-prefetch requests too, because they should be keyed under their parent prefetch's origin. Therefore, we should save this factory to the renderer's bundle for later reuse.

These ad-hoc factories should be used for any cross-origin prefetch or cross-origin preload-on-prefetch requests destined for a particular origin. Since these factories are essentially tied to an origin (via the NetworkIsolationKey), it seems reasonable to introduce a CrossOriginPrefetchLoader map to URLLoaderFactoryBundle or ChildURLLoaderFactory to store them. Currently ChildURLLoaderFactoryBundle::CreateLoaderAndStart has special-prefetch-logic that sends prefetch requests to prefetch_loader_factory_. We would extend this logic to detect cross-origin prefetches, and consult the CrossOriginPrefetchLoader map instead (or create a new entry). Same-origin prefetches will behave as normal, being routed through the existing prefetch_loader_factory_.

Security concerns: The browser is setting the NetworkIsolationKey of the request based on information directly obtained from the renderer. A compromised renderer could tell the browser to create a URLLoaderFactory with an arbitrary NetworkIsolationKey, and could insert entries into arbitrary cache partitions. This may not be too serious for main resources because their reuse is restricted to top-level navigations (see HTTP Cache Match-Time Logic), but is much more problematic for preloads-on-prefetches resources.

Preloads-on-Prefetch: Only the renderer should need to keep track of whether a ResourceRequest is a preload-on-prefetch. When it comes across a preload-on-prefetch, it should mark in such a way that ChildURLLoaderFactoryBundle routes it to the correct factory in the CrossOriginPrefetchLoader map. This factory will correspond to its parent prefetch's origin. After this, no further work needs to be done by the browser/networks service.

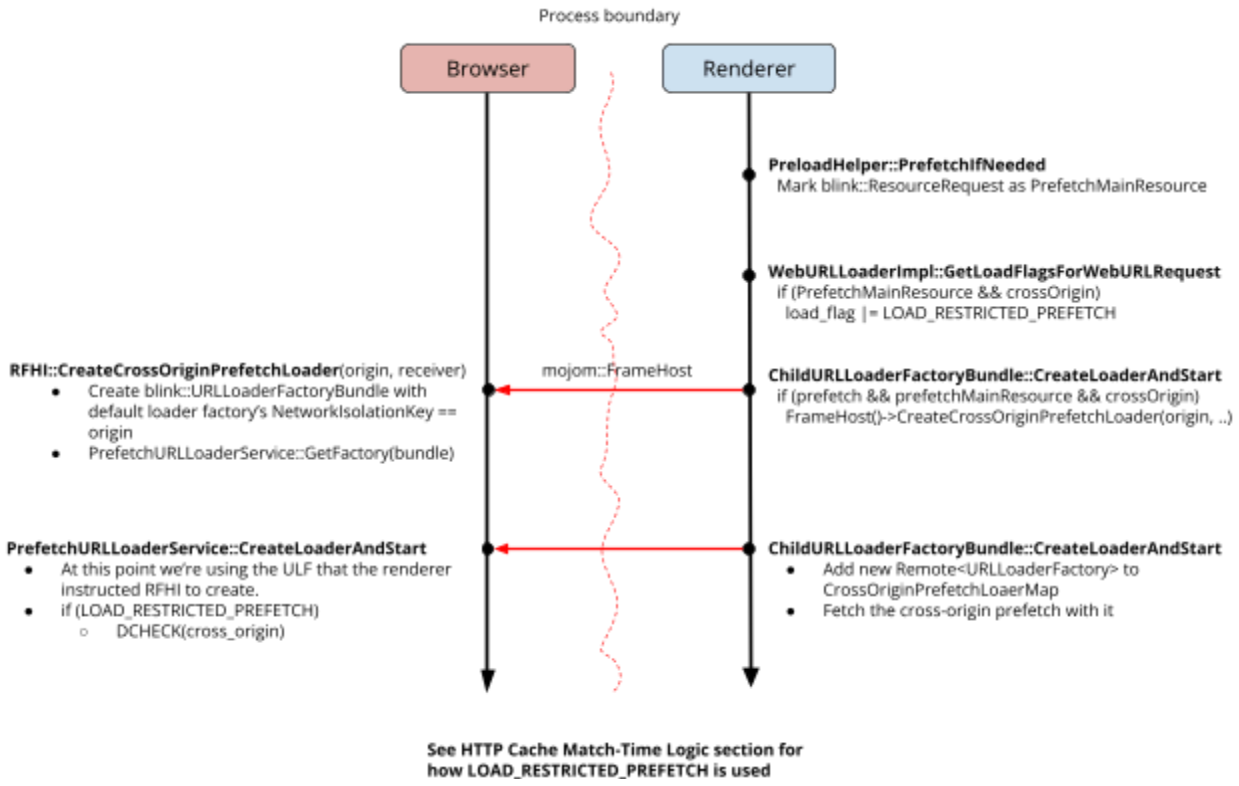
Pros:

- Network service does not need to be modified
- Complexity is scoped to renderer & browser communication. Everything else “just works”
- Preloads-on-prefetch requests only require additional routing by the renderer

Cons:

- Overhead of IPC and URLLoaderFactory creation for every new cross-origin prefetch
- Not secure enough of a proposal for preloads-on-prefetch requests. **This is the reason we cannot go with this proposal.**

Main Resource Implementation Design



(Figure illustrating new flow of cross-origin main resource prefetches)

Proof-of-concept CL: http://crrev.com/c/1735366

Renderer => Browser IPC

The renderer’s ChildURLLoaderFactoryBundle needs to communicate to the browser that it wants a new URLLoaderFactory, specifically for prefetching, with a non-default NetworkIsolationKey. I propose we extend mojom::FrameHost like so:

```
CreateCrossOriginPrefetchLoaderFactory(
  url.mojom.Origin cross_origin,
```

```
    pending_receiver<network.mojom.URLLoaderFactory> factory_receiver
)
```

CreateCrossOriginPrefetchLoaderFactory() would be called from ChildURLLoaderFactoryBundle::CreateLoaderAndStart for cross-origin prefetches. The method would be implemented by RenderFrameHostImpl, which would create a new URLLoaderFactory for subresources, in a way similar to what we do now. It would register this loader factory with the PrefetchURLLoaderService.

Creating the URLLoaderFactory with non-default NIK

Presently, RenderFrameHostImpl::CreateNetworkServiceDefaultFactoryAndObserve is used to create subresource loader factories for the renderer. This delegates to CreateNetworkServiceDefaultFactoryInternal, passing a private network_isolation_key_ to the RenderProcessHostImpl to create the factory. Therefore, all subresource factories have the same NIK. I propose adding a NetworkIsolationKey parameter to CreateNetworkServiceDefaultFactoryAndObserve, so that we can pass in our own NetworkIsolationKey when we need to, and default to the RenderFrameHost's otherwise.

Introduce CrossOriginPrefetchLoaderMap

When the renderer messages the browser (its frame host) to create a new URLLoaderFactory for a cross-origin prefetch, it should store the corresponding Remote<URLLoaderFactory> for later-use. I propose introducing CrossOriginPrefetchLoaderMap to ChildURLLoaderFactory, whose type would be std::map<url::Origin, mojo::Remote<network::mojom::URLLoaderFactory>>. It would store prefetch URLLoaderFactories associated with a given origin. The renderer consults this map when a cross-origin prefetch is encountered. If no factory for a given origin exists, it must must message its frame host to create a new one.

Marking a ResourceRequest as main-resource prefetch

We've opted to use `as=document` as a tentative signal for whether a cross-origin prefetch is intended to be used as a main resource. These requests should be routed through newly-created URLLoaderFactories in the above map.

1. Detect as=document in PreloadHelper::PrefetchIfNeeded
    a. Set a new property on blink::ResourceRequest indicating that the prefetch is a main resource
2. Convert this new property to a new load flag. WebURLRequest::GetLoadFlagsForWebURLRequest can be responsible for this.

✅PrefetchURLLoaderService routing

The above proposal creates a new URLLoaderFactory for each distinct cross-origin prefetch. These factories are stored in the renderer, which makes them more vulnerable and puts

unnecessary trust in an untrusted process. A lighter-weight and more-secure approach is to create a single URLLoaderFactory per frame, that only handles cross-origin prefetches.

The renderer can indicate that a network::ResourceRequest is a cross-origin prefetch and wishes to be cached specially. This requires the ability for the renderer to express that a network::ResourceRequest was prefetched with `as=document`, and is intended for top-level navigations. We could use LOAD_MAIN_FRAME_DEPRECATED to express this, however it is deprecated. Instead we'll introduce a new load flag LOAD_RESTRICTED_PREFETCH.

When the PrefetchURLLoaderService consumes the network::ResourceRequest, it can check for the LOAD_RESTRICTED_PREFETCH flag. If the flag is present, the request must meet the following (security) conditions:

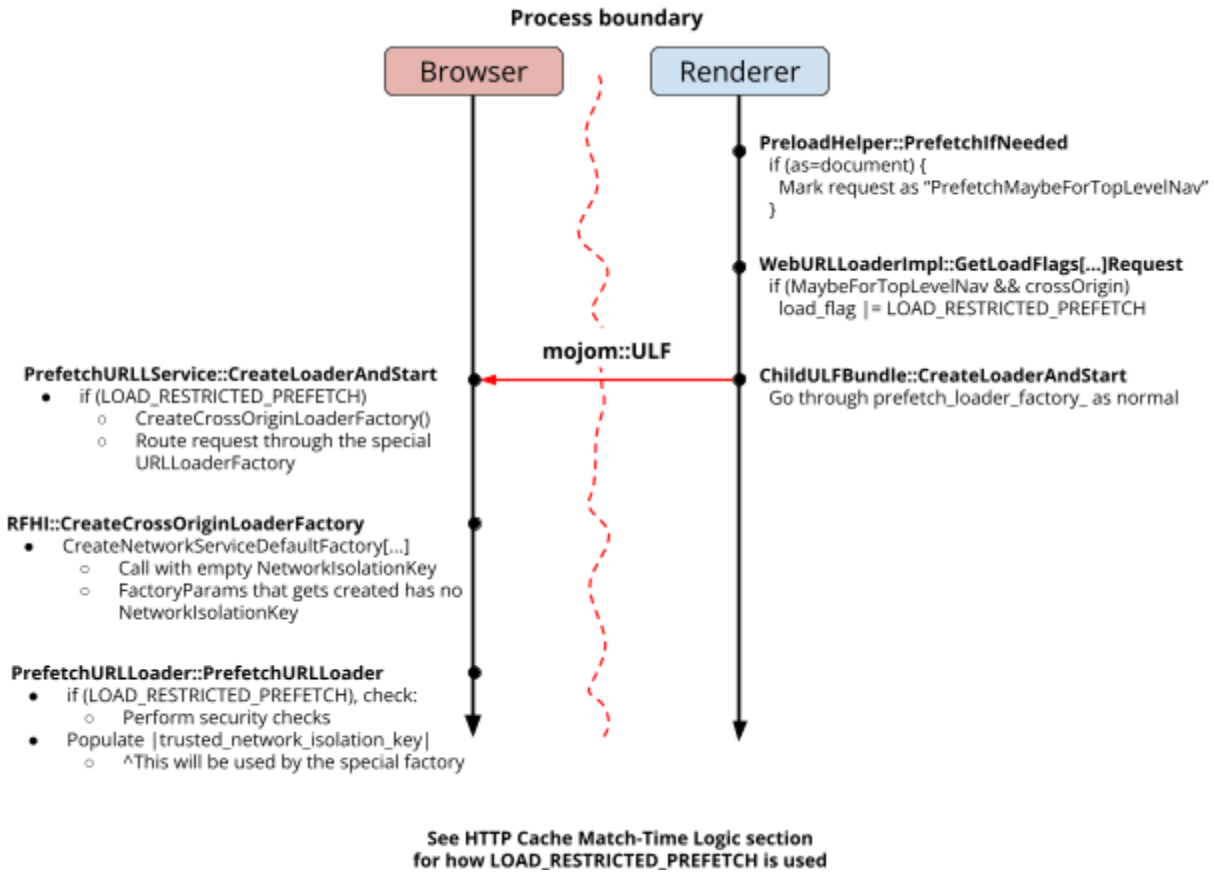
- Is a cross-origin request
- Redirect mode is kError (when the PrefetchRedirectError feature is enabled)
- Credentials mode is kNoCors
- LOAD_*CAN_USE*_RESTRICTED_PREFETCH flag is not set

If any of the above conditions are not met, the request should be completed with an error response immediate; this could indicate a compromised renderer tampering with the request. We don't want to DCHECK here, because that allows a compromised renderer to crash the browser process.

The PrefetchURLLoaderService will route this prefetch request to a special factory it maintains in its per-frame BindContext. This factory should be created lazily when necessary, as not all frames will have cross-origin prefetches. The factory will be initialized with no NetworkIsolationKey.

Instead, PrefetchURLLoaderService populates the prefetch's NetworkIsolationKey. This is because it should be cached under the partition corresponding to the prefetch's cross-origin, and the PrefetchURLLoaderService is "trusted" and knows the target partition.

Main Resource Implementation Design



(Figure illustrating new flow of cross-origin main resource prefetches)

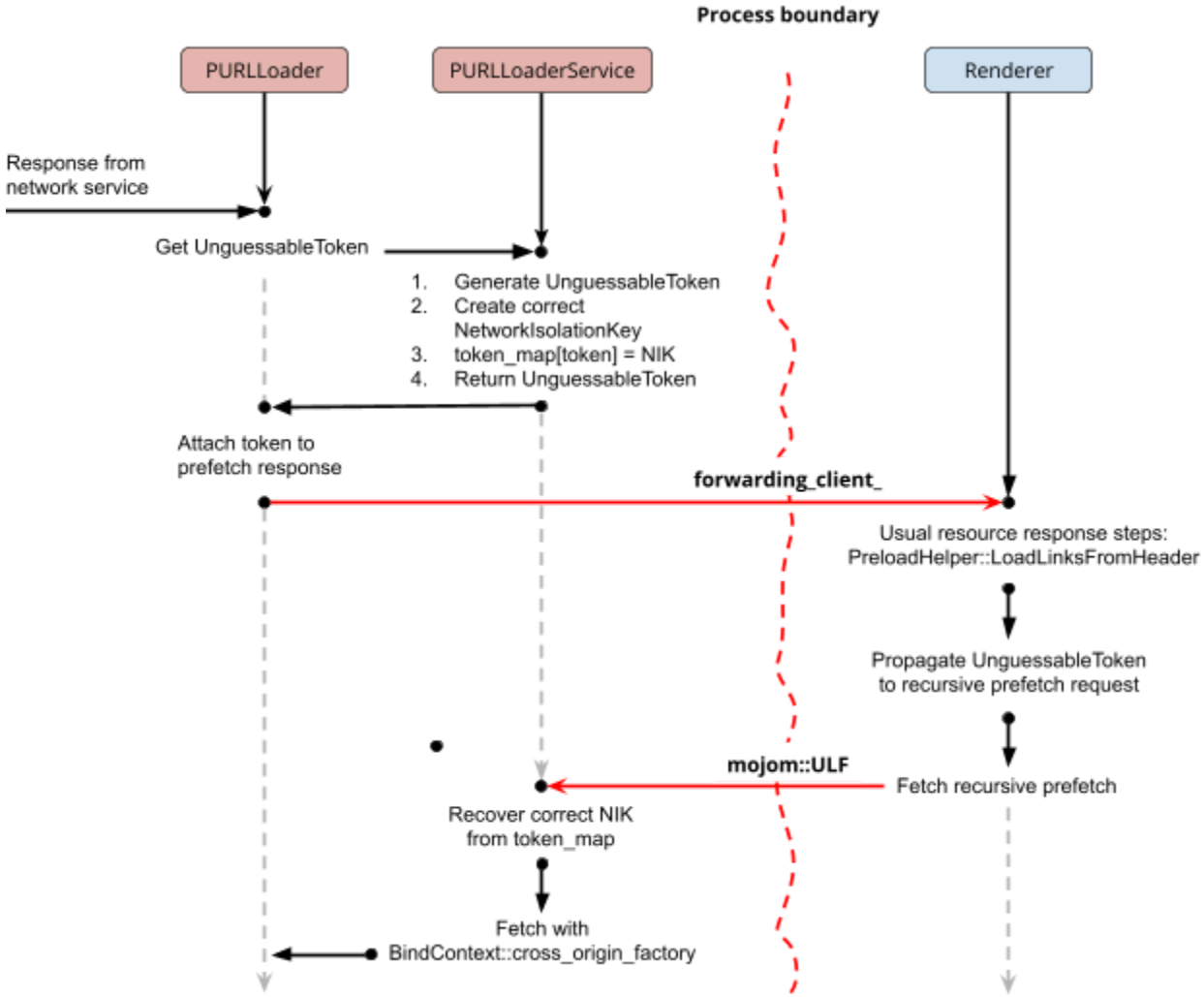
Recursive Prefetch Design

A key part of this work is ensuring that preload header requests on prefetch responses (we'll call them recursive prefetches) are cached with the parent prefetch. The renderer is responsible for parsing the `Link` header for these requests, and using frame-specific information (like viewport size) to inform these requests.

We debated on initiating recursive prefetch requests from the browser process instead of the renderer, because the browser process knows the correct NetworkIsolationKey that these should be fetched with. However, since the requests need frame-specific information (e.g., for imagesrcset/imagesizes), it makes more sense to continue to let the renderer fetch them.

We cannot simply pass the correct NetworkIsolationKey to the renderer to fetch these requests with, because we cannot trust the renderer to maintain the integrity of the key (it is "untrusted"). Instead, after the renderer fetches the request, we need to re-associate the

recursive prefetch request with the correct NetworkIsolationKey in the browser process. We do so by introducing a map to PrefetchURLLoaderService of the following shape: {UnguessableToken ⇒ NetworkIsolationKey}. When we get a prefetch response, we add the correct NetworkIsolationKey to the map via generated token. We give the renderer this token, and recover the correct NIK when renderer fetches the request.



Pros:

- **Cheap**: Less overhead than the earlier proposal; no new IPC messages
- **Efficient**: We're not creating N cross-origin prefetch loader factories for N different cross-origin prefetches. Instead we're reusing a single "special" frame-bound URLLoaderFactory for all cross-origin prefetches
- **Secure**: Relies on the renderer as little as possible

Cons:

- Complexity is somewhat less tightly-scoped; changes are not scoped to the renderer, but would likely require renderer + browser changes

Major CLs

- CL 1730572 - CreateNetwork[...] should accept a NetworkIsolationKey
- CL 1735366 - Main resource cross-origin prefetch + SplitCache
- CL 1775646 - Simplify prefetch + cross-origin prefetch logic
- CL 1772896 - Prefetch's preload headers are cached correctly
- See Testing Plan for tests

✅HTTP Cache Match-Time Logic

The above proposal describes changes that need to be made when a cross-origin prefetch request is made. This section describes changes that need to be made to when a cross-origin prefetch is to be *matched* from the HTTP cache.

Same-origin prefetch resources can be matched from the cache as usual; no changes are necessary. However, cross-origin prefetches sitting in the HTTP cache must only match top-level navigation requests, for privacy reasons. Therefore, it is necessary to be able to distinguish a cached cross-origin prefetch from others.

HttpResponseInfo
Add a new flag |restricted_prefetch|. This flag should be set when an HttpResponseInfo's properties, such as |unused_prefetch|, are set. We should have some logic like so:

```
HttpNetworkTransaction::Start (...) {
  ...
  if (request_->load_flags & LOAD_RESTRICTED_PREFETCH) {
    DCHECK(request_->load_flags & LOAD_PREFETCH);
    response_.restricted_prefetch = true;
  }
  ...
}
```

This indicates that a prefetch has been handled specially in some way, and its reuse should be restricted to requests that can explicitly use it (see below).

LOAD_CAN_USE_RESTRICTED_PREFETCH
We can mark a request that should be able to use restricted prefetches by introducing a new LOAD_CAN_USE_RESTRICTED_PREFETCH flag. This should only be set for top-level navigation requests. We can do so with the following logic:

```
(navigation_url_loader_impl.cc)
CreateResourceRequest() {
  ...
  if (request_info->is_main_frame) {
    load_flags |= net::LOAD_CAN_USE_RESTRICTED_PREFETCH;
```

```
  }
  ...
}
```

Limiting the use of restricted prefetches

To limit the reuse of cache entries marked as |restricted_prefetch|, we'll want the following cache logic:

```
HttpCache::Transaction::DoCacheReadResponseComplete() {
  ...
  if (response_.restricted_prefetch &&
      !(effective_load_flags_ & LOAD_CAN_USE_RESTRICTED_PREFETCH)) {
    TransitionToState(STATE_SEND_REQUEST);
    return OK;
  }
  ...
}
```

Main CLs for this work:

- Introduce //net RESTRICTED_PREFETCH load flags
- Stop toggling prefetch values before/after write

Special Cases

This section attempts to address some special cases regarding the reuse of prefetched resources:

1. Should prefetches (with LOAD_PREFETCH) be able to reuse prefetched responses marked as restricted?
    - ✓ No. If we allowed this, origins would potentially be able to determine whether other origins have prefetched a particular resource, which is a privacy leak
2. What should the correct behavior be when a cross-origin main-resource document is prefetched, and attempted to be reused by a cross-origin subresource *instead* of a navigation?
    - ✓ Evict the restricted prefetch from the cache, go to the network to retrieve it. Ensure that the newly-written entry is not marked as "restricted". Right when this happens essentially we should forget that the prefetch was restricted or even exists

Testing Plan

Prefetch + SplitCache

Before the work described by this doc, the following prefetch content_browsertests fail with the SplitCacheByNetworkIsolationKey feature enabled (**6 tests total)**:

- *PrefetchBrowserTest.CrossOrigin/*
- *PrefetchBrowserTest.CrossOriginWithPreload/*
- *PrefetchBrowserTest.CrossOriginSignedExchangeWithPreload/1
- *SignedExchangePrefetchBrowserTest.PrefetchMainResourceSXG_CrossOrigin/0

Tests that need to be implemented:

- Browsertest: cross-origin **document** prefetch + cross-origin navigation only downloads once
- Browsertest: cross-origin **non-document** + cross-origin navigation downloads twice
- Browsertest: cross-origin **document** prefetch + cross-origin iframe downloads twice
    - This tests that cross-origin prefetches can only be reused for top-level navigations
- Browsertest: cross-origin **non-document** + cross-origin subresource downloads once
    - This tests that cross-origin non-document prefetches are not cached under the cross-origin resource's origin, but are usable by the current origin

The above testing scenarios are implemented with CLs:

✓ Add cross-origin main resource prefetch + SplitCache tests
✓ Reorganize Prefetch + SplitCache browsertests
✓ Augment PrefetchBrowserTest.CrossOriginWithPreload

All tests are passing with the implemented proposals \o/

Prefetch Request Tests

- Prefetch + redirect mode (blocked on pending discussion of redirect mode)
    ✓ CL 175554
- Prefetch + referrerpolicy
    - `Referer` header is never sent, regardless of whether the referrerpolicy attribute exists, and its value
    ✓ CL 1781303
- Prefetch + same-origin credentials
    - Test that credentials are not sent cross-origin

    - Test that `crossorigin` attr is not honored (depending on how we treat the attribute)
- Prefetch + no-cors mode
    - Test that request mode is no-cors
    - Test that `crossorigin` attr is not honored (depending on how we treat the attribute)

Privacy Concerns

Cross-origin subresources: At this time we have no plans to support cross-origin subresources being reused on cross-origin pages, due to similar privacy concerns that prompted double-keyed caching in the first place.

Preloads-on-prefetch: The preloads-on-prefetch behavior cannot expose any more information than could be exposed by <link rel=prerender>. This is good, however I think it is not 100% determined that even <link rel=prerender> is totally safe? Furthermore, Potassium team has expressed concerns with preloads-on-prefetches, and are not totally comfortable with the concept.

Future Work

This section contains information on future work related to the project explained by this document.

- Evicting main-resources out of the cache after a certain period of time, or only allowing next-navigation in some way
- Supporting prefetch reuse on cross-origin navigations to non-document types, because there isn't a good reason to support this, and non supporting this is an implementation-detail.
- Consider `target` attribute for <link> to determine a "safe" origin. Does this buy us anything? Yoav mentioned Sleevi says it would not let us introduce credentials.
- Lower-level networking work to prevent a cross-origin server from identifying a client. See this document's comments from [1]

Post-TPAC update: See kinuko@'s TPAC Prenavigate discussion notes (2019 Sept)

[1]:

> **mmenke@**:
>
> Ok...a.com wants to share user data with b.com. It prefetches, using a cross-origin request, https://b.com/<user-id>. This uses a socket with https://b.com's network isolation key. Then it also prefetches https://b.com/, which again uses b.com's network isolation key, and the same

socket. Since the requests use the same socket, b.com can include the user ID in the root document. And thus the user ID is shared cross domain.

In fact, a.com can keep on doing this with different third party domains in the background at its leisure.

Admittedly, top level redirects with link decoration can do the same thing. Just wondering about privacy models here.

Actually, if these are for the main frame, they'd have to use first party cookies, so you wouldn't even need to rely on socket reuse. Just send a user ID.

yhirano@:

- In your example, is the user ID assigned by a.com?

- When does the undesired information sharing happen? Is it when a.commakes a prefetch, or when the user navigates to b.com?

kinuko@:

Reg: first-party cookies: It's discussed that prefetches should be made without credentials/cookies (even for main resources. how it should match with navigation requests that are by default credentialed is still being discussed).

For the socket: do you mean the server side can see the user ID and can associate two requests from the fact that they share the same socket? Could it imply that prefetches shouldn't basically use the same socket (and also # of prefetches to one domain should be probably limited ot 1)?

(I don't think we've thought through the implication of socket sharing situations)

mmenke@:

Yes, I mean reusing the socket allows two requests to be associated, even if they're both uncredentialed. It's basically the link decoration problem, except taking advantage of preconnects and socket pools, instead of decorating the main frame request.

Limiting requests to a single preconnect and not reusing the socket would fully address the issue (In combination with only allowing prefetches to be used for the main frame, at least).