Chromium Design Doc

JavaScript API: fetchLater

This is a public document

mych@chromium.org

Status: draft

Background

This document outlines the Chromium implementation of the `fetchLater` API, a JavaScript function to request a deferred fetch request. The deferred request is queued by the browser, and will be invoked in one of the following scenarios:

- The document is destroyed.
- The document is bfcached and not restored after a certain time.

The API is the spiritual successor of the experimental PendingBeacon API, but focuses only on the “deferred” intrinsics of beacon requests, and is defined as part of the existing Fetch spec.

See the explainer for API overview, and the spec (draft) for detailed behaviors. Add comments to the pull request to discuss any questions about the API.

Overview

The implementation for the JavaScript `fetchLater` API will be split over two main processes:

Renderer Process

- Provides the JavaScript API for web developers to schedule deferred fetch requests. A deferred fetch request can be configured or aborted similarly to a fetch request. The response of such a request will be dropped.
- The API is backed with the same FetchManager used by `fetch` API, which queues deferred requests, performs per-origin quota validation, and sends requests upon context destruction or upon reaching `activateAfter`.
- `FetchLaterResult` tells the `sent` state of a deferred request by looking into FetchManager.

Browser Process

- Keep deferred requests alive until they are sent, even after the render is gone.
- Handle redirects in browsers.
- Drops all responses.

Design Proposal

There are two options to implement the fetchLater API:

Option 1: Reuse PendingBeacon Implementation

The experimental PendingBeacon API is a limited version of the fetch API that only supports HTTP GET and POST requests without custom headers and any other RequestInit fields. Hence, its implementation does not take advantage of any of the fetch API's infrastructure, and instead relies on a per-Document PendingBeaconDispatcher in the renderer to serialize every PendingBeacon's URL and request body, and passes them to a per-Document PendingBeaconHost object in the browser. The request is eventually handled by a SimpleURLLoader which has the same lifetime as a PendingBeaconHost. However, its capability is pretty limited in that it only supports "simple" requests.

To implement fetchLater, the RequestInit and relevant behaviors need to be supported, and the full fetch algorithm has to be run, which are both currently backed by the complex logic under blink/renderer/core/fetch that processes fetch requests and communicates with network service directly. The PendingBeaconDispatcher is useless here.

To make deferred requests survive a destroyed renderer, there must be some counterpart in the browser process to hold the request mojo. However, PendingBeaconHost is not designed to act as a middleman between the renderer and network service. Refactoring it to do so will take more effort than choosing Option 2.

Option 2: Reuse KeepAliveURLLoaderService (Selected)

The Fetch 'keepalive' Infra Migration project introduces an in-browser KeepAliveURLLoaderService, which acts as a middleman between the renderer process and network service. For every fetch keepalive request, it goes through the service, and a KeepAliveURLLoader is created in the browser to hold the mojo pipes that connect the renderer and the network service. When KeepAliveURLLoader is disconnected from the renderer, i.e. the renderer is gone, it will take over the rest responsibility to ensure that the request and its redirects are properly handled.

A fetchLater(..) request behavior is similar to a fetch(..., {keepalive: true})'s. The main difference is that the former is only "sent" at the last possible moment by default, while the latter can be a normal fetch() request if not fired at the end of a Document. Given the fact, it means that the browser-side part of the keepalive infrastructure can be reused. Only new JavaScript wrapper and renderer-side logic for fetchLater() needs to be implemented. Furthermore, the activateAfter behavior can be solely implemented at the renderer side.

Detailed Design

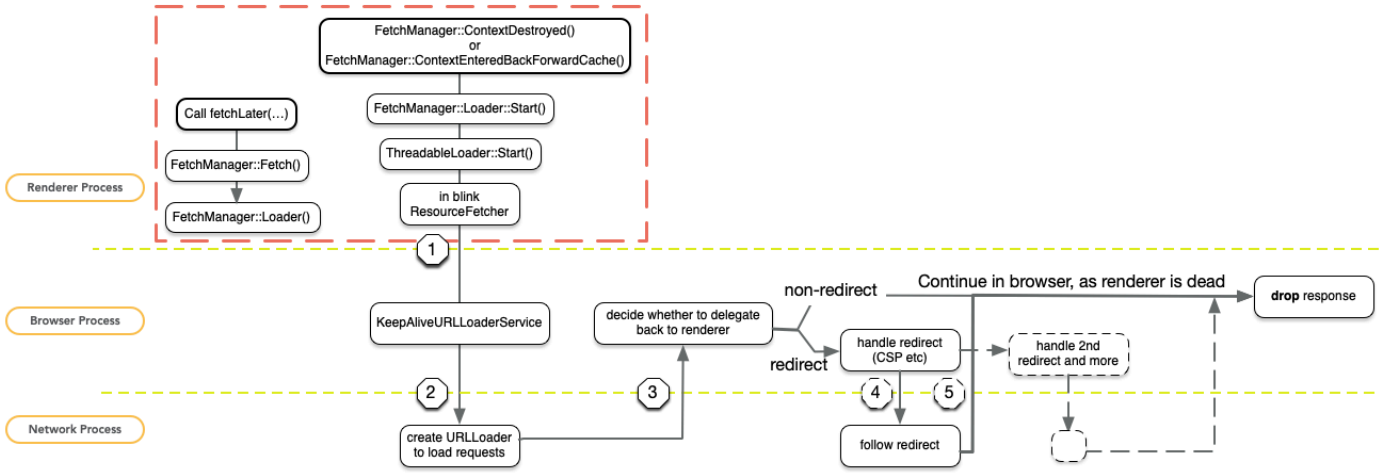


Diagram: Flow of Process Hop for a fetchLater Request

Only the components in the red box are modified by this design. The rest are from Fetch 'keepalive' Infra Migration.

IDL & Wrapper

A new V8 wrapper method, fetchLater(), under class GlobalFetch is defined for the new API, to utilize the existing ScopedFetcher implementation that associates a fetch to the right context, e.g. window, and stores total fetch counts. The main differences are that

- fetchLater returns a FetchLaterResult object instead of a ScriptPromise. Accessing the `activated` field of the object should tell if the deferred fetch has been scheduled by the browser to send.
- fetchLater takes a DeferredRequestInit, which provides an optional `activateAfter` field to accelerate request sending after calling the API. The field should be validated in a new class DeferredRequest which subclasses Request to take care of the fetchLater steps (1) ~ (6), including enforcing the keepalive field to true.

```
C/C++
[ImplementedAs=GlobalFetch] partial interface Window {
    [CallWith=ScriptState, NewObject, RaisesException] FetchLaterResult fetchLater(
        RequestInfo input, optional DeferredRequestInit init = {});
};
dictionary DeferredRequestInit : RequestInit { DOMHighResTimeStamp activateAfter; };
interface FetchLaterResult { readonly attribute boolean activated; };

class GlobalFetch {
  static FetchLaterResult fetchLater(ScriptState* state, LocalDOMWindow&, const V8RequestInfo* input,
```

```
                                    const DeferredRequestInit* init, ExceptionState& e) {
    return ScopedFetcher::From(window)->FetchLater(state, input, init, e);
  }
};

class GlobalFetchImpl : public ScopedFetcher {
  static FetchLaterResult FetchLater(...) {
    DeferredRequest* r = DeferredRequest::Create(state, input, init, e);
    if (e.HadException() return FetchLaterResult();

    auto result = fetch_manager_->FetchLater(state, r->PassRequestData(state), r->signal(), e);
    if (e.HadException()) return FetchLaterResult();
    return result;
  }
};
```

FetchLaterManager

Every call to FetchManager::Fetch() adds a new FetchManager::Loader object to FetchManager. The Loader implements ThreadableLoaderClient and holds an instance of ThreadableLoader, which is responsible for performing different types of fetch requests, e.g. http fetch or data fetch. At the end of the call, it triggers FetchManager::Loader::Start(), which in return triggers ThreadableLoader::Start() and ResourceFetcher::RequestResource(). The fetching is then kicked off in ResourceFetcher::StartLoad() after a series of ResourceRequest validation.

After some discussions with code owners of ThreadableLoader/ResourceFetcher/ResourceLoader, it appears that FetchLater requests are not welcomed to reuse those logic, as most of the request/response handling will not happen there.

Therefore, a new FetchLaterManager should be created. And a new method FetchLaterManager::FetchLater() is defined to support the deferred fetching behavior. Unlike FetchManager::Fetch(), calling FetchLater() only creates a subtype of FetchManager::Loader object, FetchMnager::**DeferredLoader**.

DeferredLoader should perform the Deferred fetching algorithm to prepare a request for the loader, including validating the total deferred bytes for every origin. The loader will not be started immediately, as specified it must “**wait until the last possible time”**, i.e. when ExecutionContext is destroyed, or after “activateAfter” time, to notify the browser to start. In implementation, it calls an IPC FetchLaterLoader::SendNow(), which will notify the intermediate loader in the browser to start sending the request.

NOTE: FetchManager::Loader() ctor takes a ScriptPromiseResolver, which does not exist in fetchLater() call. Hence, some of the Loader's logic needs to be updated to not rely on it.

```C/C++
class FetchLaterManager {
  FetchLaterResult FetchLater(...) {
    if (request_size_in_bytes > 64kB ||
        deferred_bytes_for_origin_[request.origin()] + request_size_in_bytes > 64kB) {
      e.QuotaExceededError("The provided request body exceeds the maximum supported size for the
origin");
      return FetchLaterResult();
    }
    deferred_bytes_for_origin_[request.origin()] += request_size_in_bytes;
    auto* deferred_loader =
      MakeGarbageCollected<Loader>(GetExecutionContext(), this, /*resolver=*/nullptr,
                                   request, &state->World(), signal);
    deferred_loader->Start();
    deferred_loaders_.insert(deferred_loader);
    return FetchLaterResult(this);
  }

  void ContextDestroyed() {
    for (auto& deferred_loader : deferred_loaders_) {
      deferred_loader->Dispose();
      // Update FetchLaterResult
    }
  }

  void ContextEnteredBackForwardCache() {
    // Optional: the loaders may choose to optimize to start in batch.
    for (auto& deferred_loader : deferred_loaders_)
      // Notify browser to start the deferred request.
  }

 private:
  HeapHashSet<Member<Loader>> deferred_loaders_;
  HeapHashMap<Origin, uint64_t> deferred_bytes_for_origin_;
};
```

~~TODO: As DeferredLoader inherits from FetchManager::Loader, it's worth noting that FetchManager::Loader::Dispose() detaches its threadable_loader_ for a keepalive request, which is not necessary after keepalive migration~~

~~TODO: FetchManager::Loader::Abort() cancels threadable_loader_, which will not stop the browser from sending the deferred request out.~~

FetchLaterResult

FetchLaterResult reflects whether a deferred request, i.e. a FetchManager::DeferredLoader(), has started to send or not.

```C/C++
class FetchLaterResult {
  bool sent() { return loader_.invokeState() == InvokeState.SENT; }
 private:
  Member<FetchManager::DeferredLoader> deferred_loader_;
};

class FetchManager::DeferredLoader {
 // See deferred fetch record's invoke state.
 enum InvokeState {
   DEFERRED = 0;
   SCHEDULED = 1;
   TERMINATED = 2;
   ABORTED = 3;
   SENT = 4;
 }
 private:
  InvokeState invoke_state_;
};
```

From FetchManager to ResourceLoader

As a fetchLater() call supports an extension to the same arguments as fetch() call, its arguments like a Request or a RequestInit should be applied with the same logic as fetch()'s.

Most of the initial fetch algorithms are implemented when running FetchManager::Loader, which the proposed design will also run via its subclass FetchManager::DeferredLoader. However, beyond this point, there are still many checks & potential mutations to the request before the request is actually started to load (see below). Given its current shape, it is dangerous to make assumptions and extract certain logic out of the entire call sequence. Not to mention duplicating them.

A fetch blink::ResourceRequest, after created by FetchManager::Loader, gets through the following call sequence, where each of them may perform checks or update the request:

1. ThreadableLoader::Start():
    a. Verify request.GetMode()
    b. Security check: IsNoCorsAllowedContext()
    c. Verify request.CorsPreflightPolicy() || cors::IsCorsEnabledRequestMode(request.GetMode())
    d. probe::ShouldBypassServiceWorker()
2. ResourceFetcher::RequestResource(): a huge section
    a. Constructing a new copy of ResourceRequest, seems to be used to obtain top_frame_origin for the original Resource.
    b. Call **PrepareRequest()**, and return ResourceForBlockedRequest()
    c. [not relevant] CreateResourceForStaticData() for data url or archive

    d. [not relevant] preload
    e. [not relevant?] UpdateMemoryCacheStats() and check RevalidationPolicy
    f. [not relevant?] Increase the priority of an existing request if the new request is of a higher priority.
    g. [?] DCHECK(EqualIgnoringFragmentIdentifier(resource->Url(), params.Url()))
    h. [not relevant?] MaybeSaveResourceToStrongReference(resource)
    i. [not relevant] ImageLoadBlockingPolicy-related
    j. Call **ResourceNeedsLoad()**
    k. Several Blink.Fetch histogram logging.
    l. Call **StartLoad()**
3. ResourceFetcher::PrepareRequest()
    a. Determine ResourceLoaderOptions
    b. [not relevant] AttachWebBundleTokenIfNeeded()
    c. Overwrite content type: params.OverrideContentType(factory.ContentType())
    d. Security Check: Context.**CehckCSPForRequest**()
    e. May modify URL: Context().**PopulateResourceRequest**()
    f. [not relevant] Compute ResourceLoadPriority
    g. ResourceRequest's many fields may be changed: SetPriority, SetRendererBlockingBheavior, SetCacheMode, SetRequestContext, RequestDestination, SetHeader (prefetch/prerendering only), SetAllowStaleResponse, Set**Referrer, SetAllowStoredCredential.**
    h. Context().**AddAdditionalRequestHeaders**(resource_request);
    i. Call **Context().CanRequest()**
    j. Call **Context().PrepareRequest()**
4. BaseFetchContext::CanRequest()
    a. Call CanRequestInternal()
        i. Check ShouldBlockRequestByInspector(resource_request.Url())
        ii. Add console message if checking request_mode / origin fail
        iii. Security Check: cors::CalculateCorsFlag() for same origin
        iv. Security Check: CheckCSPForRequestInternal()
        v. AllowScriptFromSource()
        vi. Security Check: ShouldBlockFetchByMixedContentCheck()
        vii. Block deprecated URL
        viii. Let the client have the final say into whether or not the load should proceed.
        ix. Warn if the resource URL's hostname contains IDNA deviation characters.
    b. DispatchDidBlockRequest() using probe::DidBlockRequest()
5. ResourceFetcher::ResourceNeedsLoad()
    a. [not relevant] Check whether is archived
    b. [not relevant] Call ResourceAlreadyLoadStarted()
        i. Check RevalidationPolicy::kUse, resource->StillNeedsLoad()
6. ResourceFetcher::StartLoad()

a. Forbids JavaScript/revalidation until start() to prevent unintended state transitions.
b. Check ShouldBlockLoadingSubResource() && IsMainThread()
c. Check total **inflight_keepalive_bytes**

Renderer -> Browser Communication

To support the “DeferredRequestInit.activateAfter” feature, the renderer needs a mechanism to tell the browser to start a deferred request immediately.
Similarly, to support an abort signal, the renderer needs a way to tell the browser to drop a deferred request. Otherwise, the browser implementation for a keepalive request will start it by default after finding itself being disconnected from the source renderer.

Option 1: Via a new associated Renderer->Browser interface [selected]

This proposal sets up a new renderer->browser communication via a new **navigation-associated** FetchLaterLoaderFactory interface, resembling URLLoaderFactory.

Without associating the factory with navigation, the unload IPC of a document may reach the browser process earlier than the IPC to create a loader if a document is quickly terminated, resulting in missing fetchLater requests. See also discussions in this thread.

New Mojo Interface

```
C/C++
module blink.mojom;
interface FetchLaterLoaderFactory {
  CreateLoader(
    pending_receiver<FetchLaterLoader> loader,
    int32 request_id,
    uint32 options,
    network.mojom.URLRequest request,
    network.mojom.MutableNetworkTrafficAnnotationTag traffic_annotation);
};
interface FetchLaterLoader {
  SendNow();
  Cancel();
};
```

There will need to be a holder of the remote of FetchLaterLoaderFactory and remotes of FetchLaterLoader. It appears that the creation of such a receiver of FetchLaterLoaderFactory requires a RenderFrameHostImpl, which is not always accessible for child frame or new windows. The implementation ends up following the existing pattern of other subresource loader

factories, that the renderer side of the remotes of FetchLaterLoaderFactory are held in ChildURLLoaderFactoryBundle, instead of using GetRemoteNavigationAssociatedInterfaces() to initialize them. See this CL description for full details.

```
C/C++
class FetchLaterManager : public Supplement<ExecutionContext> {
  static void AttachTo(ExecutionContext&, AssociatedInterfaceProvider*);
  static FetchLaterLoaderManager& From(ExecutionContext& ec);

  HeapMojoAssociatedRemote<mojom::blink::FetchLaterLoaderFactory> factory_;
  HeapHashSet<Member<Loader>> loaders_;
};
```

Calling from Renderer

While it's tempting to directly implement them within FetchManager, the fact that there are complicated checks and potential modifications to a request down from FetchManager (See From FetchManager to ResourceLoader) prevents us from doing so.

Instead, FetchLaterLoaderManager should be called around the same point where a non-FetchLater fetch request would start a loader.

```
C/C++
bool ResourceFetcher::StartLoad() {
  ...
  if (request.is_fetch_later_api) {
    FetchLaterLoaderManager(ec)::CreateLoader(std::move(request));
    return;
  }
  loader = MakeGarbageCollected<ResourceLoader>(...);
}
```

After a long discussion, it seems that adding non-resource loading related logic into ResourceFetcher is also not a good idea. The latest approach is to call relevant logic from FetchLaterManager.

Where to Initialize the Navigation-Associated FetchLaterLoaderFactory

DONE:

- Check about:blank page
- Check popup window
- Check new child frame
- Check new child frame of about:blank
- Check new child frame of parent about:blank

Implementing in Browser

The existing KeepAliveURLLoaderSerivce should be updated to manage bindings to the receiver set of FetchLaterLoaderFactory, which creates KeepAliveURLLoader as FetchLaterLoader on request by renderer.

```
C/C++
class KeepAliveURLLoader : public blink::mojom::FetchLaterLoader {
 public:
  void Abort() override { DeletSelf() }
};
```

Option 2: Via a new independent Renderer->Browser interface

Prototype CL: https://crrev.com/c/4800410 (This proposal may still have IPC sequencing issues.)

This proposal sets up a new renderer->browser communication via a new FetchLaterClient interface, in addition to the existing mojom::URLLoader.

```
C/C++
interface FetchLaterClient {
 // Requests the client to send out the deferred FetchLater request immediately.
 // `fetch_later_request_id` is the ID of the deferred URLRequest the receiver
 // should act on.
 SendNow(mojo_base.mojom.UnguessableToken fetch_later_request_id);
};

module network.mojom;
struct URLRequest {
  mojo_base.mojom.UnguessableToken? fetch_later_request_id;
}
```

A FetchManager::DeferredLoader should set up the mojo pipe in its ctor, which is roughly after a fetchLater() API is called. When the FetchManager is deactivated, every DeferredLoader in it should use SendNow() with the fetch_later_request_id to tell the browser to start the deferred request.

```
C/C++
class FetchManager::DeferredLoader {
 DeferredLoader() {
   GetExecutionContext()->GetBrowserInterfaceBroker().GetInterface(
```

```
        remote_.BindNewPipeAndPassReceiver(...));
 }
 void Schedule() { PostDelayedTask(... [](){remote_.SendNow(fetch_later_request_id_);}) }

 const base::UnguessableToken fetch_later_request_id_;
 HeapMojoRemote<mojom::blink::FetchLaterClient> remote_;
};
```

One possible issue can arise from **wrong mojo message ordering**: FetchLaterClient::SendNow() should only be called after URLLoaderFactory::CreateLoaderAndStart() has reached the browser process, after when the browser will only know which fetch_later_request_id the remote is asking for. If called before, nothing can happen. Although this is unlikely to happen, as Schedule() will generally be called when FetchManager enters BackFowardCache. The only way to enforce the ordering seems to be sending a AssociatedPendingReceiver version of FetchLaterClient via Option 2.

Option 3a: Via a new field in network::mojom::URLRequest

Attempted prototype CL: https://crrev.com/c/4793104

A network::mojom::URLRequest gets passed from renderer -> browser -> network service by URLLoaderFactory::CreateLoaderAndStart() method.

```
C/C++
interface URLLoaderFactory {
  CreateLoaderAndStart(pending_receiver<URLLoader> loader,
                       int32 request_id,
                       uint32 options,
                       URLRequest request,
                       pending_remote<URLLoaderClient> client,
                       MutableNetworkTrafficAnnotationTag traffic_annotation);
}
```

This option proposes to add a new FetchLaterParams into the URLRequest struct, which carries a pending_receiver that should be used by the browser.

```
C/C++
module network.mojom;
```

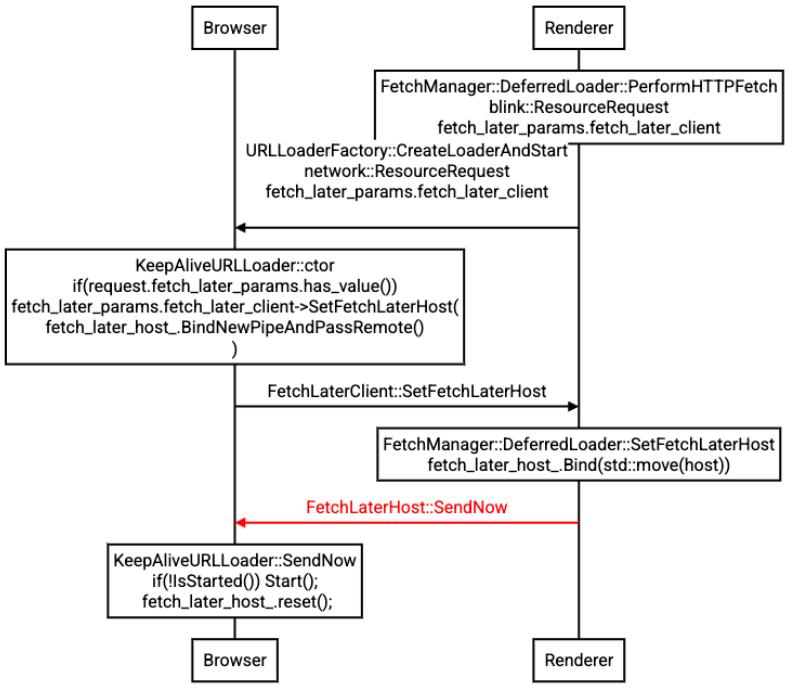
```
struct URLRequest {
  FetchLaterParams? fetch_later_params;
}

struct FetchLaterParams {
  // A client for communication from the renderer to the browser process.
  pending_receiver<FetchLaterClient> fetch_later_client;
};
```

However, there are several issues:

1. network::mojom::URLRequest is typemapped to network::ResourceRequest and typemapped from blink::ResourceRequest, and both of them are implicitly copyable. However, a mojo::PendingReceiver cannot be copied.
2. Even if we manage to support the “copying” behavior of ResourceRequest by allowing new instance to take over the ownership of the pending_receiver of fetch_later_client, the field will still be lost before it reaches the handler (KeepAliveURLLoader) in content. We are not sure how many clones will happen in between Renderer/Browser communication of a CreateLoaderAndStart() call.

Option 3b: Setting up bi-directional & one-time pipes via a new field in network::mojom::URLRequest

Attempted CL & discussions: link



To solve the issues in Option 3a, we propose to let FetchLaterParams hold a pending_remote of fetch_later_client, similar to other existing usages in URLRequest. The fetch_later_client is now copyable.

```
C/C++
module network.mojom;

struct URLRequest {
  FetchLaterParams? fetch_later_params;
}

struct FetchLaterParams {
  // A one-time Browser->Renderer communication to set up FetchLaterHost.
  pending_remote<FetchLaterClient> fetch_later_client;
};

interface FetchLaterClient {
  // The remote in the browser uses this method to set up a Renderer->Browser
  // communication.
  //
  // Calling this will close the message pipe for the interface as well, so no
  // further calls can be made.
  SetFetchLaterHost(pending_remote<FetchLaterHost> fetch_later_host);

  // To support creating a copy of the remote of this client.
  Clone(pending_receiver<FetchLaterClient> receiver);
};
```

```
// Renderer -> Browser communication
interface FetchLaterHost {
  // Asks to send out the deferred FetchLater request immediately.
  //
  // Calling this will close the message pipe for the interface as well, so no
  // further calls can be made.
  SendNow();
};
```

After receiving the pending_remote, the browser uses it to set up another Render->Browser connection to allow a SendNow() call to happen later. The pending_remote will then be dropped.

```
C/C++
class KeepAliveURLLoader : public network::mojom::FetchLaterHost {
  KeepAliveURLLoader(...) {
    if (resource_request.fetch_later_params.has_value()) {
      mojo::Remote<network::mojom::FetchLaterClient> fetch_later_client{
         resource_request.fetch_later_params->CloneFetchLaterClient()};
      fetch_later_client->SetFetchLaterHost(fetch_later_host_.BindNewPipeAndPassRemote());
      // `fetch_later_client` from the request is dropped immediately after
      // setting up the Renderer->Browser connection.
  }

  void SendNow() override { Start(); fetch_later_host_.reset(); }

  mojo::Receiver<network::mojom::FetchLaterHost> fetch_later_host_{this};
};
```

Option 4: Via existing URLLoader/URLLoaderClient/URLLoaderFactory IPC [not feasible]

These three existing network::mojo interfaces are all widely implemented, and get used across renderer/browser/network. Adding a new method into any of them specifically for Renderer->Browser communication looks infeasible.

In-Browser

Once a deferred request is started by blink::ResourceFetcher, the rest of the flow will be similar to the one described in Keep the Request Pipe Alive in the Browser Process. The main difference is that the request should NOT be started immediately when KeepAliveURLLoaderService::KeepAliveURLLoaderFactory::CreateLoaderAndStart() is called.

To support deferred loading behavior, KeepAliveURLLoaderService is updated to use the `resource_request.fetch_later_request_id` field to tell whether a request should be handled by a deferred KeepAliveURLLoader. If set, KeepAliveURLLoader is created with is_pending = true, and it will only start the request when get notified by KeepAliveURLLoaderService when the corresponding mojom::URLLoader is disconnected from a renderer.

```
C/C++
class KeepAliveURLLoader {
 public:
  KeepAliveURLLoader() : is_started_(false) { ... }
  bool IsStarted() const;
  void Start() { CHECK(!is_started_); is_started_ = true; ... }
};
class KeepAliveURLLoaderService {
  void KeepAliveURLLoaderFactory::CreateLoaderAndStart(..., resource_request, ...) {
    auto loader = std::make_unique<KeepAliveURLLoader>(...);
    if (resource_request.fetch_later_request_id.has_value()) {
      loader->Start();
    }
  }
  void OnLoaderDisconnected() {
    if (loader_receivers_.current_context()->IsStarted()) {
      loader_receivers_.current_context()->Start();
    }
  }
};
```

Privacy Considerations

This design has no impact on the existing fetch API. However, the following privacy requirements have been discussed and are important to follow:

- Deferred requests must be sent over HTTPS(?).
- Deferred requests can only be sent after the page becomes inactive, i.e. bfcached, if BackgroundSync permission is enabled. (see review feedback)

Security Considerations

Request Limit

See Permission Policy: deferred-fetch.

Mojo Interface

This design introduces two new navigation-associated mojo interfaces, FetchLaterLoaderFactory and FetchLaterLoader, which resembles the existing URLLoaderFactory and URLLoader but only for FetchLater request specific use.

The initial setup follows other subresource URLLoaderFactory patterns that a remote of FetchLaterLoaderFactory is passed from Browser to Renderer via NavigationClient::CommitNavigation(), see https://crrev.com/c/4936910. Subsequent data flows from the lower-trust Renderer to the higher-trust Browser whenever a JS fetchLater() API is called, which triggers FetchLaterLoaderFactory::CreateLoader(), and binds a FetchLaterLoader.

The browser limits the usage of FetchLaterLoader::SendNow() and FetchLaterLoader::Cancel() to single time by disconnecting itself after the first call to prevent any potential issues.

Please also refer to the Security Considerations section of the fetch keepalive migration about how the browser process handles a keepalive request:

1. When a call to fetchLater(**url**) is made, before the request is queued by browser process, the following checks are performed against **url** (in both renderer & browser):
    a. Check Safe URL
    b. Check Content-Security-Policy
    c. Check Mixed Content
    d. Check Safe Browsing
2. The request sending may be kicked off after some user-specified timeout, or by default at page unload. If subsequent request handling involves in redirected URL targets:
    a. Mixed Content will not be performed. This is not regression, it's the same for fetch(url) requests, blocked by https://crbug.com/1500989.
    b. Other checks from (1) will be performed in the browser process.