Blob URLs and Loading in a mojo world

Marijn Kruisselbrink
mek@chromium.org

Introduction

While trying to switch over how blob URLs are created/revoked to mojo, I ran into some test failures which made me realize the way loading of blob URLs is currently done is problematic. This document is my attempt of explaining the problem and trying to figure out how to best solve this problem.

Current situation

```
interface BlobURLHandle {};
interface BlobRegistry {
  // [other stuff]
  [Sync] RegisterURL(Blob blob, Url url) => (BlobURLHandle handle);
};
```

And loading of Blob URLs goes through the regular loading code path (whichever that is?) resulting in ultimately the blob URL being resolved in either BlobURLLoaderFactory::CreateLoaderAndStart or ResourceDispatcherHostImpl::ContinuePendingBeginRequest (or various other ResourceDispatchHostImpl methods).

For downloads there is some extra logic going on in the fact that content::DownloadURLParameters has its own blob handle field, which can be used instead of the blob the URL maps to (to keep the blob alive long enough for the download to start).

In any case, currently blob URLs are always resolved in the browser process.

The current situation isn't actually using the above described mojo API yet for script created (and revoked) blob URLs. Instead regular old-style (sync for register) IPCs are used. But that's what I'd like to change,

The problem

Several potential problematic cases around blob URLs (note that not all of these are actually problematic, they are just listed to try to illustrate how blob URLs might be more tricky than somebody would assume).

1. Script creates blob URL, immediately fetches it.
2. Script creates blob URL, passes URL to a different renderer via postMessage, and other renderer fetches blob URL.
3. Script fetches a blob URL, and immediately afterwards revokes the URL (before fetch has completed).
4. Script navigates iframe (or other top-level frame) to a blob URL, and immediately revoked blob URL.
5. Script navigates the frame that created a blob URL to that blob URL (the navigation away will revoke the blob URL).
6. Script initiates download of blob URL, and immediately revokes blob URL afterwards (tested in src/content/test/data/download/download-attribute-blob.html).
7. Script creates XMLHttpRequest, opens blob URL. Then revokes blob URL before "sending" the request. According to the spec URL parsing should resolve blob URLs, and thus this should work, currently it doesn't (https://crbug.com/695031).
8. Similar case to 7), but instead create a Request, and revoke the blob URL before fetching the request (https://github.com/w3c/web-platform-tests/pull/7848).
9. Script creates blob URL in one renderer, passes blob URL to different renderer via postMessage, second renderer revokes blob URL. As long as both windows are same-origin this should work and revoke the blob URL (the spec is a bit unclear in this area, although there is an example sort of describing this case).
10. Script creates blob URL in one frame, fetches blob URL from different frame in same renderer and immediately revokes blob URL in first frame (all frames same origin).

Case 1) and 2) are not currently a problem, because blob URL registration is sync in both the IPC and mojo cases.

Every other case is hard (if not impossible) to solve with the current mojo API, where the lifetime of each blob URL is represented by its own message pipe/BlobURLHandle instance, and as such revocation of blob URLs is entirely unordered relative to any other IPC/mojo call.

Case 7 and 8 are of course even more tricky to get right (but probably also less important, since they've always been broken in chrome), since it's no longer just a pure mojo/ipc race condition there, instead script can introduce arbitrarily long delays between when a blob URL is revoked and a fetch to that URL is actually started/should still succeed.

Case 9 is a minor problem with the current mojo API in that currently blob URL revocation is only possible by closing the BlobURLHandle connection, which of course doesn't work if some window/process that didn't create the URL tries to revoke the blob URL.

The solution?

Or at least part of it.

As a first step to resolving some of these ordering issues, I propose changing the mojo API dealing with blob URLs to something like:

```
interface BlobURLRegistry {
  [Sync] Register(Blob blob, Url url) => ();
  Revoke(Url url);
  Resolve(Url url) => (Blob blob);
  ResolveAsURLLoaderFactory(Url url, URLLoaderFactory& factory);
  ResolveForNavigation(Url, url, BlobURLToken& token);
};

interface BlobURLToken {
  Clone(BlobURLToken& token);
  GetToken() => (UnguessableToken token);
};
```

A couple of comments on this:

1. A separate interface from BlobRegistry. BlobRegistry itself isn't (and probably shouldn't) be tightly coupled to a document/worker, but BlobURLRegistry should most definitely be tightly coupled in that way (creating/revoking URLs should do origin checks, lifetime of URLs is tied to lifetime of frame/worker, etc).
2. Closing the BlobURLRegistry pipe will revoke all URLs that were created through it. That's consistent with the spec/current behavior (although I really should make the spec more explicit in this area).

3. Get rid of BlobURLHandle. Since closing the BlobURLRegistry pipe will revoke all the URLs registered through it (and the pipe is tied to the lifetime of the worker/document) there is no risk of leaking blob URLs. Furthermore since we need a separate Revoke method anyway to be able to revoke blob URLs created by different (same origin) globals, getting rid of BlobURLHandle simplifies the API as well as the blink implementation (no need to keep around BlobURLHandles for all registered blob urls).
4. A separate Revoke method. As mentioned above, processes/documents that didn't create a URL still need to be able to revoke the URL, so we need something other than BlobURLHandle.
5. A new Resolve method. This method should be used by loading code to resolve a blob URL (more and that later). Since resolving has to be sequenced with at least revoking URLs (and in some cases we might even need to resolve the blob URL well before the actual fetch request is send to the browser process) we can't rely on the browser process resolving blob URLs (unless we don't care about case 7 and 8 and are willing to make BlobURLRegistry associated with whatever interface is used for the actual fetch requests).
6. ResolveAsURLLoaderFactory: used for subresources, returns a URLLoaderFactory capable of only loading the specified blob URL (or failing if the URL wasn't mapped to anything).
7. ResolveForNavigation: Different from the above, because the browser process needs to be able to make sure what blob a URL mapped to when it was resolved (and can't purely trust the browser for that), more on that later.

So what about that loading code?

To be honest, I have no idea how the current code work. But I broadly see two options here:

1. Somehow let blink pass in the Blob it resolved a URL to as part of the request it sends to the browser process. This could be just the blob directly, or if that's too specific it could be a URLLoader derived from the blob, or something in between (a data pipe + whatever other API would be needed to actually load the blob?)
2. Completely bypass the browser process for loading blob URLs. Read the blob directly in the renderer (either via its existing Read/ReadAll methods, or via a new method that returns a URLLoader, or something in between).

I expect either of those should work fine for sub-resource fetches, I'm less sure about navigations and downloads (where as I understand it the browser process does need to be involved at least at some point). So maybe we need different solutions for the different types of fetches? Also not sure if there would be differences that matter between network service and non-network service code paths?

Sub resource requests

These are relatively straight-forward. All the code dealing with sub-resource request loading, including deciding what URLLoaderFactory to use already lives in the renderer process, so it is just a matter of changing how it gets the URLLoaderFactory. To do this, https://chromium-review.googlesource.com/c/chromium/src/+/893606 does three things:

1. Add an optional URLLoaderFactoryPtr to ResourceLoaderOptions. If this factory is specified it will be used to load the resource, instead of going through the normal resource loading path.
2. Add code to (Worker/Frame)FetchContext::CreateURLLoader to resolve blob URLs if they haven't been resolved yet (by calling the ResolveAsURLLoaderFactory method on the BlobURLStore for the current context).
3. Add code to fetch and XHR to resolve blob URLs as soon as the request is created, and store the resulting URLLoaderFactoryPtr. Later this factory is passed into the ResourceLoaderOptions.

Navigations

Navigations are much harder. All navigations go through the browser process at some point, but there are at least three entry point IPCs through which blink can initiate a navigation. Furthermore the browser side currently has at least three different implementations for loading navigations:

1. Legacy code path (but currently one in use) that somehow passes the result of fetching the document back to the renderer wrapped in a blob: URL (but not a blob: URL as described in this document, just a legacy usage by the "streams" concept that happens to reuse the same scheme. In the network service future these blob: URLs won't exist anymore).
2. Slightly better code path, that instead passes the fetch result back as a URLLoader/URLLoaderClient& pair.
3. Network servicified code path.

For the purpose of blob URL mojofication I think it is good enough to only focus on the third case. The reason for that is that that is going to be the one and only implementation eventually anyway, but also that's the only case where things are already broken even without blob URL mojofication.

If we want to ship mojo blob URLs sooner than network servicification (and we might, since after the changes described in this document it will be more spec compliant than the old code), it should be possible to also make case 2 work, which soon will be the default anyway.

So back to the three IPC entry points that I've identified:

1. The mojom FrameHost.BeginNavigation call. This seems to be used for most simple navigations initiated by blink. This case is addressed in https://chromium-review.googlesource.com/c/chromium/src/+/898009. In summary:
    a. Pass a URLLoaderFactory as extra parameter to BeginNavigation (could be as extra member in BeginNavigationParams mojom struct, but would require custom CloneTraits as that struct needs to be copyable, so might be easier as separate parameter.
    b. RenderFrameHostImpl passes this factory on to Navigator::OnBeginNavigation (as a separate parameter, but now wrapped in a SharedURLLoaderFactory).
    c. NavigatorImpl stores the factory in NavigationRequest(Info).
    d. Which is then ultimately used by NavigationURLLoaderNetworkService to create the ThrottlingURLLoader similar to how it can use a custom factory for webui loading.
2. RenderFrameProxy::Navigate, sending FrameHostMsg_OpenURL to RenderFrameProxyHost. This is used for navigating out-of-process windows (i.e. out-of-process iframes, but also other windows that somehow ended up in a different process because they were navigated cross origin). This case (like the next one) is addressed in https://chromium-review.googlesource.com/c/chromium/src/+/899508. In summary:
    a. RFProxyHost passes received factory on to Navigator::RequestTransferURL.
    b. RequestTransferURL adds factory to FrameNavigationEntry for this navigation.
    c. NavigationRequest::CreateBrowserInitiated pulls the factory back out of FrameNavigationEntry and sets it in the returned NavigationRequest.
    d. Same as d. in the first case.
3. RenderFrameImpl::OpenURL, sending FrameHostMsg_OpenURL to RenderFrameHostImpl. This one is by far the most complicated. It's used for various top-level navigations, and addressed in the same CL as the previous case:
    a. RFHostImpl::OnOpenURL passes factory on to Navigator::RequestOpenURL.
    b. RequestOpenURL sets passed in factory on OpenURLParams it creates.
    c. OpenURLParams end up in chrome::Browser::OpenURLFromTab, which copies factory into NavigateParams (or content::Shell::OpenURLFromTab in case of content_shell, which skips the next step and call LoadURLWithParam directly).
    d. BrowserNavigator LoadURLInContents transforms NavigateParams from previous step into NavigationController::LoadURLParams, preserving url loader factory.
    e. NavigationController::LoadURLWithParams sets url loader factory in FrameNavigationEntry it creates.
    f. Now it's the same as the previous case from step c.

This ignored how the methods that call out to the browser process actually got the url loader factory in the first place. A few things are needed for that:

1. FrameLoadRequest resolves blob URLs when the request is created, and stores the result in itself.

2. NavigationScheduler (in blink) also resolves blob URLs as soon as a navigation is scheduled, storing the result in ScheduledURLNavigation, and overwriting the later resolution of the same URL FrameLoadRequest would do itself when it is finally created.
3. Next FrameLoader::ShouldContinueForNavigationPolicy (called by CheckLoadCanStart and ShouldContinueForRedirectNavigationPolicy) extract the blob URL token from the FrameLoadRequest, and passes it on to LocalFrameClient::DecidePolicyForNavigation.
4. That method then stores the BlobURLToken in WebFrameClient::NavigationPolicyInfo (as a raw mojo message pipe, since this is at the boundary between blink and content, and thus can't use either typed interface ptr).
5. Finally that is used by RenderFrameImpl (for IPC cases 1 and 3 above) to get the token back out to send on to the browser (it's similar/simpler for the OOPIF case, where RemoteFrame directly extracts the BlobURLToken from the FrameRequest and passes that on to RenderProxyImpl).
6. When the browser receives the BlobURLToken it converts it into a URLLoaderFactory.

One more thing about navigations is that it is potentially a security hole to let a renderer tell the browser process to "navigate to this URL, but load the contents from this URLLoaderFactory" so see the security section below for why this is safe (and why the BlobURLToken exists).

Downloads

~~Downloads are easy, since recently downloads were changed to go through the navigation code path. So fixing navigations will also fix downloads.~~
Downloads are yet another code path/IPC entry point that will have to be separately supported (done in https://chromium-review.googlesource.com/c/chromium/src/+/1026808):

1. LocalFrameClientImpl::DownloadURL resolves blob URLs to BlobURLTokens.
2. Token is passed over IPC (and a couple of other method calls), ultimately arriving in RenderFrameMessageFilter's DownloadUrlOnUIThread method.
3. There token is converted into a blob SharedURLLoaderFactory.
4. Factory is passed on to DownloadManagerImpl.
5. DownloadManagerImpl::BeginDownloadInternal wraps URLLoaderFactory in a DownloadURLLoaderFactoryGetter (similar to how the existing blob url code path in there already wraps a blob URLLoaderFactory in a BlobDownloadURLLoaderFactoryGetter).

This will also then let us remove the existing blob specific code in DownloadManagerImpl, at least for the network service code path, although before doing that (in a separate CL) I'd like to make sure all entry points that currently pass in a BlobDataHandle are actually passing in a BlobURLToken as well.

More about ordering

The described solution doesn't actually solve the 10th described potential problem. Using a separate frame-bound BlobURLRegistry for each document means that requests from different frames are not sequenced, even if those frames share the same event loop (this more generally is a problem with per-frame mojo interfaces and any kind of API that involves browser-process-side state). So because of that, rather than getting BlobURLRegistry somehow associated with a frame directly, I'd likely be stuck making BlobURLRegistry be associated with the (already existing) per-thread BlobRegistry pipes. That of course means we lose some of the nice security benefits of per-frame mojo pipes, but I'm not sure I see any other way (implementing our own sequencing on top of multiple frame-bound pipes?)

Security concerns

Cross origin navigations

The main security concern here is that a navigation might let a renderer execute code on a different origin (i.e. by somehow triggering a navigation where it can specify both a cross origin URL and dictate (part of) the content that gets loaded as that URL).
Fortunately this shouldn't be too much of a concern: navigations (like all fetches) to blob URLs are only allowed if they are same origin navigations. So a website is only supposed to be able to navigate to a blob URL if the origin of the blob URL matches its own origin. And since a website already can create blob URLs for arbitrary data in its own origin it is not a problem for a renderer to provide the content a blob URL resolves to in addition to the blob URL.

Now currently in chrome the described same origin checks are not actually implemented in the browser process, instead relying on blink itself to do these checks. Unfortunately it is also not trivial in the current design for the browser to know the origin of the initiator of a navigation, so actually implementing these same-origin checks in the browser is hard if not impossible (the problematic case is RenderFrameProxyHost).

To work around that I've introduced the concept of a BlobURLToken. Blink will resolve a blob URL into a BlobURLToken, and that BlobURLToken is then used by the browser process to look up the blob and blob URL as they existed at the time the token was created.

This token isn't just a base::UnguessableToken, but instead is a mojo interface that has a method to get a base::UnguessableToken, and that is for a few reasons:

- To resolve a blob URL into a base::UnguessableToken directly would introduce an async mojo call (or worse, a sync one) at some point in the navigation code path where

currently no such asynchronously behavior exists. By instead getting a BlobURLToken interface pipe the code in blink doesn't have to change much.
- The BlobURLToken has to keep alive some state in the browser process (i.e. the url to blob mapping, and the blob itself). If the token was just some bit of data we'd have to additionally do careful refcounting or something to make sure the state in the browser process is kept alive long enough, but not too long. By instead using a mojo interface we know to keep the state alive as long as the pipe is alive, eliminating the possibility for code bugs resulting in leaks and or premature releases.

The browser process will still need to verify that the BlobURLToken actually matches the url being fetches (otherwise the browser could still initiate a cross origin navigation to a blob URL, and give a valid token for a different same-origin blob URL), but that's pretty straight forward.

Finally we could (and probably should) add the same origin checks at the point were these BlobURLTokens are created, but even without that this should result in code that is as secure as the current chrome implementation (i.e. all the same checks are still in place, and on top of that we guarantee that the blob URL being navigated to always actually matches the contents that the blob URL is supposed to refer to).

Misbehaving URLLoaderFactory implementations

Secondary there are some concerns around the fact that URLLoaderFactory is typically a interface implemented by trusted code. This means that generally loading code in the renderer doesn't have to be too careful to deal with misbehaving URLLoaderFactories. If we would let a renderer provide the endpoint for a URLLoaderFactory that another process might end up using this gets a bit more complicated (even if in a correct situation the actual endpoint of the URLLoaderFactory is the blob system in the browser process, a renderer could just give out any mojo pipe it wants). For navigations (and even more so for downloads) it is the browser process after all that is interacting with the URLLoaderFactory/URLLoader for at least part of the fetching, so rather than have that code deal with potentially misbehaving factories/loaders it is easier to just make sure the URLLoaderFactory/URLLoader themselves are always running trusted code.
For example a misbehaving URLLoaderFactory could try to trigger redirects at unexpected moments, in an attempt to bypass the security checks described above. To deal with this concern rather than have the renderer that initiates a navigation provide a URLLoaderFactoryPtr directly, the renderer will merely provide a BlobPtr. It is then up to the browser process to turn this BlobPtr into a (trusted) URLLoaderFactoryPtr.