Multi-process Resource Loading

Contents

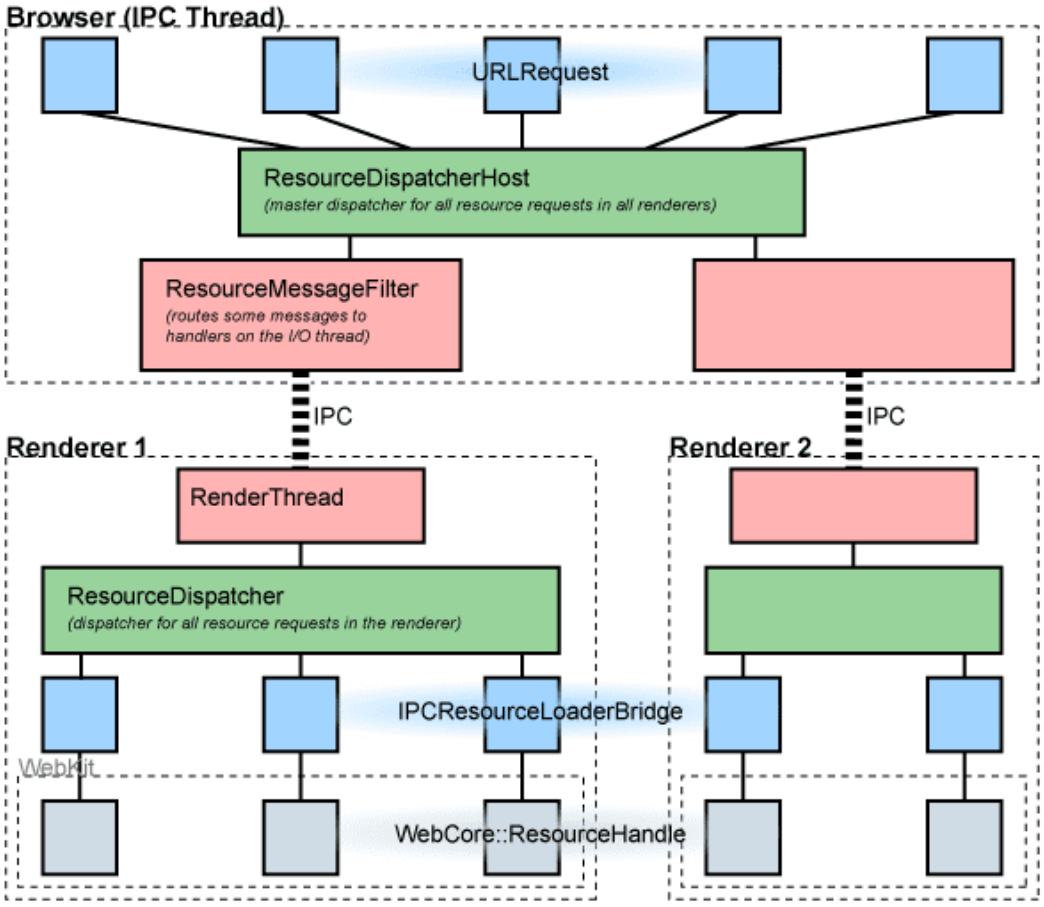
This design doc needs update. Some figures contains stale information.

Background

All network communication is handled by the main browser process. This is done not only so that the browser process can control each renderer's access to the network, but also so that we can maintain consistent session state across processes like cookies and cached data. It is also important because as a HTTP/1.1 user-agent, the browser as a whole should not open too many connections per host.

Overview

Our multi-process application can be viewed in three layers. At the lowest layer is the Blink engine which renders pages. Above that are the renderer process (simplistically, one-per-tab), each of which contains one Blink instance. Managing all the renderers is the browser process, which controls all network accesses.



Blink

Blink has a `ResourceLoader` object which is responsible for fetching data. Each loader has a `WebURLLoader` for performing the actual requests. The header file for this interface is inside the Blink repo.

`ResourceLoader` implements the interface `WebURLLoaderClient`. This is the callback interface used by the renderer to dispatch data and other events to Blink.

The test shell uses a different resource loader, so provides a different implementation, non-IPC version ~~of `ResourceLoaderBridge`, located in `webkit/tools/test_shell/simple_resource_loader_bridge`~~.

Renderer

The renderer's implementation of `WebURLLoader`, called `WebURLLoaderImpl`, is located in `content/child/`. It uses the global `ResourceDispatcher` singleton object (one for each renderer process) to create a unique request ID and forward the request to the browser via IPC. Responses from the browser will reference this request ID, which can then be converted back to the `RequestPeer` object (WebURLRequestImpl) by the resource dispatcher.

Browser

The `RenderProcessHost` objects inside the browser receive the IPC requests from each renderer. It forwards these requests to the global `ResourceDispatcherHost`, using a pointer to the render process host (specifically, an implementation of `ResourceDispatcherHost::Receiver`) and the request ID generated by the renderer to uniquely identify the request.

Each request is then converted into a `URLRequest` object, which in turn forwards it to its internal `URLRequestJob` that implements the specific protocol desired. When the `URLRequest` generates notifications, its `ResourceDispatcherHost::Receiver` and request ID are used to send the notification to the correct `RenderProcessHost` for sending back to the renderer. Since the ID generated by the renderer is preserved, it is able to correlate all responses with a specific request first generated by Blink.

Cookies

All cookies are handled by our `CookieMonster` object in `/net/base`. We do not share cookies with other browsers' network stacks (e.g. WinINET or Necko). The cookie monster lives in the browser process which handles all network requests because cookies need to be the same across all tabs.

Pages can request cookies for a document via `document.cookie`. When this occurs, we send a synchronous message from the renderer to the browser requesting the cookie. While the browser is processing the cookie, the thread that Blink works on is suspended. When the renderer's I/O thread receives the response from the browser, it un-suspends the thread and passes the result back to the JavaScript engine.