FireBreath 2.0: Browser Plugins in a post-NPAPI world

Outdated page

We're leaving this here for the discussion it contains, but please help us update the new FireBreath 2.0 section of this website.

The Beginning of the End of NPAPI (last updated in 2014)

Many of you have no-doubt seen the news that Chrome will be phasing out support for NPAPI in the next year.

The FireBreath team thinks that that is a Really Bad Idea and welcomes any of the browser vendors to contact them directly via phone or email to discuss the reasons.

Unfortunately, we at FireBreath can see the writing on the wall. NPAPI is going away. It shouldn't – there isn't a good replacement. It's frustrating, infuriating, and implies a frightening disregard for the innovation that plugins allow on the web (to be confirmed after a review of PPAPI). Recent decisions by the Mozilla and Chrome teams, however, point a very clear finger that the browser development teams are declaring a relatively quiet war of attrition on plugins and they will be gone soon. In addition, Windows 8 and later contains a modern mode (formerly called Metro) of IE that disables almost all ActiveX controls.

The conclusion is inevitable: **We will need to change our approach.**

EDIT: To clarify, FireFox has made it more difficult for NPAPI plugins, but when used properly Click to Play isn't that much of an impediment. It definitely shows that they want NPAPI plugins gone, but they haven't (yet anyway) made any announcements that indicate any sudden or imminent demise of NPAPI support. Chrome is the only browser that has, and there is not much so far to indicate exactly when that will happen.

TL;DR:

FireBreath 2.0 will support current protocols as well as Native Messaging, but hypothetically could support any of the following

- NPAPI
- ActiveX
- Native Messaging (using FireWyrm)
- NaCL/pNaCL (if we can make it build, and will use FireWyrm)
- Web Sockets (using FireWyrm)
- HTTP Ajax PULL (using FireWyrm)
- Telnet (using FireWyrm)
- IP over Avian Carrier (https://tools.ietf.org/html/rfc2549) (using FireWyrm)

Major changes will include:

- Everything will be asynchronous!
- Calling a function on the plugin will return a Promise
- Getting a property from the plugin will return a Promise
- Calling into the DOM from the plugin will also be asynchronous
    - We have FB::Deferred<T> and FB::Promise<T> types which are based (very) loosely on the Promises/A+ spec
- Drawing models will need to be developed, but will essentially be browser-based
    - WebGL, Canvas, possibly NaCL/pNaCL and/or emscripten, etc
    - Drawing models will also be asynchronous
- FireWyrmJS will provide a simple abstraction for using a FireWyrm plugin from Javascript
- FireWyrmJS will require that a WyrmHole (transport mechanism) be provided
    - The most common WyrmHoles we anticipate seeing would be ones for Native Messaging and js-ctypes
    - Many of these may require a browser Extension; we could probably create a FireBreath extension which would be usable by all

Other improvements:

- Better cmake support (run cmake on your project, not on FireBreath)

The basic plan:

FireBreath 2.0 attempt to address several long-standing complaints about the framework as well as adding a whole new type of plugin support. With the collaboration of the employees at GradeCam and other members of the community we've developed of the FireWyrm protocol. This name was carefully chosen using the same naming methodology as FireBreath itself, which is top secret. The point, though, is that the FireWyrm interface will allow a whole new type of plugin transports.

Adding support for the FireWyrm protocol involved a fair bit of complexity, but at its core it is actually a distinct thing from FireBreath -- it could be used in other projects, if someone wanted to. In essence, FireWyrm is an advanced (but suprisingly simple) mechanism for performing RPC calls over an essentially text-based asynchronous transport protocol using JSON.

What does this mean?

You have three pieces to this system:

1. FireWyrm client (FireWyrmJS will be our initial javascript implementation)
2. FireWyrm provider (FireBreath's third plugin type, a sibling to ActiveX and NPAPI)
3. A tranport mechanism for communicating between them, known as a WyrmHole

FireWyrmJS connects over a WyrmHole to the FireBreath plugin (through the provider) and gets information about the root JSAPI object; it then constructs a javascript object wrapper which can create a payload, send it over the WyrmHole, and return a Promise for all plugin calls. (If you aren't familiar with the Promise pattern, and in particular the Promises/A+ spec, you may want to read up on it a bit).

What this means is that any transport mechanism that you can find that will connect the two points of code and send strings back and forth can be used to connect to this plugin. On the C++ side a new set of FireWyrm provider entrypoints have been created which make it relatively easy to create the c++ side of the WyrmHole. The initial implementation of a WyrmHole (nearly complete) uses Native Client in order to instantiate the plugin. We'll essentially create a thin executable which will load your FireBreath 2.0 plugin using the FireWyrm entrypoints and will communicate with the browser, allowing you to use your same plugin.

Limitations:

I'm sure any of you that is actually thinking about what I'm proposing has already noticed that there are some deviations and challenges with this idea. I'm not sure I've identified them all, but here are the things I can think of off-hand that will change or need to be addressed:

- Initially, there will be no drawing model.
  - I expect this to change soon, particularly if others in the community help, but initially there will be no drawing model available to plugins using FireWyrm.
  - The first drawing model I expect to see will be implemented by sending a pixel buffer to the browser where it will be drawn on a Canvas
  - Eventually I expect that we will develop other drawing methods, such as:
    - WebGL proxied through the WyrmHole
    - Canvas proxied through the WyrmHole
    - Some other RPC-driven mechanism (perhaps we can find an existing one) compiled with emscripten and proxied through the WyrmHole
    - Some other HTML-based abstraction (controlled through the WyrmHole)
- All operations will be Asynchronous
  - We will be relying heavily on the Promise pattern, both in C++ and JavaScript
  - We will be requiring C++11 (Primarily for lambda support) in order to make this hurt less
- ***This will apply to NPAPI and ActiveX as well***! This will be a relatively HUGE change
- All property get calls will return a Promise
- *All* method calls will return a Promise, though ones that return void it will only tell you it completed
- Eventually we will attempt to find a way to allow calls to be batched in such a way that a series of operations can be sent at the same time and the code will continue when it's complete

--- Below is the previous version of this page containing brainstorms and info which was used to develop the above plans. It has not been updated in awhile ---

The alternatives, as presented by the browser manufacturers

The browser development teams seem to feel that dropping NPAPI support is acceptable for a number of reasons. One of the main reasons is that a very small percentage of people use plugins; unfortunately, for that small percentage, there are no other good options (PPAPI may be a good option - to be confirmed). Here are the options that they feel we should be able to use in their place:

(this list focusses primarily on Chrome since that's the most immediate, but the goal is to find a solution or conglomerate of solutions that can be combined to make a long term multi-browser solution)

Pepper Plugin API / PPAPI (without NaCL) (link)

We have confirmed that PPAPI will not work because while Chrome supports it they don't provide any way for third parties to install PPAPI plugins. The only method of doing so involves a command-line argument, which is unsuitable for most purposes.

However, PPAPI can be used in conjunction with NaCL - see below.

Google Native Client / NaCl (link)

- **What it is:**
  - NaCl uses a special C/C++ compiler and the Pepper API (PPAPI) to allow the creation of mostly-native code that can run "safely" in your browser.
- **Pros:**
  - Code should run pretty much at native speed
  - Sufficiently powerful graphics capabilities to run Flash
  - Can be packaged fully inside an extension and thus doesn't require a traditional installer
  - They are working towards a portable version (PNaCl) for mobile devices in the future.
- **Cons:**
  - Only works on Google Chrome
  - Does not allow direct hardware access. Although WebRTC now allows access to video and audio capture hardware, and is becoming implemented in most browsers

- 3-D graphical abilities exist (OpenGL ES2), but not full OpenGL capabilities
- Does not allow raw TCP/UDP
- Other browser vendors seem to have no interest in adding support
- No synchronous javascript API; all communication with the web page is through asynchronous "messages"

Google Native Messaging (link)

- **What it is:**
  - Native Messaging allows a Chrome Extension to exchange messages with native applications. From the docs: "Native applications that support this feature must register a *native messaging host* that knows how to communicate with the extension. Chrome starts the host in a separate process and communicates with it using standard input and standard output streams."
- **Pros:**
  - Allows running of native code, which presumably has full hardware access, native TCP/UDP access, etc
  - It should be possible to create an asynchronous bridge and abstraction to make this fairly painless from Javascript
  - Launches the application from the browser, so the user does not have to start anything themselves
- **Cons:**
  - Only works on Google Chrome, no analogue in Firefox, Safari, or Internet Explorer
  - Two separate installers; the application (native code) has to be installed, and the extension needs to be installed. The same installer could probably install the extension, but it then needs to be accepted by the user in the browser.
  - No easy way to get video back to the browser; could be a good solution to replace hidden plugins, but not anything with video.

Mozilla Extensions js-ctypes (link)

- **What it is:**
  - **js-ctypes** allows application and extension code to call back and forth to native code written in C. Basically it allows you to call into a DLL or similar from your javascript extension.
  - The probable method of using this would be to create an analogue for Google Native Messaging that works in Firefox
- **Pros:**
  - Allows calling native code from a Mozilla add-on without using NPAPI
  - Libraries may be included in the add-on bundle or located on the host system
  - Supports Javascript callbacks
- **Cons:**
  - Only works on Mozilla products (Firefox/Thunderbird)
  - Requires shims for executing C++ methods
  - Life-cycle of library objects initialized within the "shims" is per method execution (deconstructor for objects is called after each method execution)
  - No support for large chunks of binary data
  - No drawing support

Web Sockets and an external application

- What it is:
  - In theory, we could use websockets to connect to an application running on the local system and communicate with it, thus allowing that application to provide us with services that are otherwise unavailable in the browser, such as native TCP/UDP socket access, hardware access, etc.
- Pros:
  - Allows us to communicate with native code
  - Works on the latest versions of the major browsers (Chrome, Firefox, IE, Safari, Opera, and their mobile derivatives) Source
- Cons:
  - The application would need to be launched some how
  - No good solution for providing video
  - Possible security warnings with SSL sites since local server wouldn't be SSL
  - Only works on newer web browsers (TODO: what browsers support?)

--- This page is under construction ---

Features and capabilities needed that nothing above can address:

- Capturing and displaying video from an unsupported (by the browser) hardware device
- document scanning, used for document management both internal and customer-facing - requires access to TWAIN API (a 3rd party DLL).
- What else?