Alternative Method for Message Passing Between JavaScript and Flash

António Afonso
Opera Software ASA
antonio.afonso@gmail.com

ABSTRACT

This paper introduces a method to communicate with Flash using JavaScript when the `ExternalInterface` is not available in the hosting platform.

Keywords

JavaScript, Flash, ActionScript, ExternalInterface

1. INTRODUCTION

Today's web applications demand an increasing set of features to be made available to developers and users in order to close the gap between them and desktop applications.

JavaScript[1] is the de facto programming language for developing web applications. It is used alongside the DOM[2] to manipulate all aspects of a web page.

Flash was one of the first web technologies to provide users with access to rich multimedia content, before the first JavaScript frameworks for animations came to existence. It was therefore adopted by a big crowd of web designers and developers eager to provide a rich experience to their users. Flash has thus become one of the most bundled plug-ins seen in browsers, with an estimated ∼99% market penetration according to Adobe[2].

Web applications are currently developed using a mix of Flash and JavaScript to get the best of both worlds. However, browsers running in low memory devices such as the mobile phones are not able to run the standard Flash Player, and have to fall back to the less feature-rich FlashLite edition, which raises its own specific problems.

This paper explains some of these problems and proposes a possible solution to them.

2. MOTIVATION

Despite several advances in HTML5[3], like the audio and video elements[4], developers still struggle with multimedia and binary object manipulation in JavaScript. Specially because of the current scenario, where browser vendors are still debating which codecs to support and use.

Flash has been developed into a full-featured multimedia platform, giving developers everything they need to embed audio and video into their pages in a way that works in almost all desktop web browsers.

However, having part of a web application developed in Flash is not very easy to maintain because the UI will be split between the HTML/CSS and the Flash itself. Unfortunately these two elements don't integrate too well.

Having the entire page developed in Flash is also undesirable for a number of reasons that range from Flash not having good support in certain operating systems to Flash not being fully indexable by search engines such as Google.

2.1 Common Solutions

Not willing to give up, web developers started to create their own JavaScript APIs implemented in Flash. This is possible due to Flash's own ExternalInterface API[5], which makes ActionScript[6] functions available in JavaScript.

This way it is possible to develop web libraries and applications using HTML/CSS/JavaScript while still having Flash's power under the hood. One example of such a library is SoundManager 2[7], which brings MP3/AAC audio functionality to JavaScript.

The problem with current solutions is that sometimes they are not available under specific devices, specially mobile phones and video gaming consoles. These devices don't have the entire Flash framework available and are restricted to the much less featured FlashLite framework.

FlashLite 3.1 already supports the `flash.external.ExternalInterface`, however there are still some implementations that lack this feature, making it impossible to use libraries that rely on that functionality (such as the SoundManager 2 library).

This paper proposes a solution to make it possible running such libraries under platforms that are missing the `flash.external.ExternalInterface` class.

[1] JavaScript is a dialect of ECMAScript[4]
[2] Document Object Model – `http://www.w3.org/DOM/`
[3] `http://www.w3.org/TR/html5`
[4] `http://www.w3.org/TR/html5/audio.html`, `http://www.w3.org/TR/html5/video.html`
[5] `http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/external/ExternalInterface.html`
[6] ActionScript is the programming language used by Flash.
[7] `http://www.schillmania.com/projects/soundmanager2/`

3. CURRENT METHODS

The `flash.external.ExternalInterface` class provides both a way for ActionScript to invoke JavaScript functions, and a way for JavaScript to invoke ActionScript functions.

Another method for communicating with ActionScript from within JavaScript is the `SetVariable`[1] function part of the flash DOM Object, which enables the caller to change ActionScript's variables.

This variable change could then be detected in ActionScript through the `watch` function[2]. This function allows the code to be informed whenever the value of a variable has changed. This functionality alone could be used to invoke ActionScript's functions by passing the function name and its arguments as the new variable values, an example of which is shown in listings 1 and 2.

```
flashObject.SetVariable("invoke", "messageName
    =functionName&messageParams=params");
```

Listing 1: Invoking SetVariable in JavaScript

```
var functions; // available functions
watch("invoke", function(prop, oldVal, newVal)
{
    var name = parseMessageName(newVal);
    var args = parseMessageParams(newVal);

    functions[name].apply(this, args);
});
```

Listing 2: Detecting SetVariable in ActionScript

The `fscommand`[3] also allows ActionScript to invoke specific JavaScript functions, requiring the construction of proxy functions that follow the `fscommand` function naming scheme.

4. PROPOSED METHOD

Since the proposed method is meant to replace `flash.external.ExternalInterface`, there is a need for two workarounds to make libraries depending on this class to work:

1. Invoking JavaScript functions from ActionScript.
2. Invoking ActionScript functions from JavaScript.

In technical terms this means implementing `call(functionName:String, ... arguments):*` and `addCallback(functionName:String, closure:Function):void` from the `ExternalInterface` class.

[1] `http://www.adobe.com/support/flash/publishexport/scriptingwithflash/scriptingwithflash_03.html`
[2] `http://www.adobe.com/support/flash/action_scripts/actionscript_dictionary/actionscript_dictionary613.html`
[3] `http://www.adobe.com/support/flash/action_scripts/actionscript_dictionary/actionscript_dictionary372.html`

4.1 Invoking JavaScript functions from ActionScript

This solution relies on a very common way to interact with JavaScript from within ActionScript, which has been documented in several articles and forum posts on the web.

In ActionScript 2 it is possible to interact with the browser window by calling the `getURL(url)`[4] function, which provides the same functionality as clicking on a link pointing to the `url` argument.

In order to invoke a JavaScript function it is necessary to craft a url composed using the `javascript:` scheme, which makes the window execute JavaScript code in the context of the current page instead of loading a new one.

```
getURL("javascript:functionName()");
```

Listing 3: Invoking a JavaScript function

A possible implementation of `call(functionName:String, ... arguments):*` using this method is presented in Listing 4.

```
static function call(methodName:String) :
    Object
{
    var string:String = methodName+'(';
    var args:Array = [];

    for( var i = 1; i < arguments.length; i++
        ) {
        if( typeof(arguments[i]) == 'string' )
            {
            args.push('"' + arguments[i].split
                ('"').join('\\"') + '"');
        } else {
            args.push(arguments[i]);
        }
    }

    string += args.join(',') + ')';
    getURL('javascript:' + str);

    return null;
}
```

Listing 4: `ExternalInterface.call` Implementation

The limitation of this approach is the inability to receive the return value of the function call made in JavaScript. However, despite this limitation, a lot can still be achieved. For instance, the SoundManager 2 library doesn't make use of the return value of the `call` method.

4.2 Invoking ActionScript functions from JavaScript

4.2.1 `flashVars`

[4] `http://www.adobe.com/support/flash/action_scripts/actionscript_dictionary/actionscript_dictionary377.html`

The only way for JavaScript to directly pass information to the ActionScript in the Flash object is by setting the `flashVars` property of the creation[1] object. An example of passing variables `param1` and `param2` with values `value1` and `value2` respectively to the Flash object can be seen in Listing 5.

These variables will then be available under `_root.param1` and `_root.param2` in ActionScript when the Flash object is loaded.

```
<embed
    type="application/x-shockwave-flash"
    src="foo.swf"
    flashvars="param1=value1&param2=value2"
></embed>
```

Listing 5: Using `flashVars` to pass variables

Despite it being possible to change the `flashVars` attribute in JavaScript after Flash initialization, the corresponding variables in ActionScript will not be affected. This means that the variables are just copied once during Flash initialization and don't correspond to a real binding. Detecting changes in the attributes' values is also not possible so recopying is not an option.

However, this method will play a crucial part in the proposed solution alongside another technique that makes communication between different Flash objects possible.

4.2.2 `LocalConnection`

The `LocalConnection` class is provided by the Flex framework in order to allow Flash objects to invoke each other's methods[1]. It works similarly to remote method invocation, but with all participants residing in the same machine.

It configures one object as the receiver of invoke requests, and all other objects involved as the senders. The receiver object creates a named connection and the functions to invoke first (example in Listing 6) and all other Flash objects can then send invoke requests using that same named channel (example in Listing 7).

```
var lc:LocalConnection = new LocalConnection()
    ;
lc.func1 = function(param1) {
    trace("param1: " + param1);
}
lc.connect("name");
```

Listing 6: Setup of a `LocalConnection` receiver

```
var lc:LocalConnection = new LocalConnection()
    ;
lc.send("name", "func1", "value1");
```

Listing 7: Invoking an external function through `LocalConnection`

[1] `http://www.adobe.com/livedocs/flash/9.0/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00000668.html`

4.2.3 *The solution*

Using the two techniques explained in the previous sections – `flashVars` and `LocalConnection` – it is possible to pass messages from JavaScript to ActionScript using the following mechanism:

When the main Flash object is initialized a `LocalConnection` object is created and all functions to be made externally available are added to the object.

After this setup the instance is bound to a named connection, `"wii"` for instance.

When JavaScript wants to invoke an ActionScript function it creates a second Flash object – that will be called the *proxy object* – with the following `flashVars` parameter: `"messageName=<name>&messageArgs=<args>"`, where `messageName` is the function name and `messageArgs` is a JSON object representing an arguments array.

When the proxy object is initialized it reads its `flashVars` parameter and sends a message through the `"wii"` connection that was established by the main Flash object. This message will target the function name given by `messageName` and will carry the arguments parsed from the `messageArgs` variable.

Again, like its counterpart, it is not possible for JavaScript to receive the return value of the function invocation.

4.3 Implementation

The solution described is only useful for someone who is writing their own Flash objects from scratch and is able to implement such a solution, or for someone who is willing to fork his ActionScript source in order to ditch `ExternalInterface` for this solution.

This can be quite problematic on complex code sources or when the maintenance of two different versions is just not possible (resource or timewise), which is often the case.

To get around this problem a custom implementation of the `ExternalInterface` class was created, which makes the use of this mechanism a simple matter of recompiling the ActionScript source code using that version of the class.

To make this solution possible the following was needed:

- Implementation of the `call` and `addCallback` methods of `ExternalInterface`.
- The *proxy* Flash object to make the actual invocation.
- A JavaScript function to create the *proxy* Flash object in the HTML document with the correct parameters.

The implementation of the `call` method can already be found in Listing 4. The `addCallback` method just needs to add the function given as an argument to the `LocalConnection` object in order to make it available for external calls, and to define a function with the exact same name in JavaScript (this can be achieved with the already defined `call` method).

The JavaScript function that needs to be created by `addCallback` will basically call the function that creates the *proxy* Flash object.

A pseudo-code implementation can be found in listings 8, 9 and 10.

```
function callFlash(methodName, methodArgs) {
    var embed = document.createElement('embed'
        );

    embed.setAttribute("src", "proxy.swf");
    embed.setAttribute("flashvars", "
        messageName=" + methodName + "&
        messageArgs=" + JSON.stringify(
        methodArgs));

    document.body.appendChild(embed);
}
```

Listing 8: JavaScript function that creates the *proxy* Flash object.

```
static var lc:LocalConnection = new
    LocalConnection();
lc.connect("wii");

static function addCallback( methodName:String
    , instance:Object, method:Function) :
    Boolean {
    var script;

    lc[methodName] = function(args) {
        method.apply( instance, args );
    }
    script = "$('flash')."+methodName+" =
        function()";
           + "{";
           + '    callFlash("' + methodName +
               '", arguments);';
           + "}";
    js(script);

    return null;
}
```

Listing 9: `ExternalInterface.addCallback` Implementation.

```
class Proxy {
    function Proxy() {
        var lc:LocalConnection = new
            LocalConnection();
        var json = new JSON();
        var args = json.parse(_root.methodArgs
            );

        lc.send("wii", _root.methodName, args)
            ;
    }
}
```

Listing 10: *Proxy* Implementation.

A schematic of this mechanism can be seen in figure 1. When the `call` function is called, a new proxy Flash object will be created in the HTML document with `flashVars` set to "messageName=call&messageArgs=[]" (1). The proxy will then parse the `flashVars` parameters and invoke the `call` function in the main Flash object using `LocalConnection` (2). After invoking the function the proxy will remove itself from the HTML document, freeing up some of the devices's memory in the process (3). On the second call a new proxy Flash object will be created because it is no longer possible to reuse the `flashVars` of the first proxy created.

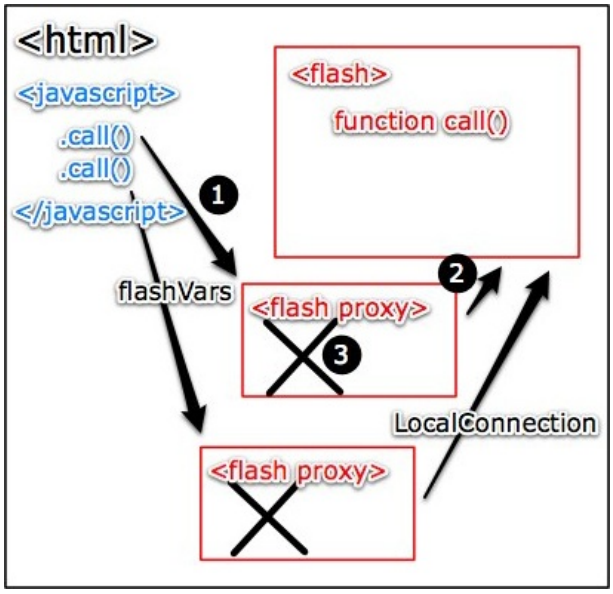

Figure 1: How ActionScript functions invocations work

In order to make both `ExternalInterface` and `Proxy` implementations completely self-contained the JavaScript function that creates the `Proxy` object is added dynamically by `ExternalInterface` during its initialization, removing the need to add it to every HTML file that makes use of it.

One of the problems of this pseudo-implementation is the order in which the messages arrive at the main Flash objects. When two consecutive Flash objects are added to the HTML document there is no guarantee of which order they will be executed and initialized in. Because of this behavior a sequential number is added to the `flashVars` parameter to make sure that the `Proxy` object will only invoke methods in the same order as they were invoked by JavaScript.

Browsers running in low-memory devices are very limited, meaning that there is a memory limit on how many Flash objects can reside on the page at the same time. Every invocation made in JavaScript to Flash will result in a Flash object that will only be used once. After several calls the browser will stop with an "Out Of Memory" error and will ask the user to reload. To stop this from happening the name of the Flash object will also be passed in the `flashVars` parameter. When the `Proxy` finally invokes the function it will remove itself from the page by calling `document.body.removeChild` using the `ExternalInterface.call` method.

Another problem is directly related to the maximum amount

of data that each `flashVars` parameter can carry. There is a 256 character upper limit per variable. In order to get around this limitation the `messageArgs` argument is split into several variables whenever it exceeds the 256 character limit – `messageArgs1`, `messageArgs2`, ..., `messageArgsN` – and reassembled at the `Proxy` Flash object.

4.4 Limitations

4.4.1 No return value

As mentioned before, this solution doesn't solve the problem of getting the returned values of the invoked functions.

Despite this limitation this method can still have some practical applications as can be seen in the 5 section.

4.4.2 Asynchronous

Since it is not possible to pause its execution in JavaScript, after invoking the ActionScript function there is no way to know if the function was actually executed. This means that the next JavaScript instruction cannot rely on the invoked function results.

4.4.3 Functions as arguments

Since all arguments passed to the `Proxy` Flash object are in the form of a JSON object, the passing of callback functions is not permitted and will be ignored by the JSON parser used.

The overcoming of this limitation might be possible by passing a special name referencing the JavaScript function and by the construction of an ActionScript proxy function that will invoke the original function using `ExternalInterface.call`.

4.4.4 Consecutive calls

The low-memory problem was mentioned in the 4.3 section and was solved by removing each `Proxy` object as soon as its work is done. However, this can still be an issue if several consecutive calls to ActionScript are made, because there will be no time for Flash to delete itself, resulting in hundreds of Flash objects alive at once.

5. RESULTS

These results were mainly achieved during the port of the "Media Player Opera Unite Application"[1] to the Wii. This Opera Unite application makes use of the SoundManager 2 library to play MP3- and AAC-encoded music files.

The main objective – to make the library work on the Wii without having to fork it and maintain customized versions of it – was successfully accomplished using the technique described in this paper. Due to the nature of the proposed method, this solution is slower than the native one since a new Flash object needs to be loaded and instantiated on every message. However, on the Opera Unite Application, there was no significant delay from the user point of view.

The SoundManager 2 was a good target because no invocation is making use of the returned value,[2] and synchronism between JavaScript and Flash isn't really a necessity. The only necessary change in the web application itself was the addition of a check to make sure that there were not too many consecutive calls being made. This problem was triggered whenever the "Play/Pause" button was clicked repeatedly.

Despite being mainly (perhaps only) used in the Media Player Unite application, this modified library can also be used by any web application that wants to target the Wii.

[1] `http://Unite.opera.com/application/162/`

[2] Only `getMemoryUse` uses the returned value, but it is only relevant for debugging purposes

6. RELATED WORK

Similar results were achieved by Mario Klingemann[3] regarding the passing of information from JavaScript to ActionScript. However, in his approach only a pair of numbers is sent since his main objective was to pass the Wiimote coordinates to Flash.

His technique involves listening to the `Stage.onResize` event that is triggered whenever the size of the Flash object residing in the HTML document is changed. He also uses a proxy Flash object and `LocalConnection` to listen to resize events and to invoke the updated coordinates event on the main Flash object[5].

Klingemann also refers to Aral Balkan and his research for communication between JavaScript and ActionScript on the Wii[3].

7. CONCLUSIONS

This paper was mainly written because of the general lack of documentation for JavaScript/Flash developers on devices that lack the `ExternalInterface`.

The possibility of using this method without having to modify the ActionScript source code also made it worthwhile to document and publish.

The SoundManager 2 author will be contacted and hopefully a version compiled with this customized `ExternalInterface.as` will be released. However, with all the recent improvements in both FlashLite and devices' browsers, it is expected that these workarounds will no longer apply in the future and that developers will be able to use `ExternalInterface` as they do at the moment on the desktop.

8. FUTURE WORK

The major bottleneck of this solution is the lack of returned values from the invocations.

No major thought was given to this problem because the SoundManager 2 library doesn't make use of returned values, however since a two way communication was achieved it might be possible to overcome this in future work.

The author believes that passing functions can be easily achieved by passing a `string` reference to the function when calling ActionScript. The `proxy` Flash object will then replace that `string` with an ActionScript function that will call back the JavaScript function with that name.

[3] `http://www.quasimondo.com/`

9. ACKNOWLEDGMENTS

The author wishes to acknowledge the entire Opera Software Team, namely the WebApps Team and in particular:

- Gautam Chandna – for giving me the chance, and trust, to solve this issue.
- Fredrik Öhrn – for pointing me in the right direction.
- Mario Klingemann – his code made me understand what was wrong with my LocalConnection usage.
- Scott Schiller – for his awesome JavaScript/Flash SoundManager 2 library.
- Filipe Cabecinhas, David Håsäther, Deniz Turkoglu, Mihai Sucan – for reviewing the paper.
- Chris Mills – for reviewing and helping with the publishing process.
- Charles McCathieNevile – for reviewing and helping with legal and standards issues.
- Sílvia Pinheiro – for all the support.

10. REFERENCES

[1] Adobe System Incorporated. flash.net.LocalConnection (Flex 2). `http://www.adobe.com/livedocs/flex/2/langref/flash/net/LocalConnection.%html`, June 2006.

[2] Adobe System Incorporated. Flash Player Version Penetration. `http://www.adobe.com/products/player_census/flashplayer/version_penetra%tion.html`, December 2009.

[3] Aral Balkan. Using the Wiimote buttons in Flash. `http://aralbalkan.com/825`, December 2006.

[4] Ecma. Standard ECMA-262, ECMAScript Language Specification. `http://www.ecma-international.org/publications/standards/ecma-262.htm`, December 1999.

[5] Mario Klingemann. how to make the wiimote work in flash. `http://www.quasimondo.com/archives/000638.php`, December 2006.

[6] Wikipedia. Wii launch — wikipedia, the free encyclopedia, 2010. [Online; accessed 24-January-2010].