Improve cross-domain communication with client-side solutions

Work around the SOP restrictions

Wang Jiaye
Hu Changchun

June 28, 2011

Introduction

Increasingly, websites need to collaborate with each other. For example, an online housing rental website needs support from Google Maps to show the location of a particular rental house. To meet such needs, different kinds of mashups have emerged. A mashup is a web application that integrates data or components from different providers to make it more useful or more customized. Mashups, or collaboration abilities, are considered an important part of Web 2.0.

Surprisingly, it's not easy to combine Asynchronous JavaScript and XML (Ajax) and mashups. Due to security restrictions imposed by browsers, it's also cumbersome to make different widgets on the page communicate with each other. Traditionally, you could solve this problem by setting up a proxy on the server side, which is not scalable. In this article, learn about some other solutions, on the client side, for cross-domain communication and data transfer.

Security restrictions

The same origin policy (SOP) prevents scripts loaded from one origin to get or manipulate properties or methods in the documents from another origin. The term origin is the combination of domain name, application protocol, and port of the document running the script. There can be misunderstandings about the SOP concept; it means more than site A couldn't get information from site B. You'll need to know what you can and cannot do under SOP restrictions.

Limitations with SOP

For example, a web page from origin A could:
• Get scripts, CSS stylesheets, or an image from origin B
• Contain an iframe/frame pointed to the page from origin B
• Send some information to origin B with the src attribute of HTML elements, such as iframe or img

A web page from origin A could not:

• Make an Ajax call to origin B
• Read or manipulate content in an iframe/frame pointed to page B

Why is this so? Mainly to protect a user's important information. The assumption is: if a user interacts with one provider, he would not want any information submitted to that site to be leaked to other sites. This kind of restriction limits cooperation among different websites, but protects users from potentially harmful attacks.

There are many solutions to deal with the problem. For example, JSONP exploits the fact that web pages can dynamically load scripts from any source. However, JSONP has two main restrictions: it does not have an error-handling mechanism, like Ajax call, and the script tag request is GET method, which has a length restriction. (For more information about JSONP, see the Related topics section.)

The following sections discuss client-side solutions to cross-domain communication and data transfer. Each solution has advantages and disadvantages; the application scenario largely affects your choice.

Cross-subdomain solution

If origin A and B share the same super domain, it's easy to let two documents access each other with the change of the document.domain property. document.domain is a read-only property in the HTML specification; most modern browsers allow it to be set to super domain (not top level). For example, a document with the URL www.myapp.com/index.html could set its own domain as myapp.com, while another document from sample.myapp.com/index2.html could also set its own domain as myapp.com. Figure 1 shows how document.domain works.

Figure 1. document.domain
With this cross-subdomain solution, origins from different subdomains could communicate under the same super domain, which is not an SOP restriction. But, strictly speaking, cross-subdomain solutions are most suitable for intranet applications.

**URL.hash(fragment id) solution**

A URL is composed of several parts, as shown in Figure 2 below:

**Figure 2. Components of a URL**

```
```

Traditionally, any change to a URL will load a new web page. The exception is a change of the hash value. Any change to the hash of the URL will not lead to a refresh of the page. Hash is already widely used in many Web 2.0 sites to bookmark each step when partially refreshing the page. In cross-domain communication, hash is also a valuable asset. Documents from different origins could set each other’s URL, including the hash value, though there are limitations on getting each other’s hash value. The documents could send messages to each other using the hash. Figure 3 shows an example.

**Figure 3. Communication using URL.hash(fragment id)**

In Figure 3, when A wants to send a message to B, it could modify the hash value of B, as shown in Listing 1:

**Listing 1. Sending message by url.hash**

```javascript
function sendMsg(originURL, msg){
    var data = {from:originURL, msg:msg};
    var src = originURL + "#" + dojo.toJson(data);
    document.getElementById('domainB').src=src;
}
```

Improve cross-domain communication with client-side solutions
A function in B will poll the hash value of itself and find out what A has sent. It could reply to A in the same way. If A wants to receive this message, it should also poll the hash value of itself.

Listing 2. Monitoring url.hash and receiving message from it

```javascript
window.oldHash="";
checkMessage = function(){
    var newHash = window.location.hash;
    if(newHash.length > 1){
        newHash = newHash.substring(1,newHash.length);
        if(newHash != oldHash){
            oldHash = newHash;
            var msgs = dojo.fromJson(newHash);
            var origin = msgs.from;
            var msg = msgs.msg;
            sendMessage(origin, "Hello document A");
        }
    }
}
window.setInterval(checkMessage, 1000);
sendMessage = function(target, msg){
    var hash = "msg="+ msg;
    parent.location.href= target + "#" + hash;
}
```

Like JSONP, this method also has a length limitation, but it can handle errors better. Some special characters, such as the question mark (?), are reserved characters in the URL and should be encoded first.

Listing 3. Sending message containing special characters by url.hash

```javascript
function sendMsg(originURL, msg){
    var src = originURL + "#" + encodeURI (dojo.toJson(data));
    ...
}
```

And, when receiving, you should decode it first.

Listing 4. Receiving message containing special characters

```javascript
function checkMsg(){
    var msgs = decodeURI(dojo.fromJson(newHash));
    ...
}
```

Cross-fragment technique

Since hash is already used in many websites for other purposes, it will be complicated for those websites to incorporate the URL.hash(fragment id) technique for cross-domain communication and data transfer. Cross-frame messaging is somewhat similar to fragment id messaging. Figure 4 shows how the cross-fragment technique works.
Figure 4. Cross-fragment technique

In the figure above, when A wants to communicate with iframe B, it will create an iframe in itself first. This iframe points to the "proxy" C that is in the same domain as B. It will include parameters or data and the frame identifier of B in the proxy's URL.

Listing 5. Sending message in cross-fragment technique

```javascript
function sendMsg(msg){
    var frame = document.createElement("iframe");
    var baseProxy = "http://www.otherapp.com/proxy.html";
    var request = {frameName:'otherApp',data:msg};
    frame.src = baseProxy+"#"+encodeURI (dojo.toJson(request));
    frame.style.display="none";
    document.body.appendChild(frame);
}
```

When C loads, it gets the request and data from A and invokes a corresponding method in B. Since B and C are in the same domain, they could directly call the other's method by the handler of another's window. A could successfully send messages to B, and B could respond in the same way.

Listing 6. Receiving message in cross-fragment technique

```javascript
window.onLoad = function(){
    var hash = window.location.hash;
    if(hash && hash.length>1){
        var request = hash.substring(1,hash.length);
        var obj = dojo.fromJson(decodeURI (request));
        var data = obj.data;
        //process data
        parent.frames[obj.frameName].getData(...);// getData in a function defined in B
    }
}
```

OpenAjax implementation

OpenAjax provides managed hub modules to support cross-domain communication and data transfer based on fragment id and a cross-frame technique. Managed hub modules include a
manager side and client side. The managed hub contains a message hub to store messages. Each widget that wants to communicate with others will set up a hub client in itself, and a corresponding container will also be set up to connect to it. The container will interact with the managed hub on behalf of the client. Client sides could send and receive messages using a subscribe/publish mechanism. The OpenAjax basic working flow is shown in Figure 5.

**Figure 5. Main workflow for OpenAjax**

![Diagram](image_url)

**Window.name solution**

Window.name is a bit of a tricky solution to cross-domain communication and data transfer. Traditionally, `window.name` is used as follows.

- Use `window.frames[windowName]` to get a child window.
- Set it as the target attribute in the link element.

Though `window.name` has a notable characteristic that makes it suitable for the "bridge" among documents from different origins, it's not a frequently used property. Whatever page it loads, the value of `window.name` remains the same. How can you use it under the SOP restriction? Figure 6 illustrates how `window.name` assists cross-domain communication.
Figure 6. window.name and cross-domain communication

When page A wants to get resources or a web service in another origin, it could add a new hidden iframe B in itself, targeting the outside resource or service. The server will respond with an HTML file, thus setting the window.name property to the data. Since A and iframe B now are not in the same domain, A still cannot get the name property of B. After B gets the data, it should be navigated back to any page that is in the same domain as A to make the name property accessible to A. After A gets the data, B could be destroyed at any time.

Use dojox.io.windowName for cross-domain communication

Dojo has provided support for cross-domain communication based on window.name. The only API is dojox.io.windowName.send(method, args), which is similar to dojo.xhrGet/dojo.xhrPost. The method could be GET or POST, and the args are similar to those in dojo.xhr. For example, you can send a cross-domain request in the client side, as shown in Listing 7:

Listing 7. Sending message by window.name

```
var args = {
  url: "http://www.sample.com/testServlet?windowName=true",
  load: function(data){
    alert("You've got the data from server " + data);
  },
  error: function(error){
    alert("Error occurred: " + error);
  }
}
dojox.io.windowName.send("GET", args);
```
You can use `dojox.io.windowName` the same way you use `dojo.xhr`. For the server side, it is suggested that, if you want resources or services accessible with `windowName` transport, you check the `windowName` parameter in the request. If the request includes such a parameter, the server should respond with an HTML document that sets its `window.name` to the data that needs to be delivered to the client. The code in Listing 8 shows an example.

**Listing 8. Back-end support for window.name message technique**

testServlet.java:
protected void doGet(HttpServletRequest request, HttpServletResponse response){
    // process request
    String returnData = ...;
    String isWindowNameReq = request.getParameter("windowName");
    if(null != isWindowNameReq && Boolean.parseBoolean(isWindowNameReq)){
        returnData = getCrossDomainStr(returnData);
    }
    response.getOutputStream().print(returnData);
}
private String getCrossDomainStr(String data){
    StringBuffer returnStr = new StringBuffer();
    returnStr.append("<html><head><script type="text/javascript">window.name='");
    returnStr.append(data);
    returnStr.append("'</script></head><body></body></html>);
    return returnStr.toString();
}

When navigating the frame back to any page in the origin domain, you should make sure the page exists in the domain. There will be problems with Internet Explorer if the page does not exist. In Firefox, you could simply use `blank.html`. In Dojo, you could specify which page to roll back with the `dojo.dojoBlankHtmlUrl` property. By default, it is set to `dojo/resources/blank.html` under the Dojo library.

The data volume transferred using `window.name` is much larger than that with `url.hash`. Most modern browsers support 16M+ data transfer based on the `window.name`.

**What's new in HTML5**

In the draft HTML5 specification, the new method `window.postMessage(message, targetOrigin)` is for safe cross-domain communication. When it is called, a message event will be dispatched, and, if the window is listening to the message event, it could get the message and the origin of the message from the event. Figure 7 shows an example.

**Figure 7. Cross-domain communication with HTML5**
In Figure 7, when the iframe wants to inform the parent window from another origin that it has been loaded successfully, it could send the message by `window.postMessage`. At the same time, it will monitor the feedback message, as shown in Listing 9:

**Listing 9. Sending message by HTML5 new method**

```javascript
http://www.otherapp.com/index.html
function postMessage(msg){
  var targetWindow = parent.window;
  targetWindow.postMessage(msg, "*");
}
function handleReceive(msg){
  var object = dojo.fromJson(msg);
  if(object.status == "ok"){
    //continue to do other things
  }else{
    //retry sending msg
  }
}
window.addEventListener("message", handleReceive, false);
window.onload = function(){
  postMessage("already loaded");
}
```

The parent document will listen to the message event. When the message arrives, it is first checked for whether it's from `www.otherapp.com`, and then the acknowledgement message is returned.

**Listing 10. Receiving message by HTML5 new method**

```javascript
http://www.myapp.com/index.html
function handleReceive(event){
  if(event.origin != "http://www.otherapp.com")
    return;
  //process data
  var otherAppFrame = document.getElementById("otherApp");
  otherAppFrame.postMessage("{status:'ok'},", "http://www.otherapp.com");
}
window.addEventListener("message", handleReceive, false);
```

The example code in Listing 10 could run in Firefox 3+, Internet Explorer 8, Google Chrome 2, Opera 9+, and Safari 4. It facilitates communication among documents from different origins. And, if your own document does not want to receive any messages from other documents, don't add message event listener and omit all the messages.
Related topics

- "OpenAjax Hub 2.0 Specification: Managed Hub Overview" (OpenAjax Alliance, 2009): Get an overview of the OpenAjax Hub, and learn more about the Managed Hub.
- "OpenAjax Hub 2.0 Specification: Managed Hub APIs" (OpenAjax Alliance, 2009): Learn to write your own cross-domain implementation.
- `dojox.io.windowName`: Read more about Dojo cross-domain communication through `windowName`.
- "Cross-domain communications with JSONP, Part 1: Combine JSONP and jQuery to quickly build powerful mashups" (developerWorks, February 2009): Understand how you can combine an obscure cross-domain call technique (JSONP) and a flexible JavaScript library (jQuery) to build powerful mashups surprisingly quickly.
- Follow [developerWorks on Twitter](https://twitter.com/developerWorks).
- [Dojo toolkit downloads](https://塍to.net/): Get Dojo (Dojo 1.6).
- [OpenAjax Hub library](https://塍to.net/): Get OpenAjax Hub.
- [IBM software](https://塍to.net/): Try IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.

© Copyright IBM Corporation 2011
[Trademarks](http://www.ibm.com/developerworks/ibm/trademarks/)