Client Logic for Firefox Component Specification

1. Design

The Client Logic for FireFox component provides client-side logic supporting Mozilla FireFox users' interaction with the Orpheus application. This client logic will be incorporated into a FireFox extension that enables FireFox to be used as an Orpheus client.

General Strategy

The overall Orpheus client will take the form of a browser extension whose user interface manifests through a browser tool bar, popup browser windows, and items added to the context menu. It will periodically poll the server for updates, and will rely on HTML pages loaded from the server for most of the content it presents. The Orpheus Client Logic for FireFox component will manifest as a Java class (or classes) that

- provides appropriate handling for UI events generated by user interaction with the client-side controls,
- monitors browser / document events to issue alerts to the user at appropriate times,
- periodically polls the Orpheus server for application state updates,
- exposes specific methods for interaction with scripts,
- provides an event listener interface by which it can notify the rest of its plugin about events, and
- as needed, pops up new browser windows in which to display information to or solicit information from the user.

This component handles all of that functionality through a large class that is accessed via LiveConnect through JavaScript. This component implements the delegation pattern by allowing that much larger class to encompass the API's of a few other classes, allowing the JavaScript to only have to deal with a single class, instead of multiple classes.

This component also provides a wrapper JavaScript file that handles a lot of the mundane tasks in JavaScript associated with using the component, allowing for much easier access to the component through JavaScript. This is additional functionality that should prove very useful to the extension implementation of this component.

This component uses the LiveConnect API quite a bit, allowing the Java component to both be called by JavaScript, as well as allowing the Java component to call JavaScript functions in the browser. We use this Java to JavaScript functionality when setting cookies, calling event handler JavaScript methods, and manipulating pop-up windows.

The concrete extension will register this component's methods as event handlers for events like "successful login" and the "Show Upcoming Games" button click. This component does not register itself, that is all handled by the host browser extension.

More information about the LiveConnect API can be found here:

http://java.sun.com/products/plugin/1.3/docs/jsobject.html#Intro

We are primarily concerned with the JSObject class, notably the “eval” and “get / set” member methods.

Developers should be aware that all code in this component is run on the client-side, hosted in the client JRE.

An example showing the Javascript to Java to Javascript calling is available in the “docs” folder. It shows a simple XUL file that uses Javascript to call into Java that then calls back out into the Javascript.

1.1. Design Patterns

The **strategy** pattern applies to the UIEventListener and ExtensionPersistence classes, as they are used generically by FirefoxExtensionHelper, regardless of their underlying implementation.

The **observer** pattern applies to the UIEventListener as implementations of that interface are alerted of UI events raised in the browser. It also applies to the OrpheusServer class and how it alerts the FirefoxExtensionHelper class when a server poll has received a response.

The **delegation** pattern applies to the FirefoxExtensionHelper class, as it contains a number of different classes whose API’s are consolidated in that single class. This makes it much easier for the JavaScript to interact with the component, as it only has to worry primarily with this class.

1.2. Industry Standards

JavaScript, DOM, HTTP, HTML, XML, Cookies, SHA1

1.3. Required Algorithms

This component is an applet that is loaded on a page via XUL. The sample XUL would look something like this:

```
<html:applet code="FirefoxExtensionHelper.class"
    codebase=…
    width="0"
    height="0"
    MAYSCRIPT=”true”>
 </html:applet>
```

This is all pretty standard, but note that the “MAYSCRIPT” attribute MUST be set to true for this component to work as expected. This allows the Java code to interact with the Javascript in the browser without any security concerns.

Loading a FirefoxExtensionHelper from configuration values:

The FirefoxExtensionHelper constructor not only loads a number of different configuration values, mostly for URL access to the server, it also ensures that the persistence and server poller are created correctly, with the Orpheus server poller being started in the constructor.

This constructor initializes the member variables in this class using values read from the configuration namespace given. This constructor should use the configuration namespace to read in the following values:

domain_parameter_name: The query parameter name used when testing whether or not a domain is host to any games

orpheus_url: The URL to the Orpheus server, used to initialize the "server" member variable

orpheus_poll_time: An integer value indicating how many minutes to wait between each server poll

orpheus_timestamp_parameter: A string parameter name for passing the timestamp along with poll requests

persistence_class: A class name, like "com.orpheus.plugin.firefox.persistence.CookieExtensionPersistence", used to initialize the persistence member variable

event_pages: Each UIEventType / string URL pair, entered into the "eventPages" member variable.

hash_match_URL: The URL to load when a given object matches the current target

hash_match_domain_parameter; The query string parameter name to use when passing the domain name to the hash_match_URL

hash_match_sequence_number_parameter: The query string parameter name to use when passing the sequence number to the hash_match_URL

target_text_parameter: The query string parameter name to use when passing the current target string to the hash_match_URL

default_height / default_width (optional): The default dimensions of popup windows

page_changed_URL: The base of the URL to request when the user requests a new page in a domain that participates in the Orpheus application

After the persistence implementation has been set, the feed timestamp should be retrieved through the getFeedTimestamp method. After all the configuration values are loaded, "server" member variable should be set to a new OrpheusServer instance, created using the URL and poll time loaded, and the feed timestamp read from persistence, as well as "this" for the listener. After the OrpheusServer instance has been created, the polling should be started through "server.Start()"

Polling the server:

This functionality is implemented in OrpheusServer.pollServerNow. It creates a request to the server for an Atom feed that is handled by this component. This method performs the actual URL request to the Orpheus Server, creating an RSSFeed instance from the Atom response, passing it to the FirefoxExtensionHelper instance contained in the listener member variable. This method should perform the following operations:

```
Create a new HttpURLConnection using the "url" member variable
Set the timestamp parameter using the timestampParameter string, and the
timestamp value
If the timestamp value is null, don't add the parameter
Get the input stream of the connection
(HttpURLConnection.getInputStream())
Read the contents of the stream into a temp string
Close the connection
Create a new RSSFeed instance based on the string read
```

```
Pass the RSSFeed instance to "listener.serverMessageReceived".
Get the atom:update timestamp value from the feed and set the "timestamp"
member variable appropriately.
```

Note that if any errors occur while performing this operation, they should be ignored, as this method is called continuously in a thread.

Processing a server poll response:

This functionality is implemented in FirefoxExtensionHelper.serverMessageReceived, called from the OrpheusServer when it receives a response. The server response with an Atom feed containing either text content or a serialized BloomFilter instance. This method processes the given Atom document received from the Orpheus Server in response to a poll request. The feed will contain a "content" node that contains one of the following 4 types: text, html, xhtml, or application/x-tc-bloom-filter.

If the content node is text, html, or xhtml, the content should be parsed and displayed to the user in a pop-up window, using the popupWindowWithContent method: popUpWindowWithContent(<atom content>, false, false, false, false, false, false, false,800,600);

If the content node is of the "application/x-tc-bloom-filter" type, a new BloomFilter instance should be created based on the serialized data parsed from the content node. The atom:update timestamp of the feed should be parsed from the feed and persisted to the client browser through "persistence.setFeedTimestamp".

Event handling:

The FirefoxExtensionHandler class has a number of event handler methods, like “logInClick()” which alert the component to events raised in the browser. These are typically handled by getting a configured URL and popping up that specific URL in a new browser window. These methods are called from Javascript in the actual extension, alerting this component to the fact that an event has been raised. These methods should do the following operations: Get the URL for the page to display to the user from the eventPages Map, for the UIEventType.* value that corresponds to the event. Popup that page to the user, the popup method will handle reusing a previous window: popUpWindow(<URL>, false, false, false, false, false, false, false, defaultHeight, defaultWidth);
For each UIEventListener in the eventListeners List, call the corresponding event handler method.

Changing a page:

This functionality is called in response to a special event that indicates that the user has changed the page in the current browser instance. The FirefoxExtensionHelper.pageChanged method encapsulates this functionality and is called directly from the host browser extension. This method is called from Javascript in the actual extension, alerting this component to the fact that the current pages has changed. We test the new page to see if it is in the "filter" member variable. Before testing the page name, we first must strip just the domain portion from the page name to use when validating the domain. For instance "http://www.google.com?q="asdf"" would be stripped to just google.com. If the domain isn't in the BloomFilter, no work is done. The domain parsed should be used to also set the currentDomain member variable. If the BloomFilter contains the domain string we parsed above,
We call server.queryDomain, passing in the configured domainParameter member variable, and the new domain we parsed. The number of active games in that domain is

returned. If the response from the method is not 0, we create a new request URL, based on the pageChangedURL, with a query parameter of the domainParameter name, with the value being the new current domain. Once we have this URL, we pop it up in a new window with minimal adornments, using popupWindow(URL, false, false, false...);

Testing if an object is the current target:

The FirefoxExtensionHelper class contains a “currentTargetTest” method that tests if a given object matches the current target of the game, and if it does, a number of operations are made to handle that particular event. This method determines if the given String DOM Element matches the current target. This method should create an org.w3c.dom.Element from the given String and then this method should traverse the element given, extracting the text content of each node, removing the markup tags of the element. For each text content item, all trailing and leading whitespace should be removed, and all sequences of more than one whitespace character should be compressed to a single space. Finally, all characters in the resultant string should be changed to lower case. Once we have the text all concatenated together and appropriately trimmed and lower cased, we encode the text to UTF-8, using java.net.URLEncode.encode(string, "UTF-8"). This also handles converting the text to the appropriate application/x-www-form-urlencoded format for posting back to the server, if necessary. Once we have the encoded string, we get a new SHAAlgorithm instance for hashing the data. Then we call SHAAlgorithm.hashToHexString(encoded string) to get the encoded data. Once we have the encoded, hashed data, we compare the hex string to the current target ID string, retrieved from persistence.getCurrentTargetID(). If those values match, we create a new URL to load, using the hashMatchURL as the base, and adding the following query parameters

The hashMatchDomainParameter name, and the currentDomain value
The hashMatchSequenceNumber name, and the persistence.getSequenceNumber value
The targetTextParameter name, and the encoded text value

Once we have the query string built , we generate a POST request for the server, which we execute and read the response from. Once we have the response and have read it in its entirety, we call popupWindowWithContent to show the content retrieved from the server. Pseudo-code for making the POST request is below:

```
// Construct data
String data = URLEncoder.encode(hashMatchDomainParameter, "UTF-8") + "="
+ URLEncoder.encode(currentDomain, "UTF-8");

data += "&" + URLEncoder.encode(hashMatchSequenceNumberParameter, "UTF-
8") + "=" + URLEncoder.encode(persistence.getSequenceNumber, "UTF-8");

data += "&" + URLEncoder.encode(targetTextParameter, "UTF-8") + "=" +
encodedString;
// Send data
URL url = new URL(hashMatchURL);
URLConnection conn = url.openConnection();
conn.setDoOutput(true);
OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream());
wr.write(data);
wr.flush();

// Get the response
BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
String line;
String response;
```

```
while ((line = rd.readLine()) != null) {
      response+=line;
}
wr.close();
rd.close();
```

Popping up a window with content:

As part of the API for this component, we have the ability to pop-up a new window to the user that we set the content in. This allows us to show the user content retrieved from Atom feeds or other non-HTML data. This functionality is implemented in FirefoxExtensionHelper.popupWindowWithContent. Note that both the popup methods will reuse a single window, allowing at the most a single popup window for the component. This method pops up a window containing the given URL, with the decoration and size options shown. This method should first see if the "popupWindow" member variable is not null. If it isn't null, it references a current popup window that we should reuse instead of opening a new one. To reuse the window, we just need to set the URL to the URL given and then resize the already opened pop-up to the size given. We don't dynamically change the decoration, just the size. To ensure the scripting interface is available in the new window, we make sure to write out the "<script src="FirefoxExtension.js"></script>".

For an already open window, call:

```
popupWindow.setMember("location.href", url);
popupWindow.call("resizeTo", new Object[]{height, width});
popupWindow.call("document.write", new Object[]{"<script...."});
```

If the window isn't currently open, we need to open it using Javascript run on the clientWindow object. We also need to make sure we keep a reference to the opened window, and set the window.opener property to point back to the clientWindow in this class. The boolean values should be concatenated together in a comma-delimited string, detailed here:
http://www.javascript-coder.com/window-popup/javascript-window-open.phtml

---

```
popupWindow=clientWindow.call("open", URL, "OrpheusChild", <options
string>);
popupWindow.setMember("opener", clientWindow);
popupWindow.call("document.write", new Object[]{"<script...."});
```

This will ensure the window is opened correctly and that we keep a reference to the child popup window, and that the popup has a reference back to this parent window.

Setting cookies:

This component uses a browser cookie to persist values between browser restarts. This is the typical way of performing persistence in a browser. The CookieExtensionPersistence class implements this functionality. It has a setCookieString method that uses a JSObject instance to call out to the browser window to set the cookie value containing the 4 persistent values we need to save. This method sets the cookie value to a string created with the timestamp, workingGameID, sequence number, and currentTargetID member variable values in the following format:

```
TIMESTAMP=<timestamp value>;WORKING_GAME_ID=<working game
ID>;CURRENT_TARGET_ID=<current target ID>;SEQUENCE_NUMBER=<sequence
number>
```

The path of the cookie should be "/", allowing all pages to access the cookie.
With the path added, the entire string set should look like this:

```
TIMESTAMP=<timestamp value>;WORKING_GAME_ID=<working game
ID>;CURRENT_TARGET_ID=<current target ID>;SEQUENCE_NUMBER=<sequence
number>;path=/;
```

If any value is missing, for instance if "timestamp" is null, it should be omitted from the cookie string. After the string has been created, it can be used to set the document cookie like this:

```
document.setMember("cookie", cookieString);
```

The calendar string written for the TIMESTAMP value should be able to be deserialized in setClientWindow to the same timestamp written.

1.4. Component Class Overview

FirefoxExtensionHelper:

This class is the main class in the component. This class implements both the Observer pattern for event handling, as well as the delegation pattern for delegating functionality to multiple different classes. We use the delegation pattern to allow this class to encompass the API that needs to be used by the Firefox extension in a single class, allowing access to the helper methods to be much easier. This class provides methods for handling events, setting persistent values, polling the server and popping up windows. This class also provides functionality for testing a given object to see if it matches the current target ID hash string and handling that particular event appropriately. When popping up windows in response to events, all windows should be opened with minimal adornments of toolbars, all the boolean parameter values should be false.
This class is mutable, as its internal state can change in a number of ways, including in response to polls made to the Orpheus server. This class extends the Applet class so it can be embedded on a page via XUL.

OrpheusServer:

This class handles making polling requests at specific intervals to a configured Orpheus server URL. This class makes the request and then calls the FirefoxExtensionHelper.serverMessageReceived method to handle the response. All this class has to do is call the server and then pass the response to the listener member variable for handling. The response should be an Atom 1.0 feed document that outlines content either as a serialized Bloom filter instance, or as text, html, or xhtml, which is displayed by the FirefoxExtensionHelper class to the user. This class is mutable, as the pollTime can change after instantiation. It implements the Runnable interface, allowing the parent FirefoxExtensionHelper class to run the polling in a separate thread.

UIEventListener:

This interface defines methods used to notify event listeners of the various events that can be raised on a page and then passed into this Java component via Javascript. Custom implementations can be used to perform various operations depending on the various method implementations. Implementations of this interface do not need to be thread safe. Note that all implementations will run client-side, and are called AFTER the FirefoxExtensionHelper has popped up a new window and made the request in response to each click event.

ExtensionPersistence:

This interface defines the contract for persistence of properties used by this component, and indirectly the FireFox extension that uses this component. As this code is run on the client side, the persistence implementations should save and read from an accessible

location. Because the code is run in a security sandbox, implementations need to be aware of the security restrictions put on them. The current implementation provided is a mechanism that saves the values to cookies on the client side, allowing the values to persist between browser restarts, provided the user doesn't clear out their cookies. Implementations of this interface do not need to be thread safe.

UIEventType:

This enumeration holds values that indicate the type of an event. These enumeration values are used in both JavascriptUIEventListener and FirefoxExtensionHelper to relate items to specific events. In FirefoxExtensionHelper, the event types are associated in a Map to the page to load in response to the event, and in JavascriptUIEventListener, the class maps UIEventType values to a list of Javascript functions to call in the client browser in response to the event types. This class is immutable and thread safe.

CookieExtensionPersistence:

This class implements the ExtensionPersistence interface to save and retrieve the values from a cookie saved on the client side. This class sets three cookie values, the feed timestamp, the sequence number, the working game ID, and the current target ID. The values should be saved in a cookie that looks like this: TIMESTAMP=<timestamp value>;WORKING_GAME_ID=<working game ID>;CURRENT_TARGET_ID=<current target ID>;SEQUENCE_NUMBER=<sequence number>
When setting and retrieving these values, care should be taken to preserve existing values so that changing one value doesn't cause the loss or corruption of one of the other two values. This class parses out the four values in the "setClientWindow" method and then sets them through the "setCookieString" method. The accessors and setters just have to use the member variables parsed, with the setters just calling "setCookieString" after the value has been set. This class is mutable, as the member variable values change after instantiation.

JavascriptUIEventListener:

This class implements the UIEventListener interface to call registered Javascript functions in the client browser in response to events raised through the FIrefoxExtensionHelper. The flow is as follows: The browser calls the registered Javascript function for a click event. That function calls the Java FirefoxExtensionHelper method for that click event The FirefoxExtensionHelper calls this class in response to the click event This class calls back out to a different Javascript function. Basically, this class allows us to have both Javascript and Java event handler functions for a click event raised by the user. This class is mutable as its registered listener functions and clientWindow can change after instantiation.

FirefoxExtensionWrapper.js:

This javascript file contains wrapper functions used to reference the Java code in the com.orpheus.plugin.firefox package. This file is additional functionality, allowing the user to access the functions of the Java component directly through Javascript.
This file handles creating the FirefoxExtensionHelper and JavascriptUIEventListener instances, as well as maintaining a reference to UIEventType, used when getting the constant names. The "init" function handles setting the security policy. Security of LiveConnect code is covered here:
http://java.sun.com/j2se/1.3/docs/guide/plugin/security.html.
The init function also handles instantiating the helper and listener member variables, as well as creating the UIEventType class for use in the functions. All the methods use those values when accessing the FirefoxExtensionHelper class. The "init" function has to

be called before any other functionality is available.

1.5. Component Exception Definitions

FirefoxExtensionConfigurationException:

This exception is thrown when the configuration values read for the FirefoxExtensionHelper class are either missing or erroneous. The constructors in this exception should wrap those of the base class.

FirefoxClientException:

This exception is the base exception for this component. All custom exceptions should extend from this exception. The constructors in this exception should wrap those of the base class.

FirefoxExtensionPersistenceException:

This exception is thrown when errors occur while setting or retrieving the persistent values through one of the ExtensionPersistence implementations. This could be due to IO or security errors, as the code is run on the client side. The constructors in this exception should wrap those of the base class.

1.6. Thread Safety

This component is not thread safe. FirefoxExtensionHelper, CookieExtensionPersistence, JavascriptUIeventListener, and OrpheusServer are all mutable. Note that this shouldn't be a problem, as one instance of this component will be associated with each browser instance, so the mutability should be confined to a single browser thread. The only issue where mutability of a property will be an issue is in the FirefoxExtensionHelper.filter value, which is set asynchronously via calls from the OrpheusServer. To make this thread safe, all access to the "filter" member variable should be synchronized to ensure thread safety. To make the component thread safe as a whole, locking would have to be added when doing things like setting persistent values, or adding event listeners.

2. Environment Requirements

2.1. Environment

- Development language: Java
- Java 1.4

2.2. TopCoder Software Components

- **Bloom Filter 1.1** is used to contain the domains that are part of the application
- **Configuration Manager 2.1.5** is used to read the configuration values from
- **RSS Generator 2.0** is used to parse and examine the Atom feed returned by the Orpheus server.
- **Hashing Utility 1.0** is used to generate an SHA hash for objects to compare to the current target.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1. Package Name

com.orpheus.plugin.firefox

3.2. Configuration Parameters

All parameters are required, unless marked "optional".

Parameter	Description	Values
domain_parameter_name	The query parameter name used when testing whether or not a domain is host to any games	Any non-empty string
orpheus_url	The URL to the Orpheus server, used to initialize the "server" member variable	Any valid URL
orpheus_poll_time	An integer value indicating how many minutes to wait between each server poll	Any positive integer
orpheus_timestamp_parameter	A string parameter name for passing the timestamp along with poll requests	Any non-empty string
persistence_class	A class name, like "com.orpheus.plugin .firefox .persistence .CookieExtensionPersistence", used to initialize the persistence member variable	Any class name that can be loaded via reflection
hash_match_URL	The URL to load when a given object matches the current target	Any valid URL
hash_match_domain_parameter	The query string parameter name to use when apssing the domain name to the hash_match_URL	Any non-empty string
hash_match_sequence_number_parameter	The query string parameter name to use when passing the sequence number to the hash_match_URL	Any non-empty string
target_text_parameter	The query string parameter name to use when passing the current target string to the hash_match_URL	Any non-empty string
default_height	The default height of pop-up windows (optional)	Any positive integer
default_width	The default width of pop-up windows (optional)	Any positive integer
page_changed_URL	The base of the URL to	Any valid URL

	request when the user requests a new page in a domain that participates in the Orpheus application	
event_pages	A list of comma-delimited event / page pairs, used to link event types with pages to load, should be specified like “LOGIN_CLICK, http://www.google.com”	A list of comma-separated values

4. Usage Notes

4.1. Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2. Demo

Here is a simple XUL file that loads the Firefox Extension Helper using the FirefoxExtensionWrapper.js file. It also registers some event handlers for specific buttons. Note that the “MAYSCRIPT” value of the applet MUST be set to true for this component to work.

```
<window
xmlns=http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul
onload=”init(myApplet)”>

<script src="FirefoxExtensionWrapper.js" type="application/x-
javascript"/>

<html:applet code="FirefoxExtensionHelper.class"
    codebase=…
    width="0"
     id=”applet name”
    height="0"
    MAYSCRIPT=”true”>
 </html:applet>

<button label="Log In" onclick="logInClicked();" />
<button label="Log Out" onclick="logOutClicked();" />
<button label="Show Active Games" onclick="showActiveGames();" />

...

</window>
```

Javascript calls

```
//Set the poll time to 10 minutes
setPollTime(10);

//Set the working game ID
setWorkingGameID(12);
```

```
//Get the working game ID, event listeners will also be notified
var ID=getWorkingGameID();

//Set the current target ID
setCurrentTargetID(“lksjlksjeoijlsiejf”);

//Check if our window is a popup
if(isPopupWindow())
	<<do something>>

//Create an element string to test against the current target
var target=....

//This could cause a popup to occur if the object given matches the
//current target.
currentTargetTest(target);

//Popup a window
popupWindow(http://www.google.com, true, false, true, false, false,
false, false, 480, 640);

//Force a poll to the server, this could cause a popup if the server
//returns a message
pollServerNow();

//The user changed a page, if somegame.com is a valid game domain, a
//popup will appear
pageChanged(“http://www.somegame.com”);


function customLogIn()
{
	alert(“Log In was clicked”);
}

//Add a custom JavaScript event handler
addLogInClickHandler(“customLogIn”);

//Remove the custom event handler
removeLogInClickHandler(“customLogIn”);
```

5. Future Enhancements

Future event listeners and persistence mechanisms can be added