Client-Server Connection Status Monitoring using Ajax Push Technology

Julien R. Lamongie*
Lawrence Technological University, Southfield, MI, 48219

This paper describes how simple client-server connection status monitoring can be implemented using Ajax (Asynchronous JavaScript and XML), JSF (Java Server Faces) and ICEfaces technologies. This functionality is required for NASA LCS (Launch Control System) displays used in the firing room for the Constellation project. Two separate implementations based on two distinct approaches are detailed and analyzed.

Nomenclature

AJAX = Asynchronous JavaScript and XML
CSS = Cascading Style Sheet
DOM = Document Object Model
EL = Expression Language
GIF = Graphical Interchange Format
HTML = Hyper Text Markup Language
HTTP = Hyper Text Transfer Protocol
JSF = Java Server Faces
JSP = Java Server Pages
LCS = Launch Control System
XML = Extensible Markup Language

I. Introduction

Launch Control System (LCS) displays will be used in the firing room of the Launch Control Center at Kennedy Space Center by Constellation[1] System Engineers (CSEs) to monitor and control the launch data coming from the launch vehicle, solid rocket boosters, etc. Due to the critical nature of the incoming data, the status of the connection between client displays and the display server needs to be monitored on a close basis. One of the requirements for the displays is to provide a visual indicator of the good health of the connection between the client and the server so that CSEs know the data they visualize is correct and up to date.

Based on the intention to make LCS displays portable, accessible, maintainable and in line with current technology, a NASA case study identified the ICEfaces[2] framework with Ajax Push as the candidate of choice for the runtime display environment. ICEfaces uses Ajax Push based on the Java Server Faces[3] life cycle – itself built upon the Java Server Pages[4] framework – for rendering server-initiated changes to the client browser.

In this paper, two separate NetBeans projects were created as working demonstrations of a client-server connection monitoring mechanism based on the previously explained technologies. The first project uses a pure server-centric approach for visually rendering updates to the client, whereas the second project uses a long running client-side JavaScript task. The pros and cons of each approach are discussed.

* System Software Engineering Intern, Launch Control System, Kennedy Space Center, NASA.

II. Underlying Technologies

A. Java Server Pages and Java Server Faces

Both Java Server Pages and Java Server Faces are specifications that are part of Java Enterprise Edition[5], a platform for server programming using the Java language.

Java Server Pages (JSP) is a technology to create web pages that display dynamically generated content. As such, it is comparable to other server-side languages such as PHP or ASP. A JSP page contains static content (e.g. HTML) along with Java code and certain XML[6]-like pre-defined tags. When they are first called by an HTTP request, JSP pages are converted to servlets, which are java programs running on the server, rather then on the browser. The server must therefore support the Java platform servlet specification. The three main features of the JSP technology are: (1) a language to develop JSP pages, (2) a language for accessing objects on the server (Expression Language or EL), and (3) a mechanism for extending the JSP language (tag libraries).

The Java Server Faces (JSF) technology is a development framework built upon JSP (see Figure 1). It is used to create user interface components based on the Model View Controller (MVC) design pattern, which separates application logic from presentation. The state of the components is saved when the client requests a new page and restored when the request is returned, following a six-phase lifecycle[7].

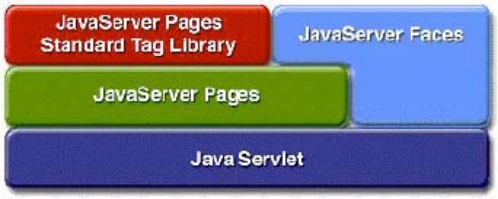

Figure 1. JSF and JSP building blocks.

B. Managed Beans

A Java Bean is a reusable software component that can be visually manipulated in application builder tools (e.g. NetBeans). From a programming standpoint it's simply a Java class with a public no-argument constructor and getter and setter methods to access class variables (properties). A managed bean (also called backing bean) is a Java Bean whose scope is managed by JSF through a configuration file.

There are three possible scopes: (1) application, which lasts until the server stops the application, (2) session, which lasts until a user's session times out, and (3) request, which ends when the response is sent back to the user. The managed bean contains the application's data and business logic. Expression Language is used to access the managed bean's elements from within the JSP page.

C. Ajax Push and ICEfaces

Ajax Push refers to the process of asynchronous updates of a web page on the client browser, based on server-side events. This is made possible through a mechanism called HTTP inversion or long polling, which consists in opening a connection to the web server and keeping it open until some state changes on the server. ICEfaces is an open-source integrated application framework that has Ajax Push natively built into all of its components.

ICEfaces replaces the standard JSF mechanism with a mechanism based on an Ajax bridge residing on both the server and the browser. The DOM, which is a standard platform and language independent object model used to represent HTML, is cached on the server and when its state changes, the server-side bridge pushes down the changes to the browser, where they are then reassembled by the browser-side Ajax bridge. This process, called Direct-to-DOM rendering, is illustrated in Figure 2.

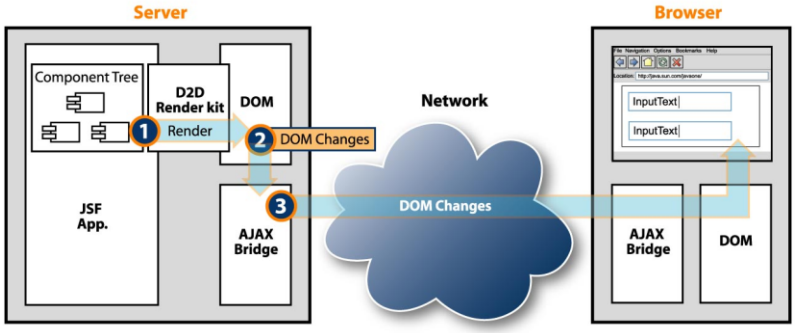

Figure 2. Asynchronous updates with Direct-to-DOM (D2D) rendering.

III. Prototype Implementations of a Connection Status Indicator

Two separate projects were created as prototype implementations of the connection status visual indicator that will be needed on each template used to create LCS displays. Both of these projects are Visual Web JSF-ICEfaces projects. The specifications for the tools used in both projects are:

- NetBeans 6.1[8]
- ICEfaces 1.7.1
- Java Runtime Environment 1.6.0_03-b05
- GlassFish v2 (web server bundled with NetBeans)[9].

A. Rendering

Both scenarios used an interval renderer to render data at 250 milliseconds, which is the required rate for display updates. Figure 3 illustrates the rendering process on the server side.

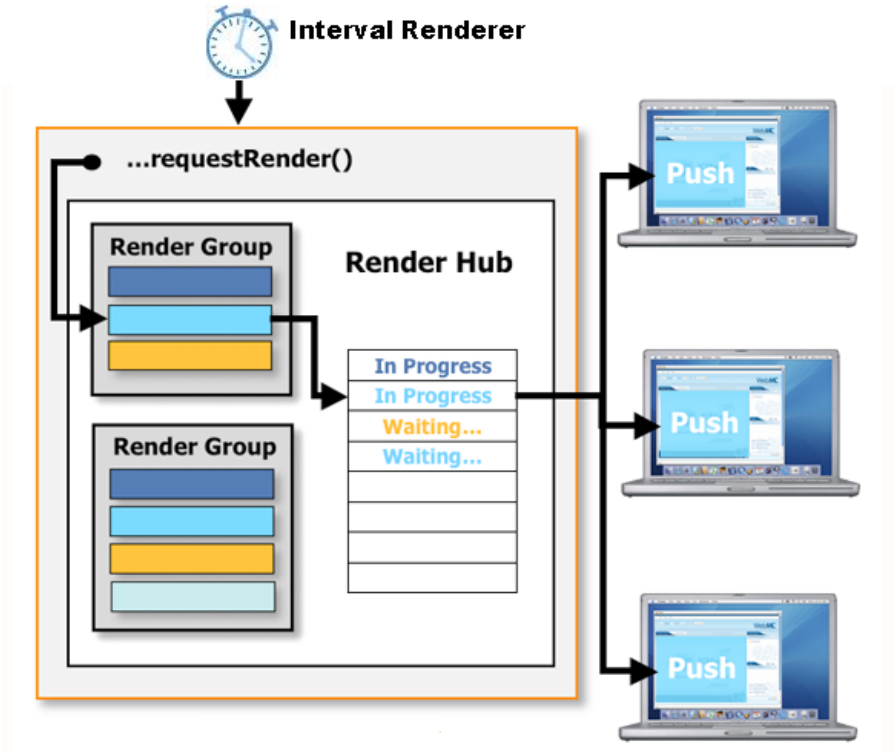

Figure 3. Interval Rendering using ICEfaces.

The rendering was implemented in a totally separate way from the connection status mechanism through a an application-scoped managed bean (named ClockBean) implementing ICEfaces' Renderable and DisposableBean interfaces and containing an IntervalRenderer object. The initial rendering is started when the JSP page loads through an invisible text field bound to this bean. The EL used for the binding is shown below. The same source code was used in both project.

```
<ice:outputText style="visibility: hidden" value="#{ClockBean.autoLoad}"/>
```

B. Server-centric Approach

In this project, the connection status is represented visually as a progress bar that fills up if the connection is healthy and that remains static if it is not. The mechanism to do so is as such: the server pushes a value down to an OutputProgress ICEfaces component on the webpage every time the page is re-rendered. This value (from 0 to 100) represents the percentage that the bar should be filled. With each push this value is increased until it reaches 100, after which it wraps back to 0 in a never ending loop as long as the connection remains healthy. If the server stops

pushing to the progress bar, the progress remains the same and thus the lack of movement alerts the user that there is a problem with the connection.

Since the purpose of the progress bar is to indicate that the connection is healthy, there is no need to make the value the same on all web pages that are rendered, as long as each page's progress bar moves as server pushes occur. This makes it possible to make the connection status a request-scoped managed bean (named ClientServerHeartbeat) and avoid the overhead of creating a server task to increment the value. The OutputProgress component's value property is bound to this bean's value variable using EL. Each time the page is re-rendered by the interval renderer, the value is incremented. The EL used for the binding is shown below.

```
<ice:outputProgress id="heartbeat" label=" " labelComplete=" "
                    styleClass="lcsClientServerHeartbeat"
                    value="#{ClientServerHeartbeat.value}"/>
```

Each time the page is re-rendered, the getter function for the value variable of the ClientServerHeartbeat class is called. This simple function is shown below.

```
public int getValue() {
   value += increaseRate;
   if (value > 100) value = 0;
   return value;
}
```

The rate that the bar increases depends on the rendering interval and on the rate of increase which is a modifiable value set in a configuration file. The visual appearance of the progress bar can be configured as well, through a CSS[10] (Cascading Style Sheet) file attached to the JSP page. The name of the CSS style to use is specified as a parameter in the ice:outputProgress tag.

C. Client-centric Approach

In this project, the connection status is represented visually as an animated GIF (Graphic Interchange Format) image when the connection is healthy and a different, static GIF image when it is not. The mechanism to do so is as such: the server pushes a value down to a hidden field on the webpage at a given set interval. A client-side JavaScript running task stores this integer and compares it with the previous value sent down by the server. It the values are the same then the JavaScript task changes the animated gif to indicate that the connection is no longer healthy. The task can be configured to allow for a set number of retries before actually changing the GIF image.

Just like in the server-centric approach, there is no need to make the value the same on all web pages that are rendered – meaning the connection status can be a request-scoped managed bean. A hidden field on the JSP page is bound to this bean's value variable using EL. Each time the page is re-rendered by the interval renderer, the value is incremented. To avoid having a never ending increase, the value is reset to 0 after a pre-determined arbitrary value, such as 100. This works because the client-side JavaScript task checks that the values are different, not that the new value is larger then the previous one.

In this scenario, the code to change the image must reside on the client side and not the server, otherwise a lost connection would mean the animated GIF would never be changed to the static one and thus the indicator would make it seem as if the connection was still healthy. This code is implemented through a simple JavaScript task that gets initialized when the JSP page first loads on the client browser using the window.onload JavaScript event:

```
window.onload = initClient;
```

The above code executes a function called initClient when the browser finishes loading the web page. The window.onload event can be called from within the JavaScript file itself without modifying the JSP page. The JSP page's header tag links to the JavaScript file and the rest is done automatically by the browser. To create a recurring task, the initClient function calls the setInterval function:

```
function initClient() {
   setInterval("heartbeat()",500);
}
```

The setInterval function executes a function at regular intervals. Here it executes the heartbeat function every 500 milliseconds. The heartbeat function has the code to check that the server has pushed new data and to check if the connection GIF is the correct one.

D. Analysis

The server-centric approach has the advantage of not requiring any client-side JavaScript. However, because the progress bar's motion is controlled by each value being pushed down from the server and because the timing for this is not perfectly the same at each push, there can be slight stuttering. The motion is not perfectly fluid like it is for an animated GIF image.

The client-side approach is the exact opposite. The motion is fluid, but this requires a long running client-side JavaScript task on each browser page connected to the display server. Because ICEfaces embeds the JavaScript functionality required for Ajax Push in its Ajax bridge, and because it uses DOM-diffing (the process of comparing two DOM versions to find the differences) on the server-side, it does not recommend changing the DOM on the client-side. In this particular instance, this does not pose a problem because the change is limited to the URL (Uniform Resource Locator) property of an image tag and not an ICEfaces component. Another potential problem with this approach is that only one function may be assigned to the JavaScript window.onload event. Future requirements to use the window.onload event would then have to account for it already being called by the connection status JavaScript code, and modifications would have to be made to that file.

ICEfaces provides a built-in connection status component. However, it monitors the status of the network connection between the page and the server and not whether actual data are being pushed down to that particular page, and so was not suitable for LCS displays. What is interesting to notice however is that their component uses the client-side approach of displaying an animated GIF image, not the server-side approach. But the JavaScript code to change the GIF image in case the connection status changes is embedded within their Ajax bridge, and thus not accessible.

IV. New Direction

After this sample implementation was completed, the ICEfaces solution was abandoned. Scalability tests results which had nothing to do with the implementations described in this paper showed that the ICEfaces rendering model would not scale to the requirements for LCS. At a rendering interval lower than 750 milliseconds, rendering was just not consistent enough for the number of displays projected to be needed in the firing room.

Some performance problems have been evidenced during testing of the two projects presented here with 250 milliseconds rendering intervals, although the prototype implementations detailed in this paper were only tested using the local web server built-into NetBeans, and not a dedicated web server. But the fact that the implementations described here were several orders of magnitude simpler than the one used for the scalability tests seemed to indicate that there could be performance and thus scalability issues for a larger and more complex project.

The technology to use for LCS displays has not been chosen as of the writing of this paper.

V. Conclusion

Ajax Push technology enables asynchronous web-based push of presentation changes to the client browser. This capability can be harnessed to create a visual indicator of the connection status in two different ways. This paper explains how this was implemented in two separate Visual Web ICEfaces projects using NetBeans. Although it was possible to create a visual indicator without the need of client-side JavaScript, the visual appearance of the indicator was not as satisfactory as an animated image residing on the client-side. However, using an animated image on the client-side requires more overhead, since it involves some additional JavaScript on the client, and also may introduce other problems or considerations.

Even though this technology presents many interesting features to overcome the programming challenges of web applications and the HTTP protocol, the underlying mechanism used on the web server, where every web page's DOM is cached and then compared, requires overhead that does not scale well for requirements such as those of NASA's Launch Control System. Even so, the mechanisms described in the two implementations might still prove useful, since the requirements for a visual connection status indicator have not changed for the LCS project.

Acknowledgments

The author would like to thank Caylyne LaPolla for her excellent mentoring. Lien Moore, Linda Crawford and Bob Rusk are gratefully acknowledged for their help and support, as is the rest of the Launch Control System team. Kennedy Space Center and the University Student Research Program are also acknowledged for making this internship possible.

References

[1]NASA's Constellation Project, URL:
http://www.nasa.gov/mission_pages/constellation/main/index.html

[2]ICEfaces, The ICEfaces open source project, URL:
http://www.icefaces.org/main/home/index.jsp

[3]Java Server Faces Specification, URL:
http://java.sun.com/javaee/javaserverfaces/reference/api/index.html

[4]Java Server Pages 2.0 Specification, URL:
http://jcp.org/aboutJava/communityprocess/final/jsr152/

[5]Java Platform, Enterprise Edition 5 Specification, URL:
http://jcp.org/aboutJava/communityprocess/final/jsr244/index.html

[6]W3C Recommendation, Extensible Markup Language (XML) 1.0 (Fourth Edition), URL:
http://www.w3.org/TR/2006/REC-xml-20060816

[7]The life cycle of a Java Server Faces Page, URL:
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSFIntro10.html#wp122219

[8]NetBeans, Open-source Integrated Development Environment and application platform, URL:
http://www.netbeans.org/community/releases/61/

[9]Glassfish, The Glassfish Open Source Java EE Application Server, Version v2ur2, The Glassfish Project, URL:
https://glassfish.dev.java.net

[10]W3C Recommendation, Cascading Style Sheets (CSS) Level 2 Revision 1, URL:
http://www.w3.org/TR/CSS21/