BRINGING CONTROL SYSTEM USER INTERFACES TO THE WEB*

Xihui Chen, Kay Kasemir, ORNL, Oak Ridge, TN 37831, U.S.A.

Abstract

With the evolution of web based technologies, especially HTML5 [1], it becomes possible to create web-based control system user interfaces (UI) that are cross-browser and cross-device compatible. This article describes two technologies that facilitate this goal. The first one is the WebOPI [2], which can seamlessly display CSS BOY [3] Operator Interfaces (OPI) in web browsers without modification to the original OPI file. The WebOPI leverages the powerful graphical editing capabilities of BOY and provides the convenience of re-using existing OPI files. On the other hand, it uses generic JavaScript and a generic communication mechanism between the web browser and web server. It is not optimized for a control system, which results in unnecessary network traffic and resource usage. Our second technology is the WebSocket-based Process Data Access (WebPDA) [4]. It is a protocol that provides efficient control system data communication using WebSocket [5], so that users can create web-based control system UIs using standard web page technologies such as HTML, CSS and JavaScript. WebPDA is control system independent, potentially supporting any type of control system.

INTRODUCTION

Nowadays, people can do many things in web browsers, such as live meetings, trading, gaming, watching movies, and more. The web browser is no longer a simple browser. It became a convenient platform for various applications. Web applications have many advantages over desktop applications: 1) Easy to access. All you need is a URL; 2) Easy to deploy and maintain; 3) Accessible from anywhere at any time. Web applications with desktop application characteristics are called Rich Internet Application (RIA) [6]. Several technologies have been invented for RIA, such as Flash, Java Applet and Silverlight, but all these technologies require separate plugin or client software installed on the user's device and even worse, they are not available on popular iOS devices such as the iPhone and iPad. Fortunately, HTML5 emerged in recent years as a standard that has been quickly adopted by all mainstream web browser vendors. HTML5 based web applications have maximum cross-browser and cross-device compatibilities.

HTML5 includes a set of new APIs such as a canvas element, WebSocket, Drag-and-Drop, WebGL, Web Worker, Web Storage, Audio, Video, and more. Among which, the canvas element and WebSocket are most important for control system UI applications. The canvas element allows for dynamic, scriptable rendering of 2D shapes and bitmap images. This makes it easy to dynamically draw control system UIs in a web browser. WebSocket provide full-duplex communication channels over a single TCP connection. Before WebSocket, HTTP strictly followed the request-response model. For each update, clients initiated a new connection. The server could not initiate an update and "push" it to the client. A number of workarounds have been used to circumvent this problem, such as polling and long polling. These required additional header data and increased latency due to the request-response model. Compared to plain HTTP, WebSocket is a naturally full-duplex, bi-directional, single-socket connection. Once the WebSocket connection is established, the server can send message to the client at any time and vice versa. This greatly reduces latency, saves bandwidth and CPU power. Besides, the WebSocket API is very easy to use because common functionality such as handshaking, framing, buffering and encoding are already defined in the specification and hence implemented by WebSocket API providers. These merits of WebSocket make them a perfect candidate as the communication protocol for real-time control system web applications.

WEBOPI

To bring control system UIs to the Web, the ideal way is to directly run existing desktop Operator Interfaces in web browsers without extra effort. This is exactly what WebOPI does. It seamlessly executes OPI files created by CSS BOY in web browsers, without any modifications (see Fig.1).

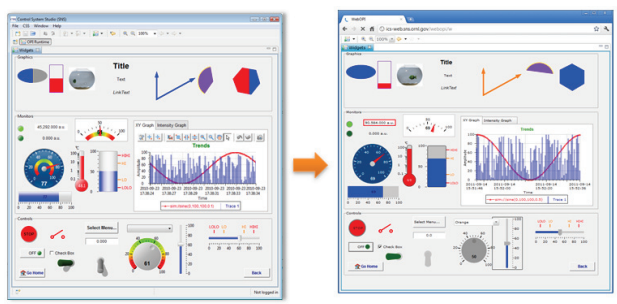

Figure 1: Comparison of same OPI running in CSS BOY and web browser.

CSS BOY is a modern graphical operator interface editor and runtime [7]. It allows users to build control system GUIs using drag and drop from over 50 widgets. It is further programmable via Jython or JavaScript. It provides extension points for extra data sources, custom

* SNS is managed by UT-Battelle, LLC, under contract DE-AC05-00OR22725 for the U.S. Department of Energy

ISBN 978-3-95450-139-7

widgets, and scripting libraries.

By reusing the OPI files created from CSS BOY, WebOPI can immediately inherit the powerful runtime functionality of CSS BOY and leverage the intuitive graphical editing capabilities of the BOY OPI Editor. Furthermore, WebOPI and CSS BOY are 90% single sourced using Eclipse Remote Application Platform (RAP) technology [8]. This allows continuous synchronized evolution of CSS BOY and the WebOPI, which means newly added features of CSS BOY are immediately available within WebOPI.

WEBOPI ARCHITECTURE

WebOPI is built on Eclipse RAP [8], which provides the capability of bringing Eclipse RCP to the web by reusing most of the existing RCP code. It achieves this by replacing the Standard Widget Toolkit (SWT) layer of RCP with the Remote Widget Toolkit (RWT) layer (see Fig.2).

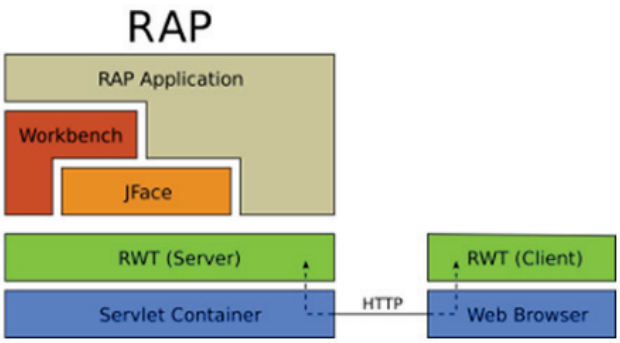

Figure 2: RAP architecture.

RWT code resides on both the server and the client side. Underneath, it uses HTTP as the communication protocol. On the server side, its Java code provides the same interfaces as SWT, so existing SWT application code can also run on RWT. On the client side, it utilizes the qooxdoo JavaScript library for native widgets, and HTML5 canvas elements for custom drawing. The client side code is responsible for the representation and event detection, while the server side code is responsible for processing the application logic. For example, when the user clicks a button in their web browser, the client code sends the click event to the server. Then the server side will process the event and reply back to the client with a result. If there are updates that the server should "push" to the client, it uses the HTTP long polling mechanism as mentioned before.

While RAP provides a convenient single-source programming model, implementers need to be aware of key differences between desktop and web applications. For example, each desktop application has a single user, while web applications allow multiple concurrent users. This requires the server to manage one UI thread life cycle per user. The server needs to verify if the user is still online, and properly dispose related resources once the user closes the web session. This is achieved by regularly checking for the long polling signal from each client. This and other small differences between SWT and RWT are handled via suitable Eclipse fragment or extension mechanisms, while the bulk of the BOY code is shared between the RCP and the RAP implementation.

WEBOPI LIMITATIONS

On one hand, the RAP single sourcing programming model provides tremendous benefits: The WebOPI can reuse existing BOY OPI files. On the other hand, there are limitations.

First of all, most of the OPI logic is executed on the server. While this reduces the client CPU load – an important consideration for small, mobile devices, including cell phones - it limits the maximum number of clients for each server. This issue is negligible for specialized control system web applications, for example related to a specific subsystem, where the number of concurrent users is small, and the advantage of easily creating a common BOY display for both local and web use by far more important.

On the other hand, the WebOPI is less suited for control system displays with a broad, site-wide audience, for example an accelerator status overview inspected by most everybody each morning.

Secondly, the RWT network traffic is not optimized for control system data. For example, on each update of a gauge widget, the server needs to send all drawing information to the web browser, instead of only the value that needs to be displayed in the gauge. We already mentioned that the long polling mechanism requires additional header data. HTTP compression can be enabled to reduce the network traffic about tenfold, albeit at the same time increasing the CPU load on both server and client.

While the WebOPI is responsive enough on desktop web browsers, the combination of these disadvantages mean that only comparably simple displays are practical on mobile devices. Higher performance control system web UIs require a more efficient protocol, optimized for control system data, which is the motivation for developing WebPDA.

WEBPDA

WebPDA is a protocol for efficient control system data communication based on WebSocket. As explained in the introduction, WebSocket has many advantages over HTTP for real-time web applications. However, WebSocket is a general protocol for transferring text or binary bytes. It is not easy to directly use it for control system web applications. WebPDA is an application level protocol and API that allows users to build control system web applications without dealing with communication details. The protocol defines and handles the communication sequence, message encoding and decoding, buffering, security, and client life cycle management. It further provides an abstract data layer on the server side so that users can extend it to arbitrary control systems.

ISBN 978-3-95450-139-7

The Protocol

In WebPDA, data is transferred as values of Process Variables (PV), using PV as defined in the EPICS [9] toolkit. The value type of a PV can be an arbitrary data structure.

The WebSocket communication between server and client is straightforward (see Fig. 3). Firstly, the client sends a regular HTTP request to the server. If the server allows, the HTTP connection is upgraded to a WebSocket connection. After the connection is established, the client sends a login command with user name and password to the server. On success, the server will mark the client as logged in. Otherwise, it will forbid further commands from this client. The client can send a "Create PV" command to the server. The server will create the PV and try to connect to that PV in the underlying control system, i.e. EPICS. Once the PV is connected, it will notify the client that the PV is connected, and from now on send value changes to the client. The client can send a "Close PV" command to server when the PV is no longer of interest. If the client connection is unexpectedly lost, the server will detect this and dispose related resources.

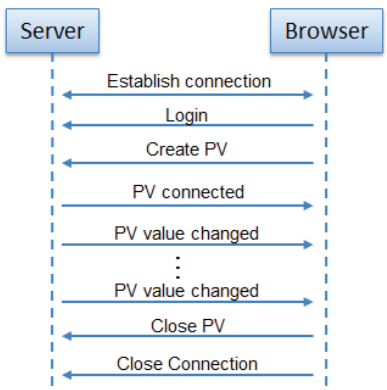

Figure 3: Typical communication sequence of WebPDA.

Since both server and client maintain the status and value of each PV, it is not necessary to transfer the whole data structure for each value update. Instead, the protocol only transfers the changed fields of the data structure. For example, the PV metadata such as units, precision, display limits, alarm limits is only transferred when it changes. Most PV updates are thus limited to network transfers of the latest value, timestamp and alarm status.

Client commands and messages from the server to the client are generally transferred as JSON [10] text, because JSON is very flexible and easily parsed in client-side JavaScript. Value updates, however, are transferred in a binary format, because a binary format is most compact and can also preserve the precision of floating point numbers. Overall, this design provides us with maximum efficiency and also flexibility.

Server Side Implementation

The WebPDA protocol does not limit implementation techniques for either server or client. In principle, any languages that support WebSocket can all be used to implement servers or clients.

Currently, we provide a server side implementation based on JSR356 [11]. JSR356 is a standard WebSocket Java API. It is currently supported by Glassfish 4 and Tomcat 8. The WebPDA core implementation is actually layered to remain independent from a specific WebSocket API, fundamentally allowing an alternate server side implementation, for example for Jetty.

The server side library is decoupled into an abstract data interface layer and a specific implementation layer, so the data interface is independent from its implementations. This allows extending WebPDA to arbitrary data sources. Currently, we provide implementation for the PVManager [12], which already has support for EPICS V3, V4, simulated PVs, local PVs and formulas. PVManager also allows extension to arbitrary control systems. User can create a new data source either on top of the abstract WebPDA data interface layer or on top of PVManager. The benefit of creating new data source on top of PVManager is that it already implemented a set of value types, queuing, throttling, encoding and corresponding decoding code on the client side.

Client Side Implementation

While the WebPDA client side can be implemented in any WebSocket-aware language, we chose JavaScript as it is currently predominant in web browsers. Corresponding to the server design, the client side also has two layers: an abstract layer that handles common communications, and a specific implementation layer that decodes the data corresponding to the server side implementation layer. If new data sources added to the server side are based on the PVManager, no additional work is needed for the client side.

The client side API hides protocol details from users, allowing users to focus on the PVs when writing web applications (see Fig 4).

```
var url = "ws://localhost:8080/webpda";
var wp = new WebPDA(url, "myname", "password");
var pv = wp.createPV("pvname", 1000, false);
pv.addCallback(function(evt, pv, data) {
    switch (evt) {
    case "conn": //connection state changed
        break;
    case "val": //value changed
        break;
    case "bufVal": //buffered values changed.
        break;
    case "error": //error occurred
        break;
    case "writePermission":
        //write permission changed.
        break;
    case "writeFinished": //writing finished.
        break;
    default:
        break;
    }
});
```

Figure 4: WebPDA client side JavaScript API.

ISBN 978-3-95450-139-7

WebPDA Widgets

To simplify the process of building web browser control system UIs, WebPDA also pre-wrapped some widgets that allow users to display the value of a PV inside a widget via a single line of code (See Fig 5). In a general HTML " " element, users only need to specify the element class as "webpda-widgets", select the widget type, for example a gauge, and specify the desired PV name. The widget will automatically connect to the PV and display its value in real-time. Based on the widget type, PV metadata such as display limits will determine the widget's range; the alarm status may affect the widget colors, and so on. Users can also wrap their own widgets as WebPDA widgets in a separate JavaScript library.

```
<div class="webpda-widgets"
data-widget="rgraph-gauge"
data-pvname="sim://noise"></div>
```

Figure 5: Pre-wrapped WebPDA widget demo.

SECURITY

Internet web applications are potentially exposed to anybody, anywhere in the world. Consequently, there may be a need to control access to WebPDA data. For authentication, both the WebOPI and the current WebPDA implementation support the Java Authentication and Authorization Service (JAAS), allowing integration with existing site-wide authentication infrastructures such as LDAP. For simpler, standalone installations, system administrators can use a server-side text file to configure users and their passwords.

The handling of authorization differs between the WebOPI and WebPDA. For the WebOPI, the server executes the application logic. PV read/write permissions are controlled by the underlying control system, such as EPICS Channel Access Security, regardless of the web-based user. While WebPDA can similarly rely on the security mechanism of the underlying control system, it allows additional configuration for each web-based user, either from a server-side file or LDAP.

To protect transferred data from man-in-the-middle attacks, TLS [13] can be used to encrypt the communication for both HTTP and WebSocket. Encrypted HTTP URLs start with https:// and encrypted WebSocket URLs start with wss://.

COMPATIBILITY

Given the plethora of mobile devices, operating systems and web browsers available on the market, it is important for control system web applications to be compatible with the major of devices and browsers. Fortunately, HTML5 as a popular standard has been quickly adopted by all mainstream browser vendors. As we write this article, both WebOPI and WebPDA are compatible with the latest versions of mainstream web browsers. Only the default Browser on Android devices may exhibit incompatibilities, but they are resolved by installing a separate Chrome, Firefox or Opera browser on the device.

SUMMARY

This article introduced two technologies that facilitate the goal of bringing control system UIs to the web. They have different characteristics, tailored for different use cases. The WebOPI makes it extremely easy to build rich control system web UIs, but its efficiency limits the number of simultaneous users. WebPDA provides maximum efficiency, but requires certain HTML and JavaScript programming skills to implement the UI. A future tool that generates WebPDA UIs using drag and drop as in the CSS BOY display editor would combine the best of both approaches.

To the end user, either technology provides access to control system data via a simple web URL on almost any web-connected device.

REFERENCES

[1] http://en.wikipedia.org/wiki/HTML5
[2] http://sourceforge.net/apps/trac/cs-studio/wiki/webopi
[3] https://sourceforge.net/apps/trac/cs-studio/wiki/BOY
[4] http://webpda.org/
[5] http://en.wikipedia.org/wiki/WebSocket
[6] http://wikipedia.org/wiki/Rich_Internet_application
[7] X. Chen, K. Kasemir, "BOY, a modern graphical operator interface editor and runtime". Proceedings of 2011 Particle Accelerator Conference, New York, NY, USA.
[8] http://eclipse.org/rap/
[9] http://www.aps.anl.gov/epics/
[10] http://www.json.org/
[11] http://jcp.org/en/jsr/detail?id=356
[12] http://pvmanager.sourceforge.net/
[13] http://wikipedia.org/wiki/Transport_Layer_Security

ISBN 978-3-95450-139-7