Another Look at Web-Enabled Instrument Interfaces

W.T.S. Deich, S.L. Allen, A. Misch

Web-enabled user interfaces for the control and monitoring of instruments and telescopes have a checkered history. The remarkable interactive speed and quality of Google Maps and Google Suggests have led us to take another look at implementing services over the Web.

These applications rely on the so-called AJAX mechanism, which enables lightweight, efficient, and responsive interfaces in nearly any modern browser.* AJAX provides a simple, standards-based means for web browsers to make asynchronous calls back to a server, and handle responses (typically but not necessarily XML) in a callback.

Here we describe webktl, a particular Web-enabled interface for KTL services. However, the next version of *webktl* will separate the KTL data sources from the core *webktl* server, and this will allow any data that can be modelled as keyword/value to be served by the *webktl* server. For example, we will be connecting interfaces to database tables as well as KTL services using *webktl*.

---

* Internet Explorer 5 and 6 do *not* count as modern browsers. And even some recent versions of Safari have bad problems with valid CSS. We find that Mozilla and Firefox browsers have been the most reliable for us.

The Virtues of AJAX

- **Low bandwidth.** Interfaces based on AJAX protocols generally have lower bandwidth requirements and greater tolerance of high-latency links than, say, remote X display servers. Thus, they can be useful for remote users.
- **Window organization.** Web-based interfaces organize nicely into separate tabbed panes in a modern browser. Very useful for monitoring health and status of several telescopes and instruments.
- **Web-based**: and everyone's a web developer! Many more people can safely experiment with and implement instrument interfaces using simple markup languages (HTML) than using programming languages such as C, Java, or Tcl. This may encourage staff to create interfaces for themselves, as they need them.
- **Clean demarcation** between content and presentation. This helps with faster, less buggy development because one can better focus on a single aspect at a time.
- **Web-based, not OS-based**. Our other interfaces generally require login to a Linux or Unix host. A web interface has no such requirements, which is attractive to the machinists who build instruments at UCO/Lick
- **Web-based – no special software required.** As long as the user has a modern browser, there's no need to install special software or use a particular client host.

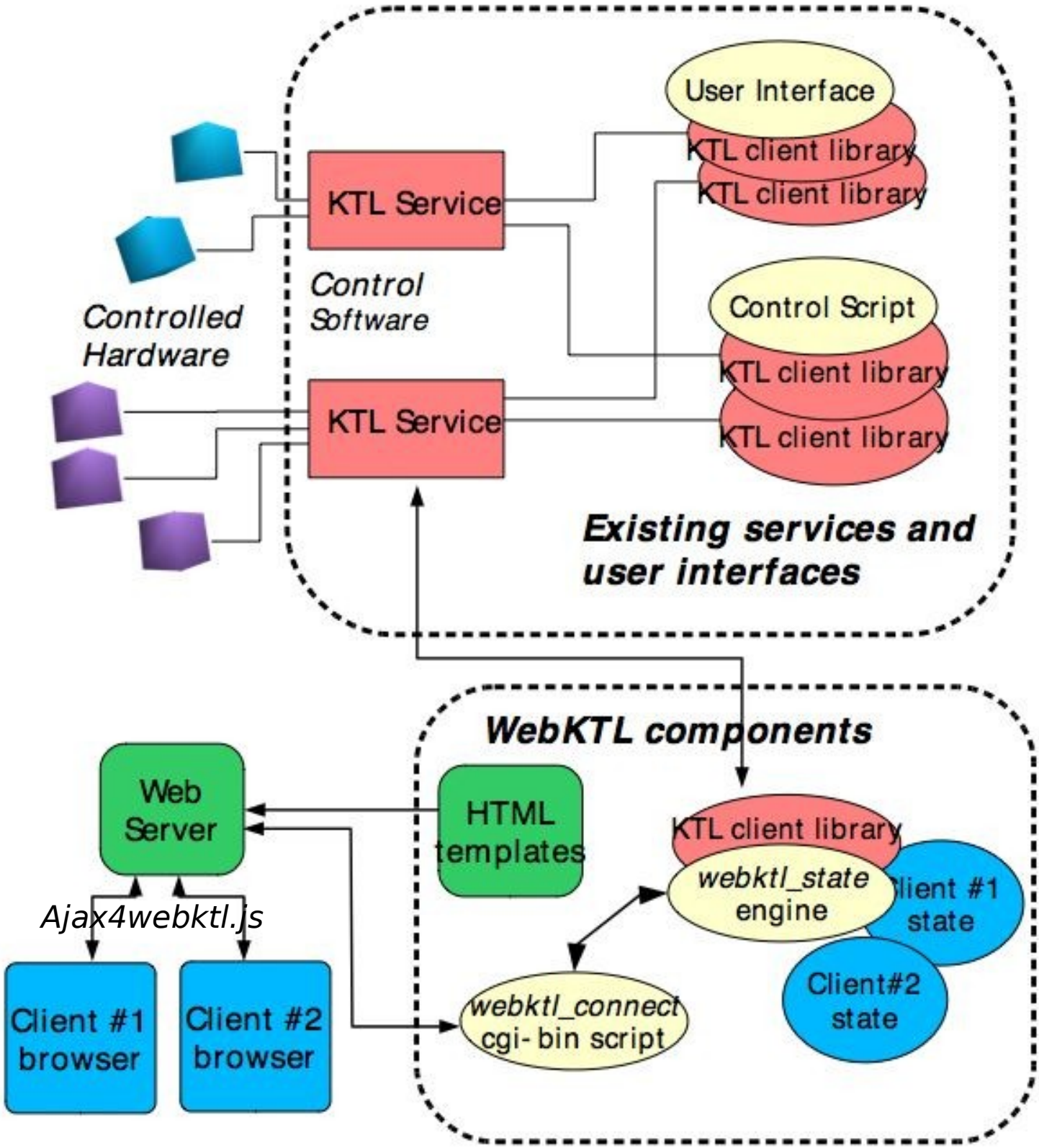
The Drawbacks of AJAX

- **Early Obsolescence?** Will AJAX quickly become obsolete? Many Web technologies have sunk without a trace. We are taking a bet that AJAX will be around for a longer while...
- **Browser-based**: it is difficult to make Web interfaces that please the end user as much as native GUI applications.
- **Web-based:** image display tools for browsers do not offer many of the capabilities of a traditional display package such as *ds9*.
- **Web-based**: few browsers implement the W3C standards in full.

WebKTL and the World of KTL Services, I

- At Keck and Lick Observatories, most controlled hardware (telescopes, instruments, etc) is represented using KTL (Keck Task Layer) services.
- KTL services represent all hardware components through sets of keyword/value (or attribute/value) pairs.
- Clients gain access to KTL services through KTL client libraries, which all follow the standard KTL API to provide uniform access to each service's keywords.
- WebKTL is an AJAX-based interface to KTL services, containing:
- HTML + CSS (Cascading Style Sheets) pages that provide the GUI layout. They use a Javascript library to fetch and refresh the keyword data displayed in the page, but many interfaces need no special Javascript of their own – it's all handled through the HTML markup.
- A Javascript library, *ajax4webktl*, that supplies all the code necessary to make the AJAXian calls and handle the responses.
- A cgi-bin script, *webktl_connect*, is invoked to handle each client refresh request or keyword modify request.
- A persistent stage engine, *webktl_state*, connects to KTL services as needed, on behalf of all web clients, and stores the state information needed to update each client.

WebKTL and the World of KTL Services, II



Fast and Easy Interface Development?

Although *webktl* is very new, and our experience with it is commensurately short, it does seem to offer very easy, very fast interface development.

- The keys to fast development are probably:

- **Small files.** So far, our AJAXian GUI's are perhaps 5-10x smaller than equivalent GUI's written in a typical language such as Tcl/Tk. Smaller files take less time to write.

- **Markup language rather than programming language**. Markup language is simpler than programming languages, and takes less time to produce.

We are hopeful that *webktl* will broaden the pool of interface developers, by empowering non-programmers to create interfaces as they see fit. For this to succeed, we must

- **keep all the logic in the *webktl* server and the *ajax4webktl* library**, so that client-side interfaces consist almost entirely of markup language, not programming language.

Data Rates

The data rates associated with *webktl* applications are small. For example, the four KTL services that make up the DEIMOS spectrograph, the largest instrument yet built by UCO/Lick Observatory, together comprise about 1280 keywords. Although the initial load by a client takes ~100KB, successive refreshes take <10KB. Thus, a *webktl* session can be readily used from a home computer over a DSL line.

Graceful Degradation – Just Say No

Good web design usually calls for "graceful degradation" – designing each web page so that it works as well as possible with whatever features are available in any browser, rather than being dependent on some particular features of newer browsers.

We deliberately ignored such considerations. Our target audience is small and it's reasonable to require them to use a modern standards-compliant browser when using a *webktl* application. Nonetheless, it is good practice to detect browsers that don't support the required features, or are known to handle them badly, and warn the user. We have not yet outfitted the *ajax2webktl* library with these features, but will do so in a future version.

Security

Security is a deep concern for web-enabled applications – one moment, telescope control is safely locked up behind a firewall, and the next moment a web application could expose telescope control to the world. The WebKTL suite of programs uses these security features:

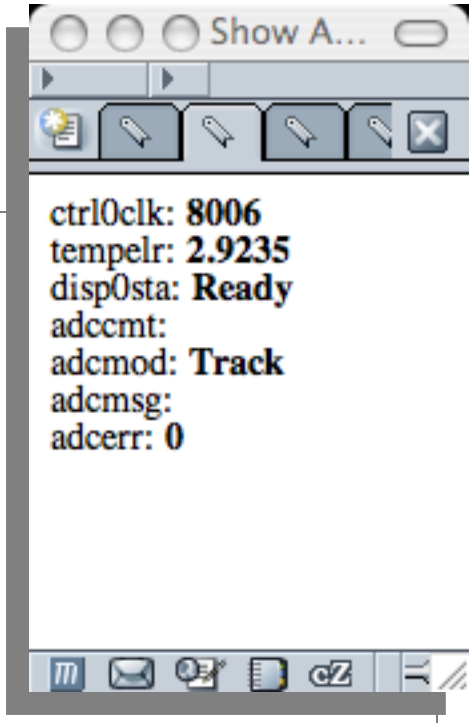
- Requires HTTPS connections to make client/browser connections harder to hijack.
- WebKTL pages should be configured to require webserver-enforced passwords (but this is outside *webktl*'s ability to enforce).
- The *webktl* server configuration files specify which hosts and/or subnets can monitor and/or modify which services and keywords.
- All communications are stringently checked for valid syntax, and for reasonable service names and keyword names before handing them off to the KTL layer for handling.

In sum, although it is possible to construct a denial of service attack against the *webktl* server, it is much more difficult to gain unauthorized access to a service.

Nonetheless, it is prudent to put critical services (e.g. telescope and dome control) on a web server that is not visible to the outside world.

Example 1. A Few Live Keywords

The first *webktl* example shows a simple example of a handful of service keywords that are made to update at about 1 Hz. The web page is shown at right, and the corresponding HTML + Javascript is given below.

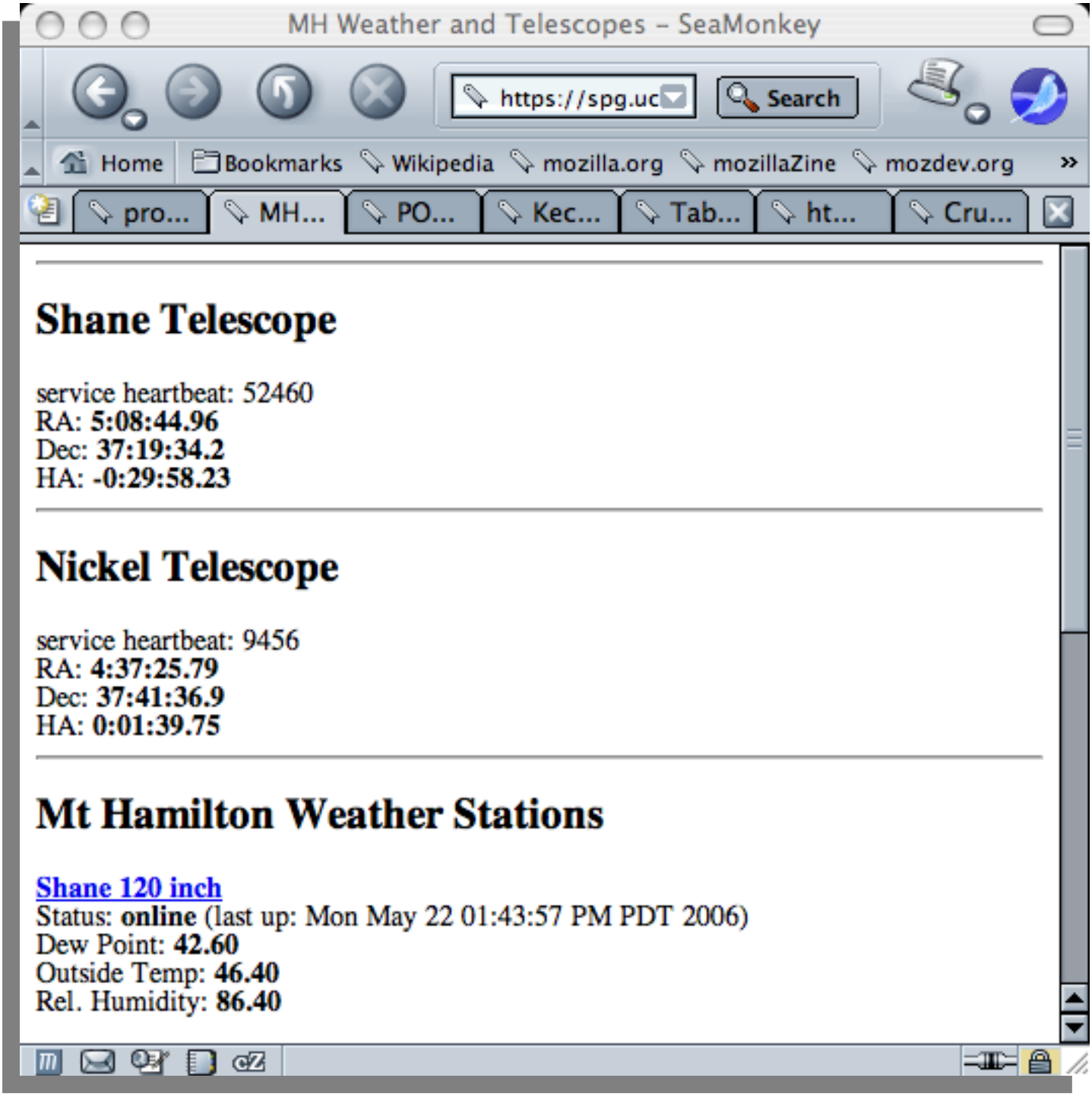



```
<html>
<head>
<script src = "ajax4webktl_v1.0.js
     type = "text/javascript"></script>

<script type="text/javascript"> <!--
    refreshServiceData('k1adc', 900);
// --> </script>

<title>Show ADC</title>
</head>

<body>
<div> ctrl0clk: <b id=”k1adc_ctrl0clk”> </b> </div>
<div> tempelr:  <b id="k1adc_tempelr" > </b> </div>
<div> disp0sta: <b id="k1adc_disp0sta"> </b> </div>
<div> adccmt:   <b id="k1adc_adccmt"  > </b> </div>
<div> adcmod:   <b id="k1adc_adcmod"  > </b> </div>

<div> adcmsg:   <b id="k1adc_adcmsg"  > </b> </div>
<div> adcerr:   <b id="k1adc_adcerr"  > </b> </div>
</body>
```

The three sections with a yellow background do the work of making this a live *webktl* page:

1. First, the required *ajax4webktl* javascript file is loaded.
2. A "refresh" of the *k1adc* service is scheduled to repeat 900 ms after each previous refresh finishes.
3. Each live-update element is identified for the *ajax4webktl* Javascript by giving it a unique id named with the magic value <***service***>_<***keyword***>, so that the *ajax4webktl* javascript knows what to insert in the companion text node.

Example 2. Using Multiple Services; Putting Live Data into Links



This page shows three data services monitored on a single page. *Webktl* is quite efficient: the visible part of this web page takes about 60 lines of HTML and javascript. (The part that is scrolled off the bottom takes about 40 more lines. See page at right for more details.)

```
<script type="text/javascript"> <!--
    refreshServiceData('met', 900);
    refreshServiceData('nickelpoco', 900);
    refreshServiceData('shanepoco', 900);

function userRefreshHandler(svc, kwd, obj) {
    if (svc == "met" && ( kwd == "m1daturl" ||
             kwd == "m2daturl" || kwd == "m3daturl" ||
             kwd == "m4daturl" || kwd == "m5daturl") ) {
        // These keywords' values are url's, so put them
        // into the href attribute, instead of the normal
        // text node replacement.
        setAttributeOfId(svc, kwd, "href", obj);
        return 0;   // don't apply normal handling to this
    } else {
        return 1;   // apply normal handling for this
    }
}
// --> </script>
...
<h2> Mt Hamilton Weather Stations </h2>
<p>
<div> <a href="?" id="met_m1daturl">
    <b id="met_m1locat"> -?- </b> </a> </div>
<div> Status:  <b id="met_m1status"> -?-  </b>
...
```

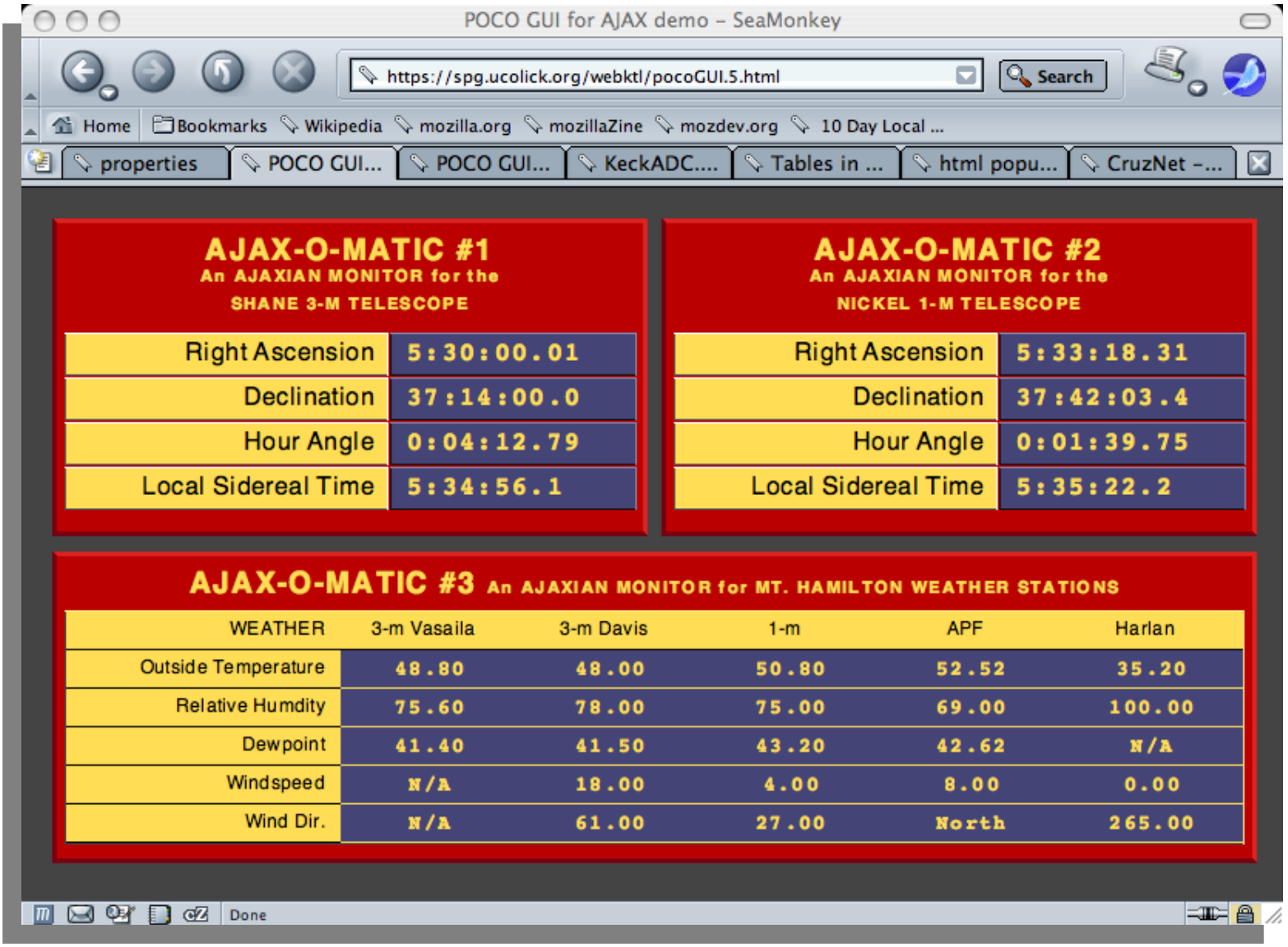
This shows the key HTML + Javascript fragments to show how *webklt* readily handles multiple services.

1. It invokes *refreshServiceData()* for each service.
2. It also shows a user-supplied callback, *userRefreshHandler()*, which intercepts updates of the keywords that define the URI's for the weather data, and puts them into an href, using a utility function *setAttributeOfId()*.
3. Finally, it shows the markup text that *webktl* uses to find the correct elements to update.

Example 3. Adding CSS to a WebKTL Page

An uninspiring page like the multi-service example above can be made much more attractive by simply adding CSS (cascading style sheets) and making some minor changes to the HTML layout. This page show status data for the Shane and Nickel telescopes at Lick Observatory, plus five mountaintop weather stations, all updating at about 1Hz.

(See page at right for CSS information.)



```
body {
  margin: 0px;
  padding: 0px;
}

#container {
  margin: 0px auto;
  width: 785px;
}

#canvas1 {
  position: absolute;
  top: 20px;
  left: 20px;
  background: #b00;
  border: 3px solid;
  border-color: #d22 ...;
  width: 380px;
  height: 200px;
  text-align: center;
}

#canvas2 {
  position: absolute;
  top: 20px;
  left: 415px;
  background: #b00;
  border: 3px solid;
  border-color: #d22 ...;
  width: 380px;
  height: 200px;
  text-align: center;
}
...
```

```
...
#teleLabels {
  position: absolute;
  top: 70px;
  left: 5px;
  width: 210px;
  text-align: right;
}

#teleData {
  position: absolute;
  top: 70px;
  left: 216px;
  width: 160px;
  text-align: left;
}

table {
  position: absolute;
  top: 35px;
  left: 5px;
  border: 1px solid;
  border-color: #fe8 #fe8 #444
#444;
  width: 765px;
}
...
```

There's not a lot to be said about this particular style sheet; it isn't affected by the webktl aspects at all. The complete style sheet file for this display is about 155 lines.

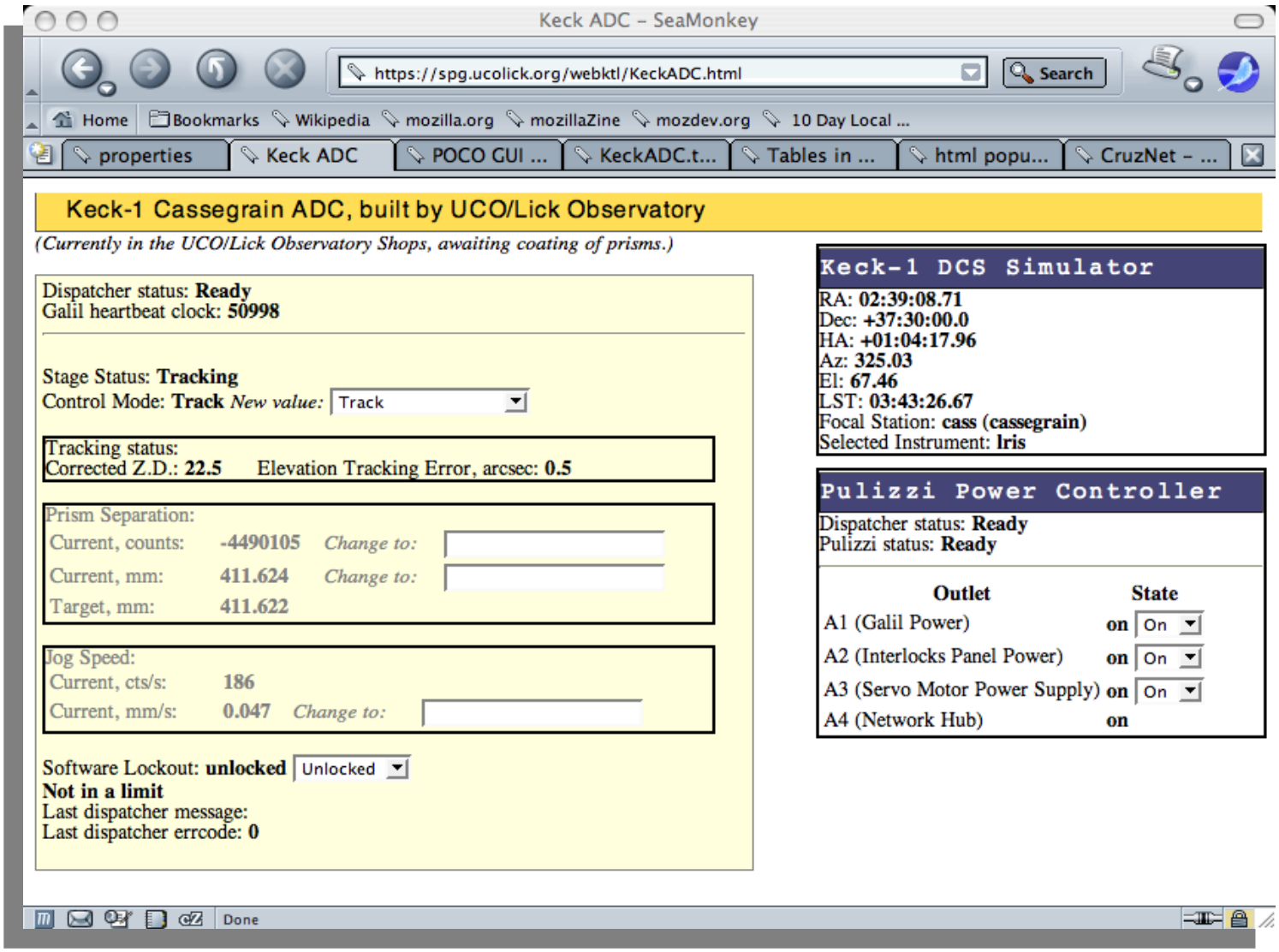
Taken together, the *webktl* back end, CSS, and HTML+AJAX do a very nice job of separating content and presentation into clean layers. With these tools in hand, an expert web designer or a casual web experimenter can create interfaces as needed, without requiring much, if any, assistance from a software developer.

Example 4. Beyond Monitoring: Instrument Control with WebKTL

This example shows a control interface for the new Keck ADC, with sub-panels that display telescope pointing data from the telescope control keyword service and the user-controllable power supply.

This interface contains a few hundred lines of custom Javascript, which is used to enable or disable various control panels depending on the ADC's control mode. As we gain experience with these controls, we'll generalize them, move them into *ajax4webktl*, and trigger them with special markup in the HTML: the goal is to keep interfaces driven by markup language, and not require special Javascript skills.

(The page to the right shows how *webktl* triggers keyword writes.)



```
<div class="Pos">
    Prism Separation:
    <br>

    <table class="Pos" style="border:0">
    <tr>
    <td>Current, counts:</td>
    <td><b id="k1adc_adcraw"> -?- </b></td>

    <td><i>Change to:</i></td>
    <td><input type="text" name="k1adc_adcraw"
            onchange="triggerModify(this)"></td>
    </tr>
    <tr>
    <td>Current, mm:</td>
    <td><b id="k1adc_adcval"> -?- </b></td>
    <td><i>Change to:</i></td>

    <td><input type="text" name="k1adc_adcval"
            onchange="triggerModify(this)"></td>
    </tr>
    <tr>
    <td id="targetpos">Target, mm:</td>
    <td><b id="k1adc_adctva"> -?- </b></span></td>
    </tr>
    </table>

</div>
```

This HTML fragment generates the box labelled "**Prism Separation**" in the Web page.

- As is done in the other examples, this uses **id=<svc>_<kwd>** to mark the elements whose text should be dynamically updated.
- Keyword writes are done by invoking the *ajax4webktl* function **triggerModify(this)** when the input changes; and
- the **name=<svc>_<kwd>** field is used by the *triggerModify* function to identify the keyword to change.

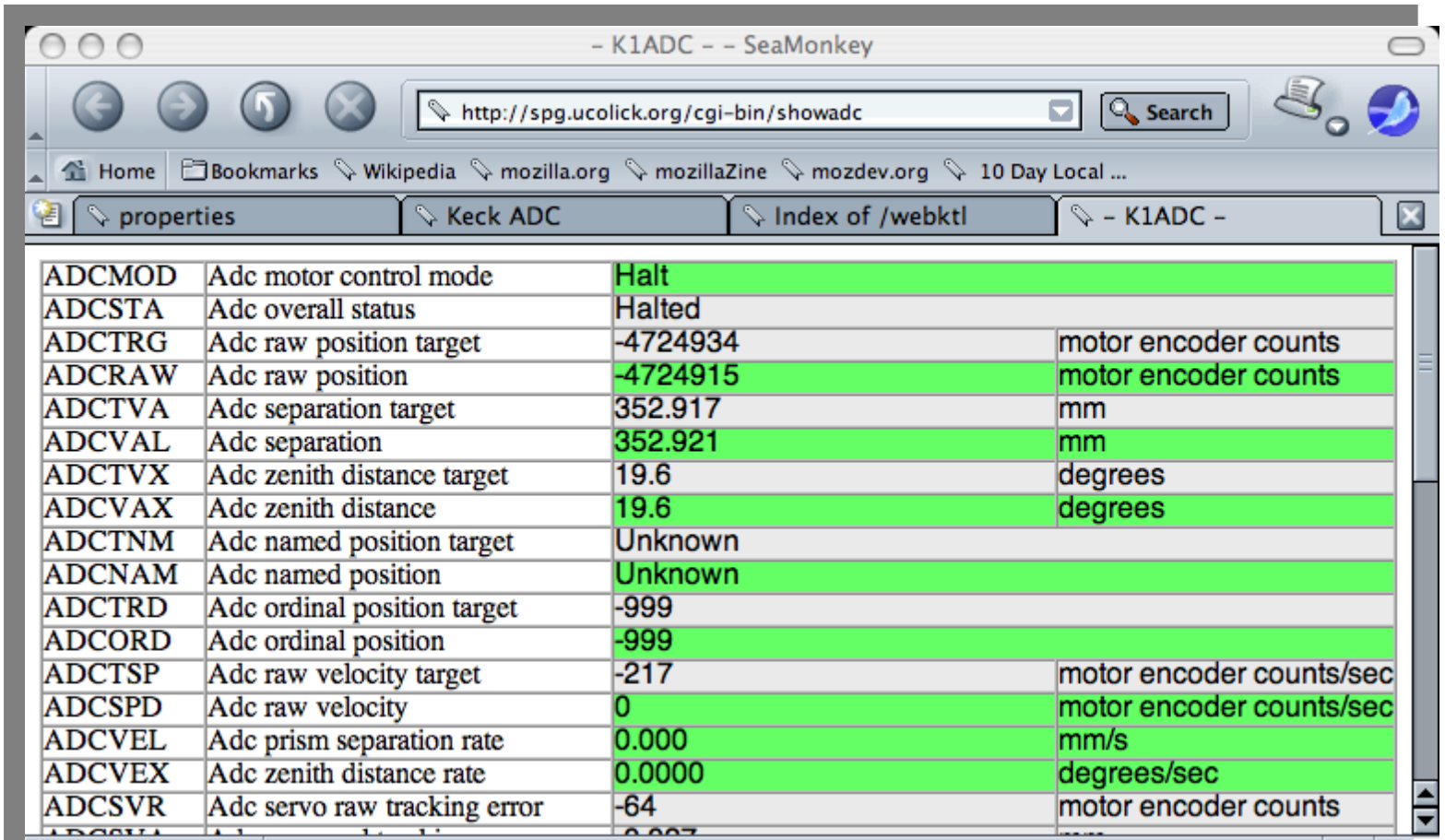
Lines of HTML + Javascript: 326. Lines of CSS: 90.

Example 5. An Autogenerated Interface

Our last example shows a prototype of an auto-generated interface. It's not pretty, but it is very simple and quick to set up. All the work is done by an *ajax4webktl* function, which queries the *webktl* server for a list of keywords and their properties, and generates an interface on the fly. Green fields represent writable keywords (click on the field and enter a new value); others are readonly.

The entire HTML + Javascript file for this web page is:

```
<html>
<head>
<script src = "ajax4webktl_v1.0.js"
        type = "text/javascript"></script>

<script type="text/javascript"> <!--
    getPropertyData('k1adc', 900); // -->
</script>

<title>- K1ADC -</title>
</head>

<body> </body>
```



ADCMOD	Adc motor control mode	Halt	
ADCSTA	Adc overall status	Halted	
ADCTRG	Adc raw position target	-4724934	motor encoder counts
ADCRAW	Adc raw position	-4724915	motor encoder counts
ADCTVA	Adc separation target	352.917	mm
ADCVAL	Adc separation	352.921	mm
ADCTVX	Adc zenith distance target	19.6	degrees
ADCVAX	Adc zenith distance	19.6	degrees
ADCTNM	Adc named position target	Unknown	
ADCNAM	Adc named position	Unknown	
ADCTRD	Adc ordinal position target	-999	
ADCORD	Adc ordinal position	-999	
ADCTSP	Adc raw velocity target	-217	motor encoder counts/sec
ADCSPD	Adc raw velocity	0	motor encoder counts/sec
ADCVEL	Adc prism separation rate	0.000	mm/s
ADCVEX	Adc zenith distance rate	0.0000	degrees/sec
ADCSVR	Adc servo raw tracking error	-64	motor encoder counts

Where Do We Go From Here?

1. **Decouple *webktl* from KTL services.** The *webktl* server is a powerful tool for serving any kind of data that can be modelled as {service+keyword}/value pairs. The next version of *webktl* will define an API for general data sources; KTL services will be just one such source. New data sources become simple plug-ins for *webktl*.

2. **Extend the *ajax4webktl* Javascript library.** The key to fast and easy interface development by a wide range of developers lies in keeping the HTML files simple. Whenever significant custom Javascript is required, it's a hint that the custom code should be moved into functions in the *ajax4webktl* library. This must be coupled with new markup elements that will trigger the custom code as needed.

3. **Increase efficiency.** On the server side, we can eliminate the *webktl_connect* process entirely; on the client side, we will try to implement versions of the *ajax4webktl* routines that require less client-side overhead.

4. **Detect lousy browsers** that don't support the requisite AJAXian features, or are known to handle them badly, and warn the user.

Web Links

All the example interfaces, plus the *webktl* source code, can be found at

`http://spg.ucolick.org/webktl/`