Another look at web-enabled instrument monitoring and control

W. T. S. Deich and Steven L. Allen,

UCO/Lick Observatory, University of California, Santa Cruz, CA 95064 USA

ABSTRACT

Web-enabled user interfaces for the control and monitoring of instruments and telescopes have a checkered history. However, the remarkable interactive speed and quality of Google Maps and Google Suggests have led us, like others, to take another look at implementing services over the Web. The so-called AJAX mechanism enables simple, lightweight, efficient, and responsive interfaces in nearly any modern Web browser. We discuss methods, security, and other implementation issues for sample interfaces that include telescope monitoring, instrument control, and weather station information.

Keywords: web services, AJAX, GUI, instrument control

1. INTRODUCTION

Since the Web first came to prominence in the early 1990s, observatory and instrument software developers have striven to take advantage of its ability to connect remote users easily with telescope and instrument monitor and control software. Early efforts at web-based user interfaces were plagued by cross-browser incompatibilities, limited capabilities of the available graphical toolkits, and uneven support for secure connections.

Early versions of Java (http://java.sun.com) were promoted in the mid-1990s for cross-platform interface development using AWT (Abstract Windowing Toolkit¹), but suffered from a primitive widget library and an inconsistent look and feel across operating systems. Furthermore, a great deal of UCO/Lick Observatory instrument-control software is Linux-based, but throughout the 1990s Java implementations on Linux systems were neither ubiquitous nor predictably functional.

Cross-browser incompatibilities were manifold. Microsoft provided asynchronous remote scripting as early as 1995, but it would only run on Windows. Software that only runs on Windows is of little interest in our Unix/Linux-oriented environment.

The introduction of Google Maps (http://maps.google.com) in 2005 impressed many Web users with its very high quality graphical user interface (GUI), which gave the lie to the assertion that Web interfaces were doomed to be second-class citizens in the world of GUIs. This sparked our interest in experimenting with the same technologies that underlie Google Maps, for use in observatory and intrument user interfaces. The so-called AJAX (Asynchronous Javascript And XML) combination of Web technologies used to implement Google Maps was not invented by Google, but the brilliance of the Google interface provided the spark that brought it to the attention of developers everywhere.

At UCO/Lick Observatory, we already have a large investment of time in our existing GUI tools. Most notably, the Tcl/Tk-based *dashboard* toolkit² has been successfully used for a series of instrument interfaces, including DEIMOS, ESI, and the HIRES exposure meter at Keck Observatory; PFCAM at Lick Observatory; and the engineering interface for the soon-to-ship Keck I atmospheric dispersion compensator. Nonetheless, we are interested in Web-based tools for a variety of reasons, including:

W.D.: Email: will@ucolick.org; Telephone: 831-459-3913; Fax: 831-459-2298;

Further author information- (Send correspondence to W.D.)

S.A: Email: sla@ucolick.org; Telephone: 831-459-3046; Fax: 831-459-2298

- Network efficiency. Interfaces based on AJAX protocols generally have lower bandwidth requirements and greater tolerance of high-latency network links than remote X display servers which often stutter and freeze. A web-based interface that is otherwise less optimal than a *dashboard* interface, may be very attractive to a remote user working over a low-bandwidth, high-latency connection.
- Window organization. Web-based interfaces naturally organize nicely into separate tabbed panes in a modern browser. This is very useful to observatory staff who may need to monitor the health and status of several telescopes and instruments. By contrast, organizing a crowded desktop displaying a collection of separate standalone applications is an often-irritating task.
- Operating system and host independence. To use a *dashboard* interface, the user must use a Unix/Linux host on which all the requisite software is installed. A web interface has no such requirements. This can be very attractive for our machinists who build the instrument hardware, but use only Windows computers.
- Web convenience. Making resources available over the web may begin as a modest convenience to users, but over time may become useful in wholly new ways. Creating web-based access to telescopes and instruments is a way to develop those conveniences and discover new capabilities.

The remaining sections of this paper give a brief description of AJAX web interfaces; compare AJAX interfaces with other remote access interfaces for observatory and instrument use; consider AJAX interfaces for local (observatory intranet) use; and describe our experimental WebKTL system for providing AJAX-based interfaces to Keck and Lick Observatory telescopes and instruments.

2. AJAX INTERFACES

During the mid-1990s, Dynamic HTML (DHTML) evolved to provide a means of creating interactive web pages. DHTML extends the foundation of static HTML pages by adding the DOM (Document Object Model, a languageneutral API that describes a web page in a uniform manner, allowing its elements and fields to be readily accessed by a programming language) and a client-side scripting language such as Javascript. Javascript can trigger on any of the usual GUI-related events such as mouse motion or key clicks, so it is possible to create client-side interfaces with good interactivity.

Although a Javascript-enabled web page could have good interactivity with the user, its interaction with the web server was relatively crippled:

- Clients would interact synchronously with the web server, so that the web page's interface would become briefly non-responsive while awaiting the response from the server.
- Typically, updates would not occur until the user explicitly pressed a button or key.
- The server would send back entire web pages. If the web page contained extensive amounts of information, it could take significant time to compose the page at the server side, transmit it to the browser, which would then have to fully parse and render the updated page.

The preferred means for overcoming these problems lies in the XMLHttpRequest mechanism, which allows code embedded in a web page to make asynchronous calls back to a web server, and use a callback to handle the response. Originally created by Microsoft as an ActiveX object in the late 1990s, it has since been re-implemented as a native Javascript object in most other browsers. Where appropriate, client-side events or timers can use XMLHttpRequest to trigger updates immediately, in the background, without explicit extra button-presses by the user. Inside the response-handling callback, Javascript can parse the response data and update the browser's copy of the page's DOM, which in turn triggers an update of the modified parts of the display.

Together, this combination of Asynchronous Javascript And XML was dubbed AJAX.* In its most common use, the XMLHttpRequest object specifies a callback handler, and then makes an asynchronous GET or POST request to the web server.³

^{*}Originally, the XMLHttpRequest was intended to retrieve data encapsulated in XML. Although XML is the most widely-used data-encapsulation method for XMLHttpRequest callbacks, in practice any sort of data encapsulation can be used, and the callback function is responsible for parsing it correctly and inserting data into the DOM.

In the past few years, nearly all modern browsers have implemented support for the DOM, Javascript, and the XMLHttpRequest Javascript object. Although the DOM is language-neutral, Javascript is of special interest, because it is available on nearly all modern browsers.

AJAX applications have all the security issues that any Web applications must overcome. They communicate via HTTP, and therefore, from a Web server's point of view, are no different from any other Web client. We discuss security implications in Section 5.6.

Programming directly with the XMLHttpRequest object is inconvenient, due to slight vendor-to-vendor variations in implementation. The Javascript code must make a variety of tests for implementation details in order to execute code that is valid in the user's browser. The best practical solution for this is to use one of many freely-available AJAX libraries that wrap all the ugly details in convenient APIs.³

Not every browser supports Javascript, and the user may deliberately disable it on other browsers. That will cause AJAX-based interfaces to fail. Most well-designed business applications will attempt to degrade gracefully to interfaces that are less attractive but will work within the user's environment. This can mean substantial additional programming for the developer. In our observatory environment, however, we simply reject browsers that do not support Javascript and enable cookies.

2.1. Non-AJAX interactive methods

AJAX is not the only means of providing efficient web-based interactive pages. Before the widespread availability of AJAX, the most popular method for enabling asynchronous interactive web pages was based on using iFrames in place of XMLHttpRequest. However, XMLHttpRequest is generally cleaner (having been purpose-built for remote calls), faster, has a more consistent API, and readily handles XML.

Another method of interest is HTTP Streaming. The connection to the web server remains open, allowing the server to stream data to the client. This, superficially, makes it an appealing model for observatory services that want to broadcast updates of telescope or instrument state. However, when latency and bandwidth are of concern, the XMLHttpRequest polling method is preferable, because it conveniently ensures that the client will ask for and thus receive updates no faster than it can handle them. More importantly, proxies in the path between browser and server may interfere with the desired flow of data.

3. REMOTE INTERFACE ARCHITECTURES

AJAX interfaces represent only one among many means of implementing remote interfaces for observatory applications. Each technique has its advantages and drawbacks. A selective listing of some of these architectures include:

- 1. Direct remote display. User interface software runs locally at the observatory, but displays to a remote window system, often using a connection that is securely forwarded over an *ssh* session.
- 2. VNC display. User interface software runs locally at the observatory, including displaying to a VNC (virtual network computing) X server at the observatory. The remote observer uses a VNC client to connect to the observatory's VNC server. As with the previous method, it is common to use *ssh* to provide a secure "tunnel" for the VNC connection.
- 3. Remote GUI. A copy of the user interface software runs on the remote user's computer, communicating with the underlying services which reside on the local observatory computers.
- 4. AJAX-based. All software runs locally at the observatory, except that the remote user interacts with the software via a web browser and AJAX.

3.1. Method 1: direct remote display

Direct remote display, tunneled over *ssh*, has been employed for remote observing at Keck Observatory.⁴

Advantages:

• Local management of software. All software except the remote user's window system and configuration resides on the observatory-managed computers. This is very important, because it is difficult to ensure that a remote user's computing environment will support the observatory's custom software.

Disadvantages:

- Slow GUI response. Typical GUIs have many widgets, and it is time-consuming to update them when network latency is large. Image displays are slow to pan and zoom, again due to network latency and/or reduced bandwidth.
- Both major and subtle configuration problems. The remote user's window manager must be configured to operate well with observatory software. Even as minor a variation as the user's choice of "click-to-type" or "click-to-place-new-window" can cause the observatory software not to function as intended.
- Display compatibility. The remote user's window system must be compatible with observatory software (e.g. both must use the X Window System).

3.2. Method 2: VNC display

VNC display is similar to Method 1, but the observatory software display output goes to a VNC (virtual network computing) X server. The remote user runs a VNC client that connects to the VNC server. This method has become the preferred method for remote observing at Keck Observatory.⁴

Advantages:

- Local management of all software, up to and including the VNC server.
- Remote users need only have a VNC client, and these exist for Unix/Linux systems, Microsoft Windows, and Macintosh systems.
- VNC offers significantly better throughput compared to a direct connection to a remote X server, when operating over long-latency and/or low-bandwidth connections.

Disadvantages:

- Remote users must have a VNC client installed (not normally a large hurdle).
- If the remote user's screen dimensions (in pixels) are not at least as large as the screen dimensions used locally at the observatory, the remote user may have to pan the VNC client's view into the VNC server's display or else not be able to see all the GUI components.

3.3. Method 3: Remote GUI

This method is significantly different from methods (1) and (2). Here, a copy of the GUI software runs on the remote user's computer, communicating with the underlying services which reside on the local observatory computers.

Advantages:

• Very good interactive performance. Mouse clicks, keystrokes, and graphical output are processed by the remote user's computer instead of travelling to the observatory computer, avoiding the latency problems that can afflict methods (1) and (2).

• Modest bandwidth requirements. Excepting image display, the bandwidth requirements are modest compared to methods (1) and (2), because transmitting the data (e.g. telescope pointing information, instrument filter selection, etc) generally requires less bandwidth than is required to transmit changes to graphical windows in methods (1) and (2).

Disadvantages:

- Management issues. The client software resides on the remote user's computer, so there are serious management headaches, such as version mismatches, hardware/software incompatibilities, missing dependent software, etc.
- Firewalls. Depending on the client software's communication protocols with the back-end services, it may be difficult to forward the client-server communications through the firewalls that protect most observatories. Some protocols can be easily tunneled over (say) *ssh* connections, but other protocols are not firewall-friendly.
- Image transfer. Current-generation CCDs and arrays of CCDs may have many more pixels than a computer display; thus, transmitting an entire FITS image to a remote client may take significantly longer than transmitting a processed display image.

3.4. Method 4: AJAX-based

Advantages:

- No special software requirements. Virtually all modern browsers have AJAX support, so no special software is required at the remote user side.
- As with Method 3, the network bandwidth requirements are modest.
- Clean demarcation between presentation and data access. It is our experience that observatory and instrument software developers are not often experts in user interface design. We speculate that the clean demarcation between presentation via a Web interface, and the AJAX methods required to access the underlying data, may lead to better observatory and instrument GUIs. A web-design expert, who is not expert in the internals of an instrument's software, can produce a high-quality interface by just learning two Javascript calls to access the data.

Disadvantages:

- It is difficult to make Web interfaces that please the end user as much as native GUI applications, because the web programming environment is less rich than a typical GUI development environment.
- As with many Web technologies, AJAX might become obsolete shortly after a developer commits to using it.
- Web-based image display tools do not offer many of the capabilities of a traditional display package such as *ds9*.

4. LOCAL ACCESS VIA AJAX

AJAX-based interfaces may be as much useful for certain local access purposes as for remote access. They may be particularly useful for system monitoring and engineering access during system development.

For more than a decade, the user interface software for all the instruments developed by UCO/Lick Observatory for both the Keck and Lick Observatories has been Unix/Linux-based. However, many of the technicians who build the instruments use Windows exclusively. We have observed that the need to log into and use a Unix/Linux system acts as a barrier to entry for the technicians, discouraging them from using the instrument

control software during the development and construction period. A web-based instrument control system would avoid these problems.

Observatory staff must monitor the health and status of a wide variety of systems. It is often a headache to manage the placement of the multiple windows that make up a suite of standalone software interfaces. By contrast, if the monitoring software is embedded in a browser, each system's web pages may be placed in a separate tabbed pane of the browser.

When system-monitoring software is embedded in a browser, an additional modest convenience is obtained because the monitor interfaces for a variety of systems can be reached by simply clicking on a link in a browser. In principle, this is not notably simpler than selecting a suitable program from a pull-down menu. In practice, however, it is our observation that many staff prefer navigating via the browser.

These issues of simplified access and screen management are relatively small things, but day-in and dayout they can spell the difference between software that makes routines a little easier for observatory staff, and software that irritates them each day.

5. WEBKTL

5.1. KTL services

At Keck Observatory, and increasingly at Lick Observatory, most controlled hardware (telescopes, instruments, etc) is represented using KTL (Keck Task Layer) services.⁵

KTL services represent all hardware components through sets of keyword (or attribute)/value pairs. A KTL service is a simple construction of very flat design, in the sense that all attributes are equally visible to all clients, and no hierarchical structure is built into the keywords. Such uniform visibility also enables efficient use of an evolved set of diagnostic tools: one may attach general-purpose software tools to a keyword to permit transparent monitoring without affecting the underlying software. When a higher-level abstraction is required, keywords for several components may be aggregated into a compound object. Together, these features mean that all software clients can have access to the same complete view of the system. This makes it simple to use the services interface to write sophisticated, hierarchical control systems; GUIs for observers, operators, and engineers; simple or complex observing scripts; and all necessary support for robotic, remote, or queued observing.

Clients gain access to a KTL service through the use of KTL client libraries. Each KTL service has a matching shared-object client library, which implements the standard KTL API to provide access to the service's keywords. A client dynamically loads the KTL client libraries for any services with which it wishes to communicate.

A well-designed set of KTL services and clients ensures that all state information is computed and stored on the service side, and clients never compute system state. This ensures that all clients can have the same, complete view of the system state. (This sounds obvious, but it is sometimes easy and expedient for a client to compute some small piece of state information. For example, a GUI may embed a rule such as "if the observer selects filter X, then the camera focus should be set to Y". This may be expedient, but it is better that the rule be put into the KTL service, rather than kept private to a client.)

5.2. WebKTL: an AJAX-based interface to KTL services

UCO/Lick Observatory's first foray into AJAX applications for instrument and telescope monitor and control is WebKTL.

WebKTL consists of these components:

- Semi-static HTML and/or server-side include (SSI) generated HTML pages that provide the GUI layout for an instrument or other device with a KTL service interface. These pages include the Javascript that fetches and refreshes the keyword data displayed in the page. (Here, semi-static means that nearly all of the content is static, except for a session cookie generated each time the page is fetched.)
- A cgi-bin script, *webktl_connect*, invoked to handle each client refresh request or keyword modify request.

• A persistent state engine, *webktl_state*, that connects to KTL services as needed, on behalf of all clients, and stores the state information needed to refresh each client with any changed keyword data upon demand.

Figure 1 illustrates the WebKTL components, and shows them in the context of the existing world of KTL services and clients. Each user interface, control script, and the WebKTL state engine is a client of one or more KTL services. As KTL clients, each loads one KTL client library for each KTL service with which it is communicating.

WebKTL's HTML pages come in two flavors. First, there is a general-purpose interface that can be used for any service: this is a simple dynamic tree for browsing and/or editing keywords. Second, there are customdesigned user interfaces for specific instruments and telescopes.

The general-purpose interface is useful mainly for engineering purposes. Although it is not an attractive interface for ordinary use, it does provide KTL service access to users who do not use the Unix/Linux hosts on which our custom client software is installed.

5.3. The webktl_connect cgi-bin script

The *webktl_connect* cgi-bin script is a small Tcl script that is invoked by an XMLHttpRequest object in the HTML page. Nearly all requests are refresh requests, sent by the client at about 1 Hz, calling for updates of any keyword values that have changed since the last request. A 'modify' request is sent when the user wants to change a keyword (for example, to select a new filter position).

When *webktl_connect* starts running, its standard input and output are connected as for any cgi-bin script: GET-method requests are passed through the environment; POST-method requests arrive over stdin; and the response is passed back over stdout to the browser. *Webktl_connect* begins by making a connection to the socket on which the *webktl_state* state engine should be listening. If it cannot connect, it *exec*'s *webktl_state*, waits for the engine to acknowledge either a successful startup or a failure to start, and then tries again to connect. A failure at this point is fatal.

After *webktl_connect* connects to *webktl_state*, it starts interpreting the request. It checks that the required session cookie exists (else returns a message explaining the need for cookies), and that the request is syntactically valid. The request is converted to a proper Tcl list, and sent over the local socket to the state engine, along with the cookie and the browser's IP address and hostname. The *webktl_connect* instance then waits for the response from *webktl_state*, formats the response as XML, sends the result back to the client, and exits.

5.4. The *webktl_state* persistent state engine

The *webktl_state* state engine starts by attempting to listen for WebKTL requests on a localhost socket. If unsuccessful, it exits; otherwise, it takes no further actions until it receives a client request. It always sends a success or failure status message to stdout. If it was started from a *webktl_connect* instance, the caller uses the status message to knows whether it should proceed with making a connection or exit with an error.

No authentication is done for incoming client connections; this implies that only trusted users are executing programs on the local host. (Existing KTL services do not provide authentication mechanisms, so KTL has always had an implicit requirement that only trusted users may use computers with direct access to KTL services.)

Webktl_state reads data from a webktl_connect client, using non-blocking I/O, until it has assembled a proper Tcl list. It syntax-checks the list for validity, and checks whether the request is originating from an IP address or hostname that has permission to make the specific request. (See the section on security, below.)

If the remote host has permission to access this KTL service, and the service is not yet being monitored by the state engine, *webktl_state* loads the service client library and begins to monitor all the keywords. KTL services asynchronously broadcast updated keyword values whenever a keyword changes (and typically throttle their broadcast rate at 1 Hz). Each time a keyword is updated, *webktl_state* records the new value and its modification time. For security and/or performance reasons, *webktl_state* doesn't "publish" all of these keywords to its clients. Instead, it consults a configuration file to know which keywords should be published for read and/or modify access, and only those keywords will be made visible to WebKTL sessions. A reasonable configuration might configure all keywords as readonly for hosts that are outside a limited part of the observatory intranet.

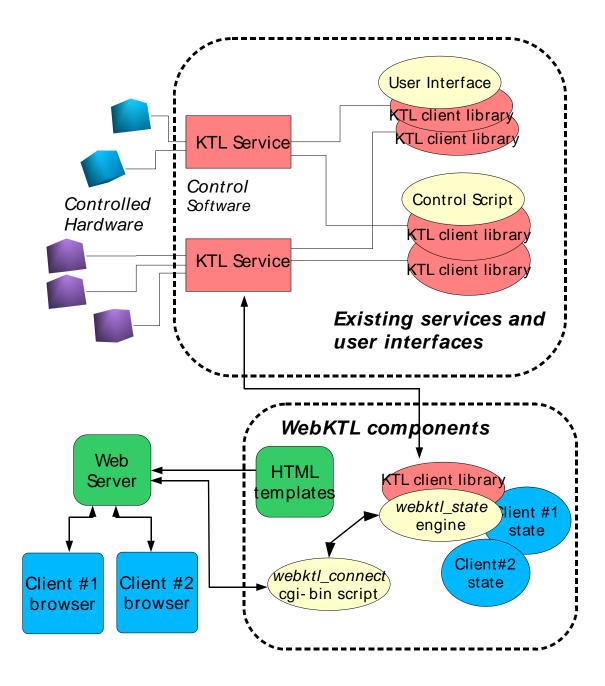


Figure 1. The WebKTL components are shown here as additions to the existing KTL services and clients infrastructure. Web browsers use a cgi-bin script to poll for frequent data refreshes.

If the connection is from a new WebKTL session — as indicated by a session cookie that is unknown to the state engine — *webktl_state* initializes a last-updated timestamp for the {session, service} pair to 0. The timestamp will be used to track the last-update time of the WebKTL session's view of the service data.

Refresh requests are handled by comparing each published keyword's modification time to the {session, service} last-update timestamp, and putting putting modified keyword/value pairs into the response data. If the request is from a new WebKTL session, the response data will also include keyword attributes for permission (readonly, writeonly, or read/write) and data type.

Modify requests are handled by checking that the keyword has been published for modification, executing the underlying KTL-modify command, and returning the KTL-modify status code and associated status message.

Webktl_state only needs to store a small amount of data for each session, consisting of the per-service last-update timestamps and a last-modify-request timestamp. The last-modify-request timestamp is described in the section on security, below. If a WebKTL session fails to make refresh requests reasonably often (about once per minute), webktl_state assumes that the WebKTL session has been closed and the client data is not needed. If a WebKTL session later re-uses the cookie, it will be regarded as a new session.

5.5. Data Rates

The initial data transfer from *webktl_state* to a WebKTL client sends all of a service's keywords and their attributes. This can be relatively large. For example, the four services that make up the DEIMOS spectrograph system, the largest instrument yet built by UCO/Lick Observatory, together comprise approximately 1280 keywords. The initial refresh-response is about 100 KB. However, the following refresh requests require much less data: fewer than 200 keywords change each second, and the data rate is less than 9 KB/s. Thus, a WebKTL session is quite parsimonious with its data rates, and can be used as readily from a home computer over a DSL line as from a computer connected via high-speed network.

5.6. 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 at large.

The most important part of WebKTL security lies outside the HTTP level itself, and is enforced by sensible configuration of the web server. First, only HTTPS (HTTP over Secure Socket Layer, or in current argot, HTTP over Transport Layer Security) connections are permitted. This helps ensure that communications between the browser and WebKTL server are not hijacked. Second, all connections must use password authentication. Third, it is helpful but not essential to configure the web server to limit access by host or domain name. *WebktLstate* always applies its own restrictions based on hostname or IP address, but the web server can limit denial of service attacks with host access control lists.

The *webktl_connect* cgi-bin script rejects any queries that are not syntactically well-formed; that use implausibly long service or keyword names; or that attempt to issue a modify request with a string exceeding 4KB. All other tests are applied by the *webktl_state* state engine. *Webktl_state* first checks that:

- The browser's hostname and/or address is acceptable.
- The browser's host is not making new-session requests excessively frequently, else the request is rejected.
- This WebKTL session is not making refresh requests or modify requests excessively fast, else the request is rejected.

Webktl_state then goes on to check KTL service name, keyword, and value (for modify requests):

- The service name and keyword must contain only letters, digits, and/or underscore;
- The KTL service and keyword names must not exceed 100 characters each;
- Modify requests must not attempt to pass strings longer than 4KB.
- Modify requests are not permitted unless the underlying KTL service grants modify permission *and* the WebKTL configuration file grants write permission to the browser's IP address.

00	http://spg.ucolick.or	g/cgi-bin/showadc	Search Search
	Bookmarks 🛇 Wikipedia 🛇 mozilla.org 🛇 m	ozillaZine 🛇 mozdev.org 🛇 10 Day Local	
) 🖓 - K1AD	C -		
ADCMOD	Adc motor control mode	Halt	
ADCSTA	Adc overall status	Halted	
ADCTRG	Adc raw position target	0	motor encoder counts
ADCRAW	Adc raw position	298770	motor encoder counts
ADCTVA	Adc separation target	0.000	mm
ADCVAL	Adc separation	1608.843	mm
ADCTVX	Adc zenith distance target	0.0000	degrees
ADCVAX	Adc zenith distance	58.4268	degrees
ADCTNM	Adc named position target		
ADCNAM	Adc named position	Unknown	
ADCTRD	Adc ordinal position target	0	
ADCORD	Adc ordinal position	-999	
ADCTSP	Adc raw velocity target	0	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	0	motor encoder counts
ADCSVA	Adc servo val tracking error	0.000	mm
ADCSVX	Adc servo vax tracking error	0.0	arcsec
ADCENE	Adc load encoder expected	-14938.500	load encoder counts
ADCENC	Adc load encoder position	-14928	load encoder counts
ADCTOR	Adc motor torque	0.000	
ADCCMT	Adc comment	and the second	
ADCTAG	Adc tag		
ADCLCK	Adc motion keyword lockout	unlocked	
ADCCAL	Adc calibration	homed	
ADCMSG	Adc lasterror message	Hand paddle has been enabled	
ADCERR	Adc modify error number	-5106	
ADCLIM	Adc limit switch status	Not in a limit	
ADCDCC	Adc disconnected cables		
ADCRCD	Adc remote control disablers	Hand paddle	
ADCCON	Adc motor control	manual	
ADCBYP	Adc motor control bypass switch	inactive	
ADCP2F	Adc prism 2 fiducial	active	

Figure 2. A simple example of a WebKTL interface. This interface requires no special programming to produce, and is useful to give technicians control of motors before a polished, dedicated user interface is created.

5.7. Sample Implementations

An early, very simple interface is shown in Fig. 2. This simple interface is not particularly attractive, but it is immediately function for any KTL service without requiring any programming. This makes it useful to technicians who typically need motor control long before a polished, dedicated user interface has been created.

A more polished interface was not available at this writing, but current implementations may be seen at http://spg.ucolick.org/webktl. These include sample code for telescope monitoring, instrument control, and displaying weather station information.

6. FUTURE WORK AND CONCLUSIONS

AJAX methods provide low-overhead methods of implementing user interfaces.

The work described in this paper is presently being extended by creating high-quality custom AJAX instrument interfaces.

A second problem that deserves attention is the updating of the KTL keyword design. The WebKTL system addresses the problem of *how* to display user interfaces efficiently over the web. However, it does not help software to know *what* is represented by the KTL keywords that it encodes, so it isn't easy to automatically generate a high-quality interface. This problem exists for any interface built for KTL services, not just AJAX interfaces.

The problem starts with the limitations of KTL keywords, which are restricted to very simple data types (integer, floating point, and small arrays of these; boolean, and character string). Instruments developed at UCO/Lick Observatory have employed a set of conventions for these keywords, thus providing a descriptive layer on top of KTL. For example, all keywords associated with a particular stage will begin with a common prefix, e.g. CFILT might be the prefix for a filter wheel stage. Similarly, there is a set of standard keyword suffixes that is used in a regular manner, e.g. the NAM suffix is used for named positions. Thus the CFILTNAM keyword refers the named positions of the CFILT stage.

The *dashboard* toolkit makes use of these conventions to simplify the construction of new instrument interfaces, but their informality still demands significant manual oversight. Thus, ordinary software, including our AJAX applications, aren't capable of auto-generating a particularly nice interface. Perhaps it would be possible to come closer to auto-generating a good interface if the detailed characteristics of instruments were formally described using XML.

There have been previous attempts to describe the characteristics of astronomical instrument control formally using XML,⁶ but these notions have not yet been employed at UCO/Lick Observatory. AJAX's support for XML invites treating groups of keywords as objects described by nested entities in XML. It also invites using objectoriented snips of javascript which produce engineering interfaces that meaningfully display all of the keywords related to a particular motor.

REFERENCES

- 1. "Abstract Window Toolkit." http://java.sun.com/javase/6/docs/guide/awt/index.html.
- D. Clarke, "Dashboard: A Knowledge-Based Real-Time Control Panel," in Proceedings of the Fifth Annual Tcl/Tk Workshop 1997, J. A. Konstan and B. Welch, eds., USENIX Proceedings, 1997.
- 3. J. Gehtland, G. Galtraith, and D. Almaer, *Pragmatic Ajax A Web 2.0 Primer*, The Pragmatic Programmers LLC, Raleigh, N.C., 2006.
- R. Kibrick, S. L. Allen, A. Conrad, and G. D. Wirth, "Optimizing interactive performance for long-distance remote observing," in *Advanced Software and Control for Astronomy*, H. Lewis and A. Bridger, eds., *Proc.* SPIE 6274, 2006.
- W. F. Lupton, "Keck telescope control system," in Astronomical Data Analysis Software and Systems IX, N. Manset, C. Veillet, and D. Crabtree, eds., ASP Conf. Ser. 216, 2000.
- T. Ames, L. Koons, K. Sall, and C. Warsaw, "Using XML and Java for telescope and instrumentation control," in Advanced Software and Control for Astronomy, H. Lewis, ed., Proc. SPIE 4009, 2000.