Practical use of KTL keywords: 2010 edition

Kyle Lanclos, William T.S. Deich
UCO/Lick Observatory
University of California at Santa Cruz, CA 95064 USA
http://www.ucolick.org

May 8, 2014

Abstract

The Keck Task Library (KTL) is an application programming interface (API) widely used at Keck Observatory and Lick Observatory. The essential goal of KTL is to simplify and standardize the mechanism used to acquire metadata from unique sources. For example, a spectrograph motor control mechanism, weather monitoring service, and dome pointing system could all provide access to unique data via KTL; a single client, or multiple clients, could then access that data strictly via the KTL API, without necessarily knowing any of the internal mechanisms used to produce the data in the first place.

This paper discusses the practical uses for KTL in the context of software intended for long-term use by diverse group of end-users and developers. What benefits does KTL provide? What classes of problems are optimally solved via KTL? How should the use of KTL be integrated with other communication mechanisms?

1 What is KTL?

The technical details and layout of the KTL API are described in several other publications. For the purposes of this discussion, KTL provides a standardized mechanism to read out and/or modify information, encapsulated as keyword/value pairs. The API limits itself to the following fundamental data types: integers, floating point numbers, and strings of characters. In addition, there are a handful of derived types: arrays of both numeric types, enumerated integers, integer masks, and boolean values.

There is no native support for binary blobs. One workaround is to transmit blobs as base64-encoded strings. Besides the immediate hit on performance and efficiency, there are message size limitations deep inside some (many) of the KTL implementations that preclude the transmission of values in excess of four kilobytes.

All keyword values have **binary** (native type) and **ASCII** (string) representations, with binary-to-ASCII translation provided by routines internal to the KTL client library. This flexibility allows for arbitrary human-readable output for a given keyword; for example, telescope pointing keywords may be transmitted in radians as their binary form, but the human readable ASCII formatting may be sexagesimal (HH:MM:SS).

KTL designates the reading of values as a

read operation; the modification of a value is a **write** operation. These are commonly referred to by the command-line tools that implement these functions, `show` and `modify`. Both synchronous and asynchronous write operations are supported, but in general only synchronous reads are supported. Continuous broadcasting of keyword values is also supported. The basic messaging system can be considered a publish/subscribe model.

A software daemon that provides access to one or more keywords is called a **dispatcher**. Individual keywords are logically bundled into **services**; a service may aggregate keywords from multiple distinct dispatchers. For example, one dispatcher may expose motor control functionality, while another dispatcher communicates with a temperature controller, but both dispatchers could be part of the same service. Any software that performs read or write operations on a KTL service is referred to as a **client**.

The essential components and build environment for KTL are UNIX-centric, with successful deployments on SunOS, Solaris, Linux, and Mac OS X. While untested, successful builds should be possible on FreeBSD, NetBSD, and OpenBSD. A Windows-native build of the current KTL architecture is not currently possible, and there is no active development towards enabling this capability.

2 When to use KTL?

Two primary design goals influence how and when to use KTL as a means to distribute information within our software environment.

2.1 Sharing live information

The most essential function of KTL in our software environment is to provide live access to the current state of a given system. Whether that information is acquired asynchronously, or on a just-in-time basis, KTL provides the framework to inspect and modify system state.

In order to be effective, it is essential that the dispatcher provide direct access to all of the data points that need to be manipulated by a client application. In general, if a given dispatcher has access to information, there will eventually be a client application that wants or needs to see it.

Similarly, it is important to expose as much of this information as possible via KTL, rather than a secondary communication mechanism. For example: if the dispatcher and the client share common state data that is written to a temporary file, or to a SQL database, that information should instead be exposed via a keyword, or via multiple keywords.

2.2 Compute state on the dispatcher

In order to maintain consistency across all listening clients, it is vital that any computation of state be performed by the dispatcher. If a client computes desirable state internally, it will be difficult (or impossible) to reliably share that information with other running clients. Additionally, if the client is computing a key piece of information, you may find yourself in a position where the client has to be running in order to express the full meaning of the keywords provided within a service.

3 When not to use KTL?

There are situations where it is not necessarily sensible for information to be shared via keywords. Three classes of content are typically not shared via keywords within our environment: application and server logs, configuration files, and camera images.

The primary goal for logging is typically to allow a developer to debug abnormal behavior in the system. Any information necessary for general operation would instead be exposed via keywords.

Configuration files are generally required to bootstrap the initial state of a dispatcher or client application; a dependency on retrieving configuration data via the network may prevent the application in question from starting in a sane fashion. Once running, the application should expose or derive as much information as possible via keywords, in order to ensure that the exposed state is as close to possible as the genuine state of the system(s) in question.

Camera images are generally too large to be distributed via keywords. Instead, images are written to a known location in the filesystem (a location advertised via keyword), or to a shared memory resource. If the architectural limitations were removed such that large binary blobs could be transmitted, it would still be necessary to determine whether the underlying transport could support enough throughput to provide the desired refresh rate, for example, if one were using keywords to distribute guide camera frames.

4 Implementing a KTL service

The KTL API itself is a very thin layer, allowing for very different systems to be present "under the hood" when it comes to messaging transport systems, or implementation of dispatchers. Over the years, many different approaches have been exercised; what is described below represents the current best practice for services deployed by UCO/Lick.

4.1 Implementing a dispatcher

Will Deich wrote a middle-ware dispatcher, `stdiosvc`, to handle all of the minutiae required for a dispatcher to receive requests via KTL. Commands or requests are received by the `stdiosvc` process, which translates these requests to newline-delimited strings. Those strings are passed to the "real" dispatcher, which interacts directly with the hardware; the dispatcher then issues newline-delimited commands to `stdiosvc` that instruct it on the proper response to the query.

Because `stdiosvc` operates via this thin command/response mechanism, the developer is free to implement their dispatcher in any programming language. At present, `stdiosvc` is in active use with both Python and C/C++ dispatchers.

In order to maximize the effectiveness of the dispatcher, it is important that it broadcast values as soon as is reasonable. For example, if a client writes a new boolean value to a given keyword, the dispatcher should immediately broadcast the change. For constantly fluctuating values, such as the value of a motor encoder on a servo motor that is in motion and thus has continuously changing values, it is more sensible to broadcast the current value on synchronous intervals, such as once a second.

A dispatcher will generally provide three distinct types of keywords: **direct** keywords, which directly report a specific, hardware-provided value; **derived** keywords, which may reflect

overall system state, or some other quantity that is computed based on one or more direct keywords; **independent** keywords, which reflect values that are computed internally by the dispatcher, without input from secondary sources.

In general, if a client makes a synchronous read request for a direct keyword, the dispatcher should acquire a current value from the hardware rather than return a cached value, as long as handling the request is within the capacity of the hardware. Upon acquiring the new value for the direct value, it should be broadcast; any keywords derived from that direct keyword should then be recomputed, and similarly broadcast.

All dispatchers should provide a set of administrative keywords, reflecting the state of the dispatcher itself. At a minimum, there should be a keyword to represent the state of the dispatcher, one or more keywords to provide detailed error reporting for dispatcher-specific problems, and a heartbeat keyword that a client can monitor to determine whether it is still connected to a running dispatcher.

4.2 Describing a service's keywords

The current best-practice for establishing a service's keywords involves the creation of Extensible Markup Language (XML) files to fully document the nature of each and every keyword in a given service. Previous generations of tools required working with a narrow view of a database-driven system; in addition, any input data that diverged from the expectations of those tools could result in unpredictable behavior. The XML-based system aims to be completely deterministic in its behavior, fully descriptive, and open to modification by any tools that can modify text files.

Beyond standard text editing, the XML is amenable to templating for repetitive sets of keywords. For example, the creation of keyword XML for Galil motor control services is heavily templated, such that an initial file containing fifty lines produces a five-hundred line XML file, containing complete descriptions for nearly fifty keywords. Because each motor stage is very similar, and there may be more than a dozen motor stages, this pays off very quickly: the motor control service for the current AO system on the Shane telescope at Mt. Hamilton contains more than 750 keywords, representing nineteen different motor stages.

Having described the keywords, the transport mechanism must also be defined. UCO/Lick exclusively uses MUSIC messaging as our KTL transport mechanism; the primary way that the KTL developer must exercise this knowledge is to define MUSIC message identification numbers for each of the standard KTL data types. The back-end daemon that facilitates MUSIC messaging is `traffic`, and a configuration file must be updated to indicate where interested clients can locate the appropriate traffic daemon.

4.3 Example dispatchers

There are a few different dispatchers implemented strictly as a tool for development. As such, they generally do not provide any direct or derived keywords, but instead offer only independent keywords. Here are two examples, the first representing a deprecated implementation in C, and the second a more modern implementation in Python that leverages then `stdiosvc` back-end:

```
cvs/kroot/kss/dummy
cvs/lroot/pie
```

Note that the pie service above serves as an example for several different best practices: in addition to the dispatcher itself, the keywords in the pie service are fully enumerated in XML, and the necessary subdirectories and build structure are in place to construct a KTL client library.

A more complex example of a modern dispatcher is the `galildisp` package, which is capable of providing access to a wide range of Galil motor control devices:

`cvs/kroot/kss/optical/galildisp`

5 Implementing a KTL client

Because the core KTL layer is implemented in C, it has been adapted for use with many different languages. Over the years, native wrappers around the KTL core were created for Tcl, Java, and Python. In addition, because there are command-line tools available, it is straightforward to invoke KTL calls from shell scripts.

5.1 Building the client-side interface

In order for a KTL service to be visible to a KTL client, regardless of the language used to implement the client, a shared library must be built and installed. The KTL service will not be visible on any computer that does not have this library installed. Once the description of the KTL XML is complete, and MUSIC message numbers have been defined for the new service, the creation of the KTL shared library is completely automated. A standard set of Makefiles are copied in, along with a small set of `.h` and `.c` files, and the client library is ready to be compiled and installed.

5.2 Transient clients

For KTL clients that are invoked for a discrete purpose, the use of synchronous reads and writes is common. For example, you may have a script that moves all of the motor-controlled stages in a spectrograph to a predefined set of known positions: that script may perform some initial preparedness-checking via synchronous reads, and then use synchronous writes to perform the modifications; when all of these calls complete, the script continues to completion.

5.3 Persistent clients

For KTL clients that persist, such as user interfaces, the use of broadcast reads and asynchronous writes tends to be more common. Leveraging broadcast reads allows a persistent client to continually update its local notion of a keyword value without blocking on a synchronous request. Similarly, using asynchronous writes allows a persistent client to remain responsive to new input, while the KTL call is processed in the background.

Persistent clients should also be careful to handle the absence or disappearance of a service's dispatcher(s) while the client is running. Asynchronous broadcasts of keyword values will resume when the dispatcher is restored, but their initial broadcasts may diverge strongly from the last known value for those keywords. By monitoring heartbeat keywords, a persistent client can alter its behavior when a dispatcher is not available; a GUI client may display a splash screen indicating the lack of an available dispatcher, or it may disable synchronous actions, such as write operations to keyword values.

5.4 Example clients

With the relevant KTL client library already installed, a simple transient client may be as short as the following:

```
#!/bin/sh
# Move three stages in service 'apfmot'
# to predefined positions:

modify -s apfmot ADCRAW='0' &
modify -s apfmot DECKERVAL='20.002' &
modify -s apfmot IODINERAW='13311' &

# Wait for all backgrounded modify calls
# to complete before exiting.
wait
```

A relatively complex example of a transient client, `show` re-written in Python, is present at the following location:

```
cvs/kroot/ktl/keyword/python/examples
```

An example of a persistent client written in C is present here:

```
cvs/kroot/ktl/kui/cshow
```

An example of a persistent GUI client written in Python is present here:

```
cvs/lroot/kast100k/gui/kast_gui.sin
```