

Using scripting languages in optical interferometry

Leonard J. Reder[†], Thomas G. Lockhart^{††}, Kenneth C. Ko, Benjamin T. Smith

Real Time Interferometry Software Group
Jet Propulsion Laboratory/California Institute of Technology

ABSTRACT

Testbeds and production systems need lightweight, capable, and rapidly developed applications. We have developed several such scripts for testing and operating the Keck Interferometer. Two stand-alone (Tcl/Tk script) applications implemented to support the Keck Interferometer are discussed. The first is a front end to automatic and manual optical alignment embedded software, developed using the Keck Observatory Keyword API extension. The second is a user interface to the Interferometer Sequencer that communicates with it via both Keywords and Common Object Request Broker Architecture (CORBA). We discuss client-side CORBA scripts implemented in Tcl, Perl and Python. These are all technologies that are either currently being used on testbeds at JPL or being evaluated for future use. Finally, a Python example demonstrating implementation of a simple CORBA server is presented.

Keywords: CORBA, Tcl, Python, Perl, Interferometer, Keck, Sequencer, Align

1. INTRODUCTION

In the 1990s scripting languages evolved rapidly to a very high level of sophistication. Extensible languages like Tcl, Perl and Python are ideal for rapid prototyping and rival the power of languages like FORTRAN, C, C++ and JAVA. At JPL we utilize these languages on several different projects. JPL's Real Time Interferometry Software Group is responsible for development of software and hardware technology to support implementation of astronomical interferometer systems. Both ground and space interferometer projects are currently in progress. The major ground based interferometer supported by the group links the two 10m telescopes of the W. M. Keck Observatory at Mauna Kea, Hawaii. In addition, several technology testbeds are being implemented for future flight missions including the Space Interferometry Mission (SIM).

The Keck Observatory telescope control system was implemented using a software package called Experimental Physics and Industrial Control System (EPICS). EPICS is a distributed control system running at Keck on single board computers using the VxWorks real time operating system. EPICS software is also used to implement the alignment software of the Keck Interferometer optics. Client-side applications communicate with EPICS via a Keyword^{*} API that is a proprietary legacy middleware component written by the Keck Observatory. Tcl is used with a Keyword extension called KTcl developed by the W. M. Keck Observatory[†] to communicate with EPICS software via Keywords. Use of this technology allows rapid prototype development of applications with minimal design time.

At JPL we have implemented a real time embedded interferometer control software which is used as the basic infrastructure for implementation of interferometer control at the W. M. Keck Observatory (see Ref. [2]) and the internal testbed projects. The software is called RTC (for real-time control) and is discussed in reference [3]. RTC is a distributed control package built within the VxWorks real-time operating system environment and utilizing CORBA⁴ for commanding and monitoring distributed controller objects. More specifically RTC uses real-time TAO (the ACE ORB) implementation of CORBA. Tcl, Perl and Python can all talk to the RTC software via various CORBA extensions. For Tcl the Mico ORB is used with the Combat extension, for Python omniORBpy is used and for Perl ORBit is used.

In this paper two stand-alone applications implemented to support the Keck Interferometer are discussed. The first application is a Keyword based application for performing automatic and manual optical alignment of the Keck Interferometer. The second is a client-side user interface (UI) to the Interferometer sequencer that communicates via both Keywords and CORBA in a single Tcl application. Within the sequencer script CORBA RPC method calls are

[†] Leonard.J.Reder@jpl.nasa.gov; phone 1-818-354-3639; fax 1-818-393-4357; Jet Propulsion Laboratory, California Institute of Technology, Mail-Stop 171-113, 4800 Oak Grove Drive, Pasadena, CA. 91109-8099

^{††} Thomas.Lockhart@jpl.nasa.gov; phone 1-818-354-7797; fax 1-818-393-4357; Jet Propulsion Laboratory, California Institute of Technology, Mail-Stop 171-113, 4800 Oak Grove Drive, Pasadena, CA. 91109-8099

^{*} The term "Keyword" with a capital "K" refers to the W. M. Keck Observatory developed Keyword API as discussed in references [1], [5] and [6]. The term "keyword" will be used as the generic term used in many other systems.

executed for commanding and RTC style telemetry is monitored via the CORBA event service. Our testbeds at JPL have implemented several smaller utility scripts in Perl and Python. For example, a Perl CORBA server for controlling power to various devices via remote calls and a Python "Ping" program for testing for live CORBA RTC objects over the network have been implemented. Additional Python infrastructure is in development at the time of this writing.

2. KEYWORDS AND EPICS

At the W. M. Keck Observatory most of the control systems built for operation of the two large telescopes have been implemented in EPICS. EPICS is a distributed control system software package, widely used within experimental particle accelerator facilities and observatories around the world for control and data acquisition within large complex systems. Other instruments within the Keck infrastructure utilize their own unique software organization and are not EPICS based. In the early 90's Keck Observatory developers recognized the need for a consistent API for integrating several instruments into the Keck environment. To this end the W. M. Keck Observatory decided to use the concept of a distributed control system using a keyword/value paradigm which was a technique being used at the Berkeley Space Science Laboratory at that time.⁵

Fundamentally the idea is that a library called the Keck Task Library⁶ (KTL) is implemented that supports event driven messaging to and from various instruments or sub-systems (referred to as "services" in KTL). Within KTL the conventional *open*, *close*, *read*, *write* and *ioctl* calls are implemented to talk to services via keywords. For our use of Keywords another interface software layer called Channel Access Keyword (CAKE) interface is implemented at Keck. This links the Keyword API to EPICS via the Channel Access network communications protocol. Channel Access is a protocol built on top of TCP/IP that provides individual EPICS processors and client applications with shared/distributed communications capability.

To implement our scripts we utilized KTcl, the Keyword Tcl extension, developed by Keck to allow Tcl scripts to directly talk to Keyword services. KTcl implements a *ktl* command within the Tcl interpreter. The *ktl* command has subcommands implemented as arguments (e.g. *ktl subcmd*); these subcommands are *open*, *link*, *monitor*, *read*, *write* and *ioctl* which are bindings into the KTL library functionality. Within the two applications discussed later the Tcl scripts define *modify* and *show* functions to write keywords and read them. The *ktl monitor* command is used within the Tcl applications to connect callback Tcl functions to keywords. When a keyword value is changed this event causes the callback to be executed. The monitor facility allows very straightforward implementation of visual panels that display numerous parameters and update them as Keyword values change.

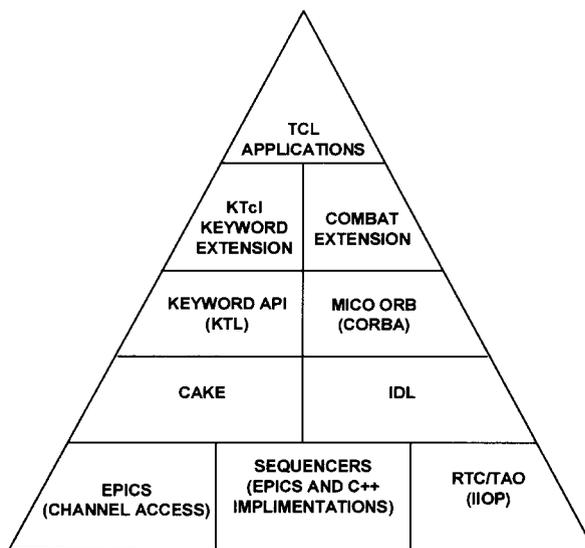


Figure 1. Software Layers within TCL Applications

The Keck Interferometer project has used the KTcl extension for the implementation of two Tcl scripts that are now used routinely in operation. These will be discussed next. The aligner script (*aligner.tcl*) is a user interface that is used for both manual and automatic optical alignment operations within the interferometer. It does all of its communications via keywords. The other application (*seq.tcl*) was initially keyword based and evolved into using both the KTcl extension and CORBA. The main function of *seq.tcl* is to provide a front end UI to the interferometer sequencer. The conceptual layers of software for TCL script implementation are shown in Fig. 1.

3. KECK INTERFEROMETER ALIGNMENT AND THE ALIGNER.TCL SCRIPT

The intent of the aligner.tcl script is to provide a single control panel that will allow the engineer to optically align the instrument, either manually or automatically from end to end. Alignment is a complicated task for the Interferometer due to the large number of optical elements within the instrument. Alignment is achieved using a series of illuminated targets that drop (move) into the optical path. Motion controlled optics (mirrors) are adjusted and alignment cameras are used to detect target position in the field of view of the optics. The overall alignment process involves an engineer stepping through each optical element and either manually or automatically aligning them to the assigned alignment target. The entire alignment system is discussed in reference [7].

This application relieves the optics engineer from having to know about underlying EPICS subsystems that provide control of all these optics in the interferometer. Scripting language technology provided an easy way to integrate all the systems into a central user interface.

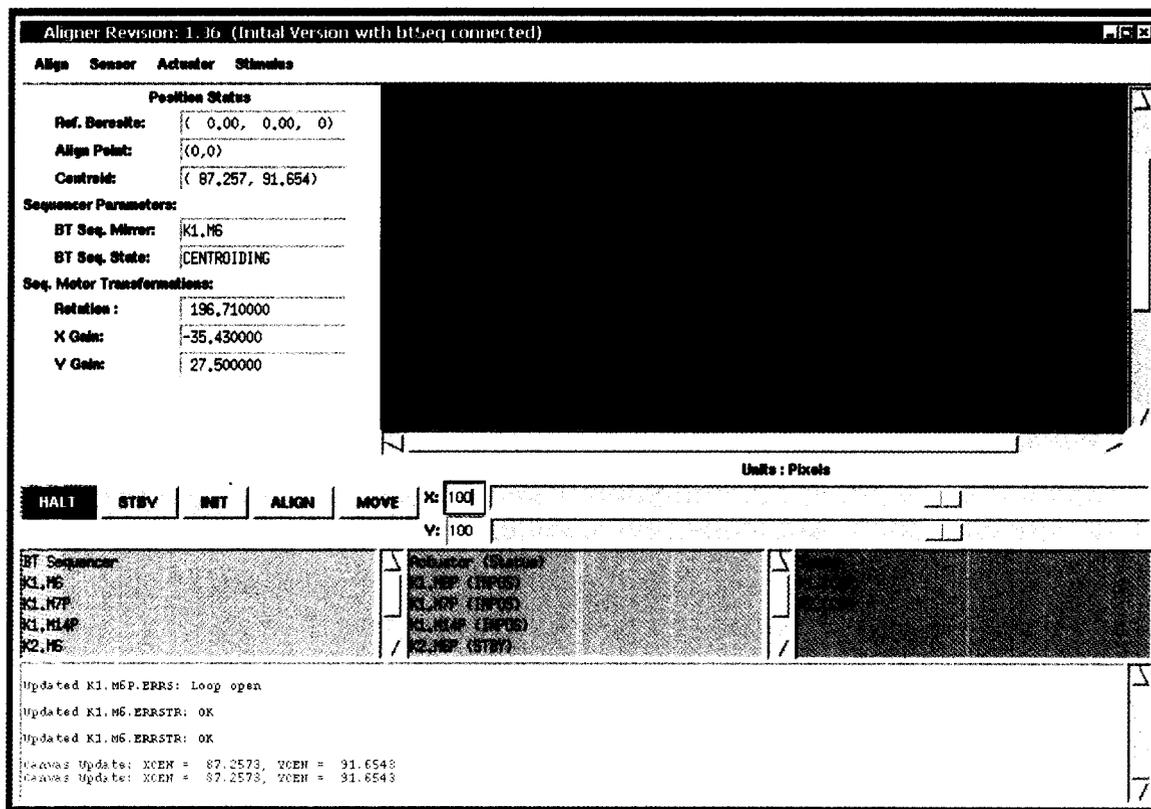


Figure 2. Automatic and Manual Optical Alignment User Interface TCL Application (aligner.tcl)

The aligner operates in two modes, a manual mode and the automatic mode. In manual mode the interface is a simple GUI front end for control of optical actuation and stimulus targets. The UI also is used for monitoring various types of alignment sensors (mostly CCD cameras). All these components have their software implemented within the EPICS environment. In automatic mode the user interface becomes the control panel for any one of several alignment sequencers. There is one instance of an alignment sequencer for each of the optics to be aligned. Each sequencer is implemented using the EPICS programmable (State Notation Language based) sequencer.

The main panel of aligner.tcl is shown in Fig. 2. The program is shown in automatic mode with the K1.M6 transport optic mirror (the first mirror in the interferometer beam train) being aligned. Central to the aligner is a Tcl canvas widget that displays an alignment image from a frame grabber (as shown) and/or a representation of alignment target location with respect to a reference boresite as detected in the field of view of a CCD alignment camera. These locations are determined by an embedded centroid routine implemented within EPICS. On the canvas widget the crosshair with a square represents a reference alignment boresite position and the crosshair with a circle is the current centroid position of the alignment target. A vector connects the two and represents the next move that the sequencer will apply to the mirror. Nominally it takes two or three iterations to automatically align an optic and this is displayed on the canvas widget as the centroided target getting closer to the boresite with each interaction until they are within a specified maximum separation

of pixels (essentially on top of each other). In typical operation the actual frame grabbed image is not displayed, and only the representation of alignment target and reference boresite position are displayed on a white background.

To the left of the canvas display are parameters associated with the selected sequencer. The **HALT**, **STBY**, **INIT**, **ALIGN** and **MOVE** buttons are assigned auto-alignment sequencer commands associated with the mirror selected in the far left hand selection list of mirrors. The center list of mirrors is an indication of the current state of the motion control for each mirror (optic). This allows the user to see if any of the motion control is in a **FAULTED** state indicating a malfunction. Thus for auto-alignment the use of the interface is very simple: one first selects a mirror and then presses **INIT**; when completion is indicated a sequencer state of **INPOS** is displayed, one pushes the **ALIGN** button and the mirror is aligned when the state of the sequencer indicates **ALIGNED**. The **MOVE** button is for manually moving a mirror while observing the move of the alignment target on the canvas widget. This functionality was included for testing. To the right of the **MOVE** button are two entry boxes and sliders for the user to enter an absolute move position in pixel units. The move of the mirror will translate an alignment target position, as seen on the alignment camera by the commanded number of pixels thus moving the associated mirror.

In the above discussion, aligner.tcl is in automatic mode. When it is switched to manual mode, the sequencer control buttons, position entry boxes and sliders all gray out and are not available to the user. The top menus that were grayed out are now activated for individual control of optical position movement, alignment stimulus targets and monitoring of sensors. There are four menu items at the top of the aligner.tcl main panel; they are *Align*, *Sensor*, *Actuator*, and *Stimulus*. The *Align* menu is used to set the mode of operation - either manual or automatic - and one can transition to a simulation mode. The simulation mode allows use of aligner.tcl without actually modifying Keywords and is included for debugging when hardware is not present. The *Sensor* menu controls operation of the frame grabbing and centroiding routines in manual mode. The user can turn on and off continuous centroid updates of targets or trigger the grabbing of a single frame from the EPICS system to observe the alignment camera's image on the canvas widget. By turning on continuous centroid updates an alignment target is continuously tracked, and its position is displayed on the canvas and in an entry box to the left of the canvas. A particular alignment camera is selected from the list presented in the far right list box showing the choices K1.CCDP and K2.CCDP.

The *Actuator* menu allows manual control of every actuated optical component within the system that is not under fast closed loop servo control (by RTC software). Under the actuator menu an external engineering GUI for Newport 850G motors that control position of large transport optic mirrors can be launched. Child Tcl windows to control such things as Picomotors, retro-reflector slides, shutters, and camera focus have also been implemented within Tcl using the Keyword API approach.

Finally the *Stimulus* menu controls all alignment stimulus sources. These include such things as LED alignment targets, and a boresight laser attenuator that controls reference boresite intensity. In addition white light source and constant term (CT) metrology panels are implemented for alignment of the fringe tracking camera and the launcher for the CT metrology respectively.

The entire implementation of the aligner.tcl UI was done using the Tk toolkit of graphic widgets discussed in reference [8]. This same toolkit was also used for seq.tcl to be discussed next.

4. SEQUENCING THE INTERFEROMETER AND THE SEQ.TCL SCRIPT

Astronomical interferometers are complex control-intensive instruments that for normal operation require intricate and repeatable sequencing of top-level events for reliable science operation. Development of sequencer software for the Keck Interferometer started by utilizing a UNIX implementation of the EPICS sequencer. The initial version enabled tracking of stellar targets by generating estimates of sidereal delay and feeding these estimates into optical delay lines. These estimates of optical delay are required so each of the instrument's optical paths will be approximately equal to achieve interference fringes for a particular star. The estimates are computed using several libraries that require universal time (UT), interferometer baseline vector coordinates, and a catalog of nominal star position information. This was a quick first implementation of sequencing the interferometer to achieve first fringes on time and within budget. To allow simple use of the initial implementation of the sequencer by astronomers, a rapidly developed UI called seq.tcl was developed. It was developed initially as a KTcl-only application. More recently the sequencer has evolved into object-oriented C++ code generated using a commercial CASE tool product called Rhapsody**. This more recent implementation controls other subsystems within the interferometer for complete science observations. The seq.tcl application has evolved to support this more complex sequencer and is shown in Fig. 3.

** More information on the Rhapsody product can be found at <http://www.ilogix.com>.

The initial version of seq.tcl was developed using only the Tk extension to command and monitor a UNIX implementation of the EPICS sequencer via Keywords. At that time neither the “Keck Angle Tracker” nor the “Fringe Tracker and FDLs” panels shown in Fig. 3 were present. These panels were added while the seq.tcl application was integrated with the EPICS-only sequencer. The panels monitor CORBA telemetry from RTC objects that control the Angle Tracker, Fringe Tracker and Fast Delay Lines (FDLs). The Angle Tracker is a specialized camera system that keeps the star within the interferometer’s optical field of view for beam combining. The FDLs adjust path delay at a high rate to maintain equal optical path length for fringes. The Fringe Tracker is a combination of camera, beam combiner optics, and software control that track a fringe peak and send high precision corrections to the FDLs to maintain fringes during actual operation. These telemetry monitor panels provided our first combined use of Keyword and CORBA within seq.tcl. CORBA interfacing was accomplished using the Combat extension (see reference [9]). Combat will be discussed more in the next section when a simple client-side CORBA Tcl script example is presented.

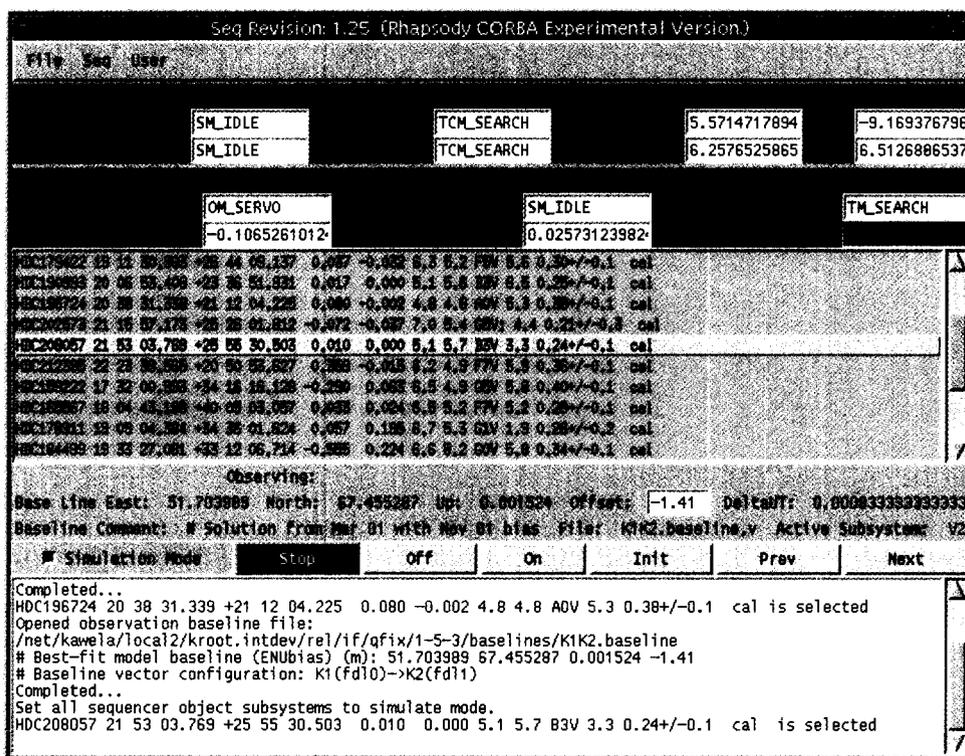


Figure 3. Interferometer Sequencer User Interface Panel (seq.tcl)

Now let us return to Fig. 3 and describe the functionality of the panel from the top to bottom. When the panel is initially launched the monitor panels are not visible and the list widget with all the HD star names is adjacent to the pull down menu items. The monitor panels shown in Fig. 3 can be turned off (collapsed so not visible) or turned on as panels (shown) and/or as separate windows. In addition similar panels (not shown) have been implemented to monitor the W. M. Keck telescope and adaptive optics systems using Keyword monitors.

There are three menu items at the top of the window: *File*, *Seq* and *User*. Under the *File* menu there are options to bring up a standard Tk library file system browser/selector window for loading either a star catalog or baseline data file. The star catalog contains a single line ASCII record for each star containing the star designation (the HD number to the left in the list widget of Fig. 3), coordinates (RA and Dec.) and other stellar parameters. After the file is loaded each of these lines is displayed in the list widget of seq.tcl. Previously grayed out buttons are now enabled. The other file loaded is a baseline file that contains baseline vector orientation coordinates and a comment line. The baseline vector is a vector from the Keck I telescope to the Keck II telescope. Both lines of the baseline file are displayed in the middle of the panel just above the buttons. The “Observing:” label displays the star catalog record just after observing has started for this object. In Fig. 3 the sequencer has not been initialized or turned on yet so no star is being observed and that line is still blank.

The *Seq* menu item contains options for the control of various parts of the overall sequencer. A series of control panel windows can be opened from this menu item. Each control panel makes CORBA method calls to command various state machine objects within the C++ sequencer framework. The control panels also monitor RTC-style telemetry generated from both the embedded RTC system and the C++ sequencer to monitor the state of operation. The *Seq* menu

also is used to select the current “Active Subsystem”. This selects which of several state machines are commanded by the **Stop**, **Off**, **On**, **Init**, **Prev** and **Next** buttons. In Fig. 3 the selected state machine controlled is the visibility squared science sequencer, the “V2”. Within our new sequencing infrastructure there are two types of state machines, those that control high-level science operations and those that control hardware sub-systems. The high-level state-machines control the hardware sub-system state machines via an internal C++ messaging scheme. All state-machine objects have a CORBA commanding interface so they can be run independently or together for testing. In the future there will be multiple types of science state machines and more hardware subsystems.

All of the subsystem state machines have a basic commanding scheme that is compatible with the row of buttons implemented on seq.tcl. The **Stop** button causes a state machine to go into a STANDBY or halted state that for most hardware systems is an initial state. The **On** buttons turns on a state machine; causing the state machine to go through an initialize state to set up an initial hardware condition and then into a STANDBY state. The **Off** button turns off the state machine causing the state machine to go through a special shutdown state to bring hardware back to a starting quiescent state. The **Init** button causes the active subsystem to do its main sequence of events for a nominal observation. For example, when the “V2” state machine is selected, pushing the **Init** button causes a complete observation sequence to happen. However, if the FDL is the active subsystem, the **Init** button causes only updated sidereal targets to be generated and fed to FDLs; no other control actions are executed. The **Init** button is tied to the selection of the star record in the list widget since this is the main input parameter for observations of any form. Also the **Prev** and **Next** button are the same functionality as the **Init** button but the selected star is either decremented (selection moved up one) or incremented (selection moved down one) in the list.

To the left of the buttons is a checkbox that enables a simulation mode within the sequencer so that all state transition sequences can be tested without required hardware to be present. Directly under the buttons is a general text widget display; primarily used to observe standard commanding status messages sent to and received from the C++ sequencer via RTC style telemetry. There is also the capability to turn on and off a telemetry monitor for each individual state machine to be displayed in the text widget.

Finally there is a *User* menu item that turns on and off the telemetry and keyword monitor panels and windows discussed above.

5. CORBA AND SCRIPTING LANGUAGE EXTENTIONS

Figure 1 shows the various software layers of both Keyword and CORBA software within a Tcl script. The Keywords are essentially networked variables that are set to values or monitored. Keywords are a procedural style of interface. However, CORBA is object oriented in that remote objects are defined in some language (not necessarily a scripting language). Remote calls are made and propagated over TCP/IP networks via the CORBA infrastructure to server objects for execution. RTC hardware control modules are server-side CORBA objects and are commanded in this manner. Each state machine object implemented within the C++ sequencer discussed above is a server-side CORBA object also commanded this way. To monitor activities we utilize one of the CORBA services called the Event Service that allows data to be pushed and pulled through a networked channel rather than using the remote procedure call model used within server objects. This allows for asynchronous data transfer. This is how we perform our monitoring of RTC telemetry in seq.tcl discussed above.

Before we continue it is important to fully understand the layers within Fig. 1. The Keyword layers are discussed in Section 2. CORBA is a language-independent standard distributed object oriented framework and allows one to implement various pieces of an application in various languages across networked machines. It allows applications written in various languages to talk to each other. These applications use an Object Request Broker (ORB) to mediate interaction between distributed components. Because CORBA is standardized there can be many different implementations of ORBs in many languages and they will be interoperable if compliant with the standard. The communications between applications performed over a TCP/IP network use a standard protocol called the Internet Inter-ORB Protocol (IIOP). It is the protocol that makes all ORBs interoperable.

But in practice how does one actually utilize CORBA? This is done through a special language that is part of the CORBA standard. The language is called the Interface Definition Language (IDL) and is used to define the interface (e.g. various methods and variables) used to communicate with server-side distributed objects. IDL code is typically compiled with an ORB-specific compiler to yield stubs and a skeleton. The stubs and skeleton are an infrastructure generated in whatever language you are working with. Stubs are utilized to build client-side applications and the skeleton is used as the basis for CORBA server objects.

Returning to Fig. 1, note that the Tcl CORBA implementations layers are shown on the right side of the pyramid. The actual script uses an extension called Combat that is an API layer to a CORBA ORB called Mico (one of several open-source ORBs). We next indicate an IDL layer that, in the case of Tcl, is not compiled into either stubs or a skeleton but is actually compiled into a static interface repository specification that is in the form of native Tcl. Interface repositories are another feature of CORBA and allow the interface to be defined dynamically and served to applications. Combat uses this facility rather than the stubs and skeletons. However most other languages use the stubs and skeletons – including the Python example we will discuss shortly. The important thing is that just below the IDL in Fig. 1 are the sequencer and RTC applications that have IDL definitions that were used to build the Tcl applications on. The RTC software and the C++ sequencer are both built using the real-time TAO ORB and it has been our experience that TAO and Mico have been fully interoperable for our remote method calls.

It is instructive at this point to consider a simple example of an interface defined for an object called *TrackerModule*. The IDL for *TrackerModule* is shown in Fig. 4. The object contains two methods called *Idle* and *Track* and neither have any arguments or return values. As can be seen the definition of methods in an IDL file uses syntax very similar to C++ prototyping of methods. Note the functionality of these methods are not specified here, only the interface. The fully functional methods will be implemented in the server-side object. This short IDL example will be used for the remainder of the paper as the basis for both client and server-side script examples.

```
module TrackerModule {
    interface Tracker {
        exception InternalError {};

        // Brings the object to a stop and leaves it idle.
        void Idle() raises (InternalError);

        // Initiates tracking behavior for this object.
        void Track () raises (InternalError);
    };
};
```

Figure 4. Interface Definition Language (IDL) Example of TrackerModule Object.

6. CLIENT SIDE CORBA SCRIPTING

The Interferometer testbeds at JPL have started to use advanced scripting languages such as Perl and Python for small applications that rely on CORBA. The seq.tcl script discussed above is actually a combined CORBA client and server implementation. All the commanding of C++ sequencer objects is done by the Tcl scripts client implementation. The server-side implementation in the Tcl was required for RTC-style telemetry handling purposes. Most of the uses to date for scripting have been in the area of client-side control.

Client example scripts written in Tcl, Python and Perl are presented next, using the above IDL code. These demonstrate the ease of implementing clients in several popular scripting languages that have CORBA support. We also show an example of a server implemented in Python. This is to illustrate how one might simulate an actual control subsystem to be implemented in the future using the RTC toolkit. The idea is that the server script could be replaced with real hardware and software at a later date, particularly useful in environments with parallel development activities.

Example Tcl Client Side CORBA Script

Figure 5 shows the complete listing of a client script that calls the *Track* and *Idle* methods (with a 2 second delay in between). It assumes that there is a *TrackerModule* server already running somewhere on the network. We will discuss server-side operation in the next section. The client script first initializes the ORB, obtaining from the command line a name of a Naming Service to help locate a server. The Naming Service is another CORBA service that is used to register CORBA objects as readable names and make an association with their rather long unique Interoperable Object

```

#!/usr/bin/env combatsh
# A simple Tracker object client (client.tcl).

# Initialize CORBA ORB and set reference
# to naming service on command line.
set argv [eval corba::init $argv]

# Source the interface repository into the TCL shell
source CosNaming.tcl
combat::ir add $_ir_CosNaming
source Tracker.tcl
combat::ir add $_ir_Tracker

# Obtain naming service IOR
set n [corba::resolve_initial_references NameService]

# Resolve object handle from naming service
set t [$n resolve {{id "spie" kind ""} {id "tracker" kind ""}}]

# Call the Track and Idle method on the object spie.tracker
# that was resolved from name service.

$t Track
after 2000
$t Idle

```

Figure 5. Tcl Client Script Example for TrackerModule Object.

Reference (IOR). The IOR is the object handle used for actually making the remote method call. Thus, the actual execution of the Tcl client script would look something like this

```
client.tcl -ORBInitRef NameService=corbaloc::machine:port/NameService
```

The `-ORBInitRef` argument allows the ORB to be assigned various parameters from the command line. In this case the Name Service “machine” and “port” number are being specified.

After initializing the ORB, the stub files `CosNaming.tcl` and `Tracker.tcl` are sourced to allow use of those interfaces. These files result from compiling IDL files using the Tcl Combat conversion script. In fact the `Tracker.tcl` file contains the IDL of Fig. 4. At this point the Tcl script now knows about what methods are available for execution but does not know about an active running object that it can call them on.

Next the CORBA package function `resolve_initial_references` is called that returns the IOR for the Naming Service object. The following line calls the `resolve` function with the name of the object to command as the argument. In this case a server has registered the object `TrackerModule` with the Naming Service as an instance named “spie.tracker”.^{***} The `resolve` call returns the IOR that is then used to execute both the `Track` and `Idle` methods remotely on object `spie.tracker`.

Example Python and Perl Client Side CORBA Scripts

Figures 6 and 7 present exactly the same example as Fig. 5, but coded in Python and Perl respectively. The Python script is implemented using the `ommiORBpy`¹⁰ ORB; a Python extension for CORBA implementation within Python. The Perl uses the `ORBit` ORB¹¹ and Perl bindings.

^{***} The dot notation used here is a custom notation for object names registered within the Name Service. Name service object naming notation as used within our software and name representation within the Name Service is discussed in Section 7.

```

#!/usr/bin/env python
# A Simple TrackerModule client

import sys, time
from omniORB import CORBA
import CosNaming
import TrackerModule

# Initialize CORBA ORB and set reference
# to naming service on command line.
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)

# Obtain naming service IOR
obj = orb.resolve_initial_references("NameService")
ns = obj._narrow(CosNaming.NamingContext)

# Resolve object handle from naming service
name = [CosNaming.NameComponent(id = "spie", kind = "")]
name += [CosNaming.NameComponent(id = "tracker", kind = "")]
obj = ns.resolve(name)
tr = obj._narrow(TrackerModule.Tracker)

# Call the Track and Idle method on the object spie.tracker
# that was resolved from name service.
tr.Track()
time.sleep(2)
tr.Idle()

```

Figure 6. Python Client Script Example for TrackerModule Object.

```

#!/usr/bin/env perl
# A Simple TrackerModule client
# ./client.pl -ORBNamingIOR=IOR:...

use CORBA::ORBit idl => [ qw(/usr/include/orbsvcs/CosNaming.idl) ];
use CORBA::ORBit idl => [ qw(TrackerModule.idl) ];
use strict;
# Ensure that we enable TCP/IP communications for the ORB
unshift @ARGV, "-ORBIIOPv4=1";

# Initialize CORBA ORB and set reference
# to naming service on command line.
my $orb = CORBA::ORB_init(@ARGV);

# Obtain naming service IOR
my $ns = $orb->resolve_initial_references("NameService");
my $nsior = $orb->object_to_string($ns);

# Resolve object handle from naming service
my $name = [{'id' => 'spie', 'kind' => ''}, {'id' => 'tracker', 'kind' => ''}];
my $Tracker = $ns->resolve($name);

# Call the Track and Idle method on the object spie.tracker
# that was resolved from name service.
$Tracker->Track;
sleep 2;
$Tracker->Idle;

```

Figure 7. Perl Client Script Example for TrackerModule Object.

Every language handles the IDL by preprocessing or precompiling. Tcl uses the Combat extension to compile the IDL into an internal interface definition. In Python the omniORB IDL compiler generates native Python stubs and skeleton code. Similarly C++ uses a preprocessor to generate full stubs and skeleton. In the ORBit Perl implementation the IDL file is simply read by the script directly and compiled dynamically.

7. A CORBA SERVER SIDE SCRIPT

We have presented simple examples of client-side CORBA scripts. This section will present an example server implementation of the *TrackerModule* object shown in Fig. 4. This example is coded as a Python script (Fig. 8), but the Perl and Tcl implementations are similar. While Python and Perl have object-oriented features built into the languages Tcl does not; a Tcl CORBA server script requires the use of `incrTcl`^{****} an object oriented Tcl extension, to allow definition of the implementation class that defines functionality of the IDL methods.

```
#!/usr/bin/env python
# Simple TrackerModule server

import sys
from omniORB import CORBA
import CosNaming
from RTC import CorbaUtils
import TrackerModule, TrackerModule__POA

class TrackerObject(TrackerModule__POA.Tracker):
    def Idle(self):
        print "Tracker is now idling"

    def Track(self):
        print "Tracker is now tracking"

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
obj = orb.resolve_initial_references("NameService")
ns = obj._narrow(CosNaming.NamingContext)

poa = orb.resolve_initial_references("RootPOA")
poa._get_the_POAManager().activate()

tr = TrackerObject()
ref = tr._this()

name = CorbaUtils.string_to_name("spie.tracker")
CorbaUtils.create_reference(ns, name, ref)

orb.run()
```

Figure 8. A Python Example Implementation of TrackerModule Server.

Examining the server-side script code of Fig. 8 shows that initially several modules are imported; a `sys` module for command line argument parsing, several modules required for CORBA support (`omniORB`, `CosNaming`), and finally the stub (*TrackerModule*) and skeleton (*TrackerModule__POA*) code generated from the IDL compiler.

In CORBA servers there is typically a software component called the Portable Object Adaptor (POA) that is responsible for managing new instances of server objects, activating and deactivating them and assigning object references. The POA may have specific user selectable-behaviors, called policies, to achieve various effects. The POA is the component of the ORB responsible for managing individual CORBA objects. The POA provides a mechanism of control over the server object to interface it with the IIOP Internet communications. Remote calls to methods are made from client ORB to server ORB via IIOP; the POA then communicates with the server ORB to dispatch events to the implementation object. In Fig. 8 the implementation object is the *TrackerObject* class where both the *Track* and *Idle* methods are implemented. Notice that the implementation inherits from both the stub and skeleton generated from the original IDL.

Directly after the implementation class is defined, the CORBA ORB is initialized, and a Naming Service IOR is obtained as was done in the client-side example (Fig. 6). The POA is registered with the ORB and activated with the two lines

```
poa = orb.resolve_initial_references("RootPOA")
poa._get_the_POAManager().activate()
```

^{****} More information on `incrTcl` and the source and executable is available at <http://incrtcl.sourceforge.net/itcl/>.

Now the ORB is active for remote calls from a client, however, an instance of *TrackerObject* has not actually been created. The next line creates the instance, implicitly activates the object, and returns a handle to the *TrackerObject*. The *TrackerObject* reference is not yet available to remote client scripts. Next a call made to *this()* to obtain the actual object reference, which is used for remote access of the object from the client-side program. The *TrackerObject* IOR is now valid for use by clients. The next several lines register the object with a Naming Service by the name "spie.tracker". These routines are not part of the Python CORBA support but rather a custom package written for JPL internal use (discussed later). The very last call is *orb.run()*; this blocks to keep the server object alive so that the script does not just exit. Since *omniORB* is multithreaded the script would have just exited and the thread destroyed, if other functionality of the script had been in place to block the main script then the *orb.run()* call would not have been needed.

Before we complete the discussion, let's return to an explanation of what is going on with the Naming Service in Fig. 8. The two lines

```
name = CorbaUtils.string_to_name("spie.tracker")
CorbaUtils.create_reference(ns, name, ref)
```

are methods within a custom class (*CorbaUtils*) that is a set of CORBA utility routines developed at JPL. Recall that the purpose of the CORBA Naming Service is to maintain a mapping of names to object references (IORs). The names are represented in the Naming Service as an "acyclic graph", but are more typically used in a hierarchical naming style. This hierarchical naming style is analogous to a DNS machine name on the Internet. There are two types of Name Service components (names), a context node and an object reference. A context node is an object that stores other context nodes and/or object references. An object reference is directly associated with a CORBA server (holding the server's IOR). Thus our object name "spie.tracker" can be thought of as first specifying a spie container (the context) with a tracker object under it. Notice the convention is from left to right rather than the right to left convention used in DNS. The CORBA standard does not put restrictions on the characters that can be used nor does it specify a dedicated separator character for names. The names are defined as a sequence (or variable length container array) of *NameComponent* types. Each name represents a node in the graph. The *NameComponent* type (defined in *CosNaming.idl*) contains *id* and *kind* members as used in the client examples. The *id* is the string name element and the *kind* is an optional description. Thus the utility method called *CorbaUtils.string_to_name* performs a translation from the "spie.tracker" string to a sequence where the naming context is "spie" and the reference name is "tracker". The *CorbaUtils.create_reference* method then iterates through the *NameComponent* sequence creating the context and reference items in the Naming Service. Clients can use this sequence to resolve IORs as shown in Figs. 5, 6 and 7, above. The CORBA calls *bind* and *rebind* are used from within *CorbaUtils.create_reference*. The "." in the string name is an arbitrary separator that we have chosen.

8. CONCLUSION

Two Tcl scripts written as part of the Keck Interferometer development effort were reviewed. They were both initially developed using the legacy Keyword/Value methodology developed by the W. M. Keck Observatory and used on most of the major telescope subsystems and instruments. Later CORBA functionality was added to the *seq.tcl* script that provided an integrated environment where combined Keyword and CORBA development could be rapidly carried out.

We presented examples of client-side CORBA scripts written in Tcl, Perl, and Python, and a Python CORBA server script. Tcl has proved to be a wonderful development environment for rapid prototyping of small to medium sized UI applications with limited functionality. The technology of scripting languages and extensions for them is continually evolving. Python is particularly well suited for rapid development of CORBA compliant functionality since it was designed from the start as an object-oriented scripting language and has a CORBA standard language mapping. JPL has several testbeds in development to demonstrate future space interferometry technology, these have begun to utilize Perl and Python script language technology. In the future scripting technologies will be used to a greater extent to create applications for the test and characterization of interferometer testbed configurations and ground-based instruments.

9. ACKNOWLEDGEMENT

The work performed here was conducted at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. The authors would like to thank the following people who contributed to this effort: Mark Colavita and Andrew Booth for helpful definition and support during the development of both the Keck Interferometer applications discussed here; Kevin Tsubota for his assistance resolving W. M. Keck Observatory software issues; and Robert Smythe for his continued moral support and assistance with a large variety of hardware gizmos.

10. SOURCE CODE

Complete source code of the example scripts presented in this paper can be obtained by email request sent to Leonard.J.Redder@jpl.nasa.gov.

11. REFERENCES

1. W. Lupton, Tcl/Tk/KTL Interface, W. M. Keck Observatory, Kamuela, HI. 27 April 2000.
2. Andrew Booth, et. El., Overview of the control system for the Keck Interferometer, SPIE Advanced Telescope and Instrumentation Control Software II Conference 4848, Waikoloa, HI. August 2002.
3. T. Lockhart, RTC: a distributed real-time control system toolkit, SPIE Advanced Telescope and Instrumentation Control Software II Conference 4848, Waikoloa, HI. August 2002.
4. M. Henning, S. Vinoski, Advanced CORBA Programming with C++, Addison-Wesley Professional Computing Series, 1999.
5. A. R. Conrad, W. F. Lupton, The Keck-Keyword Layer, Astronomical Society of the Pacific Conference Series: Astronomical Data Analysis Software and Systems II, Vol. 52, pp.203-207.
6. W. F. Lupton, A. R. Conrad, The Keck Task Library (KTL), Astronomical Society of the Pacific Conference Series: Astronomical Data Analysis Software and Systems II, Vol. 52, pp.215-320.
7. G.T. van Belle, M. Colavita, et. El., Keck Interferometer autoaligner, SPIE Advanced Telescope and Instrumentation Control Interferometry for Optical Astronomy II Conference 4838, Waikoloa, HI. August 2002.
8. Brent B. Welch, Practical Programming in Tcl and Tk Third Edition, Prentice Hall, N.J. 1999.
9. Frank Pilhofer, Combat, <http://www.fpx.de/Combat>, December 2001.
10. Duncan Grisby, The omniORBpy version 1.3 User's Guide, AT&T Laboratories Cambridge, <http://www.uk.research.att.com/omniORB/omniORBpy/>, August 2000.
11. Owen Taylor, Perl bindings for ORBit, <http://people.redhat.com/otaylor/corba/orbit.html>.