Introducing Tango CS Framework: A Python Approach

Nick Caldwell

September 22, 2015

Abstract

The purpose of this document is to support the initial discovery of the Tango Control System framework. By way of developing remote control abilities to a simplistic software defined Shift Register, we will introduce the fundemental concepts and support tools. In this manner, the reader should gain the knowledge to implement their own control system and be empowered to continue the discovery of the Tango CS framework.

1 Introduction

The intention is that this paper will serve those who are new to the Tango framework, as a practical approach to gaining exposure to the fundementals of the system and just some of its supporting tools. As such, it should act as an introduction, to build the readers confidence to continue further exploration. A brief overview of the Tango system will follow, at which point one should take the time to build their own knowledge base of the concepts that are mentioned. The document will continue and walk through the design, development and testing of a Tango control system for a software defined device. At the system level and at each component stage, the design, implementation details and test conditions will be detailed.

2 Learning Outcomes

For the most part, the following sections will find support within the Tango framwork to execute the individual tasks in implementing a remote controllable software shift register. These tasks serve the higher level learning outcomes of the fundemental Tango CS concepts and discovering some of the accompanying support tools.

The fundemental concepts/objects of the Tango CS will be covered along with how access is provided through the core Application Programming Interface (API). Specifically the API extensions developed for Python support will be detailed. How the core supports error handling though the exceptions API and logging service, will be detailed through its use in our development. Visibility on the vast array of tool support within the framework will be enhanced by introducing a few of those that are available to the Python developer.

3 Background

3.1 Tango Overview

The Tango Control framework [1] is an open source software supporting the development of distributed control systems on heterogeneous software platforms. It has implemented its remoting system on a distributed object architecture, and as a result permits program communication irrespective of programming language, platform or network. The architecture referred to is CORBA, Common Object Request Broker Architecture [2].

CORBA A distributed obect system is an approach to simulate single address space systems to tackle the problems of remoting, however solving remoting has since taken on service oriented and RESTful API approaches [3, 4] having identified flaws in the

single address space solution. Having said that, the Tango framework has tackled the traditional drawbacks such as programming complexity and weak deployment support with much success.

At a high level, CORBA implementations such as Tango, expose interfaces via a broker/ORB (Obejct Request Broker) to support remote method invocation on objects that may be distributed across a network. The ORB can offer location transparency to the client and data marshalling. To reference an object, the client making the request uses an IOR (Interoperable Object Reference), usually a structure containing remote host/process and object/method identifiers.

Advantages that are inherent in the CORBA architecture and as a result a Tango CS implementation are:

- Extensive language mappings offer precision to the data coming on and off the network
- Interface Definition Language (IDL) service descriptions decouples design and implementation
- Strong Typing supports development with early error detection
- Object Orientated (OO) Architecture abstracts underlying distributed complexity, while supporting OO design principles, tools and techniques

3.2 Tango Device Server Object Model (TDSOM)

Conceptually TDSOM [5] consists of only four basic entities, the *device*, the *server*, the *database* and the *client*. The Tango model exposes devices to clients via a common interface served by the server. Its attributes, properties and commands are locatable by a network unique namespace of the form *<domain>/<family>/<member>*.

The Device This is the key Tango concept, a device class associated with a particular piece of hardware (normally), describes the device behaviour. It can be uniquely identified by the device namespace of the form *[FACILITY/]DOMAIN/CLASS/MEMBER*. Deriving from a common device class that implements the same CORBA interface, the device when instantiated will expose common CORBA operations and attributes. The device class should be designed according the guidelines [6] supporting object reusability that can and has been taken advantage of by the Tango community.

The Server Implementing the ORB concept, the server (aka Device Server) instantiates one or more instances of one or more device classes. The server listens for client requests on behalf of the devices that it hosts, routing requests and responses as appropriate.

The Database As an IOR repository, TANGO CS implemented a *'property database'* using a MySQL [7] relational database, it itself represented as a Tango device also. It is used as a registry of the device classes, instances and device network locations hosted by a device server, to be queried primarily by Tango clients. It is also used to store configuration properties for devices, providing startup configurations to device servers for their device instances.

The Client Using a proxy the client performs requests on the devices in a location transparent manner. Making use of the Tango CS API, rich clients can be developed abstracted away from some of the more complex communication programming.

3.3 Development Support

Core API PyTango [8] exposes the core Tango API, access exists to the TDSOM objects along with added utilities, data typing and error handling support.

Error Handling Every distributed system is susceptable to failures, and Tango ensures more accurate troubleshooting and efficient maintenance via it's standard approach and purpose built API for errors.

Above system level exceptions, the Tango framework provides for the DevFailed exception defined in the IDL [9], for typical try/except error handling. A DevFailed exception is an array (variable length) of DevError objects each containing the following four fields:

- Reason: (type=str) Readable string of error type
- Description: (type=str) Plain text description of error
- Origin: (type=str) Code location (class and function) that threw the error
- Severity Level: (type=PyTango.ErrSeverity) One of WARN, ERR or PANIC giving a severity level to the error

The PyTango Exception API [10] goes further to provide more specific exception types derived from the DevFailed exception, however is not covered here.

Logging Using the Tango Logging Service (TLS[5]), we can manage the direction and consumption of log messages, giving visibility and possible vital debugging information.

Logs can be directed to 'Logging Targets' of type:

- CONSOLE: standard console output
- FILE: XML formatted file on filesystem
- DEVICE: a device implementing the Tango log consumer interface

Log levels act as filters on the log producers to control transmissions to such a target. These may be of DEBUG, INFO, WARN, ERROR, FATAL or OFF value. Thus, a log of equal or higher severity to that assigned to a target will be transmitted.

The PyTango module provides log production methods of particular severity namely *debug_stream, info_stream, warn_stream, error_stream* and *fatal_stream*. Each takes the message string to be logged, and delivers to the configured targets with the severity level and device namespace. Further support exists through method decorators that will produce logs as the execution enters and leaves the method.

PyTango.server module The PyTango module provides the developer with the Tango API, but its the PyTango.server [11] module that lowers the barrier to entry to creating Tango control systems. Doing so in a modular fashion with a simple/obvious programming syntax will prove popular with developers of all skill levels.

Taurus Client Support Tools Here is a brief description of just a subset of the tools provided within the Taurus project [12].

TaurusForm: This widget offers a standalone application which is ready to use to view and edit attribues. From a terminal, provide TaurusForm with a list of attribute model URIs and a UI form will be presented. Thus we could for our purposes do:

```
~$ taurusform virtual/shift_register/test1/registerValue
```

The GUI presented can be further customized with a right-click and modifying the form in a drag and drop fashion.

TaurusDesigner: Custom clients may be required for projects, and for that we can utilize the QT designer with Taurus widgets. The drag and drop capability of Taurus (and Qt standard) widgets combined with the ability to compose your own, makes this a fast and extensible tool.

TaurusGui: A code-free graphical solution for building Taurus user interfaces, TaurusGui offers a highly functional and simple to use wizard like tool.

4 Preparation

If not already done, any concept in the previous section that isn't fully understood should be rectified using the reference documents.

4.1 Framework Installation

The development in this document was carried out on (but not restricted to) the following software infrastructure:

Ubuntu 14.04 LTS

MySql Server & Client 5.5

OpenJDK 7 (full development kit)
Python 2.7

4.1.1 Tango Controls Kernel

Installation intructions for Tango Controls can be found online [13]. Before which one should have at hand the administrator password for the MySql server, and be prepared with a value for TANGO_HOST. This value, of the form *<hostname>:<port>* , is key to the installation detailing the location of the Tango database service(s).

5 Shift Register Device

We will implement a basic Shift Register device, a representation of a Serial-In, Parallel-Out (SIPO) Shift Register [14], allowing us to keep our attention on the Tango Controls framework. It will be restricted to the ability of shifting in a single bit into and reading out the register value.

5.1 Design

Dividing the implementation of the actual register and the device class that will define the Tango interface, we seperate the concerns of the register implementation and how it interacts with the distributed system (the Tango part). See now the accompanying code files *ShiftRegister.py* and *ShiftRegisterDS.py*.

The register an object called ShiftRegister, holds a string value of n chars to represent an n-length shift register. The initializer method takes an optional integer as input, allowing the register to be initialezed to a given size. The ShiftRegister class has just two other methods, *'readRegister'* returning the registers string value, and the other *'shiftBit'*. *'shiftBit'* takes a char to be appended to the end of the register string following the first chars removal. In this fashion it simlutes the shifting of chars from right to left.

The device class inherits from PyTango.server.Device, exposing the read-only attribute *'RegisterValue'* and command *'ShiftBit'* to shift the register. Each calling upon the ShiftRegister object to carry out the functionality. The command incorporates logging and error handling, correcting the default state and status attributes as the command is executed.

Handling errors of invalid input to the command, a DevFailed exception is thrown with 'ERR' severity and appropriate reason, description and origin strings. The device will enter a FAULT state if not already, and continue to be until the register no longer holds the possibility of a bad bit value.

Logging for the purposes of this simple project is left to the default CONSOLE target, enabled by providing the verbose (-v4) argument when starting the device server. The number 4 here configures a log level of DEBUG for the console target.

Design guidelines [6] having been considered, we have designed a generic interface, used a standard naming & nomenclature structure, increasing clarity and the possibility of class reuse.

5.2 Development

Development Support There exists a code generator called POGO [15] for device classes/servers, supporting Java, C++ and Python. However, PyTango has been extended with the PyTango.server module to simplify the manual development.

If one compares equivalent devices developed using POGO and PyTango.server, there's a stark contrast of the minimal style of the higher level PyTango.server module, traditional to Python design. To that of the generated code of POGO, that aims to mimic the original C++ Tango API structure. For this reason we will continue with the manual development with PyTango.

Coding Using the PyTango.server module the development time is minimal. Inheriting from the PyTango.server.Device class and through the use of device decorators that abstract the Tango funactionality, programming is quick and clean.

The filename is restricted to that of the server name ShiftRegisterDS, that will be used a little later to test the server. Within the file we will import the modules necessary for the device class definition and *'server_run'* module for the device server. This specific modeule is used exclusively in our main function at the end of the file, it is the server loop and takes a class list as a parameter, namely the device classes to host.

The device class defined ShiftRegisterDevice acts to connect the ShiftRegister object previously written, declaring how the server will expose it. Thus the attribute, command and device property are defined here paying attention to the data types, exception handling and logging.

Testing Interpreted languages such as Python offer faster development cycles over compiled alternatives. A drawback is however that the risk of bugs reaching production code increases. Development that is closely integrated with testing is a strong defense to this risk.

To this end testing should be performed throughout the develeopment cycle. Even at this early stage of our development we should take advantage of the *unittest* module [16] of the standard Python library. Its tools offer support to the programmer for constructing and performing unit tests. Accompanied with this document, one will find test cases ensuring the basic functionality of the device server development.

Before we run, the device server needs first to register with the Tango database. Using the small python script given, we complete the simplest of DbDevInfo objects and register it with the database using the PyTango.Database API.

To start the device server we can run the following command from the project directory:

```
~$ python ShiftRegisterDS.py test -v4
```

We can now interact with the server, through the use of a DeviceProxy object, and call on the devices functionality. This can be seen in the test script that verifies the functionality of the server and device combination.

This test case is essentially our first client program, but we shall follow up with a user interface using the support tools for Graphical UI support.

6 Shift Register Client

To develop the client to our ShiftRegister device, we will take advantage of the Taurus framework. Developed on top of PyTango (originally specifically for Tango CS systems), the Taurus API and tool suite empowers both programmers and non-programmers alike, to create feature rich user interfaces.

Taurus Core Support provides the developer with Tango models like Database, Device & Attribute. These differ in functionality to the models of PyTango that we have seen earlier. A model request is accompanied with a URI to uniqely identify the Tango object, is handled by a Taurus singleton which resolves the URI with some degree of intelligence, returning the Taurus object representation (itself a singleton object). As a result, intereacting with the Taurus objects should feel more intuitive and be less error prone to that of PyTango objects.

6.1 Design

Using the Taurus API, we can quickly develop a GUI in just a couple of Python lines. In the code provided, you will see the UI makes use of TaurusForm and TaurusCommandsForm [17], to present the attributes and commands respectively. In each case, we create the object, set the model and apply it to our Qt layout. In reality, we are merely connecting the appropriate model, view and controllers togethers for our needs. The Taurus framework has made the developers job quite straightforward.

6.2 Development

Accompanying this this text is *ClientShiftRegister.py*, the Taurus client development. If your client machine has the TANGO_HOST variable set, then like the code reads, the URIs may take their shortened form. Otherwise the Taurus documentation describes the

URI format in depth. Both of our forms created display some of the configurability via the API, namely the ability to hide/show form buttons on the TaurusForm and the ability to provide a function to filter the visible commands of the TaurusCommandsForm. This only scratches the surface, further features and options can be found in the Taurus developers guide [17].

Testing our client against the device server developed earlier, we can perform we can verify the exception handling and features incorporated into the system. With the device server running we can start the client and immediately see the attribute values for the device. Using the ShiftBit command, if we shift in a valid value (0/1) we'll see the register value attribute update. Likewise, for an invalid value, the state will become FAULT along with the appropriate updated status attribute. And viewing the console output of the server, one can see the error messages arise with specific classifiers for severity and code location.

7 Conclusion

Our goal here was to introduce the Tango framework implementing a remote control software defined shift register. The core Tango CS architecture has provided a core device server model for distributed service access. While its sophisticated suite of tools developed for developers and designers alike, has supported the development throughout.

A simplistic design, and a narrow project scope, allowed us to cover the fundemental concepts in a self contained but not all inclusive manner, while gaining visibility to elements of the toolset. The reader should now hold a understanding of the methodology, core architecture, and awareness of the supporting framework. And as a result be empowered to continue with further exploration into Tango CS.

References

[1] http://www.tango-controls.org/

[2] http://www.omg.org/spec/CORBA/

[3] http://www.w3.org/TR/soap/

[4] Fielding, R. T.; Taylor, R. N. (2000). "Principled design of the modern Web architecture". pp. 407–416.

[5] http://www.esrf.eu/computing/cs/tango/tango_doc/kernel_doc/ds_prog/tango.html

[6] ftp://ftp.esrf.eu/pub/cs/tango/tangodesignguidelines-revision-6.pdf

[7] http://www.mysql.com/

[8] http://www.esrf.eu/computing/cs/tango/tango_doc/kernel_doc/pytango/latest/index.html

[9] http://www.esrf.eu/computing/cs/tango/tango_idl/idl_html/index.html

[10] http://www.esrf.eu/computing/cs/tango/tango_doc/kernel_doc/pytango/latest/exception.html

[11] http://www.esrf.eu/computing/cs/tango/tango_doc/kernel_doc/pytango/latest/server_api/server.html

[12] http://www.taurus-scada.org/en/stable/

[13] http://www.tango-controls.org/resources/howto/

[14] https://en.wikipedia.org/wiki/Shift_register#Serial-in.2C_parallel-out_.28SIPO.29

[15] http://www.esrf.eu/computing/cs/tango/tango_doc/tools_doc/pogo_doc/index.html

[16] https://docs.python.org/2/library/unittest.html

[17] http://www.taurus-scada.org/en/stable/devel/index.html