DIVERSE USES OF PYTHON AT DIAMOND

M. G. Abbott, T. M. Cobb, I. J. Gillingham, M. T. Heron,
Diamond Light Source, Oxfordshire, UK

Abstract

Diamond Control Systems Group has used Python for a range of control system applications. These include scripts to support use of the application build environment, client GUIs and integrated with EPICS as EPICS Channel Access servers and concentrators. This paper will present these applications and summarise our experience.

INTRODUCTION

Distributed control systems at Diamond extensively use Python scripting for clients and some applications. Three examples of this are presented here.

PYTHON CHANNEL ACCESS BINDINGS

Bindings for EPICS channel access have been implemented in Python using the `ctypes` library, which allows Python to directly call all of the C functions required to implement full EPICS client functionality. Channel access is provided through three functions: `caput`, `caget` and `camonitor`.

```
caput(pvs, values, repeat_value=False,
    timeout=5, wait=False, throw=True)
caget(pvs, timeout=5, datatype=None,
    format=FORMAT_RAW, count=0, throw=True)
camonitor(pvs, callback, events=DBE_VALUE,
    datatype=None, format=FORMAT_RAW,
    count=0, all_updates=False,
    notify_disconnect=False)
```

The `caput()` function is the simplest to describe: this writes a single value to a single PV. Python values are automatically converted into channel access (DBR) format: all channel access datatypes are supported.

The `caget()` function is a bit more tricky. Support for reading multiple PVs at once is helpful, as then the library can wait for all values in parallel (the cothread framework described below makes this particularly easy and efficient). Conversion of channel access values to Python values is reasonably straightforward—values are one of string, integer or floating point. If channel access delivers an array of values then Numeric Python [1] is used to represent the resulting array. The call to `caget()` can specify the desired data format (using either Python data types or channel access enum names).

The most interesting functionality arises if timestamps, status or control values are requested. This is done by specifying the format of the required data as one of RAW (data only), TIME (timestamps with status fields) or CTRL (all extra channel access ”control” fields). All the extra values are returned as fields of the returned value, for example:

```
signal = caget('SR21C-DI-DCCT-01:SIGNAL',
    format=FORMAT_TIME)
print signal.name, signal, signal.timestamp
```

Note that the value returned by `caget()` is, in effect, a double with extra fields: Python allows builtin types to be subclassed in this way.

Finally `camonitor()` raises the most difficult issues: callbacks delivering new data from channel access can occur at any time. There are three possible solutions: allow callbacks to occur completely asynchronously (using threads), allow callbacks to occur only when polled explicitly, or allow callbacks to automatically occur when convenient. In this library we use the last approach.

Python support for threads has the disadvantage of threads that updates can occur at any time, for example when a data structure being updated is in an inconsistent state, without the compensating advantage of being able to use multiple processor cores. The decision was made to use Python's support for coroutines provided through the Greenlet library [2]. This is very low level, so it was also necessary to write a simple coroutine scheduler on top of the greenlet library.

In the end, the channel access library (catools) has turned into the primary application of a coroutine threading library (cothread). Most users don't need to be aware of the existence of coroutines, but they are there in the background. The most basic functions provided by the coroutine library are `Spawn` (starts a new coroutine or “cothread”) and `Wait` (suspends the calling cothread until a specified event occurs). The main complication of this approach has been integration with other libraries, in particular the Qt library—currently this is implemented through polling on a timer by a dedicated cothread.

CONTROL SYSTEM USER INTERFACE

A graphical user interface has been implemented at Diamond, based on the Qt interface [3] with Python (PyQt [4]) and the channel access bindings described above. Figure 1 shows an example of this applied to a photon beam front-end. Qt/Python has the following advantages over the Extensible Display Manager (EDM [5]).

- Object Oriented Design allows for easy inheritance for adding new classes of widgets and the potential to modify the behaviour of existing ones, without the need to modify or rebuild the base libraries.

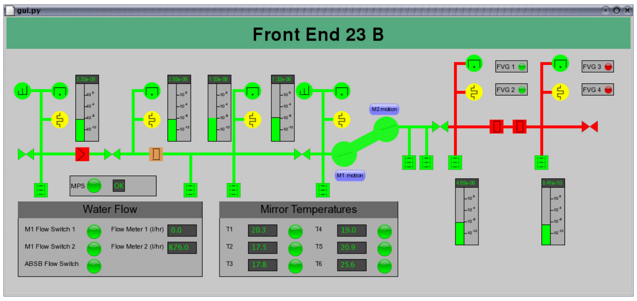

Figure 1: Example screenshot of Qt/Python application

- Good processing of large XML datasets, which we have found to be a powerful approach to application configuration.
- Scalable Vector Graphics rendering, is the basis for much of our implementation of the Qt/Python user interface. This permits easy manipulation of widgets, such as changing element fill colour, shape and position as required at run-time, through the standard Document Object Model.
- Dynamic scalability (zooming) allows the size of the user interface to be changed at will, which can significantly improve desktop real-estate, especially when multiple application windows are displayed.
- Ability at runtime, to customise a single, common application architecture, in order to support multiple systems, which are similar, but not necessarily identical (e.g. Diamond's Front Ends).
- The visualisation of complex three-dimensional scan plots and beamline geometry is being realised through Python graphical interfaces, using OpenGL.

Implementation

The class diagram in figure 2 highlights the design pattern adopted to facilitate the implemention of widgets with specific characteristics. Only three widget classes are shown as examples in this diagram.

The EPICS Channel Access (CA) interface is via the catools library described above. An example implementation (from Diamond's Front Ends interface) is abstracted in the EpicsSVGGraphic class. The application framework implements the modern Qt scene/view framework. The view layout is specified by subclassing QtGui.QGraphicsView, instantiating all widgets in the constructor, defining their positions and setting their PV identifiers.

When EpicsSVGGraphic is instantiated, it subscribes to updates of the given EPICS PV, by supplying its callback function to the CA interface. Update processing, specific to a graphical widget, is realised simply by overriding the base-class callback. For instance, a valve widget will change the fill-colour of the graphical element, depending on the new valve status.

All widgets derived from EpicsSVGGraphic, also

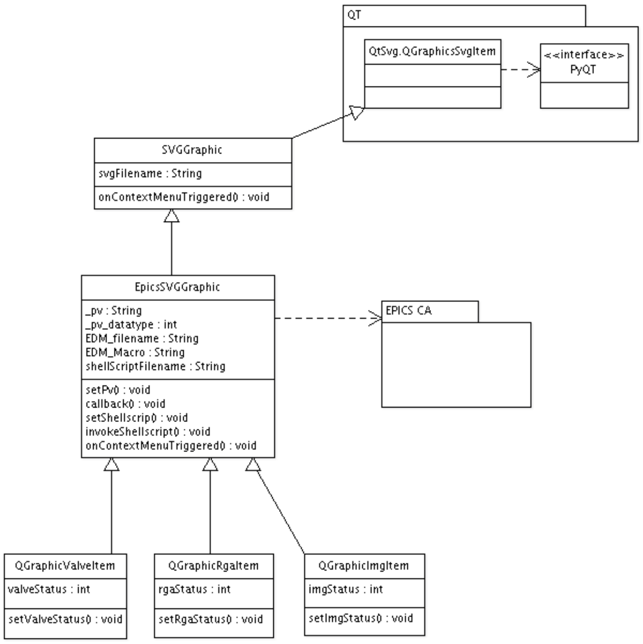

Figure 2: EPICS Qt Widget Class Diagram

inherit full clipboard copy functionality (XDND protocol), tooltips and context menus.

PYTHON IN SIMULATIONS

EPICS based photon beamlines at Diamond are increasingly using Asyn [6] as an interface layer between Device and Driver Support (figure 3 left). This abstraction allows the low level driver to be replaced with a simulation without modifying the upper levels of the structure. These simulations support early testing, not only of high level applications including EDM panels, but also core modules such as Asyn and Stream Device [7].

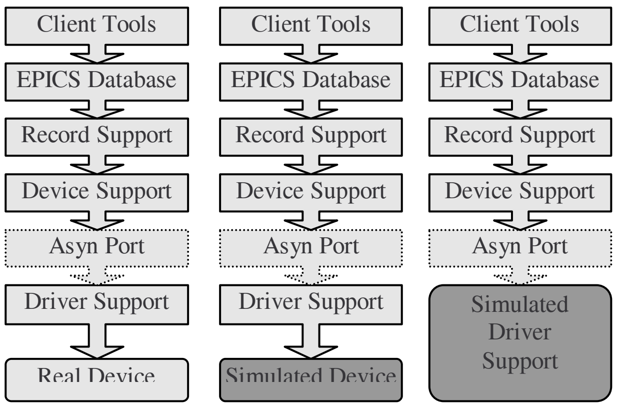

Figure 3: Asyn device structure and the two most desirable levels of simulation

The Problem

The most realistic and useful simulation is written at the lowest possible level. However, driver support is typically written in C, making simulations at that level quite time consuming.

Our Solutions

The first solution is to use the existing Driver Support but connect it to a Simulated Device (figure 3 middle). Serial devices are simulated in this way, by creating a virtual serial port at the system level and connecting a drvAsynSerialPort to it. The second solution is to replace the Driver Support with a simulation (figure 3 right). This is likely to be used for a complex device like a camera or scaler card. Both solutions can be accomplished in Python.

Creating a Simulated Device

The Python class `serial_device` wraps either a TCP server or a Linux pseudo serial port, deals with I/O and terminators, and provides scheduling functionality. The programmer is required to code a reply method suitable for the device. Figure 4 is the code for a device that has one internal value. It can be read by sending "?" and written by sending anything else. The accompanying database uses Stream Device to strip off the terminator and parse the response.

```
class my_serial(serial_device):
    # set a terminator and internal val
    Terminator = "\r\n"; val = 1
    def reply(self, command):
        # return reply to <command>
        if command=="?":
            return self.val
        else:
            self.val=command
            return "OK"
```

Figure 4: Simple serial_sim example

Creating Simulated Driver Support

The Python class `pyDrv` registers itself as an Asyn port with a variety of interfaces, provides scheduling and callback functionality and handles type conversion to and from Python native types. The programmer is required to code suitable write and read methods. Figure 5 is the code for a simple example that keeps an internal dictionary object of values, and allows access to these via a series of commands. The accompanying database has records of many types with INP or OUT links like "`@Asyn($(PORT) 0)C`".

Instances of these classes need to be created in the EPICS IOC. The "Python" command provided by pyDrv ensures the interpreter is running and executes the argument as a Python command. Figure 6 shows an excerpt

```
class my_asyn(pyDrv):
    # supported list of asyn commands
    commands = ["A","B","C","D"]
    # internal dictionary of values
    vals = {"A":1,"B":"BE","C":3.4,"D":[1,2]}
    def write(self,command,signal,value):
        # write <value> to internal dict
        self.vals[command] = value
    def read(self,command,signal):
        # return value from internal dict
        return self.vals[command]
```

Figure 5: Simple pyDrv example

from a startup script creating both kinds of port. The file my.py imported in the first line contains the code in Figures 4 and 5, the databases connect to port_a and port_b.

```
Python("from my import my_serial,my_asyn")
# start a virtual serial port
Python("a = my_serial()")
Python("a.start_serial('env_a')")
# name of port stored in environment var
drvAsynSerialPortConfig('port_a','$env_a')
Python("b = my_asyn('port_b')")
```

Figure 6: Excerpt from st.cmd startup script

CONCLUSIONS

Python has provided a flexible and powerful framework for building EPICS components, including clients (described here) and even servers (IOCs, not described). The Python language and libraries have allowed very powerful frameworks to be developed and rapidly used at Diamond—this paper only hints at the range of applications.

The language supports flexible rapid prototyping and has provided to be a valuable integration tool. The main drawback of the large scale application of Python is that it can run a *lot* slower than corresponding C code—but this has been a problem less frequently than might be expected.

REFERENCES

[1] Numeric Python, http://numpy.scipy.org

[2] Lightweight in-process concurrent programming, http://pypi.python.org/pypi/greenlet

[3] Trolltech, http://trolltech.com/

[4] Riverbank Computing, http://www.riverbankcomputing.com/static/Docs/PyQt4/html/classes.html

[5] EDM, http://ics-web.sns.ornl.gov/edm/

[6] Asyn, http://www.aps.anl.gov/epics/modules/soft/asyn/

[7] Stream Device 2, http://epics.web.psi.ch/software/streamdevice/