PvaPy: PYTHON API FOR EPICS PV ACCESS*

S. Veseli, Argonne National Laboratory, Argonne, IL 60439, USA

Abstract

As the number of sites deploying and adopting EPICS Version 4 grows, so does the need to support PV Access from multiple languages. Especially important are the widely used scripting languages that tend to reduce both software development time and the learning curve for new users. In this paper we describe PvaPy, a Python API for the EPICS PV Access protocol and its accompanying structured data API. Rather than implementing the protocol itself in Python, PvaPy wraps the existing EPICS Version 4 C++ libraries using the Boost.Python framework. This approach allows us to benefit from the existing code base and functionality, and to significantly reduce the Python API development effort. PvaPy objects are based on Python dictionaries and provide users with the ability to access even the most complex of PV Data structures in a relatively straightforward way. Its interfaces are easy to use, and include support for advanced EPICS Version 4 features such as implementation of client and server Remote Procedure Calls (RPC).

INTRODUCTION

EPICS Version 4 (EPICS4) [1] extends Version 3 [2] with features like support for complex data structures and service oriented architecture, optimized data transfers, as well as support for high level data and image processing. It also comes with a comprehensive set of C++ and Java APIs. However, what has been missing until recently is support for scripting languages. PvaPy aims to fill that gap by providing a Python API for the EPICS PV Access (PVA) protocol.

Rather than providing a direct Python implementation of the PVA protocol, PvaPy wraps EPICS4 C++ code using the Boost.Python [3] framework, a C++ library that enables seamless interoperability between C++ and Python. The main advantage of this approach is that it allows us to build on the existing EPICS4 code base and functionality, which significantly reduces PvaPy development effort.

BUILD PROCESS

Prerequisites for building PvaPy from sources [4] include the following:

- EPICS Base (v3.14.12.x, or v3.15.x) [5]
- EPICS4 C++ release (v4.4.0 or v4.5.0) [6]
- Python development header files/libraries (v2.6.x or v2.7.x) [7]
- Boost (v1.41.0 or later); installation must include the Boost.Python library [3]
- Standard set of GNU development tools (gcc, make, autoconf, etc.) [8]
- Sphinx (Python Documentation Generator) [9]; this is an optional package, generating documentation at build time is not essential.

Except for EPICS Base and the EPICS4 C++ release, all software dependencies listed above are typically included in most Linux operating system (OS) distributions. PvaPy has not been built or tested on Microsoft Windows.

PvaPy utilizes the standard EPICS build infrastructure [10]. However, unlike most EPICS modules, it also offers the possibility of configuring the software build automatically, using the GNU Autoconf [11] and a set of M4 [12] macros. Automated configuration determines compiler flags appropriate for the given operating system, and for the specific versions of Boost and Python that are installed on the build machine. Configuration scripts also determine the PvaPy API version that is suitable for the particular version of EPICS4 release, as well as prepare user environment setup scripts. User setup scripts modify PYTHONPATH environment variable so that PvaPy's "pvaccess" module can be imported within Python scripts or for interactive usage.

SOFTWARE FEATURES

PvaPy provides C++ code which calls the EPICS4 C++ libraries and defines a set of high-level classes for data objects, exceptions, and client/server interfaces. Those classes and their interfaces are exposed to users as the Python "pvaccess" module using the Boost.Python framework. PvaPy also defines a number of low-level utility and helper classes that are either required by EPICS4 APIs, or handle things like conversion between various Python and EPICS4 data structures. Note that the new high-level PVA Client C++ module [13] (available as part of the EPICS4 v4.5.0 release) greatly simplifies EPICS4 client interfaces and significantly reduces the number of internal classes that are implemented in PvaPy.

PvaPy Objects

EPICS4 C++ data types and modelling APIs are part of the PVData C++ package [14]. In PvaPy, the base class for all PV data types is *PvObject*, which represents a generic PV Structure. PvObject is initialized with a Python dictionary of PV introspection data, a set of key/value pairs describing the underlying PV structure in terms of field names and their types. The dictionary key is a string (the PV field name), and the value can be one of:

- PVTYPE: a scalar type, any of BOOLEAN, BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG, ULONG, FLOAT, DOUBLE, or STRING
- [PVTYPE]: a single element list, representing a scalar array

* Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, under contract DE-AC02-06CH11357.

ISBN 978-3-95450-148-9

- {key:value,…}: a dictionary, representing a structure
- [{key:value,…}]: a single element list containing a dictionary, representing a structure array
- (): an empty tuple, representing variant union
- [()]: a single element list containing an empty tuple, representing variant union array
- ({key:value,…},): a single element tuple holding a dictionary, representing a restricted union
- [({key:value,…},)]: a single element list containing a single element tuple of a dictionary, representing a restricted union array

In this way, we can easily describe even the most complex PV structures using standard Python data types and structures like dictionaries, lists and tuples. For example, a PV structure containing a structure array and a restricted union would be initialized as in Figure 1 below:

```
pv = PvObject({
  'sArray':[{'i':INT,'d':DOUBLE}],
  'u':({'f':FLOAT,'s':STRING},)
})
```

Figure 1: Initializing PvObject consisting of a structure array and a restricted union.

Field values for PvObject instances can be set using a dictionary keyed on the field names:

```
pv.set({
  'sArray':[
    {'i':1,'d':1.1},
    {'i':2,'d':2.2}
  ]
})
```

Figure 2: Setting PvObject's value via a Python dictionary.

The corresponding "get()" method returns a Python dictionary of all the PvObject's field values. Another way of manipulating and accessing a PvObject's fields is to use setters and getters that correspond to different field types. For example, setting a structure array can be done through the "setStructureArray()" method:

```
pv.setStructureArray(
  'sArray',
  [
    {'i':1,'d':1.1},
    {'i':2,'d':2.2}
  ]
)
```

Figure 3: Setting a specified structure array field.

Even though the PvObject class can represent any PV Data structure, PvaPy also comes with a number of specialized classes that wrap some of the standard PV Data types. Those include classes for various scalar types (*PvByte*, *PvInt*, etc.) and scalar arrays (*PvScalarArray*), unions (*PvUnion*), timestamps (*PvTimeStamp*), and alarms (*PvAlarm*). Note that wrapper classes for the EPICS4 Normative Types (NT) [15] have not yet been fully implemented.

Channel Class

PvaPy's *Channel* class provides the Python interface for communicating with PV Access channels, as well as for their monitoring. It is worth noting that this class also supports Channel Access (the EPICS Version 3 protocol) as well as PV Access. As of the EPICS4 release v4.5.0, the Channel class implementation is based on the PVA Client C++ package [13].

Users have the ability to retrieve and set process variable values through a Channel's "get()" and "put()" methods. The "get()" method returns a PvObject representing the current value for the given process variable. The "put()" method accepts either a PvObject or a standard Python data type as input for setting the process variable. For example, when "doubleArray" is the name of a PV channel for a structure containing an array of doubles, the following Python statements will initialize the Channel object and set its PV value:

```
c = Channel('doubleArray')
c.put([1.0,2.0,3.0])
```

Figure 4: Initializing the "doubleArray" Channel object and setting its PV value via a Python list.

The Channel class' monitoring functionality allows users to subscribe to PV value changes and process them with a Python function that takes a PvObject as an argument and has no return value. The code in Figure 5 monitors the above "doubleArray" channel, and prints the sum of the array's values after every change:

```
def sum(pv):
  s = 0
  for d in pv.get()['value']:
            s += d
  print s
c.subscribe('sum',sum)
c.startMonitor()
```

Figure 5: Monitoring PV channels.

Note that one can subscribe to PV value changes with an arbitrary number of monitor processing functions.

RPC Server and Client

The *RpcServer* class is used for hosting one or more PVA Remote Procedure Call (RPC) services. Users define an RPC processing function (which may be a Python class member), and register it with an RpcServer instance. The RPC processing function takes a client request PvObject as input, and returns a PvObject containing the processed result. Figure 6 below illustrates code for defining and registering a simple RPC service that returns sum of two numbers provided in an RPC request:

ISBN 978-3-95450-148-9

```
def sum(pvRequest):
  a = pvRequest.getInt('a')
  b = pvRequest.getInt('b')
  return PvInt(a+b)
srv = RpcServer()
srv.registerService('sum',sum)
srv.listen()
```

Figure 6: A simple RPC service returning the sum of two numbers from the client's request.

A single RpcServer class instance can host multiple RPC services, each accessible on their own PVA channel whose name is given in the "registerService()" call. The RpcServer can be started in its own thread by invoking the "startListener()" method instead of the blocking "listen()" function call shown above. This is typically used for multi-threaded programs, or for testing and debugging in Python's interactive mode.

RpcClient is a client class for PVA RPC services. Users initialize an RpcClient object giving the service's channel name, prepare a PV request object, and then invoke the service as in the following example:

```
c = RpcClient('sum')
request = PvObject({'a':INT,'b':INT})
request.set({'a':1,'b':2})
sum = c.invoke(request)
```

Figure 7: An RPC client for the "sum" service.

The result returned by the above call will be a PvObject containing the sum of the two numbers in the request.

Exceptions

PvaPy's "pvaccess" module exposes a number of exception classes that may be raised by the API under different error conditions. Examples of these are *FieldNotFound*, *InvalidDataType*, *InvalidRequest*, etc. Note that all PvaPy's exceptions derive from the base PvaException class, and that the exception hierarchy is preserved from C++ to Python using custom exception translator code and Boost.Python's translator registration mechanism (see [3] for examples).

Documentation

All exposed PvaPy classes and methods have been documented in the code, relying on Boost.Python's support for user-defined docstrings [3]. API reference documentation can be generated from the docstrings in various formats at build time, using the Sphinx documentation generator. Alternatively, users can access the official documentation [4] generated by the EPICS4 automated builds [16].

FUTURE PLANS

The most recent PvaPy version is bundled with EPICS4 release v4.5.0 [6]. Although it is fairly functional, there are quite a few desired features, development process improvements, and performance enhancements planned for the future:

- Implementation of wrapper classes for all Normative Types [15]; the current software only supports a few NT wrapper types
- Full support for PVA channels; at the moment operations like "putGet()" and "getPut()" are not supported
- Support for Python 3; at the moment PvaPy only supports recent Python 2 versions (2.6.x or 2.7.x)
- Support for NumPy arrays [17]
- Channel monitor performance and usability enhancements; at this time, processing monitor data on multiple CPU cores requires a significant amount of user-written code
- Test framework integration and test suite development; at the moment all testing is done manually
- PVA Server implementation

Note that the above list is not exhaustive and only includes some of the most important planned features.

CONCLUSION

PvaPy is the EPICS4 Python API for PV Access. It relies on the underlying EPICS4 C++ libraries and Boost.Python framework for interfacing Python and C++.

In addition to providing Python tools for EPICS4 application developers, one of PvaPy's goals is to help promote EPICS4 usage by making it more accessible to new users. As the examples presented in this paper illustrate, PvaPy interfaces have been designed with the end user in mind: to be as simple, flexible and intuitive as possible, while still retaining all capabilities and features provided by the PVA protocol.

ACKNOWLEDGMENT

I would like to thank A.N. Johnson for his work on ensuring that PvaPy's build conforms to EPICS standards, M. Kraimer and M. Davidsaver for their work on prototyping support for PV unions, M. Kraimer for the development of pvaClientCPP package, K. Vodopivec for his early feedback and suggestions, as well as to R. Lange and D. Hickin for their work on automated builds and preparing software release. I would also like to thank N.D. Arnold and the entire EPICS 4 working group for their support and encouragements during PvaPy development.

REFERENCES

[1] G. White et al., "Recent Advancements and Deployments of EPICS Version 4", this conference proceedings; EPICS4 website: http://epics-pvdata.sourceforge.net;
[2] EPICS website: http://www.aps.anl.gov/epics
[3] Boost website: http://www.boost.org; Documentation for the Boost.Python module can be found at http://www.boost.org/doc/libs/release/libs/python

ISBN 978-3-95450-148-9

[4] Similar to other EPICS Version 4 modules, PvaPy project is hosted in GitHub: https://github.com/epics-base/pvaPy; Documentation corresponding to the most recent code can be found at http://epics-pvdata.sourceforge.net/docbuild/pvaPy/tip/pvaccess.html
[5] EPICS Base releases can be downloaded from http://www.aps.anl.gov/epics/download/base/index.php
[6] EPICS4 production releases can be found at http://sourceforge.net/projects/epics-pvdata/files
[7] Python website: http://www.python.org
[8] GNU website: http://www.gnu.org
[9] Sphinx website: http://sphinx-doc.org
[10] M.R. Kraimer et al., "EPICS Application Developer's Manual", Chapter 4. For EPICS Base 3.15.2, the manual can be downloaded from http://www.aps.anl.gov/epics/base/R3-15/2-docs/AppDevGuide
[11] GNU Autoconf project website: http://www.gnu.org/software/autoconf/autoconf.html
[12] GNU M4 project website: http://www.gnu.org/software/m4/m4.html
[13] PVA Client C++ project website (GitHub): https://github.com/epics-base/pvaClientCPP
[14] EPICS4 PV Data C++ v4.5.0 reference document: http://epics-pvdata.sourceforge.net/docbuild/pvDataCPP/4.5.0/documentation/pvDataCPP.html
[15] EPICS4 Normative Types specification document: http://epics-pvdata.sourceforge.net/alpha/normativeTypes/normativeTypes.html
[16] EPICS4 CloudBees Jenkins Build Server website: https://openepics.ci.cloudbees.com; APS Jenkins website: https://jenkins.aps.anl.gov;
[17] NumPy is Python package for scientific computing; http://www.numpy.org

ISBN 978-3-95450-148-9