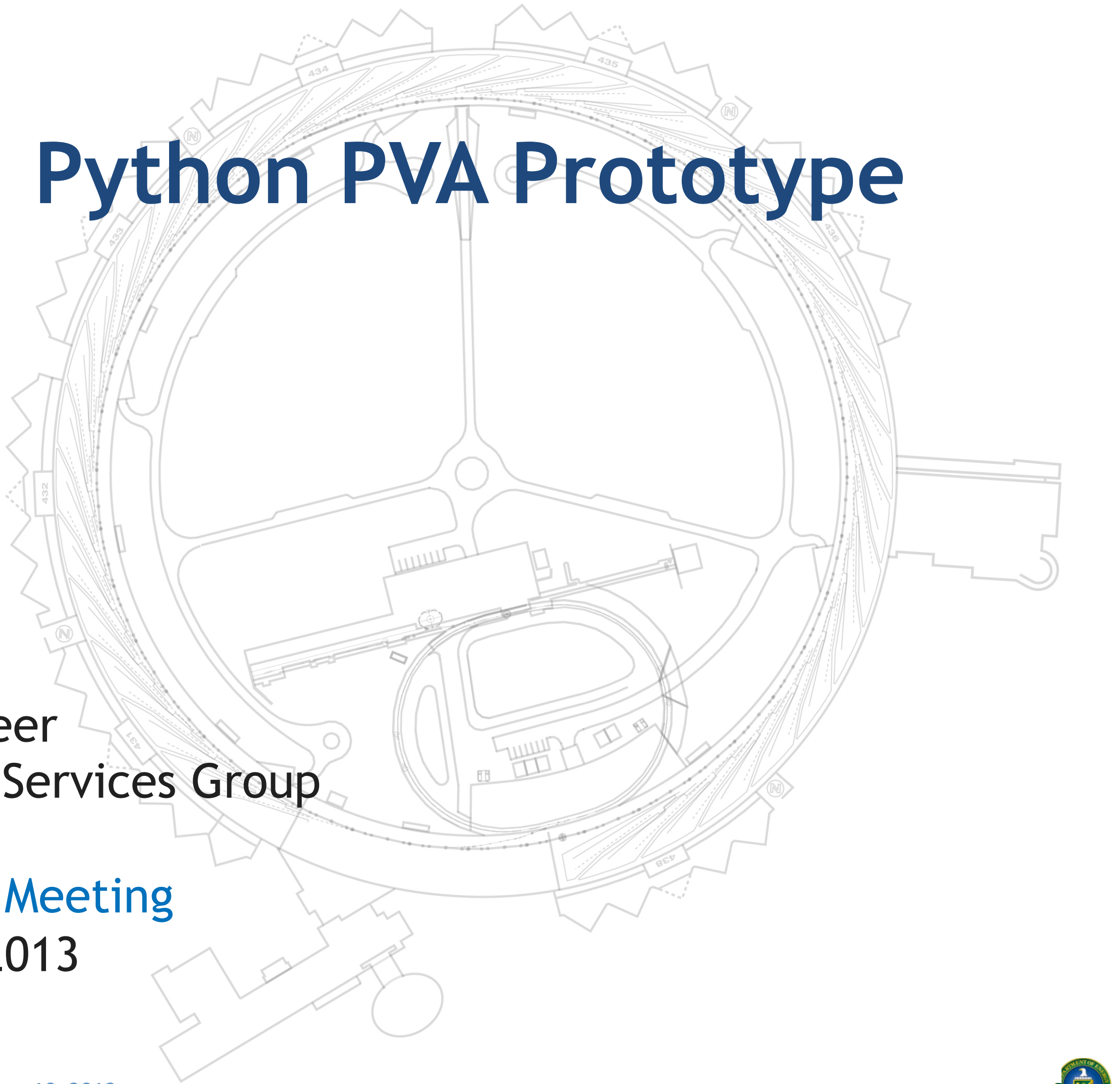
Python PVA Prototype



Siniša Veseli
Software Engineer
AES / Software Services Group

EPICS v4 Group Meeting
November 19, 2013

Outline

- Introduction
- Build
- Basic Usage
- Objects
- Advanced Usage
- Implementation Details
- Future Work

Introduction

- Goal: provide python API for PV Access that is:
  - Simple to build and use: one should be able to get started in minutes
  - Flexible: retain full PV Access functionality, anything that can be done via C++ APIs should be doable with python PVA API
  - Python look and feel: enable easy conversion to and from python objects (dictionaries, lists, etc.), use python operators, etc.

- Strategy: use boost.python to wrap PV Access C++ libraries
  - Enables one to leveraging existing functionality and reduce implementation effort
  - Simplifies maintenance: future improvements in C++ infrastructure should benefit python PVA API
  - Problem: No well-defined high level C++ API => must start almost from scratch

Build

- Prerequisites (prototype uses versions listed in parenthesis)
  - EPICS base release (v3.14.12.3)
  - EPICS4 CPP release (v4.3.0): nothing special needs to be done for build
  - Python development header files/libraries (v2.7.3)
  - Boost (v1.54.0): build must have boost_python library
- Current build process:
  1) Edit configure/RELEASE.local and add environment variables pointing to external dependencies
  2) make
  3) Create soft link pvaccess.so => libpvaccess.so in lib/$EPICS_HOST_ARCH
  4) Prepare setup file (PYTHONPATH needs to have entry pointing to $PVAPY_DIR/lib/$EPICS_HOST_ARCH)
- Plan:
  - Eliminate steps 1, 3 and 4
  - Provide standard python module build/packaging scripts

Basic Usage

- Source setup file (or export PYTHONPATH=$PVAPY_DIR/lib/ $EPICS_HOST_ARCH:$PYTHONPATH)
- Inspect package contents:

```
$ python -c "import pvaccess; print dir(pvaccess)”
```

- Start test IOC from $EPICS4_DIR/pvaSrv/testApp/iocBoot/testDbPv directory:

```
$ ../../bin/linux-x86_64/testDbPv st.cmd # in terminal 2
```

- Use Channel class to get/set PVs:

```
$ python
>>> from pvaccess import *   # never do that in scripts
>>> c = Channel('int01')
>>> print c.get()
uri:ev4:nt/2012/pwd:NTScalar
  int value 0
>>> c.put(PvInt(7))
>>> print c.get()
uri:ev4:nt/2012/pwd:NTScalar
    int value 7
```

PVObject Class

- Base for all python PVA objects is `PvObject`
  - PV structure, initialized via python dictionary of key:value pairs that describe underlying PV structure: key is a string and value one of PVTYPE, [PVTYPE], {key:value,…}, [{key:value,…}]
  - PVTYPE: `BOOLEAN, BYTE, UBYTE, SHORT,…,STRING`
  - Scalar Array: represented as a list with a single PVTYPE element describing element type, e.g. `[INT]`
  - Structure Array: represented as list a with a single dictionary element describing element structure, e.g. `[{‘x’ : INT, ‘y’ : FLOAT}]`
  - Has setters/getters for all field types, e.g.
    - > `setInt(key, value)`
    - > `getInt(key)`
    - > `setScalarArray(key, value)`
    - > `getScalarArray(key)`
    - > `setStructure(key, value)`
    - > `getStructure(key)`

PvObject Examples

```
>>> pv = PvObject({'i' : INT, 's' : STRING})
>>> print pv
structure
    int i 0
    string s

>>> # Can set entire object with key/value dictionary
>>> pv.set({'i' : 12, 's' : 'abcd'})
>>> print pv
structure
    int i 12
    string s abcd

>>> # Can use getters/setters for each field
>>> pv.getString('s')
'abcd'
>>> pv.setString('s', 'xyz')
>>> pv.getString('s')
'xyz'
```

PvObject Examples

```
>>> pv = PvObject({'i': INT, 'slist' : [STRING], 'dict' : {'b' :
BOOLEAN, 'dict2' : {'d' : DOUBLE}, 'flist' : [FLOAT]}})
>>> print pv
structure
    int i 0
    string[] slist []
    structure dict
        boolean b 0
        float[] flist []
        structure dict2
            double d 0
>>> # Can use incomplete dictionaries to set fields
>>> pv.set({'i' : 15, 'dict' : {'flist' : [1.1, 2.2, 3.3]}})
>>> print pv
structure
    int i 15
    string[] slist []
    structure dict
        boolean b 0
        float[] flist [1.1,2.2,3.3]
        structure dict2
            double d 0
```

PvObject Examples

```
>>> # Conversion to dictionary
>>> pv.toDict()
{'i': 15, 'slist': [], 'dict': {'b': False, 'dict2': {'d': 0.0},
'flist': [1.100000023841858, 2.200000047683716,
3.299999952316284]}}

>>> # Get structure field
>>> pv.getStructure('dict')
{'b': False, 'dict2': {'d': 0.0}, 'flist': [1.100000023841858,
2.200000047683716, 3.299999952316284]}

>>> # Get original structure dictionary
>>> pv.getStructureDict()
{'i': pvaccess.PvType.INT, 'slist': [pvaccess.PvType.STRING],
'dict': {'b': pvaccess.PvType.BOOLEAN, 'dict2': {'d':

pvaccess.PvType.DOUBLE}, 'flist': [pvaccess.PvType.FLOAT]}}
```

Derived Object Classes

- Each scalar type has its own class: `PvBoolean, PvByte, …, PvString`
  - Can be initialized using scalar value
  - Have setters/getters

```
>>> s = PvString('abc')
>>> print s
abc
>>> d = PvDouble(123.456)
>>> print d
123.456
>>> l = PvLong(123456789012345678L)
>>> print l
123456789012345678
>>> l.get()
123456789012345678L
>>> l.set(13L)
>>> l.get()
13L
```

Derived Object Classes

- Scalar array type class: `PvScalarArray`
  - Initialized using scalar type
  - Setter/getter

```
>>> array = PvScalarArray(INT)
>>> print array
structure
    int[] value []
>>> array.set([1,2,3,4,5])
>>> print array
structure
    int[] value [1,2,3,4,5]
```

Channel Class

- Provides get/put functionality

```
>>> c = Channel('bigstring01')
>>> c.put(PvString('Very Big String'))
>>> print c.get()
uri:ev4:nt/2012/pwd:NTScalar
    string value Very Big String
c = Channel('intArray01')
>>> print c.get()
structure
    int[] value []
>>> print array
structure
    int[] value [1,2,3,4,5]
>>> c.put(array)
>>> print c.get()
structure
    int[] value [1,2,3,4,5]
```

RPC Client Class

- Client for RPC service
- Start v4 test RPC service from $EPICS4_DIR/pvAccessCPP/bin/linux-x86_64/

```
$ ./rpcServiceExample # in terminal 2
```

- RPC test channel is “sum”:

```
>>> rpc = RpcClient('sum')
>>> request = PvObject({'a' : STRING, 'b' : STRING})
>>> request.set({'a' : '11', 'b' : '22' })
>>> print request
structure
    string a 11
    string b 22
>>> response = rpc.invoke(request)
>>> print response
structure
    double c 33
```

RPC Server Class

- Allows creating PVA services in python
- In terminal 1

```
$ python # in terminal 2
>>> from pvaccess import *
>>> srv = RpcServer()
>>> def echo(x):   # x is instance of PvObject
...     print 'Got object: ', x
...     return x   # service must return instance of PvObject
>>> srv.registerService('echo', echo)
>>> srv.listen()
```

- In terminal 1

```
>>> rpc = RpcClient('echo')
>>> response = rpc.invoke(request)
>>> print response
structure
    string a 11
    string b 22
```

RPC Client/Server Example

- In terminal 2

```
$ python
>>> from pvaccess import *
>>> srv = RpcServer()
>>> def sum(x):
...     a = x.getInt('a')
...     b = x.getInt('b')
...     return PvInt(a+b)
>>> srv.registerService('sum', sum)
>>> srv.listen()
```

- In terminal 1

```
>>> rpc = RpcClient('sum')
>>> request = PvObject({'a' : INT, 'b' : INT})
>>> request.set({'a' : 11, 'b' : 22})
>>> print request
structure
    int a 11
    int b 22
>>> response = rpc.invoke(request)
>>> print response
structure
    int value 33
```

RPC Client/Server Example

- In terminal 2

```
>>> from pvaccess import *
>>> srv = RpcServer()
>>> def hash(x):
...     import hashlib
...     md5 = hashlib.md5()
...     md5.update(str(x))
...     h = md5.hexdigest()
...     dict = x.getStructureDict()
...     dict['hash'] = STRING
...     response = PvObject(dict)
...     response.setString('hash', h)
...     return response
>>> srv.registerService('hash', hash)
>>> srv.listen()
```

- In terminal 1

```
>>> rpc = RpcClient('hash')
>>> request = PvString('abcd')
>>> print rpc.invoke(request)
structure
    string hash 0a380e7375d8c3f68d1bbe068141d6ce
    string value
```

Exception Handling

- At the moment all exceptions are mapped into UserWarning

```
>>> c = Channel('notthere')
>>> c.get()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UserWarning: Channel notthere timed out
```

- Plan: decent exception hierarchy

Implementation Details/Issues

- Current functionality implemented in under 5K lines of code
- Actual python pvaccess module (`pvaccess.cpp`) so far has less than 350 lines
- Lots of work went into defining C++ API that would be easy to wrap using boost.python
- Fair amount of python PVA C++ code is now duplicated from utilities like pvget and pvput
- Common code (e.g., various default requester impl classes, parsing utilities, etc.) could be extracted into high level PVA C++ API that would be easier to use and more attractive for an average user (RPC Service/Client C++ classes are an excellent example)
- All PVA command line/test utilities could be built on top such API (e.g., `src/pvaccess/testClient.cpp` retrieves PV from a given channel in about 20 lines of code)
- Promotes code reusability, easier maintenance, etc.

Future Work

- Complete python PVA API functionality (e.g., channel monitor)
- Build/packaging
- More exception classes
- More work on usability:
  - Additional Object constructors
  - Python operators (especially for scalar types)
- Python docs
- Test suite
- …