EPICS V4 IN PYTHON*

Guobao Shen#, Marty Kraimer, Michael Davidsaver, BNL, Upton, NY 11973, U.S.A.

Abstract

Interest in Python as a rapid application development environment continues to grow. Many large experimental scientific facilities have adopted Python for beam commissioning and the operation. The EPICS control system framework has become the de facto standard for the control of large experimental facilities, where it is in use in over 100 facilities. The next version of EPICS (EPICS V4), under active development will extend the support for physics applications, data acquisition, and data analysis. Python support for EPICS V4 will provide an effective framework to address these requirements. This paper presents design, development and status of activities focused on EPICS V4 in Python.

MOTIVATION

As part of the EPICS V4 [1] initiative, there is increased interest in expanding the conventional low level hardware support to include high level applications such as physics applications, data acquisition and data analysis [2]. The support consists of the following main modules: (1) pvData, a memory resident real time database with a predefined data structure; (2) pvAccess, a network protocol for transferring data over wire; (3) pvIOC, a processing engine; (4) pvService, a collection of pvIOC instances implemented as a service.

The high performance/low latency of EPICS V3 for stream data and instrumentation control has been thoroughly demonstrated in many facilities. To implement the proposed extensions to EPICS V4, especially data acquisition, a major concern is its performance. Benchmark tests have been conducted at NSLS-II project to compare the performance of pvAccess with that of Channel Access [3][4]. These benchmarks were performed using pvAccess implemented in Java, and Channel Access implemented in C/C++. The results show that for scalar data, pvAccess has identical performance when processing large numbers channels, and a slightly reduced performance when processing only a few (for example 10) channels. For array processing, pvAccess outperformed Channel Access. More detailed benchmarking results are given in [3]. The benchmark results satisfied the requirements to proceed with the proposed enhancements.

A modular infrastructure based on the client/server model has been designed for NSLS-II for beam commissioning, physics study, and beam operation applications [5][6][7]. In this approach, data flow is separated to 2 parts: (1) acquire data from hardware, and (2) consume data in the application. The 2 flows are implemented with a well-defined API. An advantage of this design is that hardware data flow API is implemented by developers, who have more experience on data and hardware control. The data consumption is handled by physicists who are experts in data analysis.

The client/server implementation is based on EPICS V4, which provides a good technical framework for the infrastructure, the flexibility of data structure design, and satisfactory performance for the physics applications. The 3-tier architecture is shown in Fig. 1. The detailed explanation of this architecture can be found in [8][9].

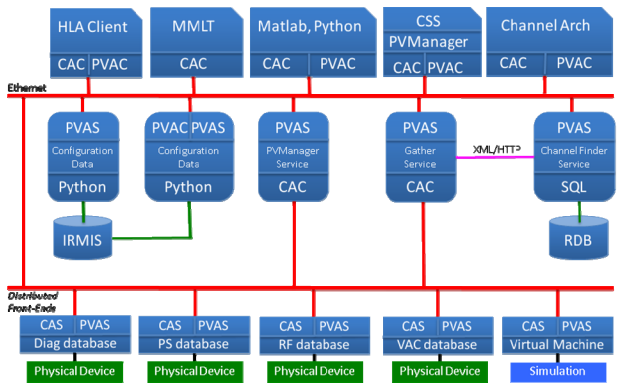

Figure 1: System Architecture.

Figure 1 illustrates the application development environment provided by EPICS V4. Developers have the flexibility to select their favourite technologies to develop back end applications.

At NSLS-II, Python has been selected as the primary development language for physics applications. This motivated us to provide client side Python support. In addition, as shown in Fig. 1, all static data stored in the IRMIS database are served through the V4 server layer. A Python API is under development for access to the IRMIS database. To utilize this API, it is required also to have a Python support on server side.

These 2 factors provided the motivation to have full Python support for EPICS V4. In this paper, we report the results and present current development status for Python support using 3 separate approaches.

PYTHON IMPLEMENTATION

As an interpreted, high-level programming language with dynamic semantics and native support of extensive scientific libraries, Python is very attractive for rapid application development as well as for use as a scripting or glue language. There is a growing interest in the community to use Python as the physics application development platform. At the NSLS-II project, Python has been selected as primary language for physics application development.

EPICS V4 was developed originally in Java, and provides a comprehensive set of features. Later as

*Work supported under auspices of the U.S. Department of Energy under Contract No. DE-AC02-98CH10886 with Brookhaven Science Associates, LLC, and in part by the DOE Contract DE-AC02-76SF00515

#shengb@bnl.gov

requested by the community, a C++ binding was developed and now has full functionality in both the pvData and pvAccess modules, and basic support in the pvIOC and pvService modules. The current language bindings are shown as Table 1.

Table 1: EPICS V4 Language Bindings.

Module	Java	C++	Python
pvData	Y	Y	Y/Limited
pvAccess	Y	Y	Y/Limited
pvIOC	Y	Y/Limited	N
pvService	Y	Y/Limited	N

In this paper, we will focus on the development in Python. For the Python binding, there are several possible solutions: (1) utilizing Python/C API; (2) using existing wrapping tools; and (3) developing a native binding all in Python.

In this section, we will discuss the first two, followed by a discussion of the native binding development the next section.

Python/C

Python/C provides an API for a programmer to build an external module callable in Python. It is a relatively well-understood process to writing an extension module. Most Python/C API functions have one or more arguments as well as a return value of type PyObject*. The returned type is a pointer to an opaque data type, which represents an arbitrary Python object. All Python object types are generally treated the same way in the Python language.

We have to pay attention that all Python objects have a type and a reference count. When exposing an API to Python, the developer must explicitly manage the reference count in the C code. When an object's reference count becomes zero, the object is de-allocated, and collected by garbage collector.

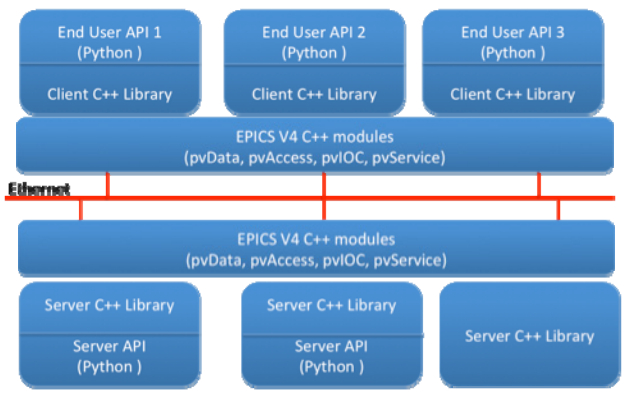

Figure 2: Software Structure Using Python/C.

A realistic approach, shown in Fig. 2, for the end user client interface is to develop a client library in C++, and wrap the C++ library as a Python interface. Each client library is standalone. The benefits are that most codes are in C++, and in general we can gain performance in Python as good as that in C++. However, changes in EPICS V4 C++ modules usually affect the client C++ library. As the project grows, the number of end user client libraries will increase. Long-term maintenance will become an issue.

Wrapping Tools

To minimize the effect to the client library caused by changes of V4 C++ modules, another approach is to wrap each C++ module. This software architecture is as shown in Fig. 3. The end user API is developed in native Python. Any change in the EPICS V4 library affects only the wrapper library, and is transparent to the end API developer/user. Another advantage with this solution is that both client and server can use the same wrapping library.

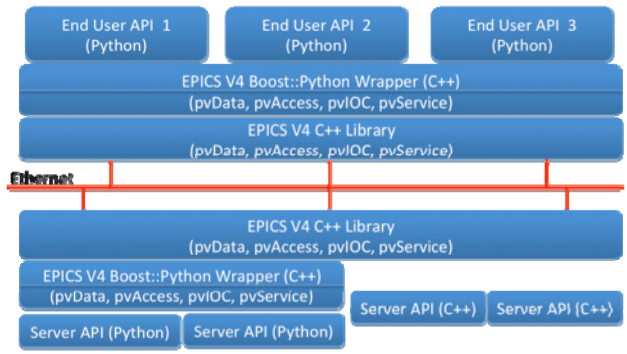

Figure 3: Software Structure by Wrapping V4 Modules.

Tools such as SWIG, SIP, and Boost Python [10] can be used to expose C++ API to Python. Essentially, they are a wrapper for the Python/C API, and provide a more realistic solution to solve many well-known problems. For example, using the Python/C API, the developer has to deal with pointers passed between Python and C++. The developer is responsible for pointer management, particularly when the referenced object has been deleted. Using wrapping tools, those problems are taken care by the tools.

At NSLS II, 2 tools are evaluated, SIP and Boost Python respectively. Because SIP does not support the STD::TR1 smart pointer well, Boost Python was selected in the end. The Boost Python is one member of the Boost C++ library collection, and binds C++ and Python in a mostly-seamless fashion. It also provides a set of policies to track the ownership of an object in both C++ and Python domains. With the Boost Python Library, the developer can quickly and easily export C++ to Python.

A disadvantage with Boost Python library is that a boost library must be installed, and care has to be taken to switch between the boost library and the STD library. Moreover, the Boost Python library does not fully support STD::TR1 library, and provides a simulated interface set for STD::TR1.

Native Implementation

Either the Python/C approach or exposing C++ API to Python using wrapping tools is a workable solution.

Although exposing the C++ API allows users to focus their development on native Python, this heavily relies on the Boost Python Library. This library dependency forces all of EPICS V4 to be compiled against the Boost library. Moreover, tracking and passing object ownership to/from between C++ and Python domain is difficult.

Although we believe Boost Python is best approach for exposing the EPICS V4 API to Python, its template meta-programming mechanism pushes compilers to their limits. Often it is necessary to increase template-depth limit settings of the compiler, memory usage can be vast, compile times are long, and error messages are often difficult to decipher.

A new design is under consideration to implement EPICS V4 in native Python to take full advantage of Python. For example EPICS V4 defined an efficient way to describe complex data structures and the data protocol. In the current implementation in either C++ or Java new data structures must be constructed entirely from the primitive data types. Using Python, it is much simpler to map the data structure into a numpy [11] array. The software architecture is shown in Fig. 4.

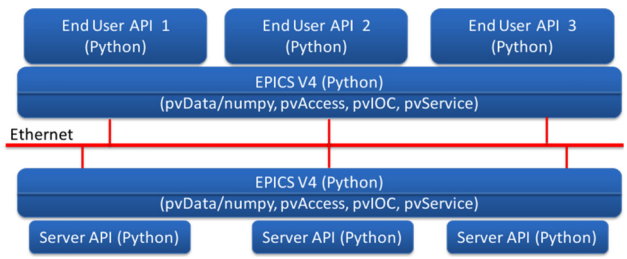

Figure 4: Software Structure in Native Python.

CURRENT STATUS

Python/C

As above described, each library developed in Python/C is a standalone library. In another words, its development relies on each service development. Currently, one library is developed for a service implemented under EPICS V4 in Java, a 'gather' service. The library supports all functions required by clients, and supports all fundamental data manipulation such as get, put, and monitor.

Boost Python Wrapping

With Boost Python library, basic APIs have been implemented in Python for the pvData and pvAccess modules. The end user interface for accessing the gather service is under development. Design and implementation issues related to thread safety and locking have been found and resolved in the C++ modules.

Basic unit tests have been performed for the exposed APIs. More detailed testing is necessary to improve the stability and robustness.

Native Implementation

This is still under design, and the development is at very early stage. Some preliminary code has been developed to verify basic ideas, for example mapping EPICS V4 data structures into numpy arrays.

SUMMARY

This paper presented activities on Python development for EPICS V4. Several different approaches have been conducted at BNL and the status is described.

At the present stage, a workable approach is to utilize Boost Python library. In this approach, there is a trade off solution between performance and flexibility/scripting. Performance benchmarking is necessary.

ACKNOWLEDGEMENT

The authors would like to thank Matej Sekoranja at COSYLAB for his contributions on epics-pvdata development, especially the pvAccess implementations in both Java and C++. They want to give their thanks to Leo Bob Dalesio at BNL for his continuous support and encouragement.

REFERENCES

[1] http://epics-pvdata.sourceforge.net/
[2] L. Dalesio, *et al*, "EPICS V4 Expands Support to Physics Application, Data Acquisition, and Data Analysis", This proceedings, FRBHMULT06, Grenoble, France (2011)
[3] G. Shen, "Performance Analysis of EPICS Channel Access and pvAccess", NSLS-II Tech Note 082 (2010)
[4] G. Shen, *et al*, "Server Development for NSLS-II Physics Applications and Performance Analysis", Proc. of PAC11 (2011), MOP252, New York, USA
[5] G. Shen, "A Software Architecture for High Level Applications", Proc. of PAC09 (2009), FR5REP004, Vancouver Canada
[6] G. Shen, "A Modular Environment for High Level Applications", Proc. of ICALEPCS09 (2009), THP094, Kobe Japan
[7] G. Shen, *et al*, "A Novel Approach for Beam Commissioning Software using Service Oriented Architecture", Proc. of PCaPAC10 (2010), WEPL037, Saskatoon Canada
[8] G. Shen, *et al*, "Prototype of Beam Commissioning Environment and its Applications for NSLS-II", Proc. of IPAC10 (2010), WEPEB026, Kyoto Japan
[9] G. Shen, *et al*, "NSLS-II High Level Application Infrastructure and Client API Design", Proc. of PAC11 (2011), MOP250, New York, USA
[10] http://www.swig.org/; http://riverbankcomputing.co.uk/software/sip/intro; http://www.boost.org/
[11] http://numpy.scipy.org/