ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA153

PYTHON AND MATLAB INTERFACES TO RHIC CONTROLS DATA *

K.A. Brown†, T. D'Ottavio, W. Fu, A. Marusic, J. Morris, S. Nemesure, A. Sukhanov,
Collider-Accelerator Department, BNL, Upton, NY, USA

Abstract

In keeping with a long tradition in the BNL Collider-Accelerator Department (C-AD) controls environment, we try to provide general and simple to use interfaces to the users of the controls. In the past, we have built command line tools, Java tools, and C++ tools that allow users to easily access live and historical controls data. With more demand for access through other interfaces, we recently built a set of Python and MATLAB modules to simplify access to control system data. This is possible, and made relatively easy, with the development of HTTP service interfaces to the controls. While this paper focuses on the Python and MATLAB tools built on top of the HTTP services, this work demonstrates clearly how the HTTP service paradigm frees the developer from having to work from any particular operating system or develop using any particular development tool.

INTRODUCTION

The C-AD controls system [1,2] was developed in the mid-1990's and built largely in C, C++, and Java (some legacy code in C with new code all in C++ and Java). The basic design is based on the Accelerator Device Object (ADO) model, which is similar to the TACO (predecessor to TANGO [4]) model. The ADO model, shown in Fig. 1, is a client-server model that uses TCP/IP as the communcation protocol and RPC at the server level.

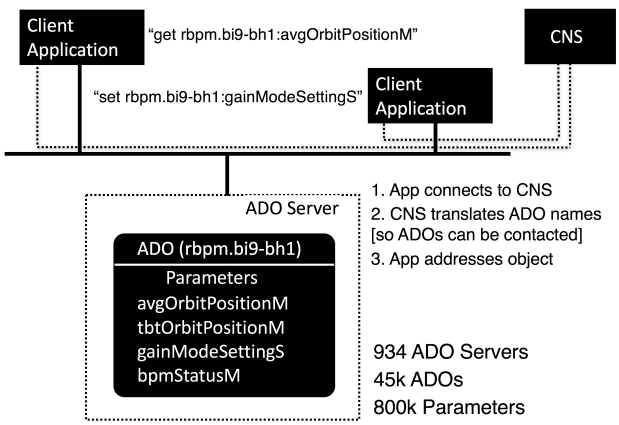

Figure 1: C-AD Controls ADO communication model.

In the ADO communications model, a client application just has to know the name of the ADO and device parameter in order to establish a connection to that parameter. Communication is done using *get* and *set* calls. Asynchronous requests can also be established. In the example shown in Fig. 1, two clients are trying to communicate with the same ADO, rbpm.bi9-bh1, but to different parameters. The ADO server handles the client requests as they are received. Since the communication is through TPC/IP, the client needs to establish a direct connection to the ADO server. We use a specialized Controls Name Server (CNS) to tranlate ADO names into host names and RPC program/version numbers, so those ADOs can be contacted.

RESTFUL SERVICES MODEL

A Representational State Transfer (REST) architectural style of communication utilizes the communication protocol at the internet level, to use textual resource representations and predefined stateless operations. HTTP is an application protocol and the next communication layer up from TCP/IP, a transport protocol, and provides a clean, standard, and well-defined protocol for this communication. The use of HTTP servers provides the ability to abstract away the control system and even the operating system dependencies from the client application [3]. The client can now be an iOS app, an Android app, a Python script, a MATLAB® script, or even just a simple web page built to interface to the HTTP server. In each case a client just has to know the HTTP specific protocol interface and the names of the control system ADO's and devices. Figure 2 shows the communications model for a Python application communicating to the controls through an HTTP Device server interface and through an HTTP Data server interface. The Device Server allows direct *set* and *get* communication to the ADO parameters while the Data Server provides access to logged data.

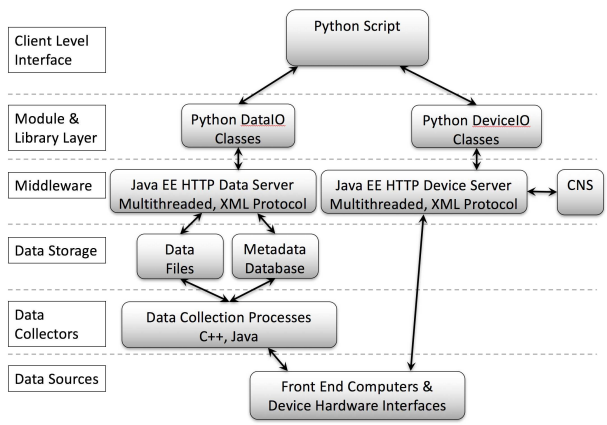

Figure 2: Example of HTTP communications model.

Data Server

In a REST communication model, the HTTP message includes a request (such as *GET*, and *PUT*) along with the meta

* Work performed under Contract Number DE-SC0012704 with the auspices of the US Department of Energy.
† kbrown@bnl.gov

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA153

data needed to complete the request. The server returns the data for synchronous calls, or, in the case of asynchronous calls, a handle for the client to use for completing the communication transaction.

The Data server provides a REST interface to data collected and stored by the C-AD controls system data logging facilities [5]. To get data from the server, an http message must be constructed by the client. This message is broken up into distinct parts. First there is the base path part of the message, which might look like,

`http://servername.bnl.gov:port/DataServer/`

This is followed by specific fields that construct the request. If we want to get data from the RHIC Schottky logs, the message would contain the name of the Schottky ADO manager (request = SchottkyMan), the name of the desired parameters (params = horz.blue:tuneM), the maximum number of points to return, and other parameters. An actual request might look similar to this:

`http://servername.bnl.gov:port/DataServer/SchottkyMan?user=kbrown&client=path/LogClient&host=loghostname&start=unixtime&end=unixtime&data=horz.blue:tuneM&count=20000&context=SyncStdAdo.`

Expecting general users to construct such messages is still too low level of a protocol, so we have built Python and MATLAB® packages that make it much easier to request data from the logs. An end user just has to know the name of the log server for the data they want, the start and end times for the data (in human readable form), and the names of the parameters. Much of this interface is specific to the C-AD control system and how data is logged and organized. Nevertheless, it is now a fairly simple process to bring log data to other platforms, adding great flexibility in how end users access controls information.

Device Server

The Device server provides a REST interface for direct interaction with ADO parameters. To simplify the interface for end users, Python and MATLAB® packages have been built. So now, an application just makes a simple *set* or *get* request using the Device IO class interface. Figure 3 shows an example of a Python script using this interface.

Data is packaged into a Python dictionary, allowing the returned data to be addressed by the controls ADO and parameter names.

PYTHON TOOLS

Python has been used in the C-AD controls network for many years, primarily for offline data analysis. There has been increased interest in the use of Python at C-AD starting in 2016. Python's extensive library of open source packages, including data analysis packages, makes it very popular and attractive for use at C-AD.

```
from deviceio import pyado
pyado.LogToFile = True

# use the device server
adodata = pyado.useHttp()

request = [(adoname, adoparam1),
           (adoname, adoparam2)]
value = adodata.get( request )
for r in value:
    for k in value[r]:
        print k,  value[r][k]
```

Figure 3: Structure of Python interface to Device Server.

Although Python as a language has its deficiencies (e.g., thread limitations) it has a large community of users and significant high-quality code base.

There are six basic functions for interfacing to the C-AD controls system through Python. In addition, an application can choose whether to go through the Device server (REST API interface) or use direct calls using a pure Python API interface. From the application point of view either approach looks exactly the same except for one call that defines which interface to use (e.g., see Fig. 3).

The six basic functions are:

*getMeta(*args,**keywords)*

This function pulls the meta data for the given ADO, which is useful if an application needs to know any of the properties associated with a given ADO or its parameters. Examples of its use are:

1. *getMeta('adoname')* returns a list of parameters for the ADO
2. *getMeta('adoname', all=True)* returns a dictionary of all metadata related to the ADO
3. *getMeta('adoname', parname')* returns a dictionary of all the properties associated with the given ADO parameter

*get(*args,**keywords)*

This function returns a dictionary of entries, keyed with 'adoName:parameterName', each entry is in turn a dictionary of returned properties with keys 'value', 'timestamp' and others. Examples of its use are:

1. *get('adoName','parName')* returns value and timestamp of the parameter.
2. *get('adoName','parName.propertyName')* returns parameter's property.
3. *get([(adoName1,parName1),(adoName2,parName2), ...])*
   returns values or properties of several parameters.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA153

*set(*args)*

The set function sends a new value or set of values to the specified ADO/device(s). Examples of its use are:

1. *set('adoName','parName',value)* sets the parameter value.
2. *set([(adoName1,parName1,value1), (adoName2,parName2,value2), ...])* sets values of several parameters.

*getAsync(callback, *args,**keywords)*

This sets a monitor on the parameter, which will cause a call of a callback function each time the value changes.

The callback function can be the same for several calls of the getAsync. The callback function is invoked as:

callback(*args)

where the args are similar to the return of the get([(adoName1,parName1),(adoName2,parName2),...])

cancelAsync()

Cancel the monitor, set by *getAsync()*.

*sendAlarm(message,**keywords)*

Send an alarm message to the RHIC Controls alarm servers.

Example Keyword Arguments:

host = 'rhicnotifyserver.pbn.bnl.gov'
program = 2000004
topic = 'ADO:adoName:parameter'
category = notify.CATEGORY['NOTIFY_OK']

One of the driving motivations to make greater use of Python in C-AD is the ability to take code from other facilities and convert it over for our use. Two recent examples are a MATLAB® module developed at Cornell for conditioning a high voltage electron gun and a Python application (suite) developed at NSLS II for beam orbit commissioning and studies. In each case, the conversion for use on our controls was made fairly simple by changing controls interfaces from the original applications to use our HTTP servers with our device names. The main focus, then, was put into altering the look and functionality of the application, based on the end user requirements.

MATLAB® TOOLS

The MATLAB® system has several ways to interface with accelerator control systems (hardware and software). Some applications need additional tool boxes and others require a significant amount of programming effort to realize the seamless communication between the controls system and MATLAB®. By taking the advantages of the REST API services, we developed a MATLAB® script library which bridges the gaps between control data resources and MATLAB® system. With this library, MATLAB® users and application developers can easily communicate with

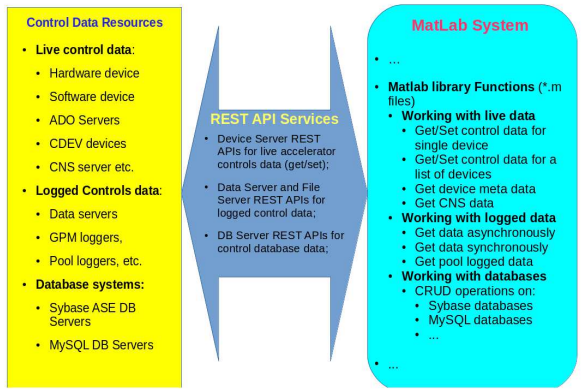

Figure 4: MATLAB® library interface.

accelerator control system interfaces, including hardware, software, file logging systems and database systems. Once the control data has been brought into the MATLAB® environment, developers can process and analyze the data with MATLAB®'s powerful features. Figure 4 shows how this MATLAB® library works:

Figure 5 shows the MATLAB® library functions, getDeviceValue and getListDeviceValue, as examples of the interface to the C-AD ADO controls.

```
function [device_value] =
    getDeviceValue(device_name,
                   parameter, ppm_user)
function device_values (array) =
    getListDeviceValue(
                   device_list (array))
```

Figure 5: MATLAB® library functions interfacing to controls devices.

The returned values of the functions can be any data type depending on the data types of control parameters of the devices. The returned data can also include the time stamp of the device values if desired.

Figure 6 show how to get asynchronous data from logged controls data.

```
function [title, XLabel, YLabel,
          results_cell_array] =
    getData(logReq, dataItems,
            start_date, end_date,
            count, filter)
```

Figure 6: MATLAB® library function interfacing to controls data.

In the *function*, users just specify the location of the logged data (file names), time range, device names and parameters (array of items), and data filter to filter the data (if desired),

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA153

and get the query result with the details in data title, X-Y data titles, and cell data corresponding to the array of device items and time stamps.

For database data, the MATLAB® function simply passes the database server name, database name, and SQL statement, then get the execution results of the SQL statement. The library function and the REST API services underneath handles all database operations. The returned data is in JSON format.

This MATLAB® library provides to MATLAB® users or application developers a simple interface for connecting to the control system and get control data using function calls. Figure 7 shows an example of a RHIC Control application that uses this MATLAB® library in the last RHIC run_fy17: The program named "Camera Image Monitor" communicates with a accelerator control camera device through the MATLAB® library functions and makes the camera image data available in the MATLAB® system. It then makes use of the MATLAB®'s powerful data analysis, process and visualization capability to create the rich GUI and present the data in a very meaningful way.

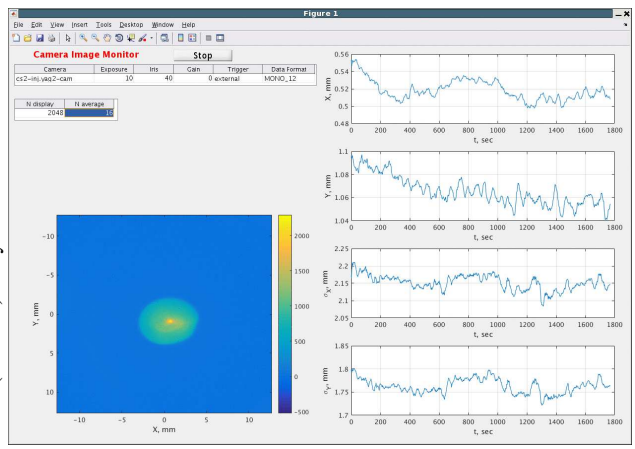

Figure 7: CeC Camera Image Monitor App [8].

APPLICATIONS

DC Gun Auto-conditioning

For the Low Energy RHIC electron Cooling project (LEReC) [9], a high voltage electron gun was built by Cornell. To condition this device, which needs to operate at over 400 kV, a significant conditioning process must take place. Cornell developed a simple MATLAB® script that automated some aspects of the conditioning. Since the MATLAB® script depended on the EPICS [6] MATLAB® interface, it would not work with the C-AD controls system. The simplest solution was to convert the MATLAB® code to Python. This was fairly easy to do. However, the commissioners wanted the new script to do more than the Cornell script, so it evolved.

Since the script had some parameters hard-coded into it, for the Python version these parameters were turned into ADO parameters, allowing commissioners to change the script behavior without having to restart it.

In the end, the new Python script was much different from the MATLAB® script. To perform properly required multiple threads and new logic was required (i.e., an auto turn on mode was included, in case the power supply tripped off during conditioning). However, the core logic of the original script remained in place and that gave some 'comfort' to the commissioners, who had concerns about re-writing the entire application from scratch.

So, in this case, code was taken from another facility and really was used as a guide to repurpose to our facility. However, it was extremely useful to have that original code and the focus of the development was not on the controls interface but on the function and look of the new code. The gun conditioning application graphics window is shown in Fig. 8.

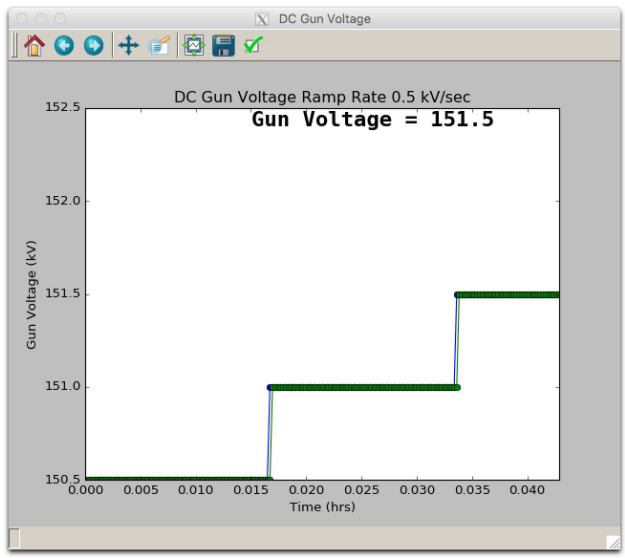

Figure 8: Graphic display of DC Gun Conditioning.

LEReC & CEC Online Models

Both LEReC and the Coherent Electron Cooling proof of principle experiment (CeC) [10] are electron accelerator systems, with a significant level of complexity. Each has electron guns with low energy sections and beam transport systems composed of standard accelerator components, including transport solenoids. Each is instrumented with current monitors, beam position monitors, and various types of profile monitors. A more detailed description of the online model interface is described by Brown [7]. The CeC model application graphics page is shown in Fig. 9.

FUTURE PLANS

The new RESTful tools for HTTP protocol interfaces opens up a world of applications. These combined with Python and MATLAB modules provide a simple and easy to use interface to the controls systems. As is the culture of the C-AD Controls system, we try to empower our end

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA153

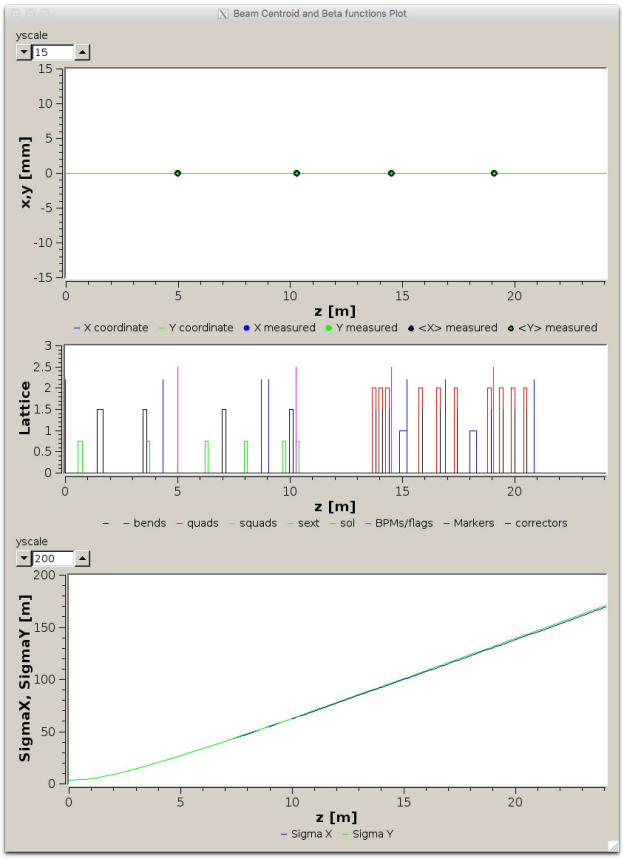

Figure 9: CeC Model page.

users as much as possible, enabling the operators, physicists, and other support groups to build their own interfaces to the controls. As these other groups make use of these new tools we expect feature requests, as well as bug reports.

What is more exciting is how these tools position the C-AD controls for the future, when it comes time to build the eRHIC controls. High level interfaces no longer have deep connections to low level controls interfaces. As the REST applications grow, we build a suite of interfaces that will require little modification for commissioning and operating eRHIC.

REFERENCES

[1] J. Skelly and J. Morris, in *Proc. ICALEPCS'99*, Trieste, Italy, Oct. 1999, pp. 42-24.

[2] L. Hoff and J. Skelly, in *Proc. ICALEPCS'93*, Berlin, Germany, Oct. 1993, Nucl. Instr. and Meth. A, p. 185, 1993.

[3] T. D'Ottavio *et al.*, presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper TUPHA157, this conference.

[4] TANGO, `http://www.tango-controls.org/`.

[5] T. D'Ottavio, B. Frak, S. Nemesure, and J. Morris, in *Proc. ICALEPCS'11*, Grenoble, France, Oct. 2011, pp. 40-43.

[6] EPICS, `http://www.aps.anl.gov/epics/`.

[7] K. Brown *et al.*, presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper TUPHA135, this conference.

[8] I. Pinayev, private communication.

[9] A. Fedotov *et al.*, in *Proc. NAPAC'16*, Chicago, IL, USA, Oct. 2016, pp. 867-869.

[10] V.N. Litvinenko *et al.*, in *Proc. IPAC'11*, San Sebastian, Spain, Sep. 2011, pp. 3442-3444.