AN UPDATE ON CAFE, A C++ CHANNEL ACCESS CLIENT LIBRARY, AND ITS SCRIPTING LANGUAGE EXTENSIONS

J. Chrin, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

CAFE (Channel Access interFacE) is a C++ client library that offers a comprehensive and easy-to-use interface to EPICS (Experimental Physics and Industrial Control System). Functionality is provided for the synchronous and asynchronous interaction of individual and groups of low-level control data, coupled with an abstraction layer to facilitate development of high-level applications. The code base has undergone major refactoring to make the internal structure more comprehensible and easier to interpret, and further interfaces have been implemented to increase its flexibility, in readiness to serve as the CA host in fourth-generation and scripting languages for use at SwissFEL, Switzerland's X-ray Free-Electron Laser facility. An overview of the structure of the code is presented, together with an account of newly created bindings for the Cython programming language, which offers a major performance improvement to Python developers, and an update on the CAFE MATLAB Executable (MEX) file.

INTRODUCTION

CAFE (Channel Access interFacE) [1, 2] is a modern, C++ client library that provides an intuitive and multifaceted user interface to the EPICS (Experimental Physics and Industrial Control System) [3] native C-based Channel Access (CA) Application Programming Interface (API) [4]. It allows for remote access to control data, encapsulated in Process Variables (PVs) residing in EPICS Input/Output Controllers (IOCs), while sheltering the user from the intricacies of programming with the native CA library. CAFE's conception arose from requirements foreseen by SwissFEL, Switzerland's X-ray Free-Electron Laser [5], coupled with the desire to avoid deprecated CA APIs propagating into a new project. Its development has since encompassed a renewed effort as requirements for application development for the forthcoming commissioning phase became apparent [6]. In particular, it was recognized that the effort afforded to a complete API, that further provided abstract layers for beam dynamics applications, could be readily incorporated into any C/C++ based scripting languages of choice. The main advantages to this approach are:

- The inherent simplicity and convenience of maintaining a single CA interface code. New CA functionalities from future EPICS 3 releases need only be integrated into a single base library.
- A uniform response to errors and exceptions that facilitates trace-backs.
- The CA class is well separated from the internals of the domain language meaning that bindings to other scripting and domain-specific libraries are vastly simplified.

An overview of the structure of the code is given, together with a discussion on design features that provide control over configurable components that govern the behaviour of interactions, and guarantee that the outcome of all method invocations are captured with integrity in every eventuality. Newly created bindings to Cython are then presented, and the improvement in performance yielded to Python developers is emphasized. An updated version of CAFE's MATLAB Executable (MEX) file [7], that exposes new functionalities to MATLAB [8] users, has also been made available.

CAFE C++ IMPLEMENTATION

The C++ interface to the EPICS Channel Access client library follows sound practices in CA programming [9] by placing careful attention to:

- Management of client-side CA connections.
- Memory optimization, particularly when connections are restored.
- Separation of data retrieval from its presentation.
- Strategies for converting between requested and native data types.
- Caching of pertinent data related to the channel and its state.
- Aggregation of requests for enhanced performance.
- Adaptive correction procedures, e.g. for network timeouts.

CAFE provides functionality for synchronous and asynchronous interactions for both single and groups of channels. All transactions report their data to a multi-index container provided by the Boost C++ libraries [10]. Here, the container takes ownership of instances of the "Conduit" object, each of which acts as the storage location for all data related to its associated PV. Multiple, distinct interfaces (or indices) provide convenient access to the object elements, allowing their data to be quickly retrieved or modified. The handle index (or object reference) has been configured with a unique key, and as such, acts at the definitive reference to the resource's data. To facilitate data access the underlying container is also indexed by PV, PV Alias, and the CA Channel Identifier. Callback functions have been implemented on all operations involving connection handlers, event handlers and access right handlers. Their invocation triggers their data to be written into the "Conduit" container object. In this way, the connection state of the channel, and all the channel's parameter values, are recorded with integrity. Memory to hold the channels data is allocated dynamically on first connection, and re-examined in the event of re-connection. Only one connection per channel is ever established, unless the channel is also member of a synchronous group (which itself

ISBN 978-3-95450-148-9

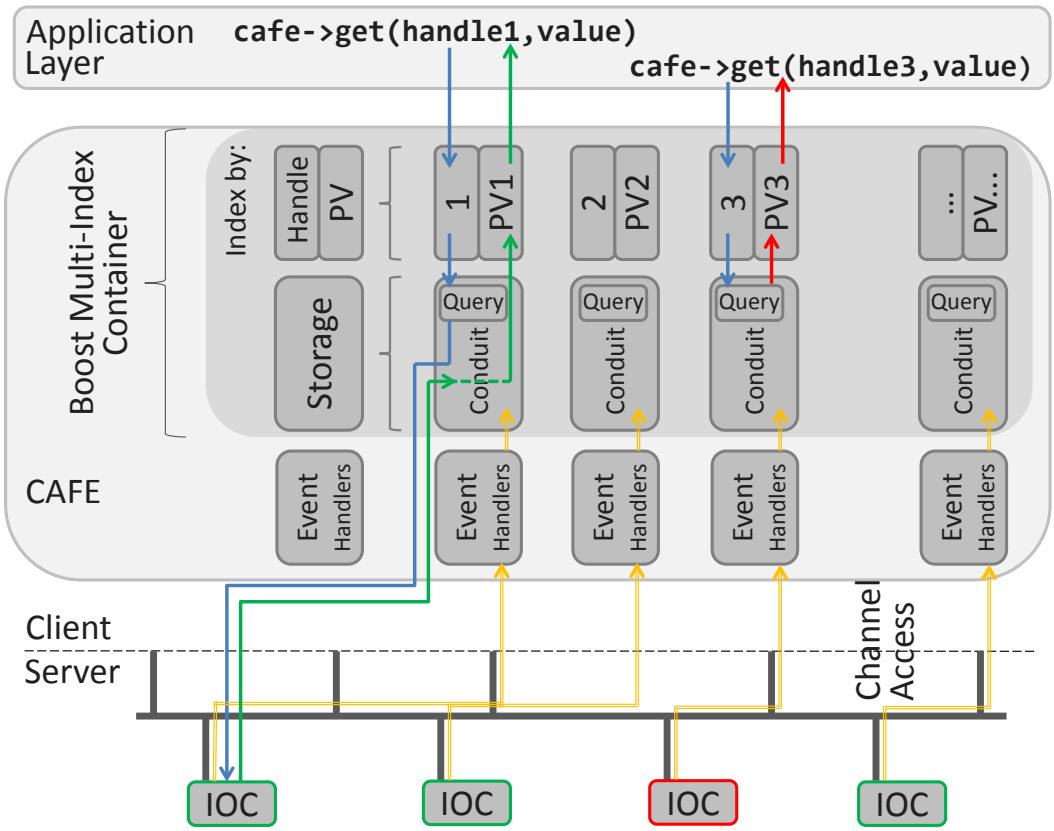

Figure 1: The information flow for a cafe method invocation in the case of a connected channel, PV1 (green), and a disconnected channel, PV3 (red). The multi-index container ("Conduit" object) serves as the data store for the full complement of the PV's data, whether static or dynamic. The handle (index) is the reference to the resource's data. PV data emitted from the IOC is recorded within the container (yellow); cafe method invocations first query the container to assess whether the prerequisites for launching a message have been met.

exists as a logical software entity), in which case a separate virtual circuit is created and a new handle assigned.

All commands first query the container to establish whether a message need be sent over the network. This is illustrated in Fig. 1, which shows the response to a method invocation on a connected channel, PV1, and disconnected channel, PV3. Only if the prerequisites for the method invocation are satisfied, e.g. the channel is connected, is the message sent to the IOC. A hierarchy of software procedures ensures that the most appropriate, and immediate, error message is reported to the client. For instance, a message to PV3 will report either that the channel is presently disconnected or has never been connected, if that be the case.

Policy classes provide control over configurable components that govern the behaviour of interactions, either on a global or individual basis. These include policies that determine procedures to establish connectivity, blocking or non-blocking method invocations, a strategy for converting between data types that are requested and offered, and allow for network time-outs to adapt dynamically.

An XML configuration mechanism provides a convenient framework for users to define and initialize CAFE groups. Transactions on CAFE group objects are invoked through simple intuitive method invocations that reference groups through their identifier.

In addition to refactoring the internal structure of the code, the CAFE interface has also been extended to facilitate bindings to scripting languages (e.g. C++ vectors map directly onto Python lists). Among the new features that have been added is a mechanism to enable and simplify multi-dimensional scan procedures.

SCRIPTING AND DOMAIN-SPECIFIC LANGUAGE EXTENSIONS

CAFE, by design, was not only intended to be sufficiently expressive to provide the required abstractions in a convenient way, but also flexible enough to act as a CA gateway for other, underlying C/C++ based programming languages. As discussed in [2], the idea of providing a single C++ library for use across a number of scripting and domain-specific languages (DSLs) enforces a logical boundary between components of the CA API and the specifics of a given domain's C/C++ extension framework. In this way, the CA classes are not confined to the system in which they execute. Code reusability avoids repetition, reduces maintenance,

ISBN 978-3-95450-148-9

and vastly simplifies the creation of bindings. Indeed, exposing CA functionality to a given domain begins to follow a definite and recognizable pattern, which, after careful validation of input arguments, largely reduces to a mapping of CAFE/C++ data types to their domain equivalent. The CAFE C++ library has sufficient breadth and maturity to act as a suitable host for scripting languages. Whatever it may be missing, should be typically straightforward to address. Bindings have been provided for Python and MATLAB, which are the languages of choice for physics application developers at SwissFEL.

CyCafe: Syphoning CAFE with Cython

Cython [11] is a programming language that offers a Python-like style of coding while maintaining the performance advantages of C. Its prime capabilities are to compile Python code into C or C++ source code, and to interface with external C/C++ libraries. With both the powerful Cython (version 1.22) constructs and the full Python language available for disposal, an optimized Python interface has been developed for CAFE (CyCafe). The Python extension module is named PyCafe. A number of CAFE methods exposed to PyCafe are shown in Listings 1 and 2. The improvement in performance for a single channel operation is a healthy factor of four compared to a pure Pythonic channel access operation. Some important and novel particularities of providing the CyCafe API are revealed in the following.

Python Buffer Protocol: The introduction of a new buffer protocol allows a number of Python built-in types, such as `memoryview`, the `ndarray` arrays from the much used NumPy scientific computing package [12], and other objects that implement the protocol, to share their data without the need for copying. Cython can similarly extend the buffer protocol to work with data arising from external libraries. CyCafe consequently provides interfaces that cater for the new buffer protocol. In particular, their use in reading/writing waveforms results in a marked improvement in performance.

C Function Pointers and Callbacks: Cython supports C function pointers allowing C functions, that take function pointer callbacks as input arguments, to be wrapped. This feature allows users to pass a Python function, created at runtime, to control the behaviour of the underlying C function. Using this methodology, a Python callback function may be easily supplied for any asynchronous CA interaction. This is highlighted in Listing 2, where a monitor on a PV is activated. A novel aspect of the CyCafe interface, here, is that only the handle (i.e. object reference) is, and need be, reported back to the callback function. Since the CAFE API takes the provision to cache the data in its internal storage area, defined by the multi-index container, the user may call upon any one of a number of CAFE methods that retrieve data directly from the cache, as show in line 4 of Listing 2. The precedence of sifting through Python dictionaries is obviated.

Listing 1: PyCafe Read/Write Examples

```
import PyCafe
cafe = PyCafe.CyCafe()
cyca = PyCafe.CyCa()

#handlePV=<handle/'pvName'>
#dt=<'int','float','str','native'(default)>
#hpvList=<hList/pvList> i.e. handle/'pvName' list
#s gives overall status, sList is a status list
pvList=['pv1','pv2','pv3','pv4']
try:
  handle= cafe.open('pvName') #returns obj. ref.
  hList = cafe.open( pvList ) #ret. obj. ref. list
except Exception as inst:
  print inst

#Synchronous Single Channel Operations
try:
  #get value in native type
  value = cafe.get(handlePV)
  #returns structured data
  pvData= cafe.getPV(handlePV, dt='float')
  #write operation
  cafe.set(handlePV, pvData.value+0.001 )
  #waveform, return list in native type
  valList = cafe.getList (handlePV)
  #waveform, return memoryview of floats
  memview = cafe.getArray(handlePV, dt='float')
  #waveform, return numpy array in native type
  npArray = cafe.getArray(handlePV, asnumpy=True)
  #set waveform; input [values] may be any of
  #list, memoryview, numpy.ndarray, array.array
  cafe.set(handlePV,[values])
  #Get cached controls data
  pvCtrl = cafe.getCtrl(handlePV)
  pvCtrl.show() # print all control parameters
  print "units = ", pvCtrl.units
  print "precision = ", pvCtrl.precision
  print "enum options (if any): ", pvCtrl.enum
except Exception as inst:
  print inst

#Synchronous Multiple Channel Operations
valList,s = cafe.getScalarList(hpvList)
s,sList   = cafe.setScalarList(hpvList, valList)

#Asynchronous Single/Multiple Channel Operations
s,sList = cafe.getAsyn(hList)
s,sList = cafe.waitForBundledEvents(hList)
pvData  = getPVCache(hList[0])

#Synchronous Groups
#gHandleName=<groupHandle/'groupName'>
s = cafe.defineGroup('groupName', pvList)
gHandle = cafe.openGroup('groupName')
valList,s,sList = cafe.getGroup (gHandleName)
s,sList = cafe.setGroup (gHandleName, valList)
#returns list of structured data
pvgList = cafe.getPVGroup(gHandleName,dt='str')

cafe.terminate() #tidy up
```

ISBN 978-3-95450-148-9

Listing 2: PyCafe Monitor Example

```
import PyCafe
cafe = PyCafe.CyCafe()
cyca = PyCafe.CyCa()

#Callback function
def py_callback(handle):
  #Any method that retrieves data from cache
  pvData=cafe.getPVCache(handle)
  pvData.show()
  return

#Start Monitor
monID=cafe.monitorStart(handle, cb=py_callback,
    dbr=cyca.CY_DBR_TIME, mask=cyca.CY_DBE_VALUE|
    cyca.CY_DBE_ALARM)
...
status=cafe.monitorStop(handle, monID)

cafe.terminate() #tidy up
```

Global Interpreter Lock (GIL): Cython uses the CPython/API to access C-level code. CPython's memory management is not, however, thread-safe, necessitating a dedicated mutex, the Global Interperter Lock (GIL), to ensure that only one native thread executes Python bytecodes at any given time. External C code that does not interact with Python objects can, however, be executed without the GIL in effect, and thus achieving thread-based parallelism. All methods that access the low-level hardware through CA are done so in a non-GIL context. Without taking this necessary step of releasing the GIL, PyCafe will otherwise hang, particularly in cases where callbacks are involved. A monitor callback can still be invoked even while another CAFE operation is being invoked. All CyCafe methods that interact over the network are consequently called in a non-GIL context, i.e. with the GIL released.

MOCHA: A MATLAB Executable File for CAFE

MOCHA (MATLAB Objects for CHannel Access) is a MATLAB Executable (MEX) file that provides CA functionality to MATLAB through CAFE. MOCHA supports all MATLAB data types, and benefits from MATLAB's 64-bit indexing functionality, granting cross-platform (32/64-bit) flexibility. MOCHA methods may be invoked directly without the need to run additional MATLAB scripts.

The package has been put into good effect at the SwissFEL Injector Test Facility (SITF) [13], where it proved to be a stable and robust API that served the extensive needs of applications developers. The MOCHA MEX file, previously presented in [7], has since been consolidated with the updated CAFE library. As is the recommended practice, a separate MEX file has been compiled for each release version of MATLAB.

SUMMARY

The CAFE C++ Channel Access interface library has been refactored, extended and consolidated, in preparation to serve as the Channel Access host to scripting languages in use for beam dynamics applications at SwissFEL. In particular, CAFE's new Cython interface exposes an extensive set of CA functionality to Python developers and offers a significant improvement in performance. The entire CAFE package may be downloaded from the CAFE website [1]. It would be a relatively straightforward task to extend the scope of this work to other C/C++ based languages once the specifics of the domain's C/C++ extension framework have been grasped.

REFERENCES

[1] CAFE, http://ados.web.psi.ch/cafe/.

[2] J. Chrin and M.C. Sloan, "CAFE, A Modern C++ Interface to the EPICS Channel Access Library", in *Proc. 13th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'11)*, Grenoble, France, Oct. 2011, paper WEPKS024, pp. 840–843.

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

[4] J.O. Hill and R. Lange, "EPICS R3.14 Channel Access Reference Manual", http://www.aps.anl.gov/epics/docs/ca.php

[5] "SwissFEL Conceptual Design Report", R. Ganter, Ed. PSI, Villigen, Switzerland, Rep. 10-04, Version Apr. 2012.

[6] T. Schietinger, "Beam Commissioning Plan for the SwissFEL Hard X-ray Facility", presented at the 37th Int. Free-Electron Laser Conf. (FEL'15), Daejeon, Korea, Aug. 2015, paper MOP017.

[7] J. Chrin, "MATLAB Objects for EPICS Channel Access", in *Proc. 14th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'13)*, San Francisco, CA, USA, Oct. 2013, paper MOPPC146, pp. 453–456.

[8] MATLAB®, http://www.mathworks.com/products/matlab/.

[9] D. Zimoch, "Channel Access Client Programming", EPICS Collaboration Meeting, 27-29 Jul. 2009, NFRI, Daejeon, Korea, http://www.aps.anl.gov/epics/meetings/2009-07/.

[10] Boost Multi-index Containers Library, http://www.boost.org/libs/multi_index/.

[11] Cython C-Extensions for Python, http://cython.org/.

[12] NumPy, http://www.numpy.org/.

[13] T. Schietinger *et al.*, "Commissioning Experience and Beam Physics Measurements at the SwissFEL Injector Test Facility", in preparation.

ISBN 978-3-95450-148-9