16th Int. Conf. on Accelerator and Large Experimental Control Systems ICALEPCS2017, Barcelona, Spain JACoW Publishing
ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA174

CUMBIA: A NEW LIBRARY FOR MULTI-THREADED APPLICATION DESIGN AND IMPLEMENTATION

Giacomo Strangolino, Elettra, Trieste, Italy

Abstract

Cumbia is a new library that offers a carefree approach to multi-threaded application design and implementation. Written from scratch, it can be seen as the evolution of the *QTango* library [1], because it offers a more flexible and object oriented multi-threaded programming style. Less concern about locking techniques and synchronization, and well defined design patterns stand for more focus on the work to be performed inside *cumbia activities* and reliable and reusable software as a result. The user writes *activities* and decides when their instances are started and to which thread they belong. A token is used to register an *activity*, and activities with the same token are run in the same thread. Computed results can be forwarded to the main execution thread, where a GUI can be updated. In conjunction with the *cumbia-tango* module, this framework serves the developer willing to connect an application to the TANGO control system. The integration is possible both on the client and the server side. An example of a TANGO device using *cumbia* to do work in background has already been developed, as well as simple *QT* [2] graphical clients relying on the framework.

COMPONENTS

Cumbia Modules

Cumbia is a set of distinct modules; from lower to higher level:

- *cumbia*: defines the *Activities*, the multi thread implementation and the format of the data exchanged between them;
- *cumbia-tango*: integrates *cumbia* with the TANGO control system framework, providing specialised *Activities* to read, write attributes and impart commands;
- *cumbia-epics*: integrates *cumbia* with the EPICS control system framework. Currently, only variable monitoring is implemented;
- *cumbia-qtcontrols*: offers a set of *QT* control widgets to build graphical user interfaces. Inspired by the *QTango*'s *qtcontrols* components, they have been enhanced and sometimes rewritten to look more stylish and friendly. The module is aware of the *cumbia* data structures though not linked to any specific engine such as *cumbia-tango* or *cumbia-epics*.
- *qumbia-tango-controls*: written in *QT*, is the layer that sticks *cumbia-tango* together with *cumbia-qtcontrols*;
- *qumbia-epics-controls*: written in *QT*, the component pairs *cumbia-epics* to *cumbia-qtcontrols*.
- *qumbia-apps*: a set of applications written in QT that provide elementary tools to read and write values to the TANGO and EPICS control systems.

Combining together the modules allows to instantiate a control system engine and build command line or *QT* graphical user interfaces effortlessly. Engines can coexist within the same application to seamlessly control devices belonging to separate control systems. Figure 1 shows how modules are interrelated.

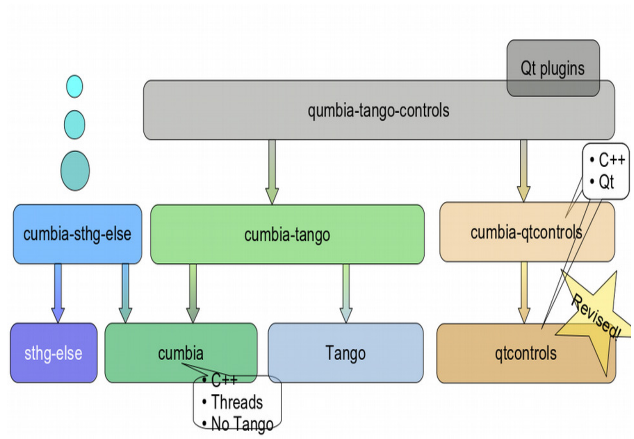

Figure 1: Relationships amongst *cumbia* modules.

CUMBIA

Cumbia is the name of the lower layer of the collection, as well as the name of a single object every application must hold in order to use its *services*.
In asynchronous environments, *threads* have always posed some kind of challenge for the programmer. Shared data, message exchange, proper termination are some aspects that cannot be overlooked. The *Android AsyncTask* [3] offers a simple approach to writing code that is executed in a separate thread. The API provides a method that is called in the secondary thread context and a couple of functions to post results on the main one.

Activities

Cumbia *CuActivity*'s purpose is to replicate the carefree approach supplied by the *AsyncTask*. In this respect, a *CuActivity* is an interface to allow subclasses to do work within three specific methods: *init*, *execute* and *onExit*. Therein, the code is run in a separate thread. The *publishProgress* and *publishResult* methods hand data to the main thread. To accomplish all this, an *event loop* must be running. By an initial parametrization, either a custom one (such as *QT*'s, used in *qumbia-qtcontrols*) or the builtin *cumbia CuEventLoop* can be installed. New activities must be registered in the *CuActivityManager* service, and unregistered when they are no longer needed. In this way, a *token* can be used to group several activities by a smaller number of threads. In other words, activities with the same token run in the same thread. Thread

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA174

implementation in *Cumbia* requires a compiler supporting the C++11 standard.

Services

By means of the reference to the *Cumbia* instance, that must be maintained throughout the entire life of an application, you can access services. They are registered in the *CuServiceProvider* and accessed by name. The activity manager, the thread and the log services are some examples, but others can be written and installed, as long as they adhere to the *CuServiceI* interface (e.g *cumbia-tango*'s *CuActionFactoryService* and *CuDeviceFactoryService*). *Cumbia* can be subclassed in order to provide additional features specific to the engine employed. *CumbiaPool* allows to register and use multiple engines in the same application. Services have been conceived with the *service provider* design pattern in mind.

Data Interchange

Data transfer is realised with the aid of the *CuData* and *CuVariant* classes. The former is a bundle pairing keys to values. The latter memorises data and implements several methods to store, extract and convert it to different types and formats. The *cumbia-qtcontrols* module handles these structures to provide a primary data display facility, unaware of the specific engine underneath (*TANGO, EPICS, ...*)

CUMBIA-TANGO

cumbia-tango integrates *cumbia* with the TANGO control system framework, providing specialised activities to read, write attributes and impart commands.

Implementation

The *CumbiaTango* class is an extension of the *Cumbia* base one. Its main task is managing the so called *actions*. An *action* represents a task associated to either a TANGO device attribute or a command (called *source*). Read, write, configure are the main sort of jobs an action can accomplish. More types of actions are foreseen, such as multiple readings or writings in sequence. *CuTangoActionI* defines the interface of an action. Operations include adding or removing data listeners, starting and stopping an action, sending and getting data to and from the underlying thread (for example retrieve or change the polling period of a *source*). *CuTReader* implements the interface and holds a reference to either an *activity* intended to receive events from TANGO or another one designed to poll a source. Figure 2 describes these relationships.

Activities is where the TANGO connection is setup, database is accessed for configuration, events are subscribed, a poller is started or a write operation is performed. This is done inside the thread safe *init*, *execute* and *onExit* methods, invoked from another thread. Progress and results are forwarded by the *publishProgress* and *publishResult* methods in the activity and received in

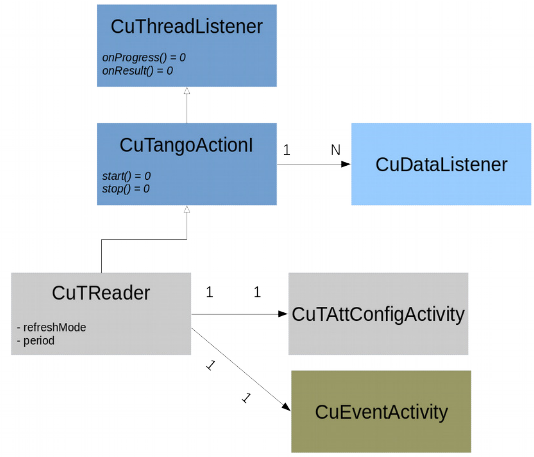

Figure 2: Diagram for the relationships between objects making up a TANGO reader.

the *onProgress* and *onResult* implemented by the action. Therein, *CuDataListener*'s *onUpdate* method is invoked with the new data. Reception safely occurs in the main thread. As previously stated, activities identified by the same *token* (a *CuData* object) belong to the same thread. *cumbia-tango* groups threads by TANGO device name.

CUMBIA-QTCONTROLS

This module combines *cumbia* and the *QT* cross platform software framework, offering graphical control system components. Labels, gauges and advanced graphs are supplied, as well as buttons and boxes to set values. As mentioned earlier, elementary data representation is provided, due to the component unawareness of the *cumbia* engine lying beneath. In order to display real data on the controls, you have to combine different building blocks at the moment of setting up each reader or writer in your application, as described later. When data is ready, it is delivered to the main thread through the *onUpdate* method that the control component (such as a label) must implement, for the reason that it inherits from the *CuDataListener* interface (see Figure 3).

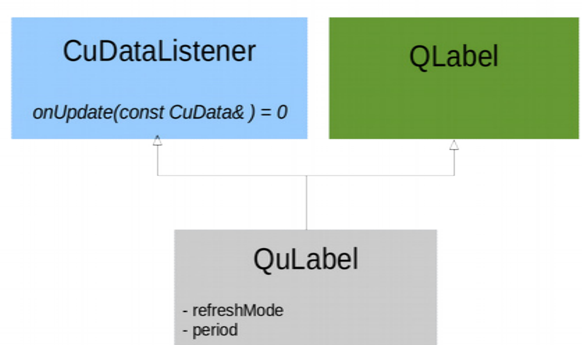

Figure 3: Diagram for the relationships between the classes involved in a graphical control widget design.

For an *event loop* must be executing, messages are posted to the main thread relying on an implementation of the *CuThreadsEventBridge_I* interface. In *QT*, we use *QCoreApplication*'s event loop in conjunction with

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA174

cumbia-qtcontrols' *QThreadsEventBridge*, which exploits *QCoreApplication*'s *postEvent*, a familiar scheme for *QT* developers. From within *onUpdate*, data is extracted and presented to the user by way of the control widget.

Conceived to associate with widgets, even though not related to them, are a couple of abstract classes that define an interface to readers and writers, namely *CuControlsReaderA* and *CuControlsWriterA*. Implementations of methods to set and remove *sources* and *targets* of execution are required, as well as means to send and receive messages to and from *actions*. They also keep references to the currently active *Cumbia* and data listener instances. Object composition to accomplish the set up of a TANGO (EPICS) reader (writer) will be discussed later.

The *strategy* design pattern [4] offers a method to install engine specific interpreters on the available widgets. For instance, a TANGO aware plot can provide a strategy to correctly deal with with *warning* and *alarm* ranges and the attribute history.

QUMBIA-TANGO-CONTROLS

qumbia-tango-controls, a component written in QT, combines *cumbia-tango* with *cumbia-qtcontrols*.

CuTControlsReader and *CuTControlsWriter* are the implementors of the previously discussed *CuControlsReaderA* and *CuControlsWriterA* abstract classes. Their *sources* and *targets* are TANGO attribute and command names, written with the same syntax as that adopted by *QTango*. They operate on a *CumbiaTango* instance, which is in charge of creating and registering *actions* exploiting the *CuActionFactoryService*, finding *actions* already in use and managing installation and removal of *CuDataListener* elements. These features specialise *CumbiaTango* with respect to the *Cumbia* base class. Refer to Figure 1 for a graphical representation of the relationships amongst *cumbia* modules and the aggregating role of *qumbia-tango-controls*.

Object Composition to Create a TANGO Reader

Figure 4 shows a class diagram illustrating the relationships between the objects involved in the set up of a TANGO reader intended to display values on a label.

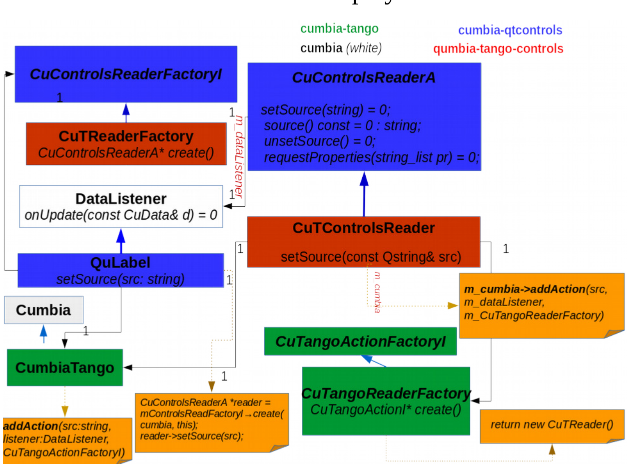

Figure 4: Relationships between the objects involved in the setup of a TANGO reader for a *QuLabel*.

A first observation concerns the mediation of *factories* to provide instances of specific objects. From the compositional and chronological perspective, *qumbia-tango-controls*' *CuTReaderFactory* creates a *CuTControlsReader* within the *setSource* method of a *QuLabel* that has been set up with *CumbiaTango* and *CuTReaderFactory* as arguments. *CumbiaTango* and the *QuLabel* (as a *CuDataListener*) parametrize *CuTControlsReader* too at creation time through the *CuTReaderFactory*. *QuLabel*'s *setSource* finally invokes the method with the same name on the *CuTControlsReader*.

The latter requests *CumbiaTango* to find an existing *action* or to create a new one for the desired *source*. Since *qumbia-tango-controls* and *cumbia-tango* are decoupled, TANGO *action* creation is again delegated to a factory, one of the implementations of *CuTangoActionFactoryI*, in this case a TANGO reader factory. Once a reference to a new or preexisting *action* has been obtained, the listener (*QuLabel*) is added. The last consideration implies that different listeners attached to the same *source* share the same *action*. Changing the settings for that action (read mode, polling period and so on) affects therefore all listeners.

Figure 5 graphically illustrates what has just been described. The sequence diagram of the initialization of a reader by means of the *setSource* method is represented. Note that, after the asynchronous message *registerActivity* at the bottom of the chart, *QuLabel* will receive updates from TANGO as an implementor of *CuDataListener*. Inside the very same method, the *token* chosen to register the activity is used to determine whether a new thread has to be created or an existing one can be adopted. *CumbiaTango*'s choice is to group threads by device name.

Object Disposal

Object creation alike, the disposal of a *cumbia* reader requires a certain degree of complexity in order to grant components independence. Taking again *QuLabel* as example, upon deletion the reader factory is first destroyed, then *CuTControlsReader*. It calls *unsetSource* to unlink the the *QuLabel* as a data listener of its associated *action* (i.e. *CuTReader*). If there are no more listeners, *CuTReader* stops itself issuing an *unregisterActivity* request to *CumbiaTango*.

The analysis hitherto developed shows how *cumbia* modules are independent from each other. Object composition is required to make elements aware of a specific control system.

QUMBIA-APPS

qumbia-apps module provides a set of base applications to perform elementary actions on *sources*, such as readings and writings. The *generic_client* tool is a graphical panel able to read and write from both TANGO and EPICS, using labels to display the current value and plots to show the trend over time or the present values, if the format is a vector. Figure 6 is a screenshot of the *generic_client* reading a TANGO scalar attribute, a TANGO spectrum and an EPICS *analog input*.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA174

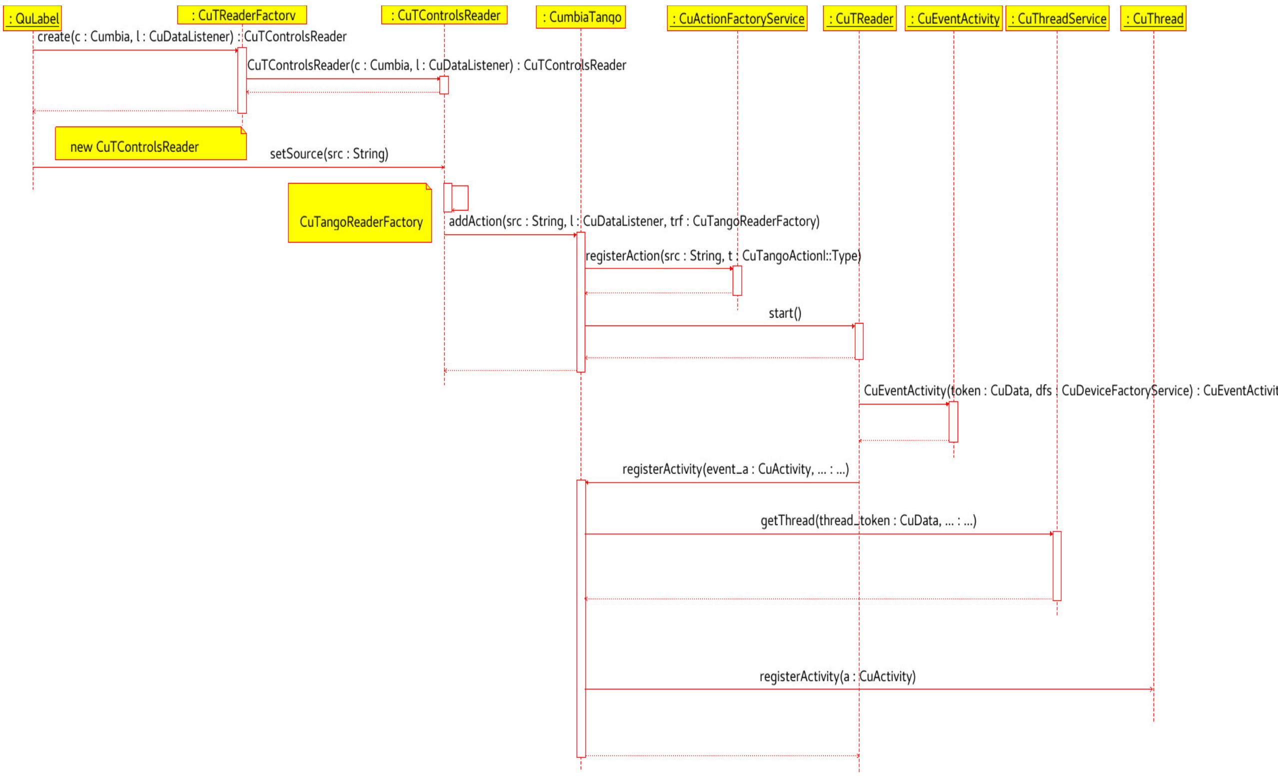

Figure 5: Sequence diagram of the initialization of a reader. After the asynchronous message *registerActivity* at the bottom, *QuLabel* receives updates from TANGO as a *CuDataListener* implementor.

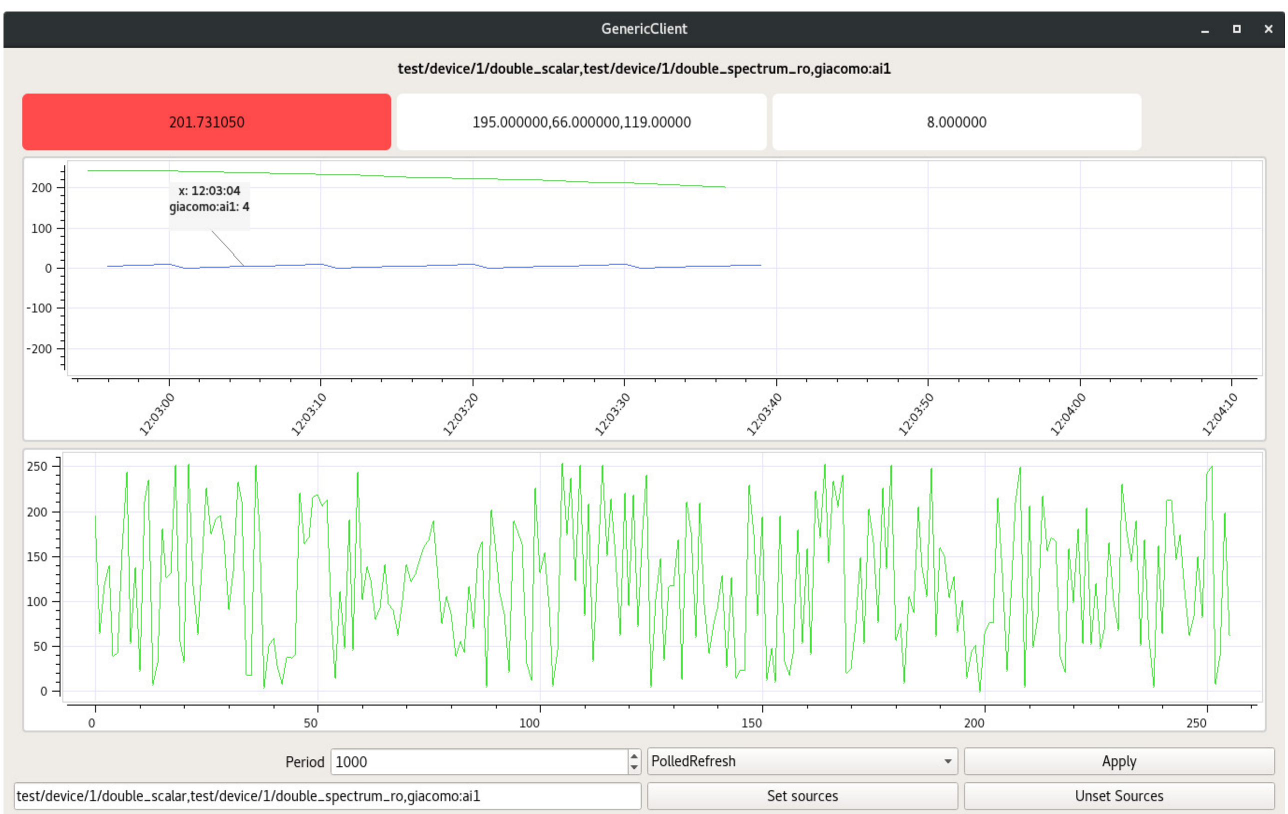

Figure 6: *generic_client* reading a TANGO scalar and spectrum attribute and an EPICS analog input.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA174

CONCLUSION

The *QTango* library, currently in use at the Elettra Synchrotron Radiation Facility, Trieste, Italy, has proved to be stable, reliable and efficient throughout the years. It makes TANGO development easy and fast, handing the programmer a set of widgets already covering the great majority of needs to build control room applications. Extending existing *QTango* components is very easy and creating new readers and writers is just a matter of subclassing and reimplementing one or two methods. All the control room applications for the FERMI@Elettra, the seeded free electron laser (FEL) facility, rely on the *QTango* framework. Nevertheless, many of the features offered are not required daily whereas some of them are not easy to implement (e.g. multiple serialised readings). *QTango* is tightly bound to TANGO, the architecture is somehow complicated and the code is not modular nor reusable enough. On the other hand, *cumbia* is made up of standardized units for easy construction or arrangement. Its lowest level can be seen as a bare C++ library suggesting another approach to multi threading, the so called *activities* (see the *Cumbia* section). They allow to simply group workers by means of a *token* and define a simple dictionary based structure for *thread safe* data interchange. The other components use QT, TANGO, EPICS in conjunction with the base library to fulfil more specific tasks. In other words, you can use *cumbia* to write a client-server chat application, *cumbia* and *cumbia-tango* to write a TANGO device server or a C++ command line program, *cumbia*, *cumbia-tango*, *cumbia-qtcontrols* and q*umbia-tango-controls* for a graphical user interface. *Cumbia* has been conceived to be lightweight, fast, scalable and easily extensible in the future. Adding characteristics is a matter of writing *activities*, registering and deregistering them in *cumbia*. The extensive adoption of the *bridge* design pattern ([4] and [5]) in the interior of most classes ensures binary compatibility at every stage of the future development. The C++ code employs the *listener/callback* pattern for asynchronous notifications, while the QT modules avail themselves of the *signal/slot* model. The *abstract factory* and *factory method* models [4] do away with the coupling between components. Finally, the *strategy* pattern can be applied to tailor generic graphical components to individual control system engine characteristics. Just as *QTango*, *cumbia* is equipped with QT designer *plugins* to quickly shape a graphical user interface for control systems.

ACKNOWLEDGEMENT

I would like to thank Stefano Cleva for his support for the initial set up of a minimal EPICS environment.

REFERENCES

[1] G. Strangolino *et al.*, "Control Room Graphical Applications for the Elettra New Injector", *Proceedings of PCaPAC08*, Ljubljana, Slovenia, 2008.

[2] QT, Cross-platform software development for embedded & desktop, *https://www.qt.io/*

[3] Android AsyncTask, from the Android developer guide, *https://developer.android.com/*

[4] Erich Gamma *et al.*, *Design Patterns, Elements of Reusable Object-Oriented Software*, October 1994.

[5] D-Pointer or opaque pointer design pattern, *wiki.qt.io/D-Pointer*