INTEGRATION OF ALMA COMMON SOFTWARE AND NATIONAL INSTRUMENTS LABVIEW

K. Žagar, A. Žagar, Cosylab, Ljubljana, Slovenia
B. Bauvir, G. Chiozzi, P. Duhoux, European Southern Observatory (ESO), Garching, Germany

Abstract

Among the candidate technologies for the Extremely Large Telescope (E-ELT) are ALMA Common Software (ACS) and LabVIEW. ACS is a CORBA-based control system infrastructure that implements a container-component model. It allows developers to focus on development of components that define application logic, with ACS-provided containers addressing infrastructural issues of distributed control systems such as remote procedure calls, logging, configuration, etc. LabVIEW is a commercial solution provided by National Instruments which allows rapid construction of user interfaces and control loops. Control loops can execute on Windows and Linux operating systems, as well as real-time control systems and FPGA circuits. In this paper, we present an approach for integration of ACS and LabVIEW. We accessed ACS from a LabVIEW user interface (both sending of data into ACS, and receiving data from ACS). Also, we accessed a real-time LabVIEW process (parts of which were executing in FPGA) from ACS – again in both directions. From the LabVIEW perspective, the approach is platform-independent as it is based on a Simple TCP/IP Messaging protocol.

INTRODUCTION

ACS [1] facilitates development of complex control systems where many types of devices need to be installed, monitored, controlled and managed, and where numerous instances of each device type exist. Also, ACS facilitates development of higher-level, non-real time control algorithms, used to coordinate work of many devices in the system as mandated by an organizational workflow.

Development in ACS is component-based: each entity in the system, whether driver interacting with a physical device, or a higher-level controller, is represented as a component. Component is defined by a name that uniquely identifies it within the system, interface (what operations and attributes it supports) and the code-base which implements the interface. Also, each component can have configuration data associated with it that specifies information relevant at run-time (e.g., hardware addresses). This approach simplifies composition of complex systems, as components can only interact with each other through well-defined interfaces.

Technologically, ACS is built atop CORBA middleware, and components can be developed in C++ (where real-time behavior must be achieved), Java (higher-level logic) and Python (testing, scripting).

Apart from remote procedure calls for interaction between components (physically implemented with CORBA invocations), ACS also provides a message-oriented approach where components can publish messages or subscribe to them. CORBA Notification Service is used as the underlying technology.

ESO started development of ACS in year 2000. It is now used by several astronomy projects and a synchrotron radiation source. The ACS is available under and open-source license.

LabVIEW [2] is a product developed by National Instruments which features an environment for developing control graphical user interfaces and control loops.

LabVIEW enables developers to efficiently construct graphical user interfaces. The composition of the user interface is visual (*what-you-see-is-what-you-get*). The rich assortment of graphical widgets results in a visually appealing panel with which the operator interacts (see Figure 1). Also, the performance and responsiveness of the LabVIEW user interface is in most cases sufficient. LabVIEW user interfaces can run on Windows and Linux platforms, and require a LabVIEW execution run-time.

LabVIEW provides a graphical programming

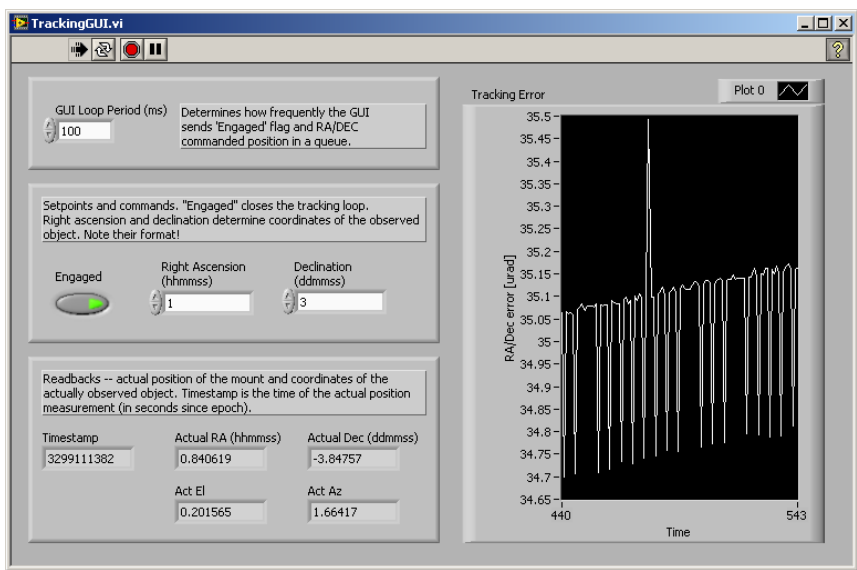

Figure 1: Example of a graphical user interface constructed with LabVIEW.

environment where a developer can represent data flows with wires and nodes (virtual instruments – VIs): a node can have several input wires, performs an operation (e.g., an addition node would add the values of all inputs), and produces the result on the output wires. Graphical widgets (controls and indicators) can also be used as sources/sinks for wires. VIs exist that can interact with inputs and outputs (e.g., analog/digital I/O, serial communication, dedicated devices, etc), and VIs for common mathematical, statistical, and other operations are pre-prepared. Through graphical programming, it is thus possible to construct control loops that can be controlled through user interaction, and their graphical representation mimics the traditional notation used in control engineering.

Control loops can execute on various platforms where LabVIEW runtime is available, among them: Windows and Linux, real-time extension for Windows (RTX), and real-time operating systems (PharLap ETS and VxWorks). With LabVIEW, it is also possible to develop control loops that execute on FPGA and interact with real-time CompactRIO processes.

The two technologies are thus largely **complementary**. Were it not possible to use LabVIEW to develop control loops and user interfaces, and ACS to manage scalable distribution of data across the control system? Investigation of approaches on how to achieve this was the goal of the project whose findings we present here-in.

APPROACHES

We have studied three approaches to integrate ACS with LabVIEW.

In the first approach, one would use the LabVIEW's ***Call Library Node*** **mechanism**, which allows invocation of functions that must be supplied in dynamically loadable/shared libraries (DLLs). These functions, usually written in C, would then interact with ACS[*]. For example, to make an invocation of an operation on an ACS component, the C function would first retrieve a reference to the component from ACS (which it would refer to by name, given as one of the inputs to the call library node), and then invoke a method. In C++, this mechanism can not easily be made generic – CORBA *Dynamic Invocation Interface* (DII) mechanism would need to be used for a generic solution, or else the method name would need to be hard-coded. This approach has been used in EPICS [3], the Virgo projects [4] and TANGO [5], where a generic solution was more easily achievable, as these control system infrastructures use a narrow interface.

In ACS, this approach is more applicable for message-oriented communication, where the API to the notification service remains unchanged from application to application, and only structure of messages is application-specific. Conversion of message structures from LabVIEW clusters to C++ structs can be generalized, as LabVIEW is capable of providing meta-data describing its clusters. However, if the C++ structure would not match the LabVIEW cluster (either in order or type of its members – a likely scenario especially during development), this would result in a serious run-time failure.

As ACS provides significant infrastructure at all of its nodes (container providing transparent access to logging, configuration, alarms, remote procedure calls, etc), significant portions of that infrastructure would need to be embedded in the LabVIEW process. For platforms where ACS is readily supported (e.g., Linux) this would not have been problematic. On Windows, a large portion of ACS would need to be adapted to support the platform. Easiest way to achieve this is to compile using libraries that expose the Windows API with Linux-style interface, e.g., *Cygwin* [6]. This approach was taken by R. Lemke, and he reports it to be largely successful, but there are issues with clashes of *Cygwin* library and LabVIEW. On other LabVIEW-supported platforms (e.g., VxWorks), yet a different solution would be necessary.

In the second approach, LabVIEW provided mechanisms for inter-process communication would be leveraged, such as ***data sockets*** [7] or ***shared variables*** [8].

Data sockets mandate an architecture where a data socket server runs on a Windows host, and with which all interacting nodes (LabVIEW GUIs or control loops) communicate to write or read data. The socket server is also available via an ActiveX interface. Though readily available, the solution has some drawbacks: its scalability and reliability is impaired (a single data sockets server), the communication protocol provides limited mechanisms for error detection and fault tolerance, mandates a Windows host and the data sockets API implementation is insufficiently portable.

Shared variables are a more novel approach to sharing process variable data between processes. LabVIEW offers a wide range of options to configure them, including ability to achieve real-time performance over a dedicated Ethernet network by carefully scheduling transmission times of messages.

At this time, however, no API outside LabVIEW exists that would enable external processes to exchange data with LabVIEW. Also the on-the-wire protocol (based on UDP/IP or TCP/IP) is proprietary. Furthermore, the level of support for shared variables varies from platform to platform, and on Linux, for example, it is not yet adequate.

ARCHITECTURAL OVERVIEW

Our application (controlling azimuth and elevation axes of a telescope) consisted of the following building blocks:

- A LabVIEW FPGA loop for motion control.
- A LabVIEW user interface for monitoring and controlling the position of the telescope.

[*] Integration with C++ is also possible, but name mangling must be prevented (e.g. with "extern C" declarations or by explicitly listing DLL exports).

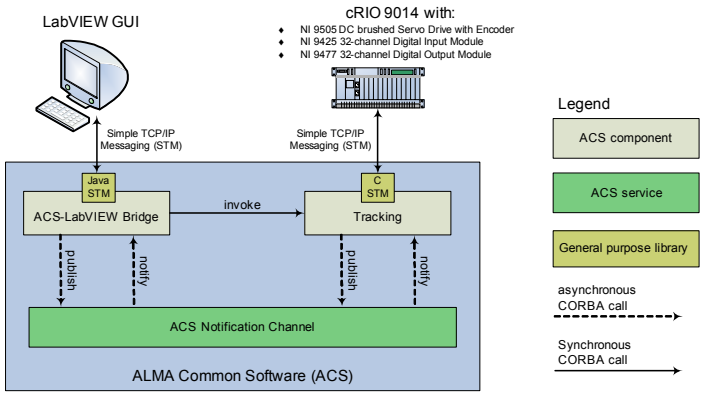

Figure 2: Architectural overview of the chosen approach.

- A LabVIEW real-time process implementing the tracking loop. This process was running on CompactRIO platform (VxWorks operating system) and used SLA library [9] for computing the apparent position of tracked objects.
- A Java ACS component with which the LabVIEW GUI communicates to read and write data from/to ACS.
- A C++ ACS component which communicates with LabVIEW control loop on CompactRIO to provide reference values and read the actual position.

The building blocks are graphically depicted in Figure 2. To implement communication between LabVIEW and a C++/Java ACS component, we have used a lightweight protocol atop TCP/IP mechanism which National Instruments calls *Simple TCP/IP Messaging* (STM, [10]).

National Instruments provides LabVIEW virtual instruments (VIs) for reading and writing data via this protocol. Atop of these, we have built ACS-specific VIs that allow subscribing to an ACS notification channel, publishing to a channel, and invoking an ACS method.

We also implemented a C and Java API that implements the STM protocol†. Our implementation is independent of ACS, and is thus applicable for integration with other control system infrastructures or independent processes as well.

The chosen approach has an advantage that it does not require ACS to be embedded in a LabVIEW process. Thus, there are were no issues relating to portability: the LabVIEW process can run from any platform capable to host it (Linux, Windows, VxWorks), and no prior preparation (compiling DLL libraries, placing them in appropriate places, configuring LabVIEW call function node VIs to locate them, etc.) is required. Also, the coupling between the GUI and the control loop is very loose: it is possible to transparently control a LabVIEW control loop from an ACS process (e.g., a Java GUI or a Python script), and to control an ACS process (e.g., a custom device whose driver is implemented in C++) from a LabVIEW GUI.

† The API is developed in C, and can thus be used either in C or C++ projects.

The drawback of the approach is performance: in current implementation, a message from LabVIEW GUI to LabVIEW control loop traverses two ACS components, possibly executing on different nodes on the network. In applications where some amount of latency (several milliseconds) is tolerable, this is nonetheless feasible.

The approach is scalable, however, and could be made fault-tolerant as well. Namely, ACS could instantiate more than one instance of the C++/Java components.

CONCLUSION

We have successfully demonstrated an approach to integrate LabVIEW GUI or control loop into an ACS-based system. We opted for loose coupling of ACS and LabVIEW processes via a simple, open TCP/IP based protocol. Our approach focused on portability and architectural flexibility, allowing to delay fault-tolerance, scalability and cumulative throughput considerations until deployment time. The drawback is increased latency of communication. For applications where latency requirements are not high, this remains a feasible option.

The approach is suitable for integration into non-ACS systems as well. Also at the implementation level, we have ensured that the Java and C API for STM is independent from ACS, and thus directly reusable.

REFERENCES

[1] J. Schwarz et al, “The ALMA Common Software – Dispatch from the trenches”, SPIE Astronomical Telescopes and Instrumentation 2008, Marseille, France
[2] National Instruments: “LabVIEW”
[3] D. Thompson and W. Blokland, “Shared Memory Interface between LabVIEW and EPICS”, ICALEPCS 2003, Gyeongju, Korea
[4] F. Carbognani, B. Lopez, D. Sentenac, “A GUI Builder Environment based on LabVIEW for the VIRGO Project”, ICALEPCS 2007, Knoxville, TN, USA
[5] J-M Chaize, A. Götz, W-D. Klotz, J. Meyer, M. Perez, E. Taurel and P. Verdier, “The ESRF Tango Control System Status”, ICALEPCS 2001, San Jose, CA, USA
[6] Cygwin, http://www.cygwin.com
[7] NI Developer Zone, “Connecting Measurement Studio User Interface ActiveX Controls to Remote Data”
[8] NI Developer Zone, “Using the LabVIEW Shared Variable”
[9] P. Wallace, “SLALIB – Positional Astronomy Library”
[10] NI Developer Zone, “A Simple TCP/IP Messaging Protocol for LabVIEW”