Towards a Distributed Control Framework with Real-Time Support

Mauricio Araya, Rodrigo Tobar, Jorge Avarias, and Horst H. von Brand

Departamento de Informática, Universidad Técnica Federico Santa María
Avenida España 1680, Valparaíso, Chile

Abstract. ALMA Common Software is a distributed CORBA-based framework for controlling complex distributed devices. As control tasks usually need hard or soft real-time support, a framework designed for control should include real-time properties. This paper presents the design of a real-time service that the framework should implement to achieve hard and soft real-time, without constraining the distributed properties such as centralized logging, error propagation and message passing.

1 Introduction

Real-time systems are not theoretical laboratory constructions, but systems with properties that the real world needs. Most of the advances in this area came from the automotive, avionics, particle accelerator and astronomical instrumentation applications. In this last application area, real-time systems have been widely used to control telescope mounts and other critical hardware devices, but each observatory (and even each telescope) has its own custom real-time software. This extends to general astronomical applications, because each instrument of each telescope uses its own custom software. Therefore, a major trend in astronomical instrumentation software is to provide generic software patterns and applications to be used for several telescopes and related hardware.

The ALMA project [1], a joint venture between astronomical organizations in Europe (ESO), North America (NRAO) and Japan (NAOJ), will consist of at least fifty 12-meter antennas operating together to explore the millimeter and sub-millimeter wavelength range. Due to the software challenges of controlling and coordinating the antennas remotely, this project is the perfect opportunity to gather common patterns and algorithms of different software cultures and approaches, towards a generic framework for astronomical applications. This challenge has been assigned to the ALMA Common Software (ACS) Team, which is in charge of building the generic framework on which the whole ALMA software is to be built. After seven years of development, ACS has become a robust, generic and distributed framework for control, that can be used for non astronomical applications too.

ACS currently does not support real-time properties at all [2], and even for local and very basic real-time requirements, the framework fails to met the

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 1445-1454, 2008.

ALMA real-time requirements. The novelty of this paper is the design of an ACS extension for hard and soft real-time, which can equip ACS components and clients with real-time properties without constraining the set of distributed ACS services.

This paper is organized as follows: Sect. 2 summarizes the base real-time technology that ACS could use to achieve real-time. Sect. 3 presents the ACS Framework architecture and operating system support. The design of an ACS Real-Time service is introduced on Sect. 4. Finally, Sect. 5 discusses the conclusions and future work on the ACS Real-Time service.

2 Hard and Soft Real-Time

When the time constraints are so important that a catastrophic event occurs if the time window is not met, the system has *hard real-time requirements*. Such systems should be built in a deterministic fashion to ensure deadline compliance [3]. There are other kinds of real-time systems that have less critical constraints called *soft real-time systems*. These systems execute real-time tasks according to a desired schedule "on average", but with deadlines. They usually work with some non-deterministic devices, such as Ethernet. Distributed systems working over those devices can achieve only soft real-time performance.

2.1 Portable Local Hard Real-Time Support

There is a common standard for portable operating systems as a part of the Portable Operating System Interface standard (POSIX) 1003.1 [4]. Its extension for real-time (POSIX-1003.1b) [5] includes process primitives, process environment, file/directory access, I/O primitives, device functions, synchronization, memory management, scheduling, clock and timers, and message passing functions. Most current real time systems (such as VxWorks [6] or QNX [7]) implement Real-Time POSIX. A survey on the most popular commercial POSIX compliant real-time operating systems can be found in [8]. During the past few years, several efforts have been developed towards a hard real time Linux kernel. Probably the best known projects nowadays are RTLinux [9] and RTAI [10]. The main problem of these approaches is real-time POSIX compliance, because standard POSIX calls are taken by the Linux kernel, and a parallel real time API has to be designed. A complete new approach is currently under development by the Linux kernel community, called Linux-RT and led by Ingo Molnar [11]. This POSIX-based new approach supports kernel preemption, priority inversion, threaded interrupts, hard IRQs, high resolution timers, and a full port of blocking spin-locks to preemptible mutexes.

2.2 Object Oriented Distributed Soft Real-Time

CORBA (Common Object Request Broker Architecture) is a vendor-independent architecture and infrastructure specification for distributed middleware systems.

CORBA has been standardized [12] by the Object Management Group (OMG). It provides common directives to transparently call remote objects and methods.

CORBA is not tied to a specific programming language, being a compatible specification for several implementations. This is done by providing an *Interface Definition Language* (IDL), which specifies the public interface of an object to be used by a distributed entity. CORBA specifies a *mapping* from IDL to several programming languages, making the interaction of heterogeneous objects that have been developed in different languages possible.

The specification dictates that each application should use a specific language-dependent implementation called an ORB (Object Request Broker). An ORB is a realization of CORBA, providing the namespaces, services and policies to publish an object into the network, and to access other ORBs objects transparently.

In 1999, a real-time CORBA specification was released by the OMG [13], to cover the ongoing tendency of using CORBA for real time setups. This specification enhances CORBA towards end-to-end system predictability.

The best known real-time ORB nowadays is Douglas Schmidt's TAO (The ACE ORB) [14]. This ORB is based on the Adaptive Communication Environment (ACE) library. ACE is an open-source object-oriented library, that implements concurrent communication patterns across a range of platforms. The idea of ACE is to simplify the development of heterogeneous object-oriented network applications, by providing reusable C++ wrapper façades for each platform, with a proven high-performance and real-time support. From here, it is almost natural to think on a real-time ORB implementation for the C++ language using ACE.

3 ALMA Common Software Framework

ALMA Common Software (ACS) [15] is a software infrastructure for the development of distributed systems based on the Component/Container model [16]. This framework was built for the complex control requirements of coordinating the ALMA (Atacama Large Millimeter/submillimeter Array) radio-telescopes. ACS also stands for Advanced Control System, as the framework is geared towards any system that requires complex and distributed control [17].

3.1 ACS Architecture

ACS is a set of congruent set of packages that provides development tools, common services, and the patterns needed to successfully build distributed systems. Most of the ACS features are provided using off-the-shelf software and ACS itself provides the packaging and the glue between them. For instance, ACS is based on the CORBA specification, which provides the whole infrastructure for the exchange of messages between distributed objects, and open source ORBs are used to provide this functionality. Common development tools, such as compilers, interpreters, unit-testing classes, or XML parsers, are also taken from the

open source world and consistently integrated. These basic tools are the first software package layer of ACS, as the Fig. 1 shows.

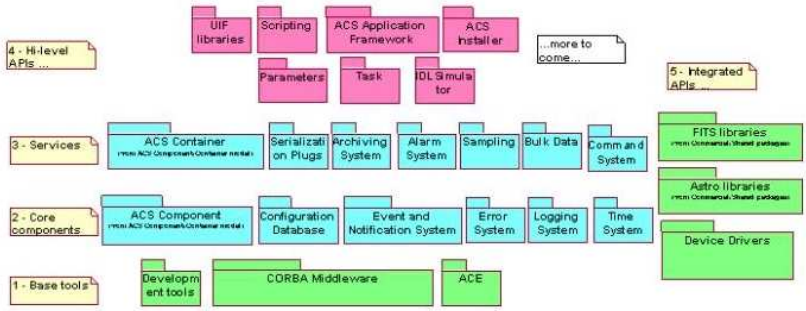

Fig. 1. ACS package diagram. It shows the layer separation and the services provided by ACS.

The next two layers are the actual ACS code called *Common Software*, which is a set of libraries and applications that support the development of distributed applications based on the *Component/Container Model* (CCM). Currently, ACS supports components developed in three major object-oriented languages, C++, Java, and Python. The main advantage of using ACS besides the CCM, is that each service is implemented for the three supported languages (at least partially). The used ORBs are TAO, JacORB, OmniORB and some Mico services.

The last layer provides high-level APIs and tools to configure and manage the system deployment and run-time. These tools are non-essential, but offer a clear path for the implementation of applications, with the goal of obtaining implicit conformity to design standards and maintainable software.

The Component/Container Model. The ALMA Common Software is structured as *components* which run inside *containers*. A *container* is a run-time entity that loads and manages several software *components*. It provides a clear division between domain code (components) from the service code (container), such as socket manipulation, system life-cycle, or data channels.

The Component/Container Model (CCM) goes beyond the original CORBA idea, and a Component/Container implementation was needed to fulfill the ALMA requirements. Unfortunately, back in the year 2000 when ACS was designed, no off-the-shelf implementations where available. Therefore, a custom Component/Container implementation was developed for ACS over the standard CORBA capabilities. The ACS CCM has now been refactored to include OMG's terminology and concepts, and currently the ACS Development Group is investigating the binary-level compatibility with CCM and J2EE.

The C++ Container (called `maciContainer`) uses an ACS Thread to implements the life-cycle of each component. An ACS Thread is an ACE Thread wrapper interface which uses POSIX Threads beneath. This Container is the most suitable candidate to support real-time properties, because the Java real-time support is still under heavy development, and Python does not offer real-time support at all.

The ACS Manager. The run-time entities of ACS are containers and clients that interact with each other using the CORBA specification and the specific ORB implementation. Clients are end-user applications designed to interact with components, while containers are designed to host components and manage their life-cycle. A possible deployment of a complex system can include lots of containers and probably several clients, so a global coordination service is needed to locate components, manage data channels, and control the life cycles of components and containers. This coordination service is called *Manager*, and is driven by a centralized Configuration DataBase (CDB), that is filled at the deployment stage.

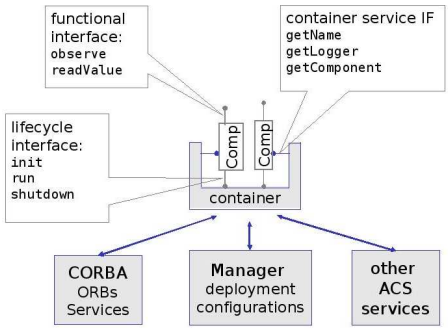

Fig. 2. ACS container/component diagram.

Component Docking. A component is an instance of a class that implements an IDL functional interface and extends a generic life-cycle implementation inherited from a basic `ACSCommponent` class, to be docked in a container. The functional interface is the public methods that the component provides to be used by other components or clients. This is defined by a CORBA IDL file, so other run-time entities can use this component distributedly and using any supported programming language. In Fig. 2, IDL defines the `observe` and `readValue` methods. There is a service interface that provides the ACS services through the container. These services are wrappers of the ORB services, information from other components through the Manager and CDB, or other ACS services like a centralized logger.

The ACS Time System. The ACS Time System consists of time conversion helper classes and two C++ ACS components: The *TIMER* component, and the *CLOCK* component. An ACS C++ Component is a shared library written in C++, with a general CORBA interface to be loaded and run in a generic C++ container (life-cycle interface). The *TIMER* functional interface provides a method to register a callback, which is called from the (possibly remote) component. This callback can be a "one-shot" call, or a periodic call at a given interval. The *CLOCK* component provides several time conversions and global time for the distributed system.

3.2 ACS Operating System Support

As ACS is based on basic off-the-shelf software (the first layer in Fig. 1), the operating system support depends on the portability of these basic packages. ACS uses a specific version of each tool, and if these versions can be installed on an operating system, ACS can be build there. But installing the basic tools is not an easy task, as the system typically offers different versions of them. Currently, ACS is officially supported only on two specific Linux distributions, RHEL 4.x (Red Hat Enterprise Linux) and SL 4.1 (Scientific Linux). In the past, ACS was also supported on Solaris and Windows, but this support was dropped due to lack of manpower and clients. The case of VxWorks is much more complex, because there are current VxWorks users, but the support for this platform is broken. This is caused mainly by the ACE+TAO VxWorks support, which has been very unstable during the past few years. Some other Linux distributions have been tested by the ACS-UTFSM Group and other users, including Debian, Ubuntu, Arch and Fedora [18]. Our group has successfully installed ACS over Ingo Molnar's RT_PREEMPT Linux kernel, which offers the possibility of using the real-time POSIX calls on the framework. Support for VxWorks and QNX is still under development.

3.3 Control Framework Alternatives

Besides ACS, there are two frameworks actively used for telescope control systems: Experimental Physics and Industrial Control System (EPICS) [19] and TAco Next Generation Object (TANGO) [20]. EPICS is a software environment that provides a pseudo-object oriented control toolkit, which is based on attributes that are stored in databases on each computer over a large network providing control and feedback. This framework supports soft real-time applications (C Language) over several real-time operating systems, but with very limited object-oriented support. TANGO is a object oriented control system toolkit based on CORBA, which hides CORBA complexity by providing a custom device pattern to develop control software. Each device implementation run as a server process that stores and retrieves information from a database. TANGO currently supports C++ and Java, uses IDL for CORBA communication, but there is no explicit support for real-time applications.

4 Towards an ACS Real-Time Service

ACS does not use the soft real-time that the TAO ORB provides, and does not offer an alternative interface for supporting local hard real-time requirement either. These two issues could be integrated in an ACS Real-Time Service that replaces the current component-based ACS Time System. The idea of this new service is to provide a common infrastructure for real-time software using ACS. At one side a hard real-time service (for local use), as a wrapper of the POSIX real-time calls. This service should provide a common method of defining thread priorities, scheduling policies, etc. In the other side, a distributed soft real-time service could be developed by using the ACE/TAO bindings for real-time. Currently, this service can only be developed for the C++ Container (i.e., `maciContainer`), but if the Java real-time support improves, the service could be ported to this platform.

4.1 Hard ACS-RT Service Design

This service must provide the same functionality of the current Time System, but using the local clock and the real-time POSIX interface. A real-time timer and an accurate timestamp function must be implemented. Following the same idea, the timer must execute a callback function periodically in real-time.

The service is designed to be used locally by a Component or by an ACS Client. Therefore, this service should coexist with the rest of the ACS subsystems such as the general Container Services, the Logging System, and the Error System. Most of the calls are not even soft real-time, so the services should be managed by a non real-time thread. Therefore, a real-time component or client will have three different threads as Fig. 3 shows. The Component/Client Imple-

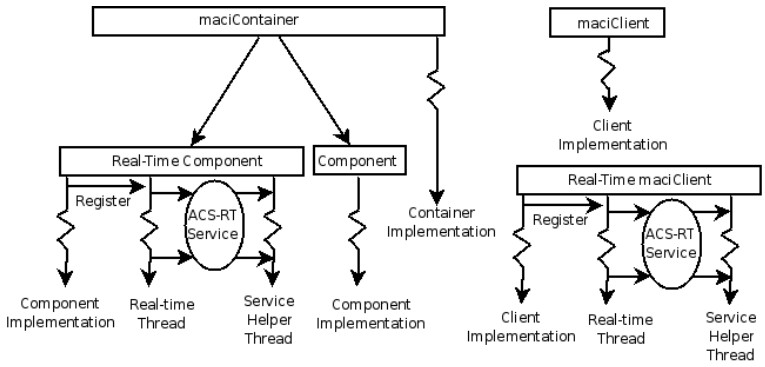

Fig. 3. Hard ACS-RT service threads

mentation is the non real-time thread that the programmer implements. This

service can register a real-time callback that will be executed by the Real-Time (RT) Thread. This RT-thread is set up with high real-time priority and a real-time scheduling policy. It must execute the registered callback functions, where the real-time timestamp function can be used to perform real-time calculations. To use non real-time services such as the Logging System or the Error System, the callbacks must use an specialized API called ACS-RT Service, which executes the calls in the Service Helper Thread. This non-RT thread executes the non real-time calls that the Real-Time Thread uses. The idea is that the real-time thread does not call blocking methods that could depend on non-predictable devices such as an Ethernet network.

As POSIX specifies a set of real-time calls with nanosecond precision, the real-time thread must use these calls to execute the registered callbacks. Therefore, the periodic loop must be implemented using the `clock_nanosleep()` syscall, and the timestamp function must be the `clock_gettime()` syscall. The usage of other POSIX calls such as the `itimer` family or the `gettimeofday()` call could produce unwanted latencies on the thread.

The ACS-RT Service API must be implemented using shared memory queues with a producer/consumer policy. The producer is the real-time thread that pushes objects on the queues, while the consumer Service Helper thread is in charge of popping objects and executing the actual calls. The objects are instances of the classes that each ACS service provides, such as a log entry, an error, an alarm or a notification message. This way, our real-time callback could use distributed logging, error propagation and message passing methods without timing penalties.

4.2 Soft Real-Time Service Design

As ACS is based on CORBA, it is natural to think of an ACS extension to real-time using real-time CORBA. This extension will be based on the ACE/TAO real-time CORBA implementation, since there is no other real-time ORB implementation used by ACS. This service will be soft real-time, because ACS was designed to be deployed over Ethernet local area network, which is not a deterministic-driven protocol.

This work presents the general design guidelines to implement a Container with soft real-time capabilities. The idea is to offer communication between ACS Components assuring a real-time QoS. Therefore, the real-time patterns on the ACE/TAO ORB [21] must be raised to the ACS Container implementation. These patterns include a real-time ORB abstraction, priority mappings for CORBA threads, priority model policies and scheduling policies [22]. An implementation of this ACS soft real-time Container or Client must:

1. Extend the ACS Basic Control Interface IDL (BACI [23]) to support a CORBA Real-Time Portable Object.
2. Implement the BACI real-time interface on the C++ `libBaci` library.
3. Extend the ACS Management and Access Control Interface IDL (MACI [23]) to set the parameters of the new `RTmaciClient` and the `RTmaciContainer`,

using a TAO priority mapping manager to setup the priority model and the scheduling policy.
4. Offer real-time configuration of the component thread priorities and policy configurations using the CDB Data Access Layer that ACS provides.
5. Implement the MACI real-time interface on the C++ `libMaci` library, mapping the ACE/TAO *lanes* and *bands* priorities to Components threads.
6. Configure the CORBA Notification Channel and Telecom Log Service to support real-time CORBA specification. This will allow real-time performance on the ACS Notification Channel, Logging System, Archiving System, Sampling System and Alarm System, because they are based on the two named CORBA services [24].

5 Conclusions and Future Work

This paper presents an introduction to the ACS framework, and describes the general trend of implementing a hard and soft ACS Real-Time Service. We are currently implementing this service, and future work will compare this service to other alternatives, producing benchmark results to validate this design.

The main contribution of this paper is the hard real-time service threading model, which merges the POSIX real-time calls with a complex distributed framework with no timing penalties for the hard real-time thread.

Many of the ACS services are based on the CORBA Notification Channel and Telecom Log Service, so the soft real-time extension depends only on the real-time ORB support. Alas, the Error System and Bulk Data Channel are not directly based on these CORBA services, so the real-time support for these services must be custom designed.

Acknowledgment

This work was financed by project ALMA-CONICYT 31060008, and could not have been done without the support of ALMA, ESO, NRAO and UTFSM. Mauricio Araya's and Rodrigo Tobar's work is supported by UTFSM *Iniciación Científica* grants. We thank the reviewers of this paper, for their detailed corrections and insightful comments.

References

1. Tarenghi, M.: The Atacama large millimeter/submillimeter array: Overview & status. Astrophysics and Space Science **313**(1-3) (January 2008) 1–7
2. Araya, M.: Verifying real-time periodic properties on the ALMA common software time system. Master's thesis, Universidad Técnica Federico Santa María (2008)
3. Stankovic, J.A.: Real-time and embedded systems. ACM Computing Surveys **28**(1) (1996) 205–208
4. IEEE: 1996 (ISO/IEC) Information Technology – Portable Operating System Interface (POSIX®) – Part 1: System Application Program Interface. (1996)

5. IEEE: POSIX Part 1: System API – Amend. 1: Realtime. (1994)
6. Collier, J.: An overview tutorial of the VxWorks real-time operating system
7. Hildebrand, D.: An architectural overview of QNX. In: Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures, Berkeley, CA, USA, USENIX Association (1992) 113–126
8. Baskiyar, S., Meghanathan, N.: A survey of contemporary real-time operating systems. Informatica (Slovenia) **29**(2) (2005) 233–240
9. Barabanov, M., Yodaiken, V.: Introducing real-time Linux. Linux Journal **1997**(34es) (1997) 5
10. Dozio, L., Mantegazza, P.: Real time distributed control systems using RTAI. In: Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (May 2003) 11–18
11. Rostedt, S., Hart, D.V.: Internals of the RT patch. In: Proceedings of the Linux Symposium 2007. (2007)
12. Object Management Group: CORBA 3.0 - formal/02-06-01 (2002)
13. Object Management Group: Real-time CORBA, v2.0 – formal/03-11-01 (2003)
14. Schmidt, D.C.: Middleware techniques and optimizations for real-time, embedded systems. In: Proceedings of the 12th International Symposium on System Synthesis (ISSS 99), Washington, DC, USA, IEEE Computer Society (1999) 12–17
15. Chiozzi, G., et al.: Common software for the ALMA project. In: Proceedings of ICALEPS. (2001)
16. Sommer, H., et al.: Container-component model and XML in ALMA ACS. In: Proceedings of SPIE 2004. (2004)
17. Plesko, M., et al.: ACS – The advanced control system. In: 4th International Workshop on Personal Computers and Particle Accelerator Controls. (2002)
18. Avarias, J.: Repackaging ACS for embedded systems. Technical Report 12/2007, Departamento de Informática, Universidad Técnica Federico Santa María (2007)
19. Martin R. Kraimer, Janet Anderson, Andrew Johnson, Eric Norum, Jeff Hill, Ralph Lange, Benjamin Franksen: EPICS: Input / Output Controller Application Developers Guide. EPICS Base Release 3.14.9 edn. (November 2007)
20. Gotz, A., et al.: TANGO a CORBA based control system. In: Proceedings of ICALEPS. (2003)
21. Wang, N., Schmidt, D., Levine, D.: Optimizing the CORBA component model for high-performance and real-time applications. In: Work-in-Progress session at the Middleware 2000 Conference, ACM/IFIP (2000)
22. Schmidt, D., Levine, D., Cleeland, C.: Architectures and patterns for developing high-performance, real-time ORB endsystems. In: Distributed Information Resources. Volume 48 of Advances in Computers. Academic Press (1999) 2–119
23. Chiozzi, G., et al.: The ALMA Common Software: A developer friendly CORBA-based framework. In: Proceedings of SPIE 2004. (2004)
24. Chiozzi, G., et al.: The ALMA Common Software, ACS status and developments. In: Proceedings of ICALEPS. (2005)