Applying a High-Level Real-Time Benchmark to a CORBA-based Control Framework Timer

Mauricio Araya, Jorge Avarias, and Horst H. von Brand
Departamento de Informática
Universidad Técnica Federico Santa María
Av.España 1680, Valparaíso, Chile

Abstract

ALMA Common Software (ACS) is the framework on which the distributed control system for the Atacama Large Millimeter Array will be based. ALMA has real time requirements, which are being met by ad-hoc workarounds. We present a model for defining high-level real time synthetic benchmarks for ACS, so the real time requirements can be met inside the framework in the future. We present some results of benchmarking the TIMER component of ACS on several configurations.

1. Introduction

During the past months, the ALMA Common Software Framework (ACS), has been tested on prototype antennas and specific hardware that will be part of the ALMA Observatory. In general terms, the framework fulfills the desired requirements, but some specific subsystems are using workarounds due the lack of real time component/container support and test. ACS developers have decided not to support a direct container implementation for RTOSs, since there is very little research on this specific issue, and thus the underlying risk is too high for software that soon will be officially used on production antennas. Unfortunately, these workaround solutions break the system consistency, making the distributed coordination even more difficult, including debugging and logging. This problem affects not only isolated real-time components; real-time communication with other real-time components currently can not be done using ACS services. Regardless, ACS C++ containers and services are strictly developed over the ACE/TAO middleware, so in theory, the framework should be able to fulfill real time requirements at least for C++, if a proper RTOS is setup. This paper presents a first approach towards the high-level real time benchmark needed by ALMA Common Software for further development and improvements.

The paper is organized as follows. Section 2 reviews the RTOS, CORBA, ACS and benchmarks state of art. Section 3 presents the Benchmarking Model. Sections 4 and 5 present the experiments and the results. At last, Section 6 presents the conclusions and future work planned.

2. Overview

2.1. Real-Time Operating Systems

The main difference between a General Purpose Operating System (GPOS) and a Real-Time Operating System (RTOS), is that the former does not support direct control of the microprocessor nor peripherals for timing issues. Then, the user (programmer) is unable to develop a program that meets timeliness requirements, because he has no control over the process scheduling, sleeping deadlines, or buffer flushing. A real-time kernel must provide basic support for predictably satisfying real-time constraints, thread/process preemptibility, thread/process priority, deterministic synchronization mechanisms, predefined latencies and a specialized API for accessing this functionality (i.e. RTAI [6]). There is a common standard for portable Real-Time systems as a part of the POSIX standard (1003.1) [12]. The POSIX extension for real-time (1003.1b) [11], includes process primitives, process environment, files/directory access, I/O primitives, device functions, synchronization, memory management, scheduling, clock and timers, and message passing functions. Indeed, these are also the essential parts of a general-purpose OS [27]. Most current real time systems (such as VxWorks, LynxOS or QNX) implement RT-POSIX.

During the past few years, several efforts have been developed towards a hard real time Linux distribution. A good description of the real time requirements and challenges that a real time Linux should consider can be found in [31]. Probably the most known projects nowadays are RTLinux [1, 34] and RTAI [6, 13]. The first one offers a
high-performance hard real time API bypassing the Linux scheduler and interrupt controller, by rewriting the interrupts using a special patch in the kernel. The interrupts are managed by special kernel modules designed for low latency. In the case of RTAI, the interrupts are managed by an underlying hard real time nanokernel (ADEOS), and real time tasks have to be written as kernel modules. There is also an user-space API (LXRT) that uses generic RTAI kernel modules that invoke ADEOS with an obvious latency penalty. The main problem of these approaches is RT-POSIX compliance, because standard POSIX calls are taken by the Linux Kernel, so a parallel real time API has to be designed. Therefore, the porting of real time applications written originally for a POSIX compliant RTOS becomes very complex.

A complete new approach is currently under development by the Linux Kernel community, called Linux-RT and led by Ingo Molnar [21]. This 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. The main advantage is that the system uses the same POSIX compliant API; the drawback is that the project is currently under heavy development, and there are only few benchmark results.

2.2. The Real-Time CORBA Middleware

CORBA (Common Object Request Broker Architecture) is a vendor-independent architecture and infrastructure specification for distributed middleware systems [14]. CORBA has been standardized [15] by the Object Management Group (OMG). Each specific implementation is called an ORB (Object Request Broker), and several open source and commercial implementations have been developed during the past years.

In 1999, a RT-CORBA specification was released by the OMG [16, 25], to cover the ongoing tendency of using CORBA for real time setups [9, 10, 18]. Douglas Schmidt has developed TAO [23, 24], a real time ORB based on the ACE library [22]. This ORB provides C++ support for the RT-CORBA specification, with tested high-performance on commercial RTOS.

2.3. The Advanced Control System Framework

ALMA Common Software (ACS) [3, 4, 20] is a software infrastructure for the development of distributed systems based on the Component/Container model [28, 29]. This framework was built to support the complex control requirements of coordinating the ALMA (Atacama Large Millimeter 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]. Currently, ACS support components are developed in three major object-oriented languages, C++, Java, and Python. ACS is based on CORBA to provide a language-independent middleware layer for distributed applications. Specifically, ACS uses the ACE/TAO ORB for C++, JacORB for Java, and OmniORB for Python. The runtime entities of ACS are containers and clients, that interact with each others through CORBA. There is a global coordination service called Manager, configured through a centralized Configuration DataBase (CDB), with the major task of managing the life cycle of the components and container references.

2.4. Benchmarks

One of the first real time benchmarks was Hartstone [33], for high-level Ada real time requirements. Simultaneously, Rhealstone [19] was developed to test low level requirements. These benchmarks are still the most commonly used, including several modifications for distributed systems and object-oriented programming [7, 8]. There are several CORBA benchmarks, such as [2, 32], or CORBA-RT benchmarks from the TAO tests [23], but there is no high-level CORBA benchmark, and no benchmark specifically for ACS. The main advantage of using a high-level benchmark is that the results are more “in touch” with the real time requirements, so it is easier to decide if the system is suitable.

3. Synthetic High-Level Benchmark Model

A synthetic benchmark is a set of fully-predictable tests and values to compare several systems, where “synthetic” stands for generated experiments to cover a fair amount of cases. A real-time requirement can be periodic, aperiodic or sporadic [30], so there are three different general scenarios for benchmark. This paper presents a benchmark for periodic requirements, but aperiodic and sporadic requirements can be benchmarked with a similar approach. A periodic real-time requirement can be specified by its frequency, meaning that a given action must be performed precisely every time at a fixed interval. For this benchmark model, it is assumed that the computation time needed by the action is much less than the time interval between requirements.

3.1. The “Hard” Real-Time Test

There is no such thing as a 100% deterministic machine [6], so there is no way to strictly meet a deadline. Even more, every measure of the time is bounded by the clock precision, so it is impossible to have a “pure hard” real-time system. Therefore, for each hard real-time
requirement there is a tolerance degree, that defines the threshold where the deadline is still met. Usually, the tolerance is a few orders of magnitude less than the precision, so the acceptance function (1) can be defined.

\[ A_{x,y}(t_i) = \begin{cases} 1 & \text{if } t_i \leq 10^{-x}10^{-y} \\ 0 & \text{else} \end{cases} \]  

(1)

In equation (1) \( x \) is the degree of timing precision (DTP), \( y \) the tolerance degree and \( t_i = |t_{\text{exp}} - t_{\text{real}}| \) the time difference between the expected and the real measured time. Note that \( 10^{-x} \) is the requirement frequency, therefore, for a given \( y \), a system fulfills a DTP of \( x \) if and only if:

\[ \prod_{T} A_{x,y}(t_i) = 1 \]

Here \( T = \{t_1, t_2, \ldots, t_n\} \) is the set of differences between the expected and the real measured times. As the \( T \) values depend on the frequency of the requirement, to define a timing precision value rather than a DTP is not a practical approach. The base-10 logarithmic scale permits a discrete comparative value for hard real-time systems, even though \( x \) supports decimal values for a more fine-grained comparison. Figure 1 shows the graphical representation of the DTP test. Each \( D_i \) is a periodic deadline, and each \( T_i \) have a 0 or 1 value, to indicates if the response is inside the \( y \) threshold or not.

For instance, if \( z = 10^{-z}/2 \), each response has a \( 10^{-z} \) [s] window to be useful, as shown in figure 2. The idea is to calculate the mean service utility for a given \( x \) and \( z \) over \( T \).

\[ M_{x,z} = \sum_{T} S_{x,z}(t_i) \quad \#(T) \]  

(3)

Figure 2. A linear isochronal SUF

3.3. Periodic Harmonic and Non-Harmonic Series

Distributed real time systems should support concurrent real time requirements, so a synthetic real time benchmark should include controlled concurrency tests. This benchmark model is aimed at comparing high-level systemic properties, rather than specific system real time services or components. Therefore, the harmonic and non-harmonic Hartstone-like [33] series are used to test concurrent real time tasks.

Following [5], base-2 harmonic frequencies are used for concurrency tests. For example, if we are testing a DTP of 3, the main frequency test is of 100 [Hz], and the harmonic tasks frequencies are 50 [Hz], 25 [Hz], 12.5 [Hz], etc. Generally, the \( i \)-th harmonic frequency will be \( 10^x/2^i \) [Hz]. For non-harmonic series, the Fibonacci series was selected because it produces a predictable non-harmonic phenomena, where the \( i \)-th non-harmonic frequency is \( 10^x/F_i \) [Hz].

3.4. Systemic vs Instantaneous Time Difference

There are at least two different views to setup the expected time value. The first one is the systemic approach which considers the history of all the time differences. This approach allows to identify systemic errors, and to have a fair picture of how the system is responding to a controlled hardware. The systemic \( T_{\text{sys}} = \{t_1, t_2, \ldots, t_n\} \) set is calculated from the accumulated theoretical value as follows.

\[ t_i = (r_i - t_0) - i \ast \delta \]  

(4)
Here \( r_i \) is the UTC measured time, \( t_0 \) the UTC instant when the experiment begins, and \( \delta = 10^{-x} \) the timer interval. If a deadline is missed, the error is carried on for the next values, so this approach is suitable for hard real time benchmarking.

The second approach is to consider the instantaneous difference between the timer interval and the measured interval. This approach permits a general QoS (Quality of Service) measurement, and a histogram analysis because the relative independence of the data. The instantaneous \( T_{\text{ins}} \) set can be calculated as follows:

\[
t_i = (r_i - r_{i-1}) - \delta
\]

Obviously, this approach is more useful for soft real time requirements, because a significant statistic can be calculated with independent events.

4. Experimental Setup

This paper presents the results of applying the described high-level benchmark model to a specific service: The ACS Time System. A few setups are described as an example of using the benchmark model. These experiments are the first step towards a requirement-based real time benchmark for ACS.

4.1. The ACS Time System

The ACS Time System consists of some 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 (lifecycle interface). For each component a specific interface must be written in CORBA Interface Definition Language (IDL) that provides the specific component methods that other components and client could use (functional interface). The TIMER functional interface provides a method to register a callback in the client program-space, and it is called from the (possibly remote) component. This callback can be a “one-shot” call, or a periodic call for a given interval. The CLOCK component provides several time conversions and global time for the distributed system.

4.2. The Client Test Program

ACS components can interact with another distributed components, or with stand-alone clients. In terms of usage, clients and components share the same communication methods, so a test client program is enough for this benchmarking effort.

The client used here is a C++ application that connects to the ACS Manager and asks for the TIMER and CLOCK component references. Then the client registers a callback with a given interval for a periodic call in the TIMER component. When the callback is invoked, the client saves in memory the current time that the CLOCK component delivers. This process repeats for the given duration of the experiment, and at last the client saves the results in a file.

The client also spawns new clients using fork, which simultaneously use the TIMER component with different frequencies, as the figure 4. The starting time of the experiment is also coordinated using the CLOCK component, and a preliminary global wait period is set to avoid startup distortions.
4.3. External Time Measurement

As the measured time values of the clients are based on the same system that is been benchmarked, they are not reliable. Therefore, the parent client process also sends a signal through the RS232 (serial) port (see figure 4). An external machine reads the signal and registers the timestamp using a high resolution timer. This monitoring program was developed using the standard POSIX real time specification, and deployed over a QNX Neutrino RTOS.

4.4. Three Deployment Examples

The general idea is to test the ACS Time System, and all the heavy infrastructure behind it, with different scenarios as an example of the benchmarking model. Two scenarios are benchmarked: Different RTOSs, and Distributed vs Local CORBA. For the first scenario a standard Linux 2.6.23.9 kernel is compared with Linux 2.6.23.9 patched for real-time (Igno Molnar’s patch rt13). This comparison should help to know if the rt13 patch helps ACS to support real time requirements. The second scenario is a comparison between running the TIMER and CLOCK components on the same machine with the client, and running the components in a distributed configuration as figure 5 shows.

5. Preliminary Results

The following preliminary results were obtained on homogeneous machines with an Intel Celeron 2.80 [GHz] CPU and 512 [MiB] RAM, over a Gigabit Ethernet.

5.1. The ACS CLOCK vs the External Measurement

The first result appears just from comparing the external measurement with the ACS CLOCK. In figure 6, the systemic differences were plotted for the ACS CLOCK and for the external serial port measurements. A clearly erratic behavior of the ACS CLOCK can be seen, but the external measurement indicates that the ACS timer is working almost flawlessly. Note that the jumps of the ACS CLOCK plot are almost for a fixed value (1000 [µs]), which gives clues for fixing this difference.

Also, a slight slope can be seen on the external measurements, meaning that the hardware clocks are not perfectly synchronized.
5.2. The “Hard” Real-Time Test

This test, shows that neither plain Linux, nor Linux-RT pass the 1000[Hz] barrier for hard RT. In figure 7, the three single client experiments were plotted against the frequency. The dependent axis shows the amount of passed hard RT tests. The first plot shows the plain Linux results, and the second one the Linux-RT variant. As can be shown, Linux-RT has a better performance than plain Linux. But the measurements are highly below the sub-millisecond expected values. This can be a Linux-RT problem, a kernel misconfiguration, or just because general-purpose hardware is used.

![Figure 7. The Hard RT Test Summary](image)

5.3. The Soft Real-Time Test

In the soft real time test, the general result is that a good QoS can be obtained for values below the 1000[Hz] barrier, using the instantaneous difference data. Also, Linux-RT provides better QoS that plain Linux by a little margin. For instance, the following table shows the results for each frequency with a $z = 10^{-x}/2$.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>S.U. Linux</th>
<th>S.U. LinuxRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [Hz]</td>
<td>0.9577</td>
<td>0.9658</td>
</tr>
<tr>
<td>10 [Hz]</td>
<td>0.9411</td>
<td>0.9424</td>
</tr>
<tr>
<td>100 [Hz]</td>
<td>0.9275</td>
<td>0.9626</td>
</tr>
<tr>
<td>1000 [Hz]</td>
<td>0.7248</td>
<td>0.8603</td>
</tr>
</tbody>
</table>

The general behavior of $z$ is plotted in figure 8 in a base-2 logarithmic scale. The figure shows that Linux-RT also gives a better curve, meaning that the data is closer to the deadlines.

![Figure 8. $z$-Utility Function](image)

5.4. Concurrent Tasks Influence

The Harmonic and Non-Harmonic tests were run for all the cases, and generally there is not much difference between the different platforms.

But in all the cases identifiable distortions were detected on both concurrent tests, as the figure 9 shows. The first plot, shows a single client behavior of the instantaneous difference. The second and third plots, shows the behavior using Harmonics and Non-Harmonics respectively.

There is also a bad QoS when the concurrent clients are used. The following table shows the comparison between the single client, and the mean between the harmonic and non-harmonic case.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>S.U. Linux-RT</th>
<th>S.U. Linux-RT Conc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [Hz]</td>
<td>0.9988</td>
<td>0.9979</td>
</tr>
<tr>
<td>10 [Hz]</td>
<td>0.9987</td>
<td>0.9975</td>
</tr>
<tr>
<td>100 [Hz]</td>
<td>0.9937</td>
<td>0.7578</td>
</tr>
<tr>
<td>1000 [Hz]</td>
<td>0.8735</td>
<td>0.0142</td>
</tr>
</tbody>
</table>

5.5. Distributed Deployment Influence

The most important result of this example is that there is no penalty at all on using the distributed setup, having almost the same QoS, same DTP test, and same concurrent clients behavior. The figure 10 shows that there is no significant difference between single client behavior (two first plots), and even a little improvement in the concurrent version (two last plots).
6. Conclusions and Future Work

Clearly, the use of an external clock for measurements was required, as the ACS CLOCK is influenced by the activity on the system. The ACS TIMER component, on the other hand, is much more stable. In the low-frequency tests, the TAO ORB performed as expected, even in the distributed case. Also, the setup with LinuxRT is somewhat better than the vanilla one, but not significantly so. We suspect that careful tuning of the kernel could make a difference here.

Below around 1000 [Hz], the quality of service is very good. But if there are many concurrent processes using the TIMER component it degrades near 100 [Hz]. As this component will be critical in any real time setup, the reason for the drop in performance has to be studied, and perhaps the interaction with this component will have to be redesigned.

Future work will define higher-level benchmarks for ACS, including realistic control loops with aperiodic and sporadic workloads. Work on other RTOSes is currently in progress.

7. Acknowledgments

This work was supported by project ALMA-CONICYT 31060008, and could not been done without the support of ALMA, ESO, NRAO and UTFSM. Mauricio Araya’s work was supported by a UTFSM Iniciación Científica grant.

References


