ACS Sampling System: design, implementation and performance evaluation

P. Di Marcantonio[*a], R. Cirami[a], G. Chiozzi[b]
[a] INAF-Osservatorio Astronomico di Trieste, via G.B. Tiepolo 11, I-34131 Trieste, Italy
[b] ESO, European Southern Observatory, Karl-Schwarzshild-Str.2, D-85748, Garching bei Munchen, Germany

ABSTRACT

By means of ACS (ALMA Common Software) framework we designed and implemented a sampling system which allows sampling of every Characteristic Component Property with a specific, user-defined, sustained frequency limited only by the hardware. Collected data are sent to various clients (one or more Java plotting widgets, a dedicated GUI or a COTS application) using the ACS/CORBA Notification Channel. The data transport is optimized: samples are cached locally and sent in packets with a lower and user-defined frequency to keep network load under control. Simultaneous sampling of the Properties of different Components is also possible. Together with the design and implementation issues we present the performance of the sampling system evaluated on two different platforms: on a VME based system using VxWorks RTOS (currently adopted by ALMA) and on a PC/104+ embedded platform using Red Hat 9 Linux operating system. The PC/104+ solution offers, as an alternative, a low cost PC compatible hardware environment with free and open operating system.

Keywords: Object-oriented programming, Software patterns, Real-time operating systems, CORBA

1. INTRODUCTION

This paper outlines the design and implementation issues of the ACS sampling tool and analyzes the achieved performances on two different platforms: on a VME based system using VxWorks RTOS (called hereafter LCU - Local Control Unit) and on an embedded non real-time system based on a PC/104+ platform. Since our tool uses the ACS framework we assume that typical ACS concepts/paradigms like Component, Manager, Container etc. are familiar to the reader. In brief, any logical or physical device (e.g. power supply, lamp etc.) in an ACS controlled system is represented by a *Component* (implemented as a CORBA object). A Component contains one or more *Properties* – entities that can be monitored or controlled (e.g. brightness in case of lamps, current in case of power supply) and *Characteristics,* which contain static data such name, units, description etc. For the purpose of this paper we use the term Component to refer to a Component with Properties and Characteristics. More complete and exhaustive descriptions of these concepts and of ACS in general can be found for instance in [1].
The ACS Sampling system allows sampling of every Component at a user-specified sustained frequency, limited only by the hardware. The sampling engine itself is implemented as an ACS Component and:

- can be activated on any local computer (both workstation and/or LCU)
- can be configured to sample one or more Components at given frequencies (simultaneous sampling from different Properties are supported)
- sampled values are published periodically, at low frequency, via ACS notification channel (see [2]).

Data transport is optimized. The samples are not sent one by one, but are cached on the local computer (i.e. in the sampling distributed object) and sent in packets with a user-defined frequency. The caching of data reduces network traffic and reduces the impact on the performance of the whole control system.

[*] dimarcan@ts.astro.it; phone 0039 040 3199111; fax 0039 040 309418; www.ts.astro.it

A client could retrieve the sampled data for later analysis or plotting by subscribing it to the notification channel. The plotting tool could be of any kind: a Java plotting widget, a dedicated GUI or a COTS application e.g. LabView. For our needs we developed a Java GUI based on the Abeans library[3].

2. DESIGN AND IMPLEMENTATION

2.1. Requirements

The basic requirements for the ACS sampling system can be summarized as follows:

- every ACS Property can be sampled at a specified sustained frequency limited only by the hardware (up to 1 kHz for a limited number of Components)
- the data channel transports sampling data
- data transport is optimized; data are cached and sent in packets (e.g. 1 Hz frequency) to keep network load under control
- simultaneous sampling of different Properties of Components must be possible.

2.2. Design

In order to fulfill the requirements quoted above, we design the sampling system (using the factory design pattern[4]) as composed by two entities: the ***sampling manager*** and the ***sampling object(s)***. The class diagram (derived from the CORBA IDL interface) depicted in Fig. 1 shows the basic class relationship of the ACS sampling system.

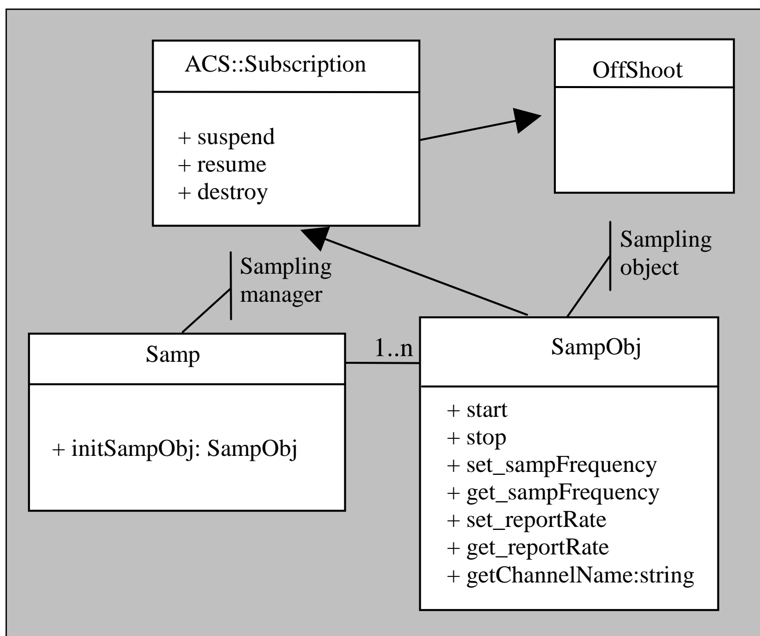

Fig. 1: UML class diagram showing the basic class relationship for the ACS sampling system

Responsibility of the sampling manager (represented by the interface *Samp*) is to accept requests coming from outside (i.e. clients willing to sample a specific Property with a specific frequency). It is a Component activated by the Container on demand. After validating the sampling request, the sampling manager creates a new “sampling object” (represented by the interface *SampObj*) and returns to the client the generated CORBA reference.

The sampling object is a CORBA object linked to the specific property which exposes to the client all methods dealing with the sampling (i.e. *start*, *stop*, *suspend*, *resume*, *set_sampFrequency*, *set_reportRate* allowing the client to fully control the sampling behavior on the specific property). As shown in Fig. 1, some of the methods are inherited from *ACS::Subscription* interface, which in turn inherits from the *Offshoot* interface. The latter one is used inside ACS to identify CORBA objects, whose lifecycle is limited to the lifecycle of the component which created them. Responsibility of the sampling object is also to create the notification channel for data delivering and to cache the data in order to optimize the network load.

Once the client connects to the sampling manager, receives the CORBA reference to the newly created sampling object and starts the sampling, then all delivered data can be retrieved from the notification channel for plotting or later

analysis. As a rule it is client's responsibility to stop the sampling and destroy the corresponding sampling object. If this is not the case, it is the sampling manager who will cleanup everything (that is, all active sampling objects) when deactivated. Such a design proves to be very flexible. A client could create as many sampling objects as required, allowing the simultaneous sampling of more Properties or even the same Property with different sampling frequencies.

3. IMPLEMENTATION

The following section will briefly illustrate the basic characteristics of all the classes involved in the ACS sampling system. The programming language used to implement them was C++ to be sure to meet all the performance requirements. Of course, client could be written using different languages like Java or Python, i.e. in all those supported by ACS.

3.1. Sampling manager

The basic functionalities of the ***sampling manager*** are achieved through its method *initSampObj* (see Fig. 1). A user, willing to sample a specific Property (e.g. the *brightness* Property of the *LAMP* Component), calls *initSampObj* with the following parameters:

- the name of the Component to be sampled (e.g. *LAMP*);
- the name of the Property to be sampled (e.g. *brightness*);
- the sampling frequency expressed as the period between two successive samples (e.g. 0.01 s for 100 Hz);
- the report rate expressed as the period between two successive deliveries on the notification channel (e.g. 1 s).

Once the user enters all the required parameters, the CORBA reference to the requested Property is obtained by means of CORBA DII (Dynamic Invocation Interface).Next step, by looking in the Interface Repository also the type of the Property is discovered (e.g. if the Property is of double or long type). Depending on the type found a corresponding ***sampling object*** is allocated and activated. Its CORBA reference is then returned to the user.
An internal list traces all the active sampling objects. By scanning this list the sampling manager will clean-up everything when deactivated from the Container (in case that the sampling object is not destroyed by the user).

3.2. Sampling object(s)

The ***sampling object*** implements all functionalities required to sample a given property (see Fig. 1). It is implemented as template class for handling arbitrary property types. Every activated sampling object is basically a CORBA object, whose name is composed by the concatenation of the passed parameters: *ComponentName_PropertyName_SamplingRate_ReportRate*. This guarantees the uniqueness, allowing also to connect as many sampling objects as required to every Property.
Two independent threads control the sampling and flushing of the samples respectively. Both threads are started as soon as the *start* method is invoked by the user. The sampling thread, which is activated at every sampling period, retrieves a value from the given Property in a synchronous way. The retrieved sample is stored in an internal buffer until a specific time interval has elapsed (the report rate). When this happens, the flushing thread flushes all the stored data on the notification channel, freeing the buffer and leaving room for new values. The internal buffer is based on the ACE message queue, which provides all the necessary synchronization mechanisms to avoid race condition between threads, optimizing the enqueue and dequeue of data.
The thread behaviour (and correspondingly the sampling behaviour) is controlled by a set of dedicated methods like *suspend*, *resume*, *stop* etc. If stopped, threads can also be restarted with a new sampling period and/or report rate. Finally the *destroy* method will stop everything, releasing all the allocated CORBA resources.
The UML sequence diagram in Fig. 2 shows the basic interactions among all the involved classes of the sampling system.

3.3. Data structure

The data structure delivered to the notification channel is composed by two fields:

1. the time stamp;

2. the sampled value;

as shown in the following code fragment taken from the CORBA IDL interface:

```
struct sampDataBlock {
        ACS:Time sampTime;
        any SampVal;
}
```

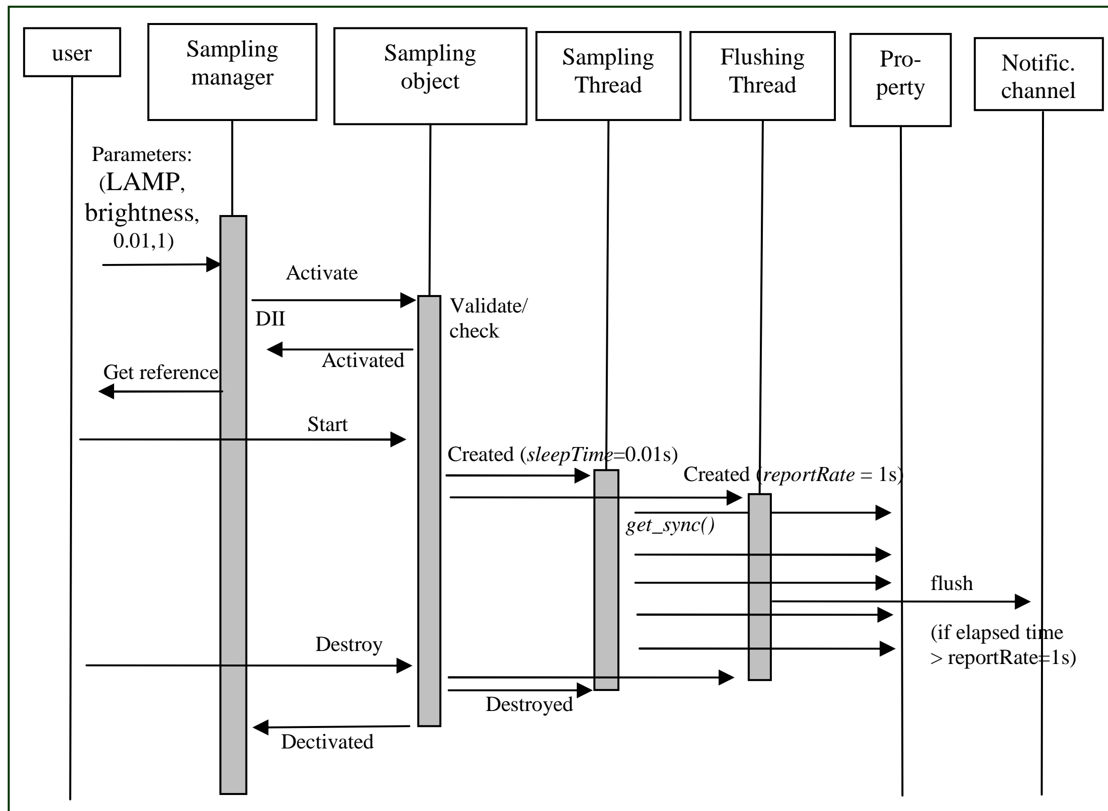

Fig. 2: UML sequence diagram showing basic interactions among ACS sampling classes

The CORBA ***any*** is used in order to accommodate sampled properties values of different types (e.g. long, double etc.) using a single structure. The slight loss of performances due to the overhead of using *any* is not noticeable down to a sampling period of 1 kHz.

Every sampling object creates its own notification channel to which data are delivered. The name of the created channel is guaranteed to be unique. In this way we avoid the possibility of mixing data coming from different sampled Properties, speeding up also the retrieval of data. Clients willing to subscribe to the notification channel can also call the method *getChannelName(),* which returns the created channel name string (see Fig. 1).

4. PERFORMANCES

To test the behavior of the ACS Sampling tool the performances were evaluated on two different platforms: on a VME based system using VxWorks as a RTOS and on a PC/104+ embedded platform running the Linux Red Hat 9 operating system (non real-time). Data were collected by two dedicated processes (written in C++):

- one server (producer), used to select the Component to sample (with a specific, user-definable, frequency and report rate);
- one client (consumer), used for storing data in a file for later analysis.

As already described, data were transported by means of the notification channel. Results and discussion about the timing and jitter of the sampled values are shown in the following two subsections.

4.1. Real-time Environment

Real-time performances were evaluated on an LCU with the following hardware/software characteristics:

- VME crate equipped with PowerPC 604 CPU with clock rate of 100 Hz;
- 128 MB RAM;
- Ethernet 100 MBit network adapter;
- VxWorks 5.5 RTOS (Tornado 2.0, gcc 2.95).

The crate VME was connected via the Ethernet adapter to a Sun Ultra 60 UltraSparc-II 250 MHz host with 512 MB of RAM. We run all the required CORBA and ACS services (interface repository, naming service, configuration database etc.) together with the Manager on the Sun host, whereas the container with the Sampling manager and several others Components was deployed on the VME. For all these tests no real hardware was connected on the LCU. All sampled values are directly read from memory, allowing measuring only the real software overhead.

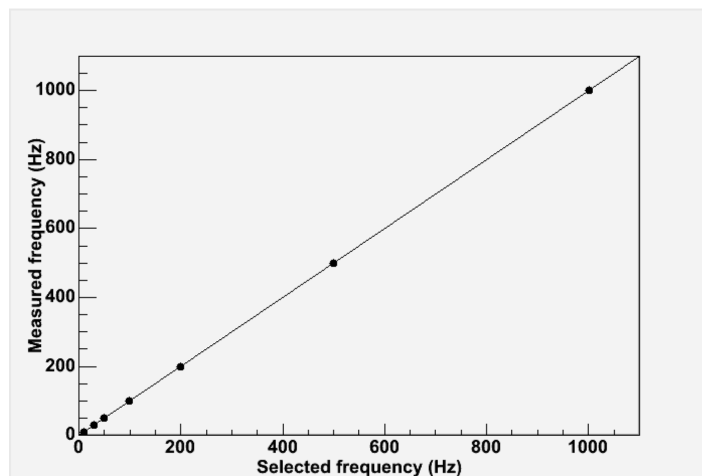

Fig. 3: Graph showing "selected frequency" vs "measured frequency" on a real-time environment. The r.m.s. of the data is less then a 1 kHz.

To disentangle the behaviour of the sampling thread from the possibly overhead introduced when delivering sampled values on the notification channel, we analyzed the gathered data in two ways. As a first step we included in our analysis only the buffered data between two successive deliveries. During this period, the flushing thread is in the sleeping state, allowing the evaluation of overheads of the higher frequency (sampling) thread.

The result of this *"pure"* sampling is shown in Fig. 3 were a graph of the "*selected frequency*" vs "*measured frequency*" is shown. Every point in the graph is an average of several thousand samples, acquired also by stressing the CPU with additional work such as activating and deactivating several Components, by calling various methods on them etc.

The graph clearly proves that there is not appreciable jitter. The nominal VME board clock rate used for our analysis was 100 Hz. This means that our system was not able to resolve samples with time stamp less than 0.01 s. Changing the clock rate by means of the VxWorks *sysClkRateSet()* system call allowed achieving a sampling period as high as 1 kHz. As can be inferred from Fig. 3, also speeding up the frequencies did not introduce any appreciable jitter up to 1 kHz. From our data, we can conclude that he *r.m.s.* for pure "sampling" is of the order of a fraction of a kHz.

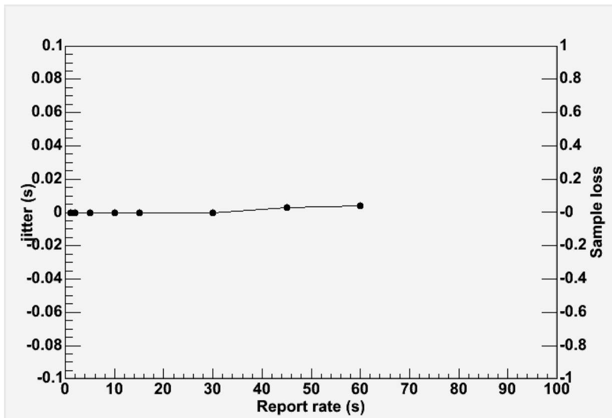

Fig. 4: Jitter vs report rate for 10 Hz sampling. The r.m.s. is of the order of 0.002 s.

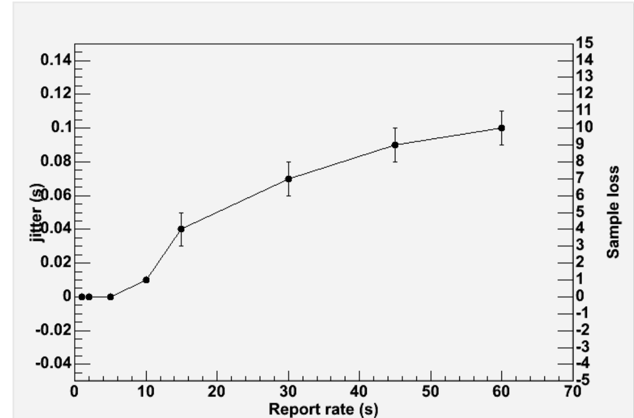

Fig. 5: Jitter vs report rate for 100 Hz sampling. A small jitter is noticeable for report rate > 10 s.

Next we included in our analysis all the gathered samples and therefore also the overhead due to the delivery of data on the network. The results are depicted in Fig. 4 and Fig. 5, where data collected for 2 sampling periods (10 Hz and 100 Hz) for several report rates are shown. Having several report rates means also delivering different amounts of data (e.g. 30 s report rate at 100 Hz means 30000 samples and corresponds to roughly 46 kB). From the collected data we have seen that for high frequency at higher report rates there is a small jitter introduced, whenever the flushing thread is activated. The figures show the average jitter, clearly indicating that we lose some data for 100 Hz sampling at report rates greater then 10 s. The number of lost samples is in any case limited (of the order of 5 – 10) thus giving a total efficiency of the order of 99.7%. For shorter report rates or lower frequencies we experienced no data loss.

The origin of this jitter is still under investigation, but this is partly expected. We are using as internal buffer the ACE message queue, which provides all the necessary synchronization mechanisms in order to avoid race conditions between the sampling and flushing thread. The message queue uses water marks to indicate when the queue has too many data on it. When the queue becomes full, it cannot enqueue new data; the sampling thread will be blocked until sufficient room is available again. In our case this is what happens for longer report rates. Of course, we can increase the size of the internal buffer, but we have to find a trade-off to avoid too much memory consumption. From the graph it is clearly seen that for a typical report rate (less then 10 s) there is no loss of samples.

It should be noticed moreover, that we deliver data with the associated timestamp. This timestamp comes directly from the synchronous read and eventually from the hardware. Therefore, even though we miss some samples and have a small jitter, the couple between a value and the actual access time is always guaranteed correct. Of course the consequence of this is an increase in the stored message size.

Note that we push our system to the limit only to gather data for the “pure” sampling. The global performances were not evaluated for clock rate higher then the nominal value (i.e. 100 Hz) in order to avoid system *thrashing* (which occurs when clock interrupts are so frequent that the processor spends too much time servicing the interrupts, preventing the application code to run). We did not also fully analyze the case of sampling more Properties simultaneously. This is matter for future work.

From the collected data we have seen that if we consider the “pure” sampling, Properties are sampled with high precision down to 1 kHz. The typical *r.m.s* is essentially limited by the precision achieved by the VME clock rate and there is no software overhead introduced by our sampling tool. When the overhead is also included in the data due to delivery on the notification channel, there appears only a small jitter at higher frequency with higher report rates (partly expected). For typical ALMA applications (sampling period of the order of 100 Hz with report rate of 1 Hz) data loss is negligible.

4.2. Embedded, non real-time environment

Modern large experimental facilities, like telescopes, accelerators etc. are highly sophisticated and complex system, where the number of controlled and monitored devices may exceed tens-of-thousands. The general trend clearly drives, among others, to most cost-effective solution: open-source, stable standards in the software field, COTS (commercial off-the-shelf), possibly less expensive hardware components. In the spirit of the aforementioned requirements the Astrophysical Technologies Group of the Astronomical Observatory of Trieste started to investigate the possibility to use

a customized version of the ACS on an embedded system in general, and on a PC/104+ platform in particular. The project is described in great detail in[5] . Here we will just outline its main characteristic and describe the performances of the ACS sampling system on our selected platform.

The PC/104+ is a low cost PC compatible hardware environment for embedded applications, characterized by a self-stacking bus, minimization of components and low power consumption. Our prototype is based on a "Digital Logic Microspace PC/104+", a miniaturized modular device incorporating the major elements of a PC/AT compatible computer with the following characteristics:

- CPU PII-MMX, 166 MHz
- 128 MB RAM
- Ethernet adapter 10/100 Mbit
- HD 10Gb
- 257 MB flash card (not used for our purposes)

To test the ACS sampling tool on our prototype, two preliminary steps were required:

1. tailoring of the Red Hat 9 Linux operating system to fit the PC/104+ characteristics (essentially the small amount of memory and lower CPU clock);
2. installing the required ACS services.

The whole porting procedure is described in[5] and will not be repeated in this paper.

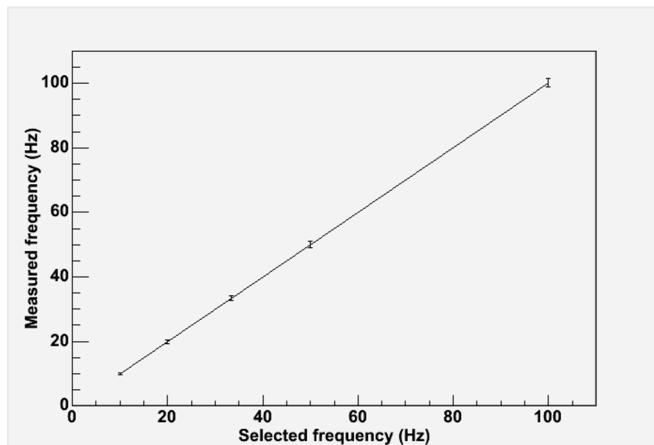

Fig. 6: Graph showing "selected frequency" vs "measured frequency" on the embedded system.

The performances of the sampling tool were evaluated with the same test suite used on the VxWorks host. We used a Linux machine (a Pentium IV PC equipped with 1 GB RAM) to run all the required CORBA and ACS services (interface repository, naming service, configuration database etc.) together with the Manager, whereas the Container with the Sampling manager and all the required Components was deployed on the PC/104+ embedded platform. The two hosts were connected via an Ethernet connection at 100 Mbit.
Data produced on the PC/104+ were sent to the client running on the Linux host and stored in files for subsequent analysis. Again we disentangle the "pure" sampling from the overhead due to delivery on the notification channel. Fig. 6 shows the graph of the "*selected frequency*" vs "*measured frequency*" and is the analogous to the Fig. 3 for the VxWorks case.

Several considerations could be drawn:

1. the highest sampling frequency is limited to 100 Hz, which reflects the maximum achievable clock rate of our platform (we couldn't change the clock rate as opposite to VxWorks case);
2. the *r.m.s* on the sampling period is greater than in the real-time case.

The latter point is of course expected. The *r.m.s* for the worst case scenario (100 Hz sampling period) is of the order of 0.01 s. Our investigation shows that this depends on the performances of the *ACE_OS::sleep()* call, which is used internally by the sampling thread. The *ACE_OS::sleep()* is simply a wrapper around the corresponding platform specific system call. In case of Linux, which does not use a real-time system, the typical implementation of the Linux sleeping function family (*nanosleep()*, *sleep()* etc.) is based on normal kernel timer mechanism, which has a resolution of 10 ms on Linux/i386. Since our estimated *r.m.s* is exactly of this order we can conclude that we are not introducing any new software overhead. We are dominated, at this level, by the resolution of the Linux sleeping function.

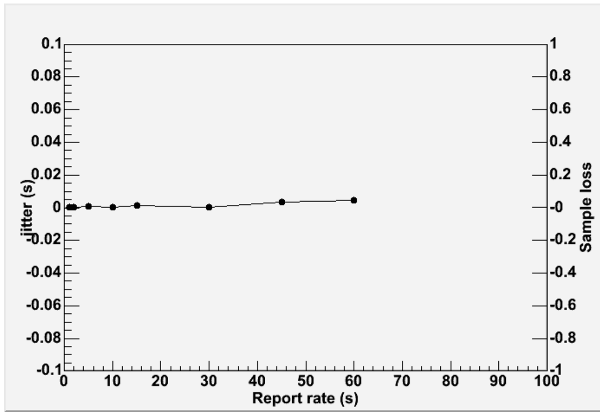

Fig. 7: Jitter vs report rate for 10 Hz sampling. The *r.m.s.* is of the order of 0.002 s.

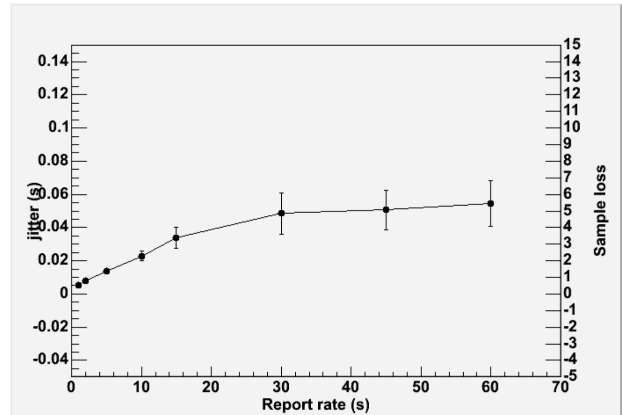

Fig. 8: Jitter vs report rate for 100 Hz sampling. A small jitter is noticeable for report rate > 10 s.

Next, as for VxWorks case, we analyzed the global overhead including also the delivering of sampled values on the notification channel. Data are shown in Fig. 7 and Fig. 8 for a sampling frequency of 10 Hz and 100 Hz respectively. The two figures and results are analogous to the real-time case and prove that the behaviour between the two platforms is consistent. Also in this case, at higher frequency with higher report rates, we observe a small jitter, which causes the loss of a limited number of values (comparable to what obtained on the real-time environment). We presume that the reason for this is analogous to what already described (see 4.1). The *r.m.s.* in this case is slightly bigger, but this takes well in account the limited time resolution for the sleeping function discussed above.

From data gathered on both platforms, the real-time one and the embedded one, we can conclude that our tool achieves the maximum allowed precision permitted by the system clock. On both platforms data are comparable and the loss is noticeable only when a longer report rate is involved. By carefully tuning the internal buffer parameters also this could be minimized to suit particular application needs.

5. ABEANS GUI

Based on the Abeans library[3] we develop a simple GUI, which controls the sampling behaviour, extracts data from the notification channel and displays them in real-time. Abeans is a library that provides simple Java beans for connection with the control system (using in case of ACS the CORBA middleware) and several useful services like logging, exception handling, configuration and data resource loaders. For our needs we use the *Spike chart*, which is a high performance Java plotting widget, able to display higher frequency data by internally buffering them. The Fig. 9 shows a graph of the sampled property value vs the corresponding time stamp. For this example the sampling frequency was 10 Hz.

6. CONCLUSION

Based on the ACS framework and adopting the factory design pattern, we developed a sampling tool, which allows high frequency sampling of one or more Properties of Components. The tool was tested on two different platforms: on a real-time environment and on a non real-time embedded system, yielding similar results. The analysis of the recorded data shows that we are limited only by the underlying hardware (i.e. by the clock rate of the used system) and that all the basic requirements are fulfilled. We experience only a small loss of data, when a bigger amount of data is to be transferred. If required by ALMA, this could be improved by optimizing the handling of the internal buffer. Together with the tools, a high performance GUI based on Abeans library is also provided.

In future, the sampling tool will be tested using Linux real-time installed on the PC/104+ embedded platform also allowing to carefully estimate performances of the whole ACS framework on such low-cost platform also.

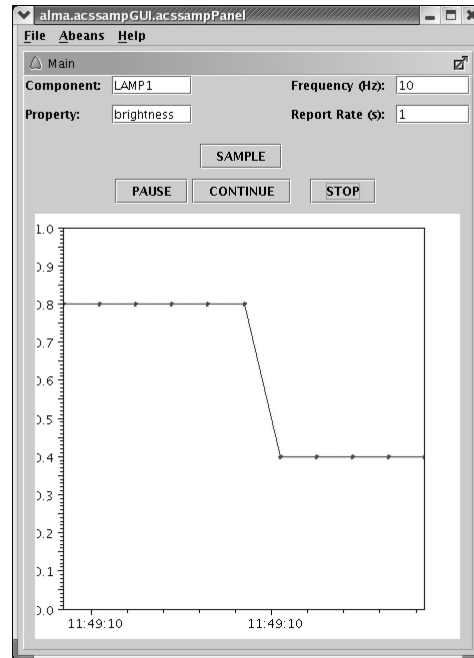

Fig. 9: ABeans GUI showing a graph of a Property sampled at 10 Hz.

REFERENCES

1. G. Chiozzi et al. , "CORBA-based Common Software for the ALMA project", *SPIE Advanced Telescope and Instrumentation Telescope Control Software II* , SPIE 4848, pp. 43-45, Kona, HW, Aug 2002
2. D. W. Fugate, "A CORBA event system for ALMA common software", *SPIE Advanced Software, Control, and Communication Systems for Astronomy*, SPIE 5496, Glasgow, June 2004
3. I. Vertovsek et al., "CosyFramework: Application Development Framework for Java", *ICALEPS 2003*, Gyeongju, Korea, October 2003
4. E. Gamma et al., *Design Patterns*, pp. 87, Addison Wesley, 2000
5. Cirami et al., "An embeddable control system for astronomical instrumentation", *SPIE Advanced Software, Control, and Communication Systems for Astronomy*, SPIE 5496, Glasgow, June 2004