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

By means of ACS (ALMA Common Software) framework we designed and implemented a sampling system which allows sampling of every Property of a Component 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.

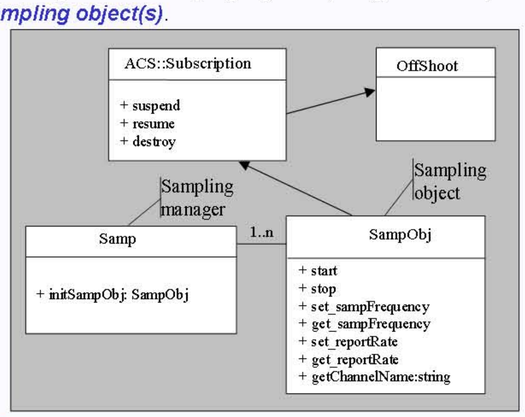
ACS Sampling System requirements

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 Properties)
- 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.

ACS Sampling System design

We designed the Sampling System (using the factory design pattern) as composed by two entities: the ***sampling manager*** and the ***sampling object(s)***.

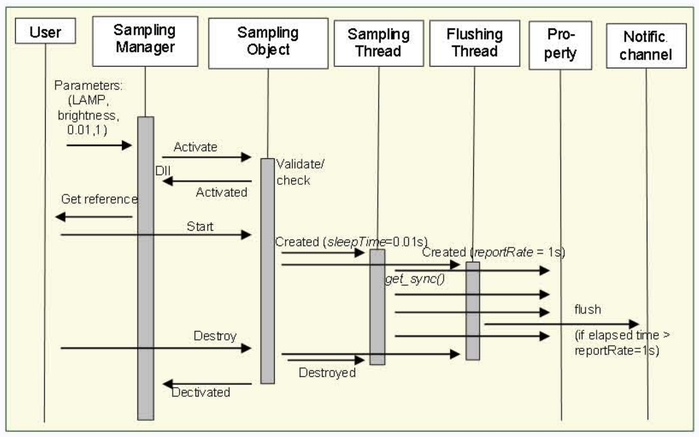


Responsibility of the *sampling manager* (represented by the interface ***Samp***) is to accept requests coming from outside (i.e. clients willing to sample a specific Property with a specific frequency). After validating the sampling request, the sampling manager creates a new "*sampling object*" (represented 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).

The *Offshoot* interface is used inside ACS to identify CORBA objects, whose lifecycle is limited to the lifecycle of the Component which created them.

ACS Sampling System implementation



The programming language used for implementation is **C++** to be sure to meet all the performance requirements. Of course, a client could be written using different languages like Java or Python, i.e. in all those supported by ACS.

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 (the *Property value* and *the associated timestamp*) 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.
Note that the timestamp comes directly from the synchronous read and eventually from the hardware. This guarantees that the coupling between a value and the actual access time is always correct.

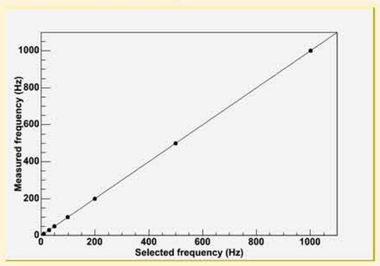
ACS Sampling System performances

Real-time environment

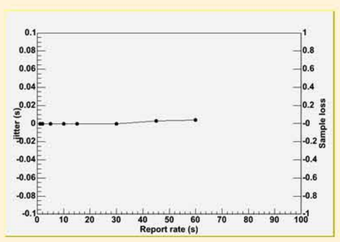
Real-time performances were evaluated on a system with the following hardware/software characteristics:

- ***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)***

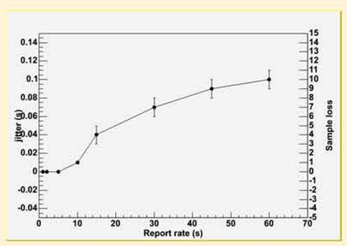
The crate VME was connected via the Ethernet adapter to a Sun Ultra 60 UltraSparc-II 250 MHz host with 512 MB of RAM running all the required *CORBA* and *ACS services* (interface repository, naming service, configuration database etc.).



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 the figure on the left where 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 while stressing the CPU with additional work such as activating and deactivating several Components, by calling various methods on them etc. ***The r.m.s. is less then 1 kHz.***



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



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. From the collected data we have seen that for higher frequencies at higher report rates a *small jitter* is 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.

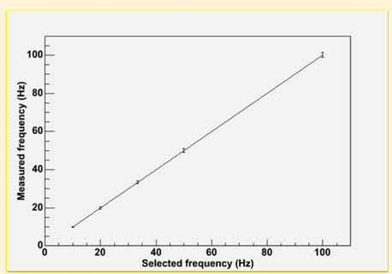
The origin of this jitter is still under investigation, but this is partly expected. We are using as internal buffer the ACE message queue, which uses water marks to indicate when the queue has too much data on it. When the queue becomes full, it cannot enqueue new data; the sampling thread will be blocked 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. The jitter amount could also be reduced by carefully tuning the threads priority (not implemented for this test). ***The number of lost samples is anyway limited (of the order of 5 – 10) thus giving a total efficiency of the order of 99.7%.*** *Moreover, for shorter report rates or lower frequencies we experienced no data loss.*

Embedded, non real-time environment

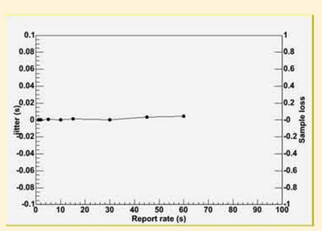
The embedded system is based on a "Digital Logic Microspace PC/104+", a miniaturized modular device incorporating the major elements of a PC/AT compatible computer with the following characteristics:

- ***CPU PII-MMX, 166 MH***
- ***128 MB RAM***
- ***Ethernet adapter 10/100 Mbit***
- ***HD 10GB***
- ***256 MB flash card (not used for our purposes)***

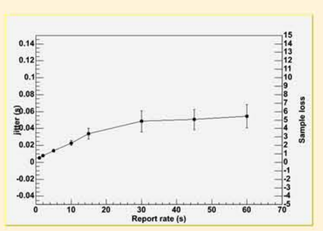
The PC/104+ platform was connected to a Linux machine (a Pentium IV PC equipped with 1 GB RAM and Ethernet at 100 Mbit) were all the required *CORBA* and *ACS services* were deployed.
To test the ACS sampling tool on our prototype we used the same test suite we adopted on the VxWorks host. Moreover, 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



Again we disentangle the "pure" sampling from the overhead due to delivery on the Notification Channel (see figure on the left were a graph of the "*selected frequency*" vs "*measured frequency*" is shown). Note that:
1. the highest sampling frequency is limited to 100 Hz, which reflects the maximum achievable clock rate of our platform (we could not change the clock rate as opposite to the VxWorks case);
2. the *r.m.s* on the sampling period is greater than in the real-time case ***(of the order of 0.01 s)***. Our investigation shows that this depends on the limited resolution of the *Linux sleeping function family*, which is used internally by the sampling thread.



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



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. The two figures on the left and the results are analogous to the real-time case and prove that the behaviour between the two platforms is consistent. 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.

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