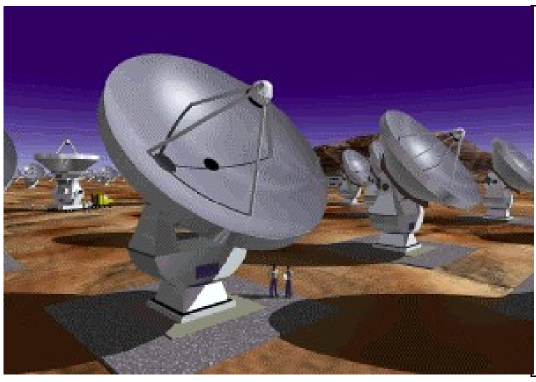


Atacama Large Millimeter

ALMA-NNNNN

Revision: 1.2

2009-04-30

User Manual

Paolo Di Marcantonio

ACS Sampling system

User Manual and How-to Manual

Paolo Di Marcantonio

INAF-Astronomical Observatory of Trieste

Roberto Cirami

INAF-Astronomical Observatory of Trieste

Owner	Paolo Di Marcantonio (dimarcan@ts.astro.it)	
Approved by:	Date:	Signature:

Change Record

REVISION	DATE	AUTHOR	SECTION/PAGE AFFECTED	REMARKS
1.0	13-11-2003	P. Di Marcantonio	All	Created.
1.1	20-11-2003	P. Di Marcantonio	5.3, 5.4	Added.
1.2	07-05-2004	P. Di Marcantonio	2.3.2, 4, 5.2, 5.4	Added

Table of Contents

1 Overview

This document describes the design of the ACS Sampling system and explains how to use it. The first sections illustrate the design and the corresponding implementation; potential users who only wish to use the sampling can safely skip those sections and read the user manual part of the document.

The ACS Sampling system allows sampling every ACS property at a user-specified sustained frequency, limited only by the hardware. This sampling engine is implemented as a characteristic component and:

- can be activated on any local computer (both workstation and/or LCU)
- can be configured to sample one or more properties at given frequencies (simultaneous samples from different properties are supported)
- sampled value are published periodically, at low frequency, via notification channel.

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 performances of the whole control system.

At this level of the implementation, data are just published on the notification channel. It is responsibility of the client to subscribe to the notification channel and retrieve the sampled values for later analysis or plotting. The plotting tool, which may be a Java plotting widget, a dedicated GUI or a COTS application, like LabView, is also not part of this release.

In order to use the sampling tool it is assumed that all ACS services (and especially the TAO notification and naming services), the Configuration Database, Manager and Container are properly configured and running. See A23 Overview and section 6 for details on how to do this.

2 Design and Implementation

2.1 Requirements

The basic requirements for the ACS sampling system are:

- every property can be sampled at a specified sustained frequency limited only by the hardware (up to 1 kHz for a limited number of Characteristic Component 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 characteristic component objects must be possible.

2.2 Design

In order to fulfill the requirements quoted above, we design the sampling system as composed by two entities: the ***sampling manager*** and the ***sampling object(s)***.

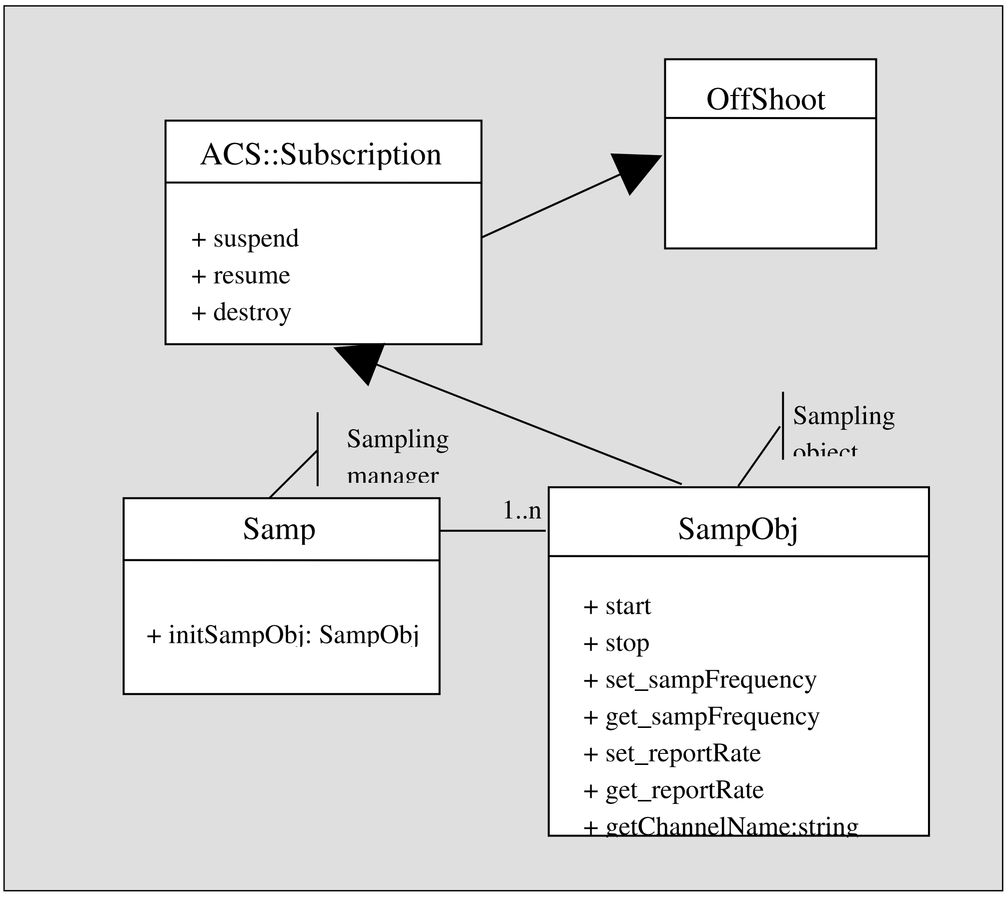
Responsibility of the sampling manager is to accept request coming from outside (typically from a client willing to sample a specific property with a specific frequency). It is a component object (using ACS terminology) having an entry in the Configuration Database (CDB) and activated by the Container on demand. After validating the sampling request, the sampling manager creates a new "sampling object" and returns to the client a CORBA reference to it.

The sampling object is a CORBA object linked to the specific property which expose to the client all methods dealing with the sampling (i.e. *start*, *stop*, *pause*, *resume*, *set_sampFrequency*, *set_reportRate* allowing the client to fully control the sampling behaviour on the specific property). 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 will not be 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 object as required, allowing the sampling of more properties simultaneously or even the same property with different sampling frequencies.

The following class diagram (derived from the idl interface) shows the basic class relationship of the ACS sampling system.



2.3 Class description

The following section will briefly illustrate the basic characteristics of all the classes involved in the implementation of the sampling system. For a detailed description refer to the corresponding man pages.

2.3.1 Sampling manager – *class acssampImpl*

The class *acssampImpl* implements the basic functionalities of the ***sampling manager*** through its method *initSampObj*. 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. LAMP1);
- The name of the property to be sampled (e.g. brightness);
- The sampling frequency expressed as the period between two successive sample; units are 100 ns (e.g. 1000000 means a period of *0.1s* i.e. 10 Hz);
- The report rate expressed as the period between two successive deliveries on the notification channel; units are 100 ns (e.g. 10000000 means a period of *1 s;* data are cached for 1s before being sent on the notification channel).

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, by looking in the Interface Repository also the type of the property is discovered (e.g. RWdouble or ROlong). Depending on the found type a corresponding ***sampling object*** is allocated and activated (as a CORBA object). Its reference is then returned to the user.

Known issue: the allocation of a new sampling object is achieved by means of a "*if…elseif...else*" block. Currently only four property types are supported: ROdouble, RWdouble, ROlong and RWlong. If a new type should be supported, then a new entry in the *"if"* block should be hard-coded. This behaviour will be changed in future (if required), in order to allow a more dynamical approach.

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. *Please note that this is just a safety mechanism*. As a rule, it should be user responsibility to destroy all the activated sampling objects.

2.3.2 Sampling object – *class acssampImplObj*

The class *acssampImplObj* (it is a template class for handling arbitrary property types) implements all the functionalities required to sample a given property. Every activated sampling object is basically a CORBA object, which name is composed by the concatenation of the passed parameters:

ComponentName_PropertyName_SamplingRate_ReportRate

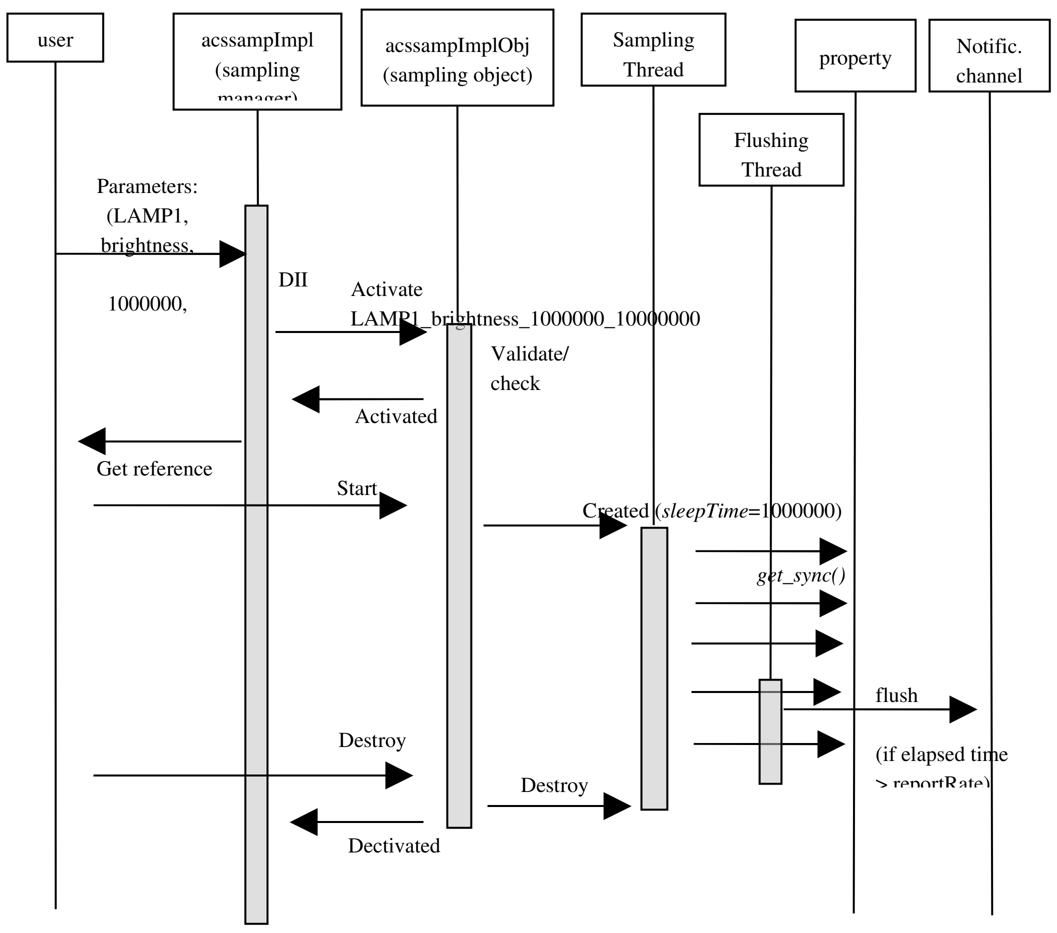
(e.g. LAMP1_brightness_1000000_10000000). This guarantees the uniqueness (at the moment we don't see the need to sample the same property with two or more sampling object with exactly the same parameters), allowing on the other side to connect to every property as many sampling objects as required.

Two independent threads control respectively the sampling and flushing of the samples. 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 mechanism 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 method like *suspend*, *resume*, *stop* etc. If stopped, the thread 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 following sequence diagram shows the basic interactions among all the involved classes of the sampling system.


user
acssampImpl (sampling manager)
acssampImplObj (sampling object)
Sampling Thread
property
Notific. channel
Flushing Thread
Parameters: (LAMP1, brightness, 1000000,
DII
Activate LAMP1_brightness_1000000_10000000
Validate/ check
Activated
Get reference
Start
Created (sleepTime=1000000)
get_sync()
flush
(if elapsed time > reportRate)
Destroy
Destroy
Dectivated

3 Data Definition

3.1 Data Structure

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

1. the time stamp;
2. the sampled value.

The following is the fragment taken from the *acssamp.idl* interface:

```
1:  struct sampDataBlock
2:  {
3:      ACS::Time sampTime;
4:      any sampVal;
5:  }
```

3 The *sampTime* is filled using *getTimeStamp()* function.

4 *any* is required in order to accommodate properties of different types (e.g. ROlong, RWdouble etc.)

Every sampling object creates its own notification channel to which data are delivered. The name of the created notification channel is a string containing the "CORBA name" of the sampling object (see 2.3.2), as follows:

NC_*samplingObjectCORBAName*

(e.g. NC_LAMP1_brightness_1000000_10000000). Clients willing to subscribe to the notification channel can also call the method *getChannelName(),* which returns this string.

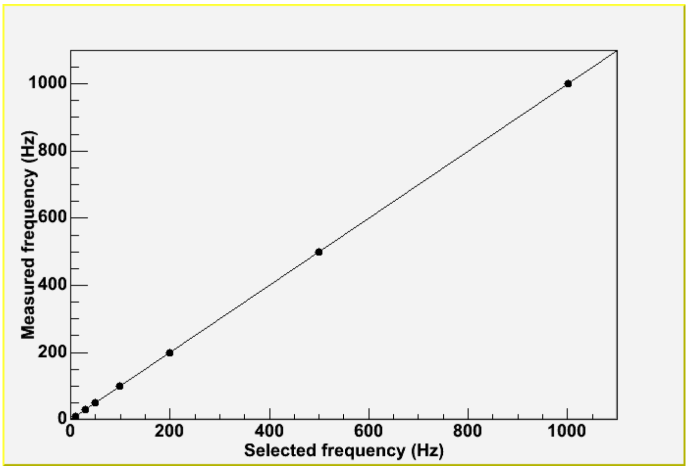
As in the case of the naming convention for the sampling object (see 2.3.2), this guarantees the uniqueness of the notification channel. Every sampling object holds its own channel avoiding in this way the possibility of mixing data coming from different sampled properties and speeding up the retrieval of data.

4 Performances

The performance of the ACS sampling system is, due to the chosen design, heavily linked to the BACI thread performances. Our analysis shows that the time resolution is ultimately given by the *sleep* function used inside the BACI thread class.

The original implementation uses the *ACE_OS::sleep()*; even replacing this call with other functions like the POSIX *nanosleep()* or more specific platform-dependent like the VxWorks *taskDelay()* limits the performances to that of the system clock (which is however reasonable). VME CPU boards currently used in ALMA operate with a clock frequency of **100 Hz**. This happens to be also the maximum speed at which we can currently sample.

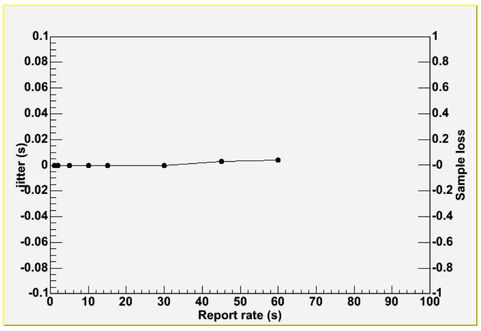
For test purpose we change the clock rate by means of the VxWorks *sysClkRateSet()* system call. This allowed achieving a sampling period as high as 1 kHz. To disentangle the behaviour of the sampling thread from the possibly overhead introduced when delivering sampled values on the notification channel, we analyzed the gathered data in two ways. As a first step we included in our analysis only the



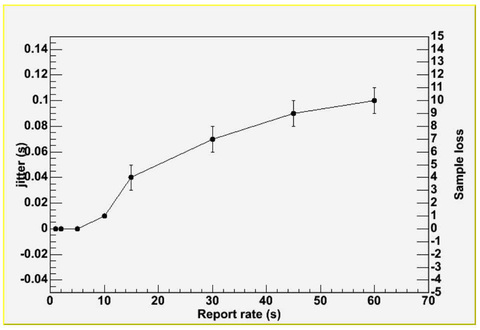
Graph showing "selected frequency" vs "measured frequency" on a real-time environment. The r.m.s. of the data is less then a 1 kHz.

buffered data between two successive deliveries. During this period, the flushing thread is in the sleeping state, allowing evaluating overheads of the higher frequency (sampling) thread.

The result of this *"pure"* sampling is shown in the above figure were a graph of the "*set 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 like activating and deactivating several components, by calling various methods on them etc. The graph clearly proves that there is not appreciable jitter.



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. The results are depicted in the following figures, where data collected for 2 sampling period (10 Hz and 100 Hz) for several report rates are shown.

Having several report rates means also delivering different amount 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 partly it is expected. As already described we are using as internal buffer the ACE message queue, which provides all the necessary synchronization mechanism 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 much data on it. When the queue become full, it cannot enqueue new data; the sampling thread will be blocked until sufficient room is again available. In our case this is what happens for longer report rate. 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.

5 User Manual

The following section illustrates the usage of the sampling system from the user point of view. The basic design guideline, since the beginning, was to keep the usage of the ACS sampling system as simple as possible. The number of operations required to get the sampling working are therefore really limited and can be summarized as follows:

1. obtain the CORBA reference to the sampling manager (which should be loaded by the Container on demand);

2. call on it the "*initSampObj*" method, with all the required parameters (i.e. passing the component name, the property to be sampled, the sampling frequency and the report rate);

3. get from "*initSampOb*j" the reference to the newly created sampling object;

4. call on it the "*start*" method.

The sampling should now start. Data are collected by the sampling object (with the given sampling period), cached and flushed to the notification channel only after the report period has expired. This will continue until the sampling is stopped by invoking the "*stop*" method.

As a rule, it is user responsibility to destroy the created sampling object, calling the corresponding "*destroy*" method.

As a safety mechanism however, the manager will clean-up all active sampling objects, when disconnected from the Container.

The following sections show the C++/Java implementation for the user server part (i.e. the part which requests the sampling) and the user client part (i.e. the part which consumes the data). Strictly speaking, both are really clients of the sampling manager, but we will still use the server/client terminology in order to distinguish them.

5.1 C++ server example

Bolded code should be adapted for the developer's particular needs. Note that all Client login/logout code has been omitted and the *acssamp* module contains the full example (file *acssampOnlyNCServer.cpp).*

```
#include <acsutil.h>

#include <maciSimpleClient.h>
#include <acssampC.h>
#include <acssampS.h>
#include <baciS.h>
#include <acserr.h>
#include <ACSErrTypeCommon.h>

   try
    {

    // obtain the reference to the SAMP (factory) object
    ACSSamp::Samp_var foo = client.get_object<ACSSamp::Samp>("SAMP1", 0, true);

    if (!CORBA::is_nil(foo.in()))
        {

        ACS_SHORT_LOG((LM_DEBUG, "Got samp descriptor()."));

         // calls the initSampObj to create dynamically a new sampling object
        ACSSamp::SampObj_ptr fooNew =
         foo->initSampObj("LAMP1","brightness",1000000,10000000);

        ACS_SHORT_LOG((LM_INFO,"*** Start to sample ***"));

                    ACS_SHORT_LOG((LM_INFO,"Not    Channel:    %s",fooNew->getChannelName()));

         // starts the sampling
        fooNew->start();


        ACE_OS::sleep(5);
34:
35:
36:         fooNew->suspend();
37:         ACE_OS::sleep(5);
38:         fooNew->resume();
39:
40:
41:         ACE_OS::sleep(6);
42:
43:         // stop and clen-up everything
44:         fooNew->stop();
45:         ACE_OS::sleep(2);
46:         fooNew->destroy();
47:
48:         CORBA::release(fooNew);
49:
50:         }
51:     } /* end main try */
52:     catch (OutOfBoundsEx & ex)
53:     {
54:     ACS_SHORT_LOG((LM_INFO, "OutOfBoundsEx exception catched !"));
55:     OutOfBoundsExImpl err(ex);
56:     err.log();
57:     }
58:     catch (CouldntAccessComponentEx & ex)
59:     {
60:     ACS_SHORT_LOG((LM_INFO,      "CouldntAccessComponentEx      exception
catched !"));
61:     CouldntAccessComponentExImpl err(ex);
62:     err.log();
63:     }
64:     catch (CouldntAccessPropertyEx & ex)
65:     {
66:     ACS_SHORT_LOG((LM_INFO,      "CouldntAccessPropertyEx      exception
catched !"));
67:     CouldntAccessPropertyExImpl err(ex);
68:     err.log();
69:     }
70:    catch (CouldntCreateObjectEx & ex)
71:     {
72:     ACS_SHORT_LOG((LM_INFO,       "CouldntCreateObjectEx       exception
catched !"));
73:     CouldntCreateObjectExImpl err(ex);
74:     err.log();
75:     }
76:    catch (TypeNotSupportedEx & ex)
77:     {
78:     ACS_SHORT_LOG((LM_INFO, "TypeNotSupportedEx exception catched !"));
79:     TypeNotSupportedExImpl err(ex);
80:     err.log();
81:     }
82:     catch (...)
83:     {
84:     ACS_SHORT_LOG((LM_INFO, "Any exception ... catched !"));
85:     }
```

1-8	All required include files.
10	Begin of a try block. We are using remote exceptions.
14	Get reference to the sampling manager. In our case it is identified by the string **SAMP1**, as written in the CDB.
18	Narrow the reference of the sampling manager. Now we can access its method.

23-24	We required to sample the **brightness** property of the **LAMP1** component with a sampling period of **0.1** sec and a report period of **1** sec.
28	This is the name of the notification channel on which data are flushed (not used by the server, but used in the client to get data).
31	Start the sampling.
37-39	Suspend and resume the sampling.
45	Stop the sampling.
47	Destroy the sampling object (release all CORBA resources and memory); strongly suggested.
49	Release CORBA pointer.
53-86	Error handling. Overabundant, but is here as an example

5.2 C++ client example

The extraction of data from the notification channel requires no special features. Interested readers should refer to the notification channel documentation for an exact explanation on how to extract them. The *acssamp* module contains one full example (file *acssampOnlyNCClient.cpp*), which could be taken as a starting point.

5.3 Java server example

Bolded code should be adapted for the developer's particular needs. Note that all Client login/logout code has been omitted and the *acssamp* module contains the full example (file *acssampSupplier.java).*

```
///////////////////////////////////////////////////////////////////////
package alma.acssamp.jtest;
///////////////////////////////////////////////////////////////////////

import java.util.logging.Logger;
import alma.acs.component.client.ComponentClient;
import alma.acs.container.ContainerServices;
import alma.ACSSamp.*;

public class acssampSupplier extends ComponentClient
{
   private Samp m_samp;
   private Logger m_logger;

   public acssampSupplier(String managerLoc, String clientName) throws
Exception
   {
      super(null, managerLoc, clientName);

    // same service interface that a component would get from the
container...
      ContainerServices csrv = getContainerServices();

      // get a logger
      m_logger = csrv.getLogger();

      String sampCurl = "SAMP1";

      // get (CORBA) reference to Samp component
      org.omg.CORBA.Object sampObj = csrv.getComponent(sampCurl);

      // use CORBA helper class for the type cast
      m_samp = SampHelper.narrow(sampObj);
   }

   /**
   Calls methods on our samp component
   */
   public void doSomeStuff()
   {

      m_logger.info("will now use the samp component...");
      try
      {
         SampObj sampObj =
m_samp.initSampObj("LAMP1","brightness",1000000,10000000);

         sampObj.start();
         m_logger.info(" ACS sampling started");

         Thread.sleep(5000);
         sampObj.suspend();
50:          m_logger.info("ACS sampling suspended");
51:
52:          Thread.sleep(5000);
53:          sampObj.resume();
54:          m_logger.info("ACS sampling resumed");
55:
56:          Thread.sleep(6000);
57:          sampObj.stop();
58:          m_logger.info("ACS sampling stopped");
59:
60:          Thread.sleep(2000);
61:          sampObj.destroy();
62:          m_logger.info("ACS sampling destroyed");
63:       }
64:       catch(Exception e)
65:       {
66:           // not handled for the moment
67:       }
68:    }
69:
70:    public static void main(String[] args)
71:    {
72:
73:       String managerLoc = System.getProperty("ACS.manager");
74:       if (managerLoc == null)
75:       {
76:          System.out.println("Java property 'ACS.manager' " + " must be
    set to the corbaloc of the ACS manager!");
77:          System.exit(-1);
78:       }
79:
80:       String clientName = "acssampSupplier1";
81:       try
82:       {
83:           acssampSupplier foo = new acssampSupplier(managerLoc,
    clientName);
84:           foo.doSomeStuff();
85:       }
86:       catch (Exception e)
87:       {
88:           e.printStackTrace(System.err);
89:       }
90:    }
91: }
```

1-8	All required include files.
25-28	Get reference to the sampling manager. In our case it is identified by the string **SAMP1**, as written in the CDB.
31	Narrow the reference of the sampling manager. Now we can access its method.
43	We required to sample the **brightness** property of the **LAMP1** component with a sampling period of **0.1** sec and a report period of **1** sec.
45	Start the sampling.
49-53	Suspend and resume the sampling.
57	Stop the sampling.

5.4 Java client example

The extraction of data from the notification channel requires no special features. Interested readers should refer to the notification channel documentation for an exact explanation on how to extract them.

5.5 Error handling

The ACS sampling system uses remote exceptions to notify errors and/or abnormal program behaviour. Apart of re-throwing all standard CORBA exceptions, the following *ACSErrTypeCommon* exceptions are also thrown (see file *acssamp.idl*):

- ACSErrTypeCommon::OutOfBoundsEx
- ACSErrTypeCommon::MemoryFaultEx
- ACSErrTypeCommon::CORBAProblemEx
- ACSErrTypeCommon::TypeNotSupportedEx
- ACSErrTypeCommon::CouldntAccessPropertyEx
- ACSErrTypeCommon::CouldntAccessComponentEx
- ACSErrTypeCommon::CouldntCreateObjectEx

Refer to the *acserr* manual for further readings.

Known issue: Currently there is **no error handling** inside the sampling thread. This means that if some sampled values are lost (from the *get_sync()* method) or there is a notification channel failure, this will not be notified to the user. As soon as ACS will support the "Multithread exception handling" (as described for example in the paper "Error Handling for Business Information Systems" by Klaus Renzel) the necessary error support will be developed.

6 Database Configuration

In order to use the ACS sampling system, the Configuration Database (CDB) must be properly configured. In particular, the following lines must be present in the *Componets.xml* file:

```
<_ Name="SAMP1"         Code="acssamp"
                        Type="IDL:alma/ACSSamp/Samp:1.0"
                        Container="Container"/>
```

The corresponding schema files *SAMP.xsd* is installed automatically by means of the *Makefile*. If this will not be the case, the file could be retrieved from the *acssamp/ws/config/CDB/schemas* directory.

7 Appendix

Your best source of information is the code itself (i.e., Doxygen). Other than that, the locations (in CVS) of all acssamp files are:

ACS/LGPL/CommonSoftware/acssamp/ws/idl/acssamp.idl

ACS/LGPL/CommonSoftware/acssamp/ws/include/acssampImpl.h

ACS/LGPL/CommonSoftware/acssamp/ws/include/acssampObjImpl.h

ACS/LGPL/CommonSoftware/acssamp/ws/include/acssampObjTemplateImpl.h

ACS/LGPL/CommonSoftware/acssamp/ws/src/acssampImpl.cpp

ACS/LGPL/CommonSoftware/acssamp/ws/test/acssampOnlyNCServer.cpp

ACS/LGPL/CommonSoftware/acssamp/ws/test/acssampOnlyNCClient.cpp

ACS/LGPL/CommonSoftware/acssamp/ws/test/alma/acssamp/jtest/acssampSupplier.java

ACS/LGPL/CommonSoftware/acssamp/ws/test/alma/acssamp/jtest/acssampConsumer.java

ACS/LGPL/CommonSoftware/acssamp/ws/test/acssampFullNCTest.cpp

ACS/LGPL/CommonSoftware/acssamp/ws/config/CDB/schemas/SAMP.xsd