Atacama Large Millimeter	KGB-SPE-01/04 Revision: 1.36 2007-07-30 *Architecture Document* Klemen Žagar

Logging and Archiving

Klemen Žagar (klemen.zagar@ijs.si)

KGB, Jozef Stefan Institute

Radostina Georgieva (rgeorgie@eso.org)

ESO

Keywords: KGB-SPE-01/04

Author Signature:	Date:
Approved by: Institute:	Signature: Date:
Released by: Institute:	Signature: Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
1.0	2000-09-10	Klemen Zagar	All
	Created		
1.10	2000-09-30	Klemen Zagar	All
	G. Chiozzi's comments taken into account. XML schema made more flat.		
1. 20	2001-03-02	Klemen Zagar	All
	J. Knudstrup's comments taken into account. <Data> element added. LogId, StackId and StackLevel attributes added. <Variable> removed. LoggingProxy interface modified. DTD schemas written (appendix).		
1.21	2001-03-16	Klemen Zagar	3.4, 3.4.1.4, 3.4.1.8, A
	Final J. Knudstrup's comments taken into account. Redefined StackId and StackLevel, updated DTD schemas, fixed minor typos.		
1.22	2001-03-26	Grega Milcinski	All
	Applied ALMA template. Listing and Identifier styles added.		
1.23	2001-03-30	Klemen Žagar	3, 3.2, 3.5, 4.1.3, 4.1.4
	Gianluca Chiozzi’s comments taken into account. COS_ macro prefix changed to ACS_. References to MACI and BACI removed (instrumentation of BACI is still described). Notification Channel usage defined. Archiving parameter types defined. Figure redrawn.		
1.24	2001-11-08	Klemen Žagar	3.4.1.6, 3.5.3
	Priority now ranges from 1 to 15. ACS_LOG example fixed.		
1.25	2001-12-20	Klemen Žagar	All
	Removed reference to “CoCoS” and “Device”. The latter has been replaced with “Distributed Object”. Removed XML schema diagrams.		
1.26	2002-01-04	Klemen Žagar	4.1.1, 4.1.5
	Archiving configuration properties made consistent with implementation.		
1.27	2002-11-25	R. Georgieva	3.4.1.8.
	Escaping delineating characters within log messages.		

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
1.28	2002-03-03	R.Georgieva	
	Java Logging API and Python Logging API added.		
1.29	2003-11-1		
	Revised for ACS 3.0		
1.30	2004-07-23	G.Chiozzi	
	Updated XML Schema		
1.31	2005-03-17	G.Chiozzi	
	Updated documentation on Archiving		
1.32	2005-05-19	G.Chiozzi	
	Removed schema documentation from appendix and referenced online documentation.		
1.33	2006-05-11	D. Fugate	
	Python section was horribly out of date.		
1.34	2007-02-12	N. Barriga	Section 3.7
	Added documentation about type safe logs usage.		
1.35	2007-02-20	N. Barriga	Section 3.5
	Fixed example on how to get a logger in Java.		
1.36	2007-07-30	N. Barriga	Sections 3.4, 3.5, 3.6
	Added new API for specifying an audience, antenna and array to the logs.		

Table Of Contents

Scope

This document describes the architecture and design for logging and archiving data generated by ACS applications. This architecture is in accordance with the Logging and Archiving section of the ALMA Common Software Architecture document [RD05].

1 Introduction

Every properly built system needs to be able to log the events that occur in it (***logging***) so that the user of the system can gain an accurate understanding of the system's state, especially in case of a failure. For example, every component should log when it is created and destroyed. For systems with more stringent security requirements it might be required that every access to the system is logged, along with the user that performed it (auditing). Also, the system should log all unusual and error conditions.

Logging is very useful during debugging. Every entering or leaving of an important function could be logged to determine whether a function has been called at all and with what parameters. This information is also useful for profiling because more frequently used functions can be easily identified and more effort can be invested in their optimization.

Logging subsystem can be built very easily in a single computer scenario: a simple `fprintf` does the job. However, in a distributed system, means must be provided to centralize all logging activity in a single place, and still not overload the network too much. Also, the temporary unavailability of the network connectivity should not cause the logging subsystem to malfunction.

Another very important issue is the consistence of log entries. If all log entries stick to the same formatting rules (for example, the name of the log entry issuer, followed by a colon, followed by an error code, …), then automated log parsers can be built to intelligently filter or transform the log output, assisting the administrator to more efficiently manage the huge amount of data contained in the log.

A control system generates a huge amount of data, such as:

- Values of controlled entities, such as the current in a power supply or position of a motor.
- Reports of alarm conditions.
- Logs that assist development and administration of the control system, such as error reports, function call-stacks, trace logs, …

The generated data can be propagated throughout the system in two ways:

- ***Polling*** (synchronous): The interested client of the data queries the data source when it needs the information.

- ***Monitoring*** (asynchronous): The interested client subscribes a callback with the data source, so that the data source notifies the client whenever a certain condition is met (e.g., the data changed for more than a specified amount, or a certain period of time has elapsed since the last notification)

Which method to use depends on the nature of the data and, more importantly, the client's Quality-of-Service requirements for the data. For example:

- The client that reports alarms needs to be notified as soon as an alarm occurs. Polling every minute would allow for too much time to pass from the moment the alarm was generated up to the time when it was reported. On the other hand, polling every second could produce too much redundant network traffic.
- The client that merely displays the current status of the system and updates it with a predetermined (slow) refresh rate could poll the data whenever it would need to display it.

Apart from the issues mentioned so far, the data generated by the control system might need to be archived for later retrieval to assist in troubleshooting the system, or to allow for statistical analysis of the data. This activity is called ***archiving***.

Like regular acquisition of data, archiving can be implemented using polling or monitoring. Furthermore, there are several alternatives available as to how to propagate the data from the source to the archive:

- There is one central archive, which collects the data in the entire system.
- There is one archiving agent per computer in a control system, collecting its host's data only, and passing it to a central archive.

The first option could generate a constant, non-neglectable load to the control system's network infrastructure. The latter could generate a huge network load, but under controlled conditions (e.g., system's off-hours or on-demand). The total load in the latter case is smaller than in the first case because data is sent in batches and not individually.

Another question is the underlying mechanism through which the data to be archived is sent. This could be one of:

- A specialized protocol for transferring archive data.
- An existing mechanism for logging (i.e., archive is nothing else but a huge log file).
- An existing mechanism for monitoring.

1.1 Reference Documents

The following documents have been referenced in this document.

[RD01] ALMA ACS Basic Control Interface Specification, M. Plesko, G. Tkacik, G. Chiozzi - (http://www.eso.org/~gchiozzi/AlmaAcs/Releases/ACS_2_0_Docs/ACS_Basic_Control_Interface_Specification.pdf)

[RD02] Adaptive Communications Environment (ACE) - (http://www.cs.wustl.edu/~schmidt/ACE-overview.html)

[RD03] OMG: CORBA Telecom Log Service Specification - (ftp://ftp.omg.org/pub/docs/formal/00-01-04.pdf)

[RD04] ALMA Common Software Technical Requirements, ALMA-TRE-ESO-XXXXX-XXXX, G.Raffi, B.Glendenning, Issue 1.0, 2000-06-05

[RD05] ALMA Common Software Architecture, G.Chiozzi, B.Gustafsson, B.Jeram **-** (http://almaedm.tuc.nrao.edu/forums/alma/dispatch.cgi/Architecture/docProfile/100017/d20021117183329/No/ALMASoftwareArchitecture.pdf)

[RD06] OMG: Notification Service Specification - (ftp://ftp.omg.org/pub/docs/formal/00-06-20.pdf)

[RD07] Java Logging Overview - (http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/overview.html)

[RD08] ACS Online Documentation for schema files - (To De Updated.html)

2 Requirements

2.1 Logging

The logging subsystem must allow for the following:

1. It must provide an easy-to-use programming interface to the application developer.
2. Every log entry has an associated priority and type (error, fatal error, information, …).
3. Every log entry is equipped with a timestamp accurate to a deci-microsecond (100's of ns).
4. The logging subsystem must be centralized, so that all log entries generated in the systems sooner or later find their way to a central log. The order of the log entries is defined by their timestamp.
5. The subsystem should allow for filtering, so that log entries with insufficient priority do not get logged, whereas those with high priority get routed to the central log immediately.
6. The log entries are consistent, so that they can be interpreted by various automated tools for filtering and transforming.
7. Log entries can be buffered locally on the machine where they were generated, and transmitted over the network to the central log on demand or when the local buffer reaches a predetermined size.

2.2 Archiving

The archiving subsystem must allow for the following:

1. Every property in the control system is eligible for archiving.
2. Archiving can be enabled or disabled on per-parameter basis.
3. Archiving can be configured on per-parameter basis.
4. Individual parameter can be archived when it changes or with a certain fixed frequency.
5. There is a central point where the archive of all parameters is kept.
6. Individual parameter's archived data can be stored on a local machine, or immediately forwarded to the central archive.
7. Each parameter can be uniquely identified through the unique distributed object name and the parameter name.

8. The archiving subsystem supplies a mechanism for retrieving the historical value of any parameter at any time, provided that the value was archived prior to that time.

3 Logging Architecture

Logging subsystem leverages three systems that have been either already built, or at least well designed and specified. These are:

1. CORBA Telecom Log Service for centralizing all log entries generated throughout the system.
2. CORBA Notification Service for distributing log entries to interested clients (consumers of logs) when the entries are submitted to the centralized logger.
3. The mechanism for generating, formatting, filtering and caching log entries.
    - C++ suppliers of logs use the ACE Logging framework with its C++ API [RD02];
    - Java suppliers of logs use the standard Java Logging API [RD07];
    - Other suppliers of logs, e.g. a Python application, can use the stand-alone ACS Log Server which provides the generic functionality.

The logging subsystem is a very basic one and it should be considered as a part of the infrastructure.

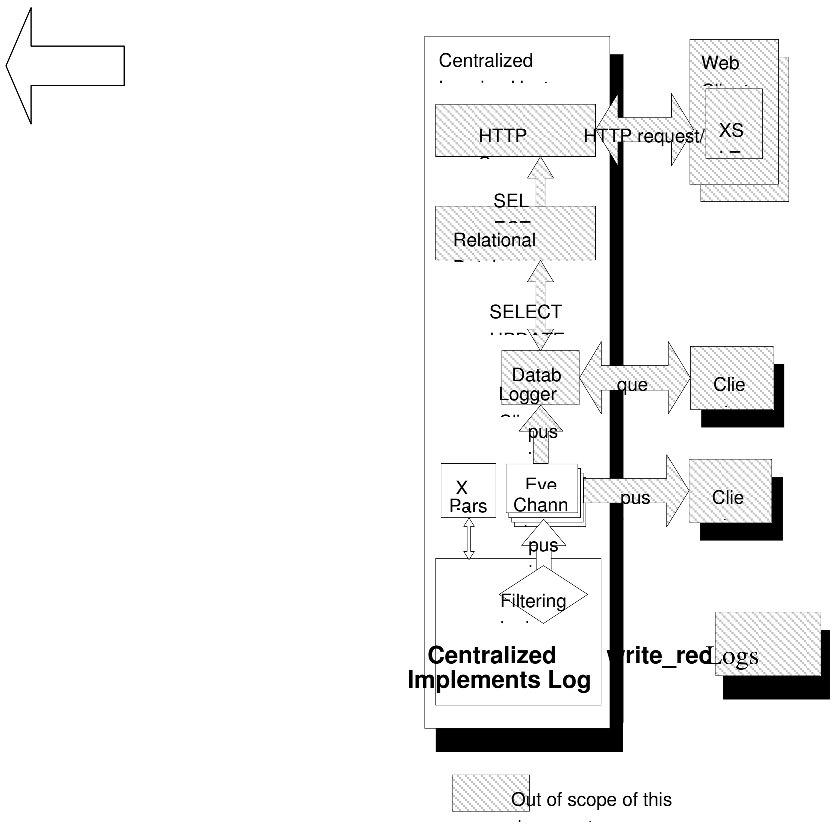

Figure 1: Architecture of the logging system.

The figure shows an overview of the ACS Logging System, based on the CORBA Telecom Log Service and The CORBA Notification Service. The CORBA Telecom Log Service has a Centralized Log object that is responsible for getting the logs from the log suppliers, validating them and submitting them to the Event Channels. The Event Channels push the logs to the subscribed log consumers. The shadowed objects are out of the scope of this document.

3.1 CORBA Telecom Log Service

The Telecom log service specification defines a set of IDL interfaces that are suitable for implementation of any kind of a log service [RD03]. The implementation using CORBA is done in the Centralized Log. All interfaces inherit from the `Log` interface which defines the following operations for submitting and querying log entries[1]:

```
interface Log
{
  // . . .
```

[1] **The listings presented here are excerpts from file** `$TAO_ROOT/orbsvcs/orbsvcs/DsLogAdmin.idl`**, courtesy of Matthew Brown.**

```
  // Write records to the log storage
  void write_records(in Anys records) raises(LogFull, LogLocked);

  // Returns all records in the log that match the given
  // constraint <c>.
  RecordList query(in string grammar,
                   in Constraint c,
                   out Iterator i)
    raises(InvalidGrammar, InvalidConstraint);

  // Retrieve <how_many> records from time <from_time> using
  // iterator <i>. Negative <how_many> indicates backwards
  // retrieval
  RecordList retrieve(in TimeT from_time,
                      in long how_many,
                      out Iterator i);
};
```

Telecom log specification represents a log entry by a structure whose IDL is:

```
  typedef unsigned long long RecordId; // RecordIds are unique within the
                                       // scope of one log.
  typedef TimeBase::TimeT TimeT; // Timestamp, as defined by the CORBA
                                 // TimeService.
  struct NVPair     // Name-Value pair
  {
    string name;
    any value;
  };
  typedef sequence<NVPair> NVList; // A set of name-value mappings
  struct LogRecord
  {
    RecordId id;      // Unique number assigned by the log
    TimeT time;       // Time when the event is logged (CORBA Time Service)
    NVList attr_list; // List of user defined name/value pairs. Not part of
                      // the event received by the log. Optional.
    any info;         // Event content
  };
```

For our purposes, the `info` element of the `LogRecord` structure consists of an XML string containing all the information about the record, such as the timestamp (`time` element is not suitable for our purposes, since it will contain the time when the log entry was logged to the central log, and not when the log entry was submitted; due to caching these two times could be significantly different). The details about the XML schema of the `info` element are described in 3.3 “Syntax of a Log Entry”.

3.2 Centralized Log

The Centralized Log that implements the CORBA Telecom Log Service is the facility that receives log entries from the entire system and dispatches them to interested clients. Particularly, it implements the `Log` interface of the Telecom Log Service. An example of an implementation of such a service is already bundled with TAO’s implementation of the Telecom Log Service and can be found under `$TAO_ROOT/orbsvcs/Logging_Service`.

The Centralized Log uses CORBA for getting the routed logs from the ***suppliers of logs (publishers)*** -applications that use the formatting, buffering and pushing capabilities of ACE API, Java Logging API, the ACS Log Service, etc. as they send log records to it.

Thus, the Centralized Logging Service receives an XML string as the “`any`” parameters in a call to `write_records from a supplier`. The XML string is formatted according to the specifications described in 3.3 “Syntax of a Log Entry”. Since the XML string already contains the timestamp information, that information should take precedence over the actual current timestamp of the receipt of the log entry.

The parsed log entry is then forwarded to appropriate event channel of the CORBA Notification Service which distributes the log entries further. The choice of the event channel depends on the log entry’s type and content. For example, log entries related to debugging could be forwarded to a different event channel than those related to archiving. The ***consumers of logs (clients)*** can access log entries by subscribing to the event channel of interest.

To store log records, the Centralized Logging Service needs a ***Database Logger client*** that has to make the log entries the Centralized Logging Service receives persistent. If such a client exists, other subscribed clients have additional options to access the database:

- Direct access to the SQL database (`SELECT`).
- Via Database Logging client’s `query` and `retrieve` methods.
- Via HTTP (the client is a web browser). HTTP server accesses the relational database, transforms the requested entries to XML, transmits them to the browser, which then uses XSLT to transform incoming XML to HTML.

3.3 Syntax of a Log Entry

Every log entry is represented as an XML document node. The schema of the XML is described in this chapter and the complete schema is provided in appendix A.

1.1.1 A Generic Log Entry

A generic log entry representation in XML looks like this:

```
<LogEntryType TimeStamp="yyyy-MM-ddThh:mm:ss.fff"
        File="filename" Line="lineno"
        Routine="routine"
        Host="hostname" Process="procname" Thread="threadname" Context="context"
        StackId="stackid" StackLevel="stacklevel"
        LogId="id" Uri="uri"
        Priority="p">
    <Data Name="name">value</Data>
    log entry message
</LogEntryType>
```

1.1.1.1 Log Entry Type

The log entries exist in different types to distinguish between the importance of information it provides. These types are described in the following sections and follow the convention specified by `$(ACE_ROOT)/ace/Log_Priority.h`.

Note that *LogEntryType* is not actually an XML tag, but merely a placeholder for the actual XML tag name such as `Debug` described in the following sections.

1.1.1.2 TimeStamp

The timestamp is a mandatory attribute of every log entry. It specifies the exact time when the log entry was submitted. The time is encoded in ISO 8601 format with a precision to one millisecond. The time is specified in TAI.

1.1.1.3 Source Code Information

The element representing a log entry is equipped with these attributes that convey the location in the source code from which the log entry was generated:

- **File**: The identification of the source file. The identification should be such that it uniquely identifies the file, and that it is possible to locate the source file with only little external information (such as project's root directory). The file name is specified relative to the root of the file system where the source file resided at the time of compilation, e.g., `/home/dknuth/ACS/motor/controller.cpp`. A more globally valid file name designation could be, for instance, `$ACS_ROOT/MACI/Activator.cpp`.
- **Line**: The line number in the source code where the log entry was submitted.
- **Routine**: The fully-qualified name of the subroutine (function) where the log entry was submitted from, for example `Activator::Init` or just `init()`.

These three attributes are optional since they cannot be provided for log entries in each language due to grammar or implementation restrictions. For example, in Java getting the source code line of the log record is very inefficient. Therefore, a log entry would only supply the file name and the method where the log entry originates from.

1.1.1.4 Runtime Context Information

The log element has six attributes that give more information regarding the runtime context in which the log entry is submitted:

- **Host**: The name of the computer on which the log entry is generated.
- **Process**: The name of the process from which the log entry is generated.
- **Thread:** The identification of the thread. For example, in C++ the identification is the name of the thread as supplied to `InitThread` (see 3.5.1.3. "Enabling the Logging Proxy").

- **Context**: Any additional context information supplied by the issuer of the log entry. For example, the name of the configuration database that is being used could be put here.
- **StackId**: Identification of a bundle of related log entries. All log entries in a bundle are caused by the same "root" log entry (e.g., the original cause of an error).
- **StackLevel**: Specification of the number of the log entries in the bundle a given entry that have caused the log entry. The root log entry has a `StackLevel` of 0, the immediate log entries caused by the root log entry have a `StackLevel` of 1, etc.

1.1.1.5 Log Entry Identification

Every log entry can be supplied with an optional `LogId` attribute which uniquely identifies the log entry's class (e.g., "file not found", "out of memory", "container starting", etc.).

`LogId` is particularly useful when used with error messages where it can be used as a key in the help system to look up detailed help/troubleshooting information. Variable sub-elements are also useful in such cases to provide more details about the source of the log entry. For example, if a file could not be found, the following XML could be generated:

```
<Error TimeStamp="2000-08-23T13:18:27.432"
       File="FileOpener.cpp" Line="131" Routine="FileOpener::Open"
       Host="Hurricane" Process="Activator" Thread="EventLoop"
       LogID="err_File_Not_Found" Priority="8">
    <Data Name="FullPath">/home/someuser/file.txt</Data>
</Error>
```

In addition to `LogId`, an optional `Uri` attribute is provided, which uniquely identifies the log entry's class. (e.g., `log://www.eso.org/acs/errors/OutOfMemory`).

1.1.1.6 Priority

The log type implies default priority of a log message. However, if the priority is explicitly specified, then the default is overridden.

Priority is measured as an integer number ranging from 1 to 15, where 1 is lowest and 15 highest priority. The value of 0 indicates the default priority.

1.1.1.7 Data Sub-element

Every log entry can contain arbitrary number of `<Data>` sub-elements. These sub-elements are useful for reporting values of individual variables to report the state of the object that submitted the log entry.

The `Name` attribute is mandatory as well as the content of the element.

1.1.1.8 Log Entry Message

The optional log entry message is a string of characters. The message can be either an XML formatted string, or a CDATA section. The only rule it must obey is not to contain a sub-string `]]> or` characters such as '<', '>' or '&', since it terminates a `CDATA` section.

- **INFO**
  log level is used to publish information of interest during the normal operation of the system.
  This information is directed to operators, engineers or anybody else working with the system.
- **NOTICE**
  logs are used to catch the attention of people (normally operators or software engineering) looking at the logging output.
  They denote important situations in the system, but not necessarily error/fault conditions.
  A NOTICE logging level should be selected with care, because many NOTICE messages weaken the attention of the reader.
- **WARNING**
  logs are used to report to readers (normally operators or software engineering) conditions that are not errors but that could lead to errors/problems.
  A WARNING logging level should be selected with care, because many WARNING messages weaken the attention of the reader.

1.1.2 Trace Log Entry

Trace logs are generated whenever a function is entered. And are used to report calls to a function.
They are used to build call trees during very critical debugging situations.
The amount of TRACE logs can be huge and will very likely affect very substantially the performance of the system
TRACE logging should be switched on only in very particular situations and for a short time.

A trace log entry (`<Trace>`) corresponds to submitting a log entry of type `LM_TRACE` to the ACE logging system. The default priority of such an entry is 2.

With trace log entries, the `Routine` attributes are mandatory. There can also be several `<Data>` sub-elements, whose purpose is to dump the function's parameters.

The log entry message is a mandatory fully-qualified name of the function that was entered, for example `MyNamespace::MyClass::MyFunction`.

1.1.3 Debug Log Entry

Debug logs are used only while debugging the system.
Therefore such logs are normally only interesting for software engineers. Analysis of DEBUG logs

should take place only while investigating problems and can put a substantial amount of load on the system.

A debug log entry (`<Debug>`) corresponds to submitting a log entry of type `LM_DEBUG` according to the ACE logging system. The default priority of such an entry is 3. Debug logs are useful for dumping object state.

With debug log entries, the `File` and `Line` attributes are mandatory. There can also be several `<Data>` sub-elements, whose purpose is to dump the object's state.

The log entry message is optional.

1.1.4 Info Log Entry

Info log level is used to publish information of interest during the normal operation of the system. This information is directed to operators, engineers or anybody else working with the system. They can also be employed for transmitting useful payload (such as archiving data).

An info log entry (`<Info>`) corresponds to submitting a log entry of type `LM_INFO` to the ACE logging system. The default priority of such an entry is 4.

1.1.5 Notice Log Entry

Notice logs are useful for logging normal, but significant activity of the system, for example startup or shutdown of individual services. They are used to catch the attention of people (normally operators or software engineering) looking at the logging output. They denote important situations in the system, but not necessarily error/fault conditions.

A NOTICE logging level should be selected with care, because many NOTICE messages weaken the attention of the reader.

A notice log entry (`<Notice>`) corresponds to submitting a log entry of type `LM_NOTICE` to the ACE logging system. The default priority of such an entry is 5.

1.1.6 Warning Log Entry

Warning logs are used to report to readers (normally operators or software engineering) conditions that are not errors but that could lead to errors/problems.
A WARNING logging level should be selected with care, because many WARNING messages weaken the attention of the reader.

A warning log entry (`<Warning>`) corresponds to submitting a log entry of type `LM_WARNING` to the ACE logging system. The default priority of such an entry is 6.

1.1.7 Error Log Entry

Error logs denote error conditions.

They are normally generated by the Error System and not explicitly use in applications by calling the logging API

An error log entry (`<Error>`) corresponds to submitting a log entry of type `LM_ERROR` to the ACE logging system. The default priority of such an entry is 8.

1.1.8 Critical Log Entry

Critical logs denote an Alarm condition that shall be reported to operators.

They are normally generated by the Alarm System and not explicitly use in applications by calling the logging API

A critical log entry (`<Critical>`) corresponds to submitting a log entry of type `LM_CRITICAL` to the ACE logging system. The default priority of such an entry is 9.

1.1.9 Alert Log Entry

Alert logs denote an Alarm condition that shall be reported to operators. This denotes a problem more important than Critical.

They are normally generated by the Alarm System and not explicitly use in applications by calling the logging API

An alert log entry (`<Alert>`) corresponds to submitting a log entry of type `LM_ALERT` to the ACE logging system. The default priority of such an entry is 10. Alerts are used for reporting errors that must be solved immediately.

1.1.10 Emergency Log Entry

Emergency logs denote an Alarm condition of the highest priority.

They are normally generated by the Alarm System and not explicitly use in applications by calling the logging API

An emergency log entry (`<Emergency>`) corresponds to submitting a log entry of type `LM_EMERGENCY` to the ACE logging system. The default priority of such an entry is 11. Alerts are used for reporting errors that must be solved immediately.

3.4 ACS C++ Logging API

1.1.11 ACE Logging

The ACS C++ Logging API for generating, formatting and filtering log entries is based on the ACE Logging API and is provided by a collection of operating system wrappers and common design pattern implementations with the following functionality:

1. A data structure that can hold a log entry (the `ACE_Log_Record`, defined in `$ACE_ROOT/ace/Log_Record.h`). The structure also holds priority, type and the timestamp of the log entry, fulfilling requirements 2.1.2 (priority and type) and 2.1.3 (timestamp). Furthermore, ACE logs filename and line number of the source code where the log entry originates from. It should be noted that priority and type in ACE can not be set separately, because type implies priority, and vice-versa.

2. A mechanism for submitting log entries. The mechanism is modeled by the `ACE_Log_Msg` class (`$ACE_ROOT/ace/Log_Msg.h`). There is one instance of this class per thread.

3. ACE's logging mechanism is extensible, allowing for custom callbacks to be registered with an `ACE_Log_Msg` object. These callbacks (implementations of a `ACE_Log_Msg_Callback` abstract class) receive all entries submitted to the logging mechanism and can process them any way they want. (Please note that the callback must be registered with the `ACE_Log_Msg` at the beginning of each thread's lifetime.)

4. ACE defines several macros which the application programmer can use to submit log entries, such as `ACE_ERROR` and `ACE_DEBUG` (defined in `$ACE_ROOT/ace/Log_Msg.h`). This fulfills the requirement 2.1.1 (programming interface).

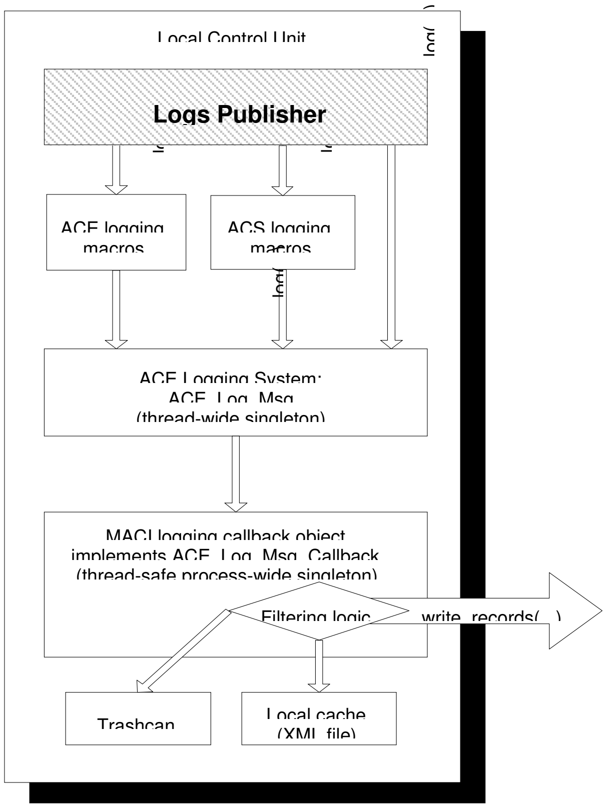
Local Control Unit
Logs Publisher
ACE logging macros
ACS logging macros
ACE Logging System: ACE_Log_Msg (thread-wide singleton)
MACI logging callback object implements ACE_Log_Msg_Callback (thread-safe process-wide singleton)
Filtering logic
write_records(...)
Trashcan
Local cache (XML file)

Figure 2: Architecture of the ACE Logging framework.

The figure gives an overview of ACE Logging framework. The ACE Logging System gets log entries that can be generic or specific (using ACS logging macros). It submits them to an object implementing `ACE_Log_Msg_Callback` that provides the filtering and the caching capabilities of the framework. The shadowed objects are out of the scope of this document.

The ACE's mechanism is flexible and high-performing and allows the implementation of objects that are specific to the ACS Logging requirements.

Important with respect to the formatting is that fact that the default logging macros of ACE (`ACE_DEBUG`, `ACE_ERROR`, etc.) already provide the logging system with the file name and the line number attributes. Additionally, the logging system outputs the runtime context along with all log entry types except for info log entry which has to be taken care of by requesting it explicitly through `LoggingProxy`'s `LM_RUNTIME_CONTEXT` flag. Though these last attributes are optional according to the XML Schema, their appearance in the log records could be quite helpful.

The implementation of the `ACE_Log_Msg_Callback` abstract class' `log` method provides with the rest of the functionality:

```
//
/// The pre-defined macro for outputting log entries. It accepts three parameters
```

```
///
/// - flags: This parameter specifies the priority and additional log-entry
/// flags, such as whether to output the runtime context (thread & process)
/// or not.
/// - routine: The fully qualified name of the routine where the log-entry is
/// being generated. Can be 0, in which case the routine name is not output.
/// - log: Formatted as (log_type, format_string, . . .). Passed as a parameter
/// to ACE's logging macros.
///
/// Usage example:
///
/// ACS_LOG(LM_SOURCE_INFO | LM_PRIORITY(7),
///       "maci::ContainerImpl::init",
///       (LM_INFO, "A sample log entry %d", i));
///
#define ACS_LOG(flags, routine, log) \
{ \
   LoggingProxy::Flags(flags); \
   LoggingProxy::Routine(routine); \
   ACE_ERROR(log); \
}

///
///
/// Manipulate priority contained in the log entry's flags. The priority can
/// be from 0 ("use default") through 1 (lowest) to 31 (highest).
///
#define LM_PRIORITY(p) p
#define LM_GET_PRIORITY(f) (f & 0x0F)

/// If OR-ed with log entries' flags, the runtime context (host name, process name,
/// thread name, context, stack ID and stack level) will also be output.
///
#define LM_RUNTIME_CONTEXT 0x00000200

///
/// If OR-ed with log entries' flags, the source code information (file name,
/// line number) will also be output.
///
#define LM_SOURCE_INFO 0x00000100

/// The Log Message Callback
class logging_EXPORT LoggingProxy : public ACE_Log_Msg_Callback
{

  public:
    /// Receives all log entries submited
    /// within the process. Thread safe!
    void log(ACE_Log_Record &log_record);

    /// Specifies the log entry type, if the output representation is different
    /// than the one implied with ACE's log entry type. Applies for the next
    /// log entry only. Pointer to the string must be stored in the thread-specific
    /// storage!
    static void LogEntryType(const ACE_TCHAR *szType);

    /// Specifies the name of the routine (function) where the following log entry
    /// will be generated. Pointer to the string must be stored in the
    /// thread-specific storage!
    static void Routine(const ACE_TCHAR *szRoutine);

    /// Set the flags that will apply to the log entry that will be submitted next.
    /// Flags must be stored in thread-specific storage! Flags are obtained by OR-ing
    /// appropriate LM_* values above. If priority is 0, the default priority
    /// associated with ACE's log entry type (LM_INFO, LM_ERROR, ...) is implied.
    static void Flags(unsigned int uiFlags);

    /// Specifies the name of the thread. Pointer to the name must be stored in
    /// the thread-specific storage!
    static void ThreadName(const ACE_TCHAR *szName);

    /// Returns the name of the thread.
    static const ACE_TCHAR *ThreadName();

    /// Specifies the name of the process. Must be stored in a process-wide global
    /// variable!
    static void ProcessName(const ACE_TCHAR *szName);

    /// Returns the name of the process.
    static const ACE_TCHAR *ProcessName();

    /// Reset the list of custom attributes. The attributes are applicable to the
    /// next log entry only.
    static void ResetAttributes();

    /// Add an attribute to the list of next log entries' attributes.
    static void AddAttribute(const ACE_TCHAR *szName, const ACE_TCHAR *szValue);

    /// Specify the LogId attribute of the log entry that follows. Can be 0 (default)
    /// in which case no LogId attribute is output.
    static void LogId(const ACE_TCHAR *szName);

    /// Specify the URI attribute of the log entry that follows. Can be 0 (default)
    /// in which case no URI attribute is output.
    static void URI(const ACE_TCHAR *szName);

    /// Specifies the stack ID of the current logical thread. Pointer to the name
    /// must be stored in the thread-specific storage! Can be set to NULL if
    /// the logical thread ID is unknown.
    static void StackId(const ACE_TCHAR *szId);

    /// Returns the the logical thread ID. Must have been set previously using
    /// StackId.
    static const ACE_TCHAR *StackId();

    /// Set the stack level in the current logical thread. The value must be stored
    /// in the thread-specific storage!
    static void StackLevel(int nLevel);

    /// Retrieve the stack level in the current logical thread.
    static int StackLevel();

    /// Set the context in which the code is operating. Pointer to the name must
    /// be stored in the thread-specific storage!
    static void Context(const ACE_TCHAR *szName);

    /// Retrieve the context in which the code is operating.
    static const ACE_TCHAR *Context();

    /// Supply data with the log entry that follows.
    /// The maximum length for AddData value is ADD_DATA_VALUE_MAX (255+\0). If it is too long
    ///  it will be truncated.
    static void AddData(const ACE_TCHAR *szName, const ACE_TCHAR *szFormat, ...);
...
};
```

An instance of the `LoggingProxy` class is created in Container's `Init` and destroyed in Container's `Done` method. It is configured from the Container's configuration record[2] using the properties listed below.

1.1.12 Logging Proxy's Configuration Data

The logging system caches logs before transmitting them to the centralized logging service. The logging is done on a per-process basis. The following parameters control logging with respect to how and what messages are published, e.g. whether they are printed or logged at all, whether they are cached locally or transferred to the logging service immediately, etc.

`ACS_LOG_STOUT` (unsigned 32-bit integer): `The environmental variable corresponding to the least priority of a log message that is to be sent to stout. By default, only log messages with priority equal or higher than LM_INFO` (3) are sent to `stout. If ACS_LOG_STOUT>0, all log messages with priority >= ACS_LOG_STOUT are also sent to stdout.`

`cacheSize`(unsigned 32-bit integer): The number of log entries to be cached before logging. When this number is reached, all log entries are transferred to the centralized logging. If network connection is not available, the local cache continues to grow, and every submitting of a log entry will attempt to flush the cache to the centralized logging.

`minCachePriority`(unsigned 32-bit integer): Minimum log priority. Log entries the priority of which is below (smaller than) the one specified with this property are ignored (neither cached nor submitted to the Centralized Logging Service). By default, the value is set to zero so that all messages are logged. *In release version of the system, this is set to `LM_INFO` (3), ignoring `LM_TRACE` and `LM_DEBUG` log entries. Debug version of the system sets this to `LM_DEBUG` (2). During development, it is set to `LM_TRACE` (1).*

`maxCachePriority` (unsigned 32-bit integer): Maximum log cache priority. Log entries whose priority exceeds (is greater than) the one specified with this property are directly transmitted to the Centralized Logging Service, bypassing the local cache. If this is set to `MinCachePriority` – 1, the local cache feature is disabled.

`centralizedLogger`(string): An IOR string representing the centralized logging object. The IOR is expected to denote a persistent object which implements the Telecom Log Service's `Log` interface, in particular the `write_records` method.

1.1.13 *`log`* Method Semantics

The log method receives an `ACE_Log_Record` object, which it first transforms into an XML string, obeying the XML schema of the 2.1.6 - "Consistence of Log Entries".

Depending on the log entry's type (and thus priority) the string is either written to a local cache (a regular XML file opened for appending and flushed for every submitted log entry, or an in-memory

[2] **This implies that logging proxy can be configured on a per-activator (i.e., per Local Control Unit) basis.**

XML string), or transmitted to the Centralized Logging Service. If the local cache contains `CacheSize` or more elements, an attempt is made to transmit the entire local cache to the Centralized Logging Service.

The reference to the Centralized Logging Service is given by the `CentralizedLogger` configuration property. To submit an entry to the Centralized Logging Service, its `write_records` method is employed, passing the XML string as the expected `any` parameter.

1.1.14 Enabling the Logging Proxy

Due to design of ACE, the callback for logging must be registered as well as unregistered with `ACE_Log_Msg` per-thread singleton for every thread individually. This is done automatically by the configuration methods of the Container i`nitThread` and d`oneThread that` have the following signatures:

```
class class maci_EXPORT ContainerImpl : : // . . .
{
  // . . .
  static void initThread(const char * threadName = 0);
  void doneThread();
};
```

The two methods are only a part of the Container servant and are not exposed through its CORBA interface.

If a nonempty string is passed as a parameter to `initThread`, a `LM_INFO` log entry is output associating the thread-ID of the current thread with its name.

1.1.15 Submitting Log Entries

As already mentioned, ACE's logging infrastructure is used for submitting log entries. It can be used at these levels:

- The macro `ACS_LOG`, or one of specialized macros `ACS_TRACE`, `ACS_DEBUG`, `ACS_DEBUG_PARAM`, `ACS_SHORT_LOG`, and `ACS_LOG_TIME, defined in the ACS include file` logging.h.
- Using `ACE_Log_Msg` and `LoggingProxy` directly.

1.1.15.1 Submitting the Source Code Information

The following code submits the source code information:

```
ACS_LOG(LM_SOURCE_INFO, // flags
        “main”,         // routine name
        (LM_INFO,       // informational log entry
         “”));          // no additional message text
```

The resulting log entry in XML would look like this (there would be no white-spaces in the actual output; they are shown below for purposes of legibility only).

```
<Info TimeStamp="2000-09-10T21:34:32.132"
      File="test.cpp" Line="131"
      Routine="main"
      Priority='4'></Info>
```

1.1.15.2 Submitting the Runtime Context

The configuration methods of the Container take care of setting up the runtime context information, e.g. the host name as well as the process and the thread information:

```
ACE_Log_Msg::instance()->local_host("host"); // set the host name
LoggingProxy::ProcessName("proc");           // called at process startup
LoggingProxy::ThreadName("thr");             // called at thread startup

ACS_LOG(LM_RUNTIME_CONTEXT,
        0,
        (LM_ERROR,
         "hello"));
```

The resulting log entry in XML would look like this:

```
<Error TimeStamp="2000-09-10T21:34:31.435"
       Host="host" Thread="thr" Process="proc"
       Priority='7'/>
  Any number 123
</Error>
```

1.1.15.3 Submitting a Variable's Value

The following code submits a value of a variable:

```
LoggingProxy::AddData("dMyDouble", "%f", dMyDouble);
ACS_LOG(0, "main", (LM_TRACE, ""));
```

The length of a value should not exceed 255 characters otherwise it is truncated.

1.1.15.4 Overriding the Default Priority

The following code overrides the default priority of a log entry:

```
ACS_LOG(LM_PRIORITY(12), 0,
        (LM_TRACE,       // Could be anything…
         "Message"))     // Could be anything…
```

1.1.15.5 Submitting an Arbitrary Message

To submit an arbitrary message, care must be taken not to break XML formatting rules (for example, < and > should be used with care). If the message contents are not known in advance and a possibility exists that they would break XML formatting rules, code like this should be used:

```
// This macro is predefined by ACS
#define LM_CDATA(t) "<![CDATA[" t "]]>"
ACE_ERROR((LM_WARNING,
           LM_CDATA("Some < text %s >"),
           szAString))
```

The unpredictable text is placed in an XML `CDATA` section.

1.1.16 Specifying an Audience, Array and/or Antenna for a log

1.1.16.1 API

- The possible audiences are defined in acscommon.idl, to use them, just access the appropriate one, for example:

```
string a = log_audience::OPERATOR;
```

- New macros have been defined in loggingMACROS.h:

```
#define LOG_FULL(logPriority, logRoutine, logMessage, logAudience, logArray,
logAntenna)
#define LOG_With_ANTENNA_CONTEXT(logPriority, logRoutine, logMessage, logArray,
logAntenna)
#define LOG_TO_AUDIENCE(logPriority, logRoutine, logMessage, logAudience)
```

1.1.16.2 Examples

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


#include <maciSimpleClient.h>

int main(int argc, char *argv[])
{
        maci::SimpleClient client;

        if (client.init(argc,argv) == 0){
                return -1;
        }
        else{
                // Log into the manager before doing anything
                client.login();
        }

        ACS_SHORT_LOG((LM_WARNING,"ACS_SHORT_LOG"));
        LOG_FULL(LM_WARNING,"main","LOG_FULL",log_audience::OPERATOR,"array01","Antenna01");
        LOG_WITH_ANTENNA_CONTEXT(LM_WARNING,"main","LOG_WITH_ANTENNA_CONTEXT","array01","Anten
na01");
        LOG_TO_AUDIENCE(LM_WARNING,"main","LOG_TO_AUDIENCE",log_audience::OPERATOR);
        client.logout();

        return 0;

}
```

3.5 ACS Java Logging API

1.1.17 JSDK Java Logging API

The official Java Logging API (java.util.logging package [RD07]) provides with a framework for generating, formatting and filtering log entries:

1. An object that can hold a log record (`LogRecord`). Its methods allow getting the level of priority, type and the timestamp as well as the filename, the process, the thread and the context of the source code where the log entry originates from.

2. An object that is used to log messages for a specific system or application component (`Logger`).

3. A mechanism for taking log entries and exporting them modeled by a `Handler` class (`ConsoleHandler, FileHandler`). There is one instance of the subclasses of this class per container[3]. Both loggers and the handlers are organized in a hierarchical namespace so that children may inherit some properties from their parents.

3. An object that provides support for formatting a `LogRecord` (`Formatter`). The formatter takes a `LogRecord` and converts it to a string.

4. An object that defines a set of standard logging levels that can be used to control logging output `(Level).` It can be applied to a log record, a logger and a handler. Specifying the lowest acceptable level acts for implementing the filtering functionality.

1.1.18 ACS Java Logging

The ACS Java Logging API is based on the official JSDK Java Logging and it has been integrated with the implementation of the CORBA Telecom Logging Service and the rest of the ACS.

[3] For more details, read about the Component-Container model in "Java Component Tutorial."

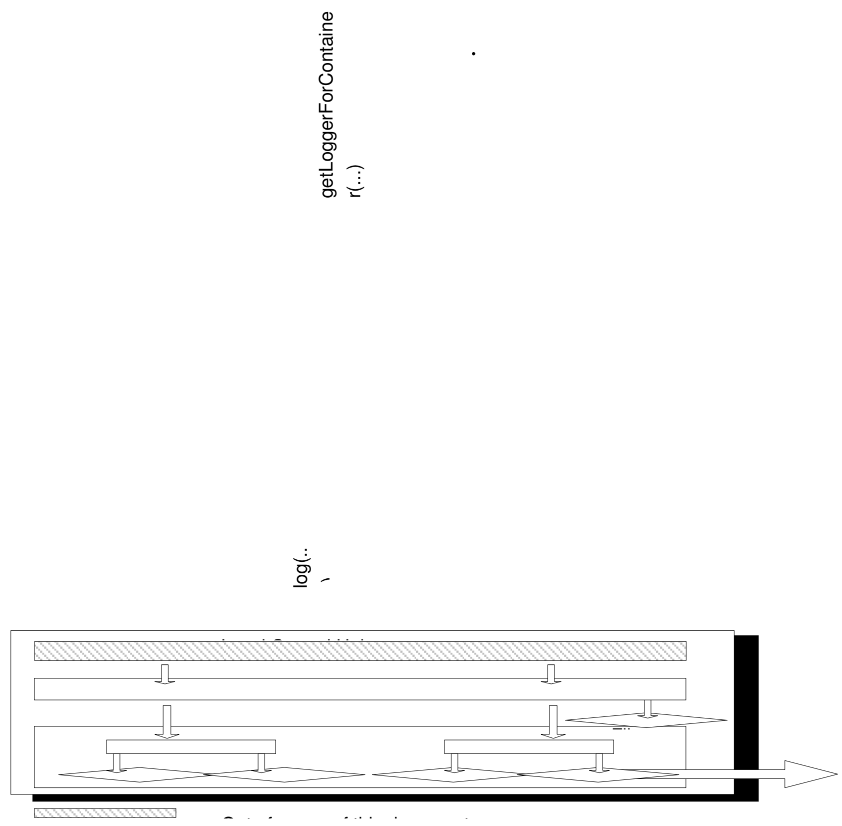

Figure 3: Architecture of the ACS Java Logging framework.

The Logger object gets log entries that it submits to a handler that provides the formatting and the caching capabilities of the framework. Filtering is done both at the logger and at the handle. The shadowed objects are out of the scope of this document.

1.1.19 ALMA Logging Configuration Data

The file `almalogging.properties` sets the specifications for the Java Logging properties. Having in mind the above, properties like the level(s) of the logger(s), the handler(s), the associated formatter(s), etc. can be all set in the configuration file. As an example consider the configuration file:

```
.level = ALL

############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################

# default file output is in user's home directory.
alma.acs.logging.AcsLoggingHandler.level = ALL
alma.acs.logging.AcsLoggingHandler.localcopy = true

alma.acs.logging.AcsRemoteHandler.cacheSize = 4
alma.acs.logging.AcsRemoteHandler.formatter = alma.acs.logging.formatters.AcsXMLFormatter

java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
# java.util.logging.ConsoleHandler.formatter = java.util.logging.XMLFormatter

############################################################
# Facility specific properties.
# Provides extra control for each logger.
############################################################

# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
alma.acs.containerstart.level = ALL
alma.acs.logger.level = ALL
```

`handlers`: is omitted since there is a default console handler which uses its simple log buffer to publish all the log records. It is implemented as part of the `AcsLoggingHandler.` It is also used for transmitting the log records to the Centralized Log that are initially stored in the log buffer. If there is no connection, the log records end up in the console. The `ConsoleHandler` uses always the console as a medium for publishing.

`.level`: should be set to one of the variables specified in the Java Logging API.

The handler's specific level, buffer size and formatter should be defined as variables too. Since the log records get published each time the buffer reaches its size and the user gets the log records as soon as the buffer is full, the size variable should not be assigned a big number.

The other specific properties include setting a specific log level for the container or other components.

1.1.20 `log` Method Semantics

Calling the `log` method allows submitting a log entry of a certain type to the Java framework of objects which then passes it on to the Centralized Logging Service. The mechanism involves a logger, a formatter and a handler. The logger passes the string further to its associated handler if the priority of the log record is above the global logging level.

The handler also deals further only with log records of higher priority than the handler specific level. Depending on the log record's type (and thus priority) the string is written to a local buffer (a formatter is called to transform the string into an XML string obeying the XML schema). If the buffer's size is reached, an attempt is made to transmit the entire local buffer to the Centralized Logging Service employing its `write_records method`.

1.1.21 *Obtaining a* `Logger`

There are several ways of obtaining a Logger object. The recommended ones are:

- For an application:

```
import java.util.logging.Logger;
import alma.acs.logging.ClientLogManager;

Logger m_logger = ClientLogManager.getAcsLogManager().getLoggerForApplication(clientName,
true); // the last parameters enables or disables remote logging
```

- For a component:

```
import java.util.logging.Logger;
import alma.acs.container.ContainerServices;

Logger m_logger = getContainerServices().getLogger();
```

1.1.22 `log` Method Use

Logging can be done in two ways: using `log` for logging log records or the level-specific method for logging messages (`finest`, `finer`, `info`, `warning`, `severe`, `all, off`).

```
m_logger.log(LogRecord.INFO, “log INFO record using the generic method log”);
m_logger.info(“log INFO records using the specific method info”);
```

In the above example, a logger that belongs to the namespace of the container – `alma.acs.container` – is instantiated. Because of the logger’s hierarchical structure, this logger is a child of the logger with a namespace `alma.acs`. In case the properties file does not specify the level for the log records to be logged with `alma.acs.container, the level of alma.acs would be considered, if specified. Otherwise the default global logging level would be considered.`

1.1.23 Specifying an Audience, Array and/or Antenna for a log

1.1.23.1 API

- The possible audiences are defined in acscommon.idl, to use them import the appropriate one, for example:

```
import alma.log_audience.OPERATOR;
String a = OPERATOR.value;
```

- Two new methods to the alma.acs.logging.AcsLogger:

```
public void logToAudience(Level level, String msg, String audience);
public void logToAudience(Level level, String msg, Throwable thr, String audience);
```

- New class alma.acs.logging.domainspecific.AntennaContextLogger

```
public AntennaContextLogger(AcsLogger logger);//constructor
public void log(Level level, String msg, String audience, String array, String
antenna);
public void log(Level level, String msg, Throwable thr, String audience, String array,
String antenna);
public void log(Level level, String msg, String array, String antenna);
public void log(Level level, String msg, Throwable thr, String array, String antenna);
```

- New class alma.acs.logging.domainspecific.ArrayContextLogger

```
public ArrayContextLogger(AcsLogger logger);//constructor
public void log(Level level, String msg, String audience, String array);
public void log(Level level, String msg, String array);
public void log(Level level, String msg, Throwable thr, String audience, String array);
public void log(Level level, String msg, Throwable thr, String array);
```

1.1.23.2 Example

```
package alma.acs.logging;
import java.util.logging.Level;

import alma.acs.component.client.ComponentClient;
import alma.acs.logging.domainspecific.AntennaContextLogger;
import alma.log_audience.OPERATOR;

public class TestAudArr extends ComponentClient{
        public TestAudArr(String managerLoc, String clientName) throws Exception {
                super(null, managerLoc, clientName);
        }

        public static void main(String args[]){
                String managerLoc = System.getProperty("ACS.manager");
                if (managerLoc == null) {
                        System.out.println("Java property 'ACS.manager' must be set to
the corbaloc of the ACS manager!");
                        System.exit(-1);
                }
                String clientName = "TestAudArr";
                TestAudArr client = null;
                try{
                        client = new TestAudArr(managerLoc, clientName);
                        AcsLogger m_logger =
(AcsLogger)client.getContainerServices().getLogger();
                        AntennaContextLogger logger = new
AntennaContextLogger(m_logger);
                        m_logger.log(Level.WARNING, "Normal Log");
                        m_logger.logToAudience(Level.WARNING, "Log with audience",
OPERATOR.value);
                        m_logger.logToAudience(Level.WARNING, "Log exception with
audience", new Exception("My dummy exception"), OPERATOR.value);
                                                        logger.log(Level.WARNING, "Log
with audience, array and antenna", OPERATOR.value, "Array01", "Antenna01");
                        logger.log(Level.WARNING, "Log with array and antenna",
"Array01", "Antenna01");
                        logger.log(Level.WARNING, "Log exception with audience, array
and antenna", new Exception("My dummy exception"), OPERATOR.value, "Array01",
"Antenna01");
                        logger.log(Level.WARNING, "Log exception with array and
antenna", new Exception("My dummy exception"), "Array01", "Antenna01");

                        Thread.sleep(1000);
                }catch(Exception e){
                        System.out.println("Error creating test client");
                }
                try{
                        client.tearDown();
                }catch(Exception e){
                        System.out.println("Error destroying test client");
                }
        }
}
```

1.1.24 Java Log Levels

The Java log levels have been remapped to comply with the ACS log levels from the XML schema. The mapping is done in the `alma.acs.logging.AcsLogLevel` class where the ACS levels, like the JAVA API levels, are specified by ordered integers. The OFF level which is not mentioned in the XML schema is included for dealing with bad levels as well as for blocking logging:

ACS Level (ACE Level)	ACS Logging Priority	Java API Level	Java Logging Priority
TRACE	2	FINEST (FINER)	400
DEBUG	3	FINE (CONFIG)	700
INFO	4	INFO	800
NOTICE	5	INFO	801
WARNING	6	WARNING	900
ERROR	8	WARNING	901
CRITICAL	9	WARNING	902
ALERT	10	WARNING	903
EMERGENCY	11	SEVERE	1000
TRACE	2	ALL	Integer.MIN_VALUE
OFF	-	OFF	Integer.MAX_VALUE

1.1.1 ACS Logging Class Diagram

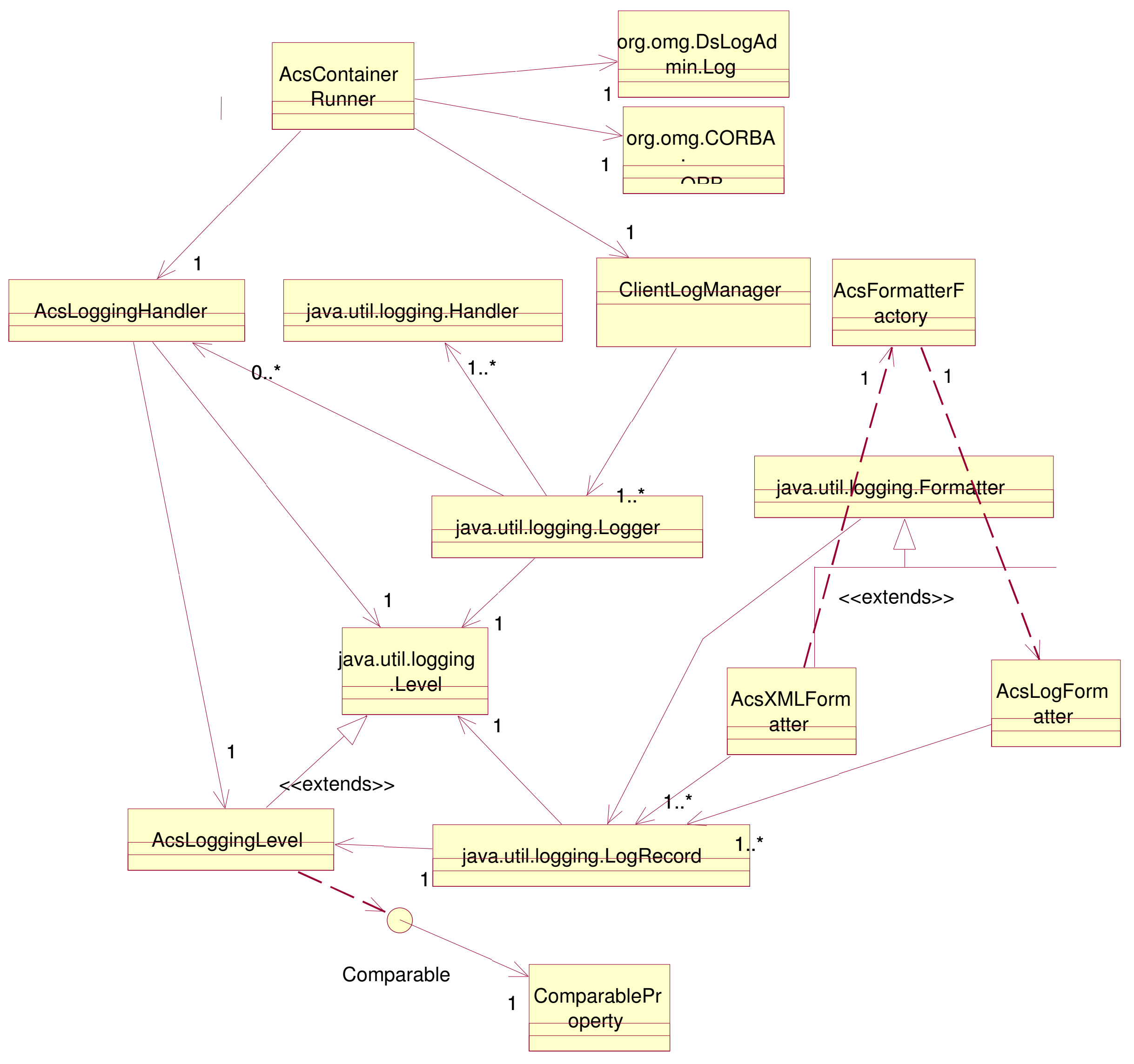

Figure 4: Class Diagram of ACS Formatters .The classes relate as shown on the class diagram above.

1.1.2 ACS Formatters

The formatters involved are named according to the ACS Logging Level. The `alma.acs.logging.AcsXMLFormatter` is a formatter object that produces a valid XML string out of a log message according to the XML schema for ACS.

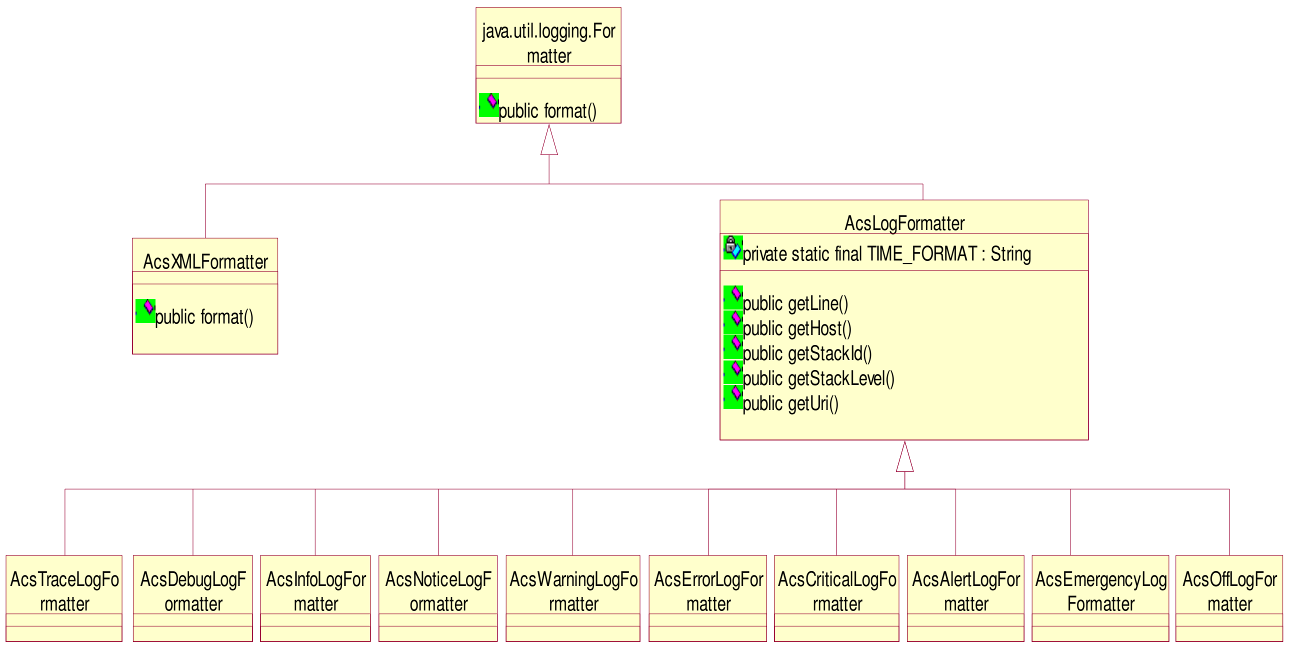

Figure 5: Class Diagram of ACS Formatters.

The AcsLogFormatter object defines methods for getting the properties needed for formatting a string into an XML string. The AcsXMLFormatter calls any of the customized formatters.

3.6 ACS Python Logging API

1.1.3 ACS Python Logging

The ACS Python Logging API provides an interface used to send logs to the (CORBA) ACS logging service object which lives within the *acsLogSvc* process. *acsLogSvc* then publishes the logs to an event channel which distributes them to all interested consumers such as the *jlog* GUI.

The standard ACS Python logger is available via a *"getLogger()"* method of *Acspy.Servants.ContainerServices* or by using the *"getLogger('logger name')"* function found in the *Acspy.Common.Log* module. The logger object returned is derived from the native Python logging class*, logging.logger*. Additionally, the ACS logger provides a set of *logXyz* methods where "Xyz" is the priority of the log (e.g., "*logInfo*"). This set of methods is provided for backward incompatibility reasons and also to automatically extract the name of the calling function, line where the log method was invoked, etc. For more information of functionality provided by the ACS Python logging API, please see the pydoc for the *Acspy.Common.Log* module.

1.1.3.1 Short Logging Example

The following consists of a trivial Python logging example. The *acspyexmpl* CVS module is literally loaded with logging useage(s) and I would highly recommend that you look there or within the pydoc for *Acspy.Common.Log* for far more comprehensive examples:

```
from Acspy.Common.Log import getLogger
logger = getLogger(“my little logger”)
logger.logTrace(“publishes logs of low priority”)
logger.logInfo(“publishes logs of normal priority with extra stuff:” + str(7))

import logging
logger.log(logging.ERROR, “and can even publish logs using native Python logging semantics”)
```

1.1.3.2 Specifying an Audience, Array and/or Antenna for a log

- API:
  - New method in Acspy.Common.Log.Logger:

```
logNotSoTypeSafe(self, priority, msg, audience=None, array=None, antenna=None);
```

- Example:

```
from Acspy.Common.Log import getLogger
import ACSLog
from Acspy.Clients.SimpleClient import PySimpleClient
import logging
from log_audience import OPERATOR
from log_audience import NO_AUDIENCE

simpleClient = PySimpleClient()

logger = getLogger("TestAudience")
logger.log(logging.WARNING, "Normal log")
logger.logNotSoTypeSafe(ACSLog.ACS_LOG_WARNING, "Log with audience, array and antenna",
OPERATOR, "Array01", "Antenna01")
logger.logNotSoTypeSafe(ACSLog.ACS_LOG_WARNING, "Log with audience", OPERATOR)
logger.logNotSoTypeSafe(ACSLog.ACS_LOG_WARNING, "Log with array and antenna",
NO_AUDIENCE, "Array01", "Antenna01")

simpleClient.disconnect()
```

3.7 Type Safe Logs

Type safe logs work on top of the free format logs defined above in the previous sections. These are logs with a formalized structure and contents, mainly thought for the implementation of operational logs of direct interest for the operator of the system, while free format logs are used for post-mortem analysis and debugging. However, this does not exclude that type-safe logs can be used for lower level logs, when deemed convenient.

The module loggingtsTypes contains standard log definitions used over the project. Before you create your own definition, check this module to see if it isn't already there.

Here follows the XML schema file, that type safe log definitions must comply to. It is located in the loggingts module, and is called ACSLogTS.xsd.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
  xmlns:loggingts="Alma/ACSLogTS"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:common="urn:schemas-cosylab-com:COMMONTYPES:1.0"
  targetNamespace="Alma/ACSLogTS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:import namespace="urn:schemas-cosylab-com:COMMONTYPES:1.0"
schemaLocation="commontypes.xsd"/>
  <xs:element name="LogDefinitionType">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="LogDefinition" maxOccurs="unbounded">
          <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
              <xs:element name="Member" type="common:Member_"/>
            </xs:choice>
            <xs:attribute name="logName" type="common:nameType" use="required"/>
            <xs:attribute name="shortDescription" type="common:shortDescriptionString"
use="required"/>
            <xs:attribute name="description" type="common:nonEmptyString" use="required"/>
            <xs:attribute name="URL" type="xs:string" use="optional"/>
            <xs:attribute name="priority" type="loggingts:priorityType" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:attribute name="name" type="common:nameType" use="required"/>
      <xs:attribute name="type" type="loggingts:logType" use="required"/>
      <xs:attribute name="shortDescription" type="common:shortDescriptionString"
use="optional"/>
      <xs:attribute name="description" type="common:nonEmptyString" use="optional"/>
      <xs:attribute name="URL" type="xs:string" use="optional"/>
      <xs:attribute name="_prefix" type="common:prefixType" default="alma"/>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="logType">
    <xs:restriction base="xs:nonNegativeInteger"/>
  </xs:simpleType>
  <xs:simpleType name="priorityType">
    <xs:restriction base="common:nonEmptyString">
      <xs:enumeration value="TRACE"/>
      <xs:enumeration value="DEBUG"/>
      <xs:enumeration value="INFO"/>
      <xs:enumeration value="NOTICE"/>
      <xs:enumeration value="WARNING"/>
      <xs:enumeration value="ERROR"/>
      <xs:enumeration value="CRITICAL"/>
      <xs:enumeration value="ALERT"/>
      <xs:enumeration value="EMERGENCY"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

An example log definition file is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<LogDefinitionType
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="Alma/ACSLogTS"
        name="SampleLog"
        type="10"
        shortDescription="Example LogDefinitionType"
        description="Longer description of the sample LogDefinitionType"
        URL="http://someurl.org"
        _prefix="alma">

        <LogDefinition
                logName="logWithMembers"
                shortDescription="Log with members"
                description="This is a somewhat more complex log using members"
                URL="http://someurl.org"
                priority="WARNING">
                <Member name="someMember"
                        type="string"
                        description="member description"/>
        </LogDefinition>
</LogDefinitionType>
```

After you have defined your logs, you must include them in your module's Makefile, adding the following line(let's assume the file is called SampleLog.xml):

```
ACSLOGTSDEF=SampleLog
```

This will produce the code necessary to use the logs from C++, Java and Python.

Example C++ usage(to compile this remember to add -lSampleLog to the LDFLAGS):

```
#include <SampleLog.h>

...

SampleLog::logWithMembers my_log(__FILE__,__LINE__,"this_function_name");
my_log.setsomeMember(“this is the value of someMember”);
my_log.log();

...
```

Example Java usage:

```
import alma.SampleLog.*

...

// m_logger is the logger of this class
// i.e. If you are in the main of a class that extends ComponentClient
// you should use this.m_logger
logWithMembers my_log=new logWithMembers(m_logger);
my_log.setsomeMember(“this is the value of someMember”);
my_log.log();

...
```

Example Python usage:

```
from SampleLog import logWithMembers

...

my_log=logWithMembers()
my_log.setsomeMember(“this is the value of someMember”)
my_log.log()

...
```

4 Archiving Architecture

Archiving combines monitors with logging service. Monitors allow triggering of callbacks whenever a certain condition in the system is met, e.g., a timeout expires or a value changes. Logging allows for reporting the value in question to a centralized archive.

Archiving is one of the services specific to control systems, which must already be aware of parameters, monitors and other BACI concepts. Archiving infrastructure is therefore a responsibility of BACI.

4.1 The Architecture

Archiving subsystem leverages:

- The logging subsystem for submitting the data that needs to be archived.
- BACI monitors for grabbing the data when required by the per-parameter archiving policy.

1.1.4 Parameter’s Archiving Configuration

Along with other parameter characteristics (minimum and maximum value, unit of measurements, …) these configuration parameters determine the parameter’s archiving policy:

`archive_priority` (unsigned 32-bit integer): The priority of the log entry that will carry the information required for archiving the parameter’s value. Default is 3 (`LM_INFO`). If the priority exceeds the value specified in the logging proxy’s `MaxCachePriority`, the archiving data will be transmitted to the centralized logger immediately. If it is below `MinCachePriority`, the data will be ignored. If it is somewhere in-between, it will be cached locally until a sufficient amount of log entries is collected for transmission to the centralized logger.

`archive_max_int`(double): The maximum amount of time (in seconds and fractions of seconds) allowed to pass between two consecutive submissions to the log. If the time exceeds the value specified here, the log entry should be generated even though the value of the parameter has not changed sufficiently.

`archive_min_int`(double): The minimum amount of time (in seconds and fractions of seconds) allowed to pass between two consecutive submissions to the log. If the time is smaller than the value specified here, the log entry is not submitted, even though the value of the parameter has changed.

`archive_delta`(same type as parameter): Defines what a change in parameter value is. If the value changes for less than the amount specified here, no log entry is generated.

`For more details see the ACS Online documentation for CDB Schema files [RD08].`

1.1.5 Extending XML Schema for Archiving

To accommodate archiving, a new log entry type is introduced to supplement the ones provided by logging itself (`<Trace>`, `<Info>`, …):

```
<Archive TimeStamp="yyyy-MM-ddThh:mm:ss.fff"
         Object="objectID" Parameter="paramID" Type="type"
         Priority="4">
    value
</Archive>
```

The `Object` attribute uniquely identifies the distributed object whose parameter's value is being archived, and `Parameter` attribute identifies the parameter within that object. The `Type` specifies the parameter's type. Possible values of `Type` are:

- `long` (also used for bit-patterns and enumerations)
- `double` (double-precision floating point values)
- `string`

The content of the element (*value*) contains the stringified representation of the current value of the parameter. Care must be taken to use XML CDATA sections if this is an arbitrary string.

1.1.6 Submitting the Archive Data

The following piece of code submits a piece of data for archiving:

```
// macro defined by BACI
#define ACS_ARCHIVE(device, param, type, value)       \
  {                                                   \
    LoggingProxy::LogEntryType("Archive");            \
    LoggingProxy::AddAttribute("Device", device);     \
    LoggingProxy::AddAttribute("Parameter", param);   \
    LoggingProxy::AddAttribute("Type", type);         \
    ACS_LOG(0, 0, (LM_NOTICE, value));                \
  }

ACS_ARCHIVE("Voltage", "double", 13.6);
```

The point where the data is obtained and where this code would be placed is discussed in the chapter "Adjusting BACI" below.

1.1.7 Archiving in the Centralized Logger

The Centralized Logger sends all `<Archive>` elements to a special ***archiving notification channel***. The reference to this notification channel can be found in the name resolution service under the name `ArchivingChannel`.

The log entries sent to the notification channel are structured events with the following properties ([6], section 2.2):

	Name	**Value**
Event Header	`domain_name`	Archiving
	`type_name`	Type of the parameter (see)
Filterable data	`time_stamp`	The time when the parameter had this value, i.e. when the log entry was generated.
	`object`	The object whose properties value is being reported.
	`parameter`	The name of the parameter within the object.
	`value`	The value of the parameter. The type of this field is the same as specified in `type_name`.

1.1.8 Archiving architecture inside BACI

Since BACI already implements monitoring mechanism, only one more monitor per property has to be set up to grab the property's data. The best place to construct this monitor (the ***archiving monitor***) is at the property's construction time. Here, BACI looks up the `archive_*` entries in the configuration database, and if at least one of `archive_min_int`, `archive_max_int` or `archive_delta` is defined, a monitor callback is constructed and registered with the property using this configuration information.

The monitor callback does nothing else but use the `ACS_ARCHIVE` macro as shown above to submit archive data to the logging proxy. The logging proxy then takes care of passing this data (either directly or via local cache) to the centralized logger, which is especially adjusted to handle `<Archive>` elements.

The monitor callback is constructed in-process relative to the property object. If TAO's optimization is used, only one virtual function call will be required from the property's monitor dispatcher to reach the archiving code in the monitor callback, making archiving a very efficient operation.

Appendix A: Logging XML Schema Definition

The documentation for the complete Logging XML schema **loggingMI.xsd** has been removed from this document and is available online at

http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACS_docs/schemas/index.html

under the **urn:schemas-cosylab-com:logging:1.0** Namespace

The original schema file is archived together with the main ACS logging module in **ACS\LGPL\CommonSoftware\logging\ws\idl\loggingMI.xsd**