Construction Project

ACADA – Array Element Logging ICD

Prepared by	Igor Oya Vallejo Digitally signed by Igor Oya Vallejo Date: 2022.12.22 14:37:12 +01'00'	2022-12-12
	I. Oya, ACADA WP Coordinator	Date
Prepared by	Alessandro Costa Firmato digitalmente da Alessandro Costa Data: 2023.01.09 11:19:34 +01'00'	2022-12-12
	A. Costa, ACADA-MON Coordinator	Date
Approved by	Digitally signed by Nick Whyborn Date: 2022.12.22 14:31:44 +01'00'	
	N. Whyborn, Lead Systems Engineer	Date
	Igor Oya Vallejo Digitally signed by Igor Oya Vallejo Date: 2022.12.22 14:37:28 +01'00'	
	I. Oya, ACADA WP Coordinator	Date
Released by	Wolfgang Wild Digitally signed by Wolfgang Wild Date: 2023.01.10 16:52:01 +01'00'	
	W. Wild, Project Manager	Date

Change Log

Issue	Revision	Date	Section/Page affected	Reason/ Remarks / Initiation Documents
1	a	2021-06-02	All	New document.
1	b	2022-12-12	Pages 10-13	CTA-CRE-SEI-000000-0005

List of Contributors

Name	Organization	Contribution
Igor Oya	CTA PO	The document and current version of the interface
Alessandro Costa	INAF	Logging Data Model; Revision and OPC UA configuration
Kevin Munari	INAF	ACS Logging API
Chiara Montanari	CTA PO	Interface overall process management
Matthieu Heller & LST team	LST	Feedback on the document
Gianluca Giavitto	SST	Feedback on the document

List of ICD actors

Actor	Role	Agreed date and signature
I. Oya	ACADA Coordinator	
D. Mazin	LST	
U. Schwanke	MST Struct	
P. Sizun	NectarCam	

M. Barcelo	FlashCam	
G. Pareschi	SST	
M. Gaug	Calibration Coordinator	

Table of Contents

List of Acronyms

ACADA	Array Control and Data Acquisition
ACE	Adaptive Communication Environmen (Not to be confused with the other usage in CTA of the ACE term for Array Comon Elements).
ACS	Alma Common Software
API	Application Programming Interface
CDB	Configuration Database
CTA	Cherenkov Telescope Array
DPPS	Data Processing and Preservation System
HMI	Human-Machine Interface
ICD	Interface Control Document
TOSS	Technical Operations Support System

1 Scope

This document specifies the requirements of interface describing the logging operations of the Array Control and Data Acquisition (ACADA) System on any CTA Array Element.

Interface management is a process to assist in controlling product development when efforts are divided amongst different parties (e.g. agencies, contractors, geographically dispersed technical teams).

This ICD describes the Logging data exchange of the interface between the Array Control and Data Acquisition (ACADA) System (the target system) and a software application contained in a generic CTA Array Element (the source system). The purpose of the ICD is to define the design of the interface(s) ensuring compatibility among involved interface ends by specifying form, fit, and function.

The ICD is managed by the CTAO Interface Manager (or their delegates) and represents an agreement between the relevant actors. The actors in this ICD are shown at the beginning of this document.

The ICD is used:

1. to document the interface definition,
2. to control the evolution of the interface,
3. to document the design solutions to be adhered to, for a particular interface,
4. as one of the means to ensure that the supplier design (and subsequent implementation) are consistent with the interface requirements,
5. as one of the means to ensure that the designs (and subsequent implementation) of the participating interface ends are compatible.

This Interface Control Document (ICD) documents and tracks the necessary information required to effectively define the interface between an Array Element and ACADA for the software logging information flow from the former to the latter, as well as any rules for

communicating with them in order to give the development team guidance on the architecture of the system to be developed.

The purpose of this ICD is to clearly communicate all possible inputs and outputs from the system for all potential actions whether they are internal to the system or transparent to system users.

Its intended audience is the Systems Engineering personnel, ACADA and Array Element development teams, and stakeholders interested in the interfacing of these systems.

2 Applicable and Reference Documents

2.1 Applicable Documents

- AD1: Common On-site Requirements in Jama
- AD2: Requirement Specification for Array Control and Data Acquisition System. Doc. No. CTA-SPE-COM-303000-0001, Issue 2, Rev. h, 2020-04-29
- AD3: Requirement Specification for Logging and Monitoring System. Doc. No. CTA-SPE-COM-303000-0009, Issue 1, Rev. h, 20-07-2020

2.2 Reference Documents

- RD1: Top-level Data Model, Doc. No. CTA-SPE-OSO-000000-0001, Issue 1, Rev. b, 2020-04-30.
- RD2: ACS Logging and Archiving. KGB-SPE-01/04. Revision: 1.36. 30-07-2007
- RD3: OPC UA Logging Infrastructure: OPC 10000-4: OPC Unified Architecture: https://reference.opcfoundation.org/v104/Core/docs/Part4/
- RD4: ACS Logging and Archiving Manual. https://confluence.alma.cl/display/ICTACS/Logging+and+archiving+Manual

3 Interface Requirement specification

3.1 Overview

CTA Array Elements (Telescopes, FRAMs, LIDARS) are composed of hardware and application software elements. Software components generate *software logs* that record events taking place in the execution the program in order to provide a recorded trail that can be used to understand the activity of the system and to diagnose (usually "post mortem") problems. They are essential to understand the activities of complex systems, particularly in the case of automatized applications with little user interaction.

The software components inside the Array Elements, as well as ACADA, generate logs through their functioning, and that are archived for the usage of the software maintenance crew. ACADA provides an infrastructure for the short-term archival of those software logs. Long term preservation and management of logs is done elsewhere (DPPS and/or TOSS, TBD).

This ICD specifies the interface requirements the ACADA and any Array Element System must meet.

This interface is a software interface, and thus it describes:

- the concept of operations for the interface,
- defines the message structure and protocols that govern the interchange of data,
- and identifies the communication paths along which the project team expects data to flow.

3.2 Assumptions

Any software log that is required to be stored by the CTA central services (ACADA, DPPS, TOSS) is transmitted via this interface.

The array Element Software components have access to the On-site Data Centre and have permission to write files in the area reserved to storage logs.

3.2.1 Constraints

No constraint was identified.

3.2.2 Functional Allocation

This interface implements the following functions:

- Gather and store logging data from the Array Elements for further handling within ACADA and other CTAO computing systems.

3.2.3 Extension of the interface

We do not foresee any need to extend this interface.

3.2.4 Data Transfer

ACADA provides a standardized infrastructure for software logging data sources within the array element control systems. This ensures the required modularity, flexibility, and performance since logging is in general a 24/7 activity that shall occur any time the corresponding software process is running.

The logging information is transferred from the data source components to ACADA in three ways:

- *Alma Common Software (ACS):* ACS components use the standard ACS logging mechanism to insert logs as described in [RD2]. ACADA has a mechanism to gather these logs, filter them and store them.
- *OPC UA*: OPC UA servers use the OPC UA logging infrastructure [RD3]. Logs are stored in ACADA by placing log entries in files which are stored in a folder structure managed by ACADA
- Low-level software logs: Low-level software processes, which are not using ACS or OPC UA, store logs in the same way as OPC UA logs.

Irrespective of the logging mechanism used, the volume of the logs produced by the software components are to be configured properly in order to avoid generating excessive logging information (see ACADA-AE-LOG-I-015) during regular operations, but also reconfigured to provide enough information during debugging campaigns. This is achieved by means of using filtering based on logging levels.

Figure 1 illustrates how ACADA and an Array Element would implement the logging data handling process. For illustration purposes, the Array Element in this example has all kind of software components supported by this interface.

It is worth mentioning that logs produced by ACS and OPC UA servers and transferred via the standard logging mechanism can be displayed to the Operator via the ACADA HMI. Low-level logs can only be inspected by accessing directly into the logging archive and cannot be accessed from the ACADA HMI.

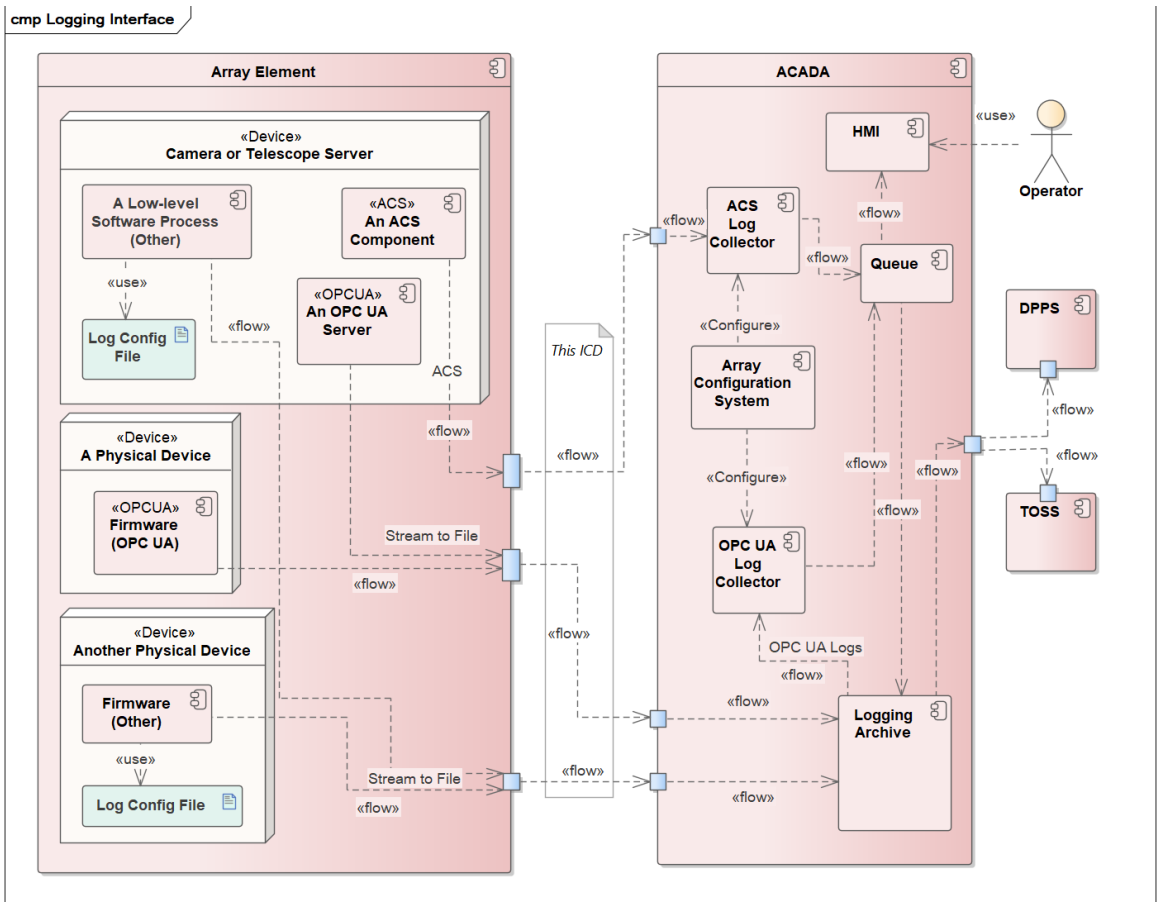

Figure 1: Diagram illustrating the logging data flow from an Array Element to ACADA, illustrating the different logical elements and protocols described in this ICD. This Array Element is an academic example that illustrates all possible variations of software elements produces of logs: i) A physical device 1 includes an OPC UA-based firmware such as a PLC; ii) Another Physical Device using a low-level software process that is not OPC UA nor ACS based; iii) An OPC UA server deployed in a Telescope or Camera Server, iv) An ACS component deployed in a Telescope or Camera Server, and v) a "custom" software process firmware that is not OPC UA nor ACS based.

3.2.5 Security and Integrity

These aspects are not addressed in this document yet.

3.3 Interface specification

This section specifies all elements of the ACADA to Array Element Logging interface.

3.3.1 Transactions

ACADA-AE-LOG-I-010. Log interfacing. Array Elements shall interface ACADA to record software logs.

ACADA-AE-LOG-I-015. Maximum Logging Throughput. An array element shall not produce more than 1 MB/s logging data volume at any given time, averaged over a time of a second.

Notes:

- This is the volume of data as produced, i.e., does not consider any kind of compression or later filtering in ACADA.
- It comprises all logs produced by all applications belonging to an Array Element. In the case of a Telescope, it is all logs produced by the applications associated to the Telescope and the Structure. Such a limit per array element allows to be compliant with C-ACADA-LOG-015 when considering all array elements and ACADA internal logs.

ACADA-AE-LOG-I-020. ACS Logging. Array Elements shall use the standard ACS logger API [RD2] corresponding to the used programming language for inserting ACS logs into the ACADA system.

Notes:

- ACS supports C++, Java and Python programming languages. See Appendix II for a description of the mapping with the existing ACS Logging API.
- Whenever using ACS components, the only logging mechanism permitted is the ACS-based logging.

ACADA-AE-LOG-I-030. OPC UA Logging. Array Elements shall write log messages to file. The logs files are then consumed by the ACADA Centralized Logging System.

Note: See Appendix II for a description supported OPC UA Logging API.

ACADA-AE-LOG-I-040. Low-level Software Logging. Array Elements shall insert low-level logs by piping log data into files located in the On-site Data Centre.

ACADA-AE-LOG-I-050. Logging File Standards. Log files used for OPC UA and low-level software logging shall be plain ASCII files UTF-8 encoding with extension ".log". In addition, the scheme presented below shall be used, depending on the used application programming language.

C++:

The file name shall follow the naming scheme:

componentInstanceName_<YYYY>-<MM>-<DD>_<HH>-<mm>-<SS>.log

for the currently active log file and for files being generated earlier, where componentInstanceName

is the name of the instance of the component creating the logs, <YYYY>-<MM>-<DD>_<HH>-<mm>-<SS> is the UTC date corresponding to the moment when the file is opened.

Note: This means that the last log of the night will be named, for example (CTAO North):

componentInstanceName_2022-10-10_07-55-23.log

And the others:

componentInstanceName_2022-10-09_08-00-00.log
…
componentInstanceName_2022-10-09_23-08-07.log
…

Java.

The file name shall follow the naming scheme:

componentInstanceName_<YYYY>-<MM>-<DD>.<num>.log

for the currently active log file and for files being generated earlier, where componentInstanceName is the name of the instance of the component creating the logs, <YYYY>-<MM>-<DD> is the UTC date corresponding to the moment when the file is opened, and <num> being integer number starting at 1 for the first log file created during the day and increasing monotonically for additional log files created the same day.

Note: This means that the last log of the night will be named:

componentInstanceName_<YYYY>-<MM>-<DD>.<i+1>.log

And the others:

componentInstanceName_<YYYY>-<MM>-<DD>.1.log
componentInstanceName_<YYYY>-<MM>-<DD>.2.log
componentInstanceName_<YYYY>-<MM>-<DD>.3.log
…
componentInstanceName_<YYYY>-<MM>-<DD>.<i>.log

Python.

The file name shall follow the naming scheme:

componentInstanceName_<YYYY>-<MM>-<DD>.log ,

for the currently active log file, and:

componentInstanceName_<YYYY>-<MM>-<DD>.<num>.log

for files being generated earlier, where componentInstanceName is the name of the instance of the component creating the logs, <YYYY>-<MM>-<DD> is the UTC date corresponding to the moment when the file is opened, and <num> being integer number starting at 1 for the first

log file created during the day and increasing monotonically for additional log files created the same day.

Note: This means that the last log of the night will be named:

componentInstanceName_<YYYY>-<MM>-<DD>.log

And the others:

componentInstanceName_<YYYY>-<MM>-<DD>.1.log
componentInstanceName_<YYYY>-<MM>-<DD>.2.log
componentInstanceName_<YYYY>-<MM>-<DD>.3.log
…

Any other programming language or logging data source.

The file name shall follow one of the naming schemes described above.

ACADA-AE-LOG-I-056. CTAO North and CTAO South: Logging Rollover. Log files, and their directories (when applicable), shall be rotated every day at 8:00 AM UTC at CTAO North, and 1:00 PM UTC at CTAO South.

ACADA-AE-LOG-I-060. Logging File Content. A software logging file shall contain one line per new log entry. A log entry shall follow exactly this structure:

<sourceTimestamp> <loggingLevel> <file> <line> <routine> <sourceObject> <logAudience> <Message>

ACADA-AE-LOG-I-065. Logging File Timestamps. Log entries shall use UTC timestamps to specify the instant when the log-file was generated, following the use ISOT (ISO 8601) time format:

%Y-%m-%dT%H:%M:%S.000 ,

where Y is year, is month, d is day, H is hour, M is minute, S is second.

Note: an example of a valid timestamp is “2021-02-05T12:49:50.250”.

ACADA-AE-LOG-I-066. Logging File Size. A software logging file size shall not exceed 20 MB. Once this file size limit is exceeded, the file shall be rotated.

ACADA-AE-LOG-I-070. Low-level Software Logging File Location. OPC UA and low-level Log files shall be stored by the Array Element software processes in the following folder structure, depending on the used application programming language:

C++ and Java.

/DATA/R1/<ARRAYELEMENT>/<SUBSYSTEM>/logs/

Where:

- <ARRAYELEMENT> is the official name of the array element instance (e.g. LSTN-01)
- <SUBSYSTEM> is the name of the subsystem producing the logs (e.g. camera)

Python.

/DATA/R1/<ARRAYELEMENT>/<SUBSYSTEM>/logs/<YYYY>/<MM>/<DD>/

Where:

- <ARRAYELEMENT> is the official name of the array element instance (e.g. LSTN-01)
- <SUBSYSTEM> is the name of the subsystem producing the logs (e.g. camera)
- <YYYY>/<MM>/<DD> is the UTC date,

changing every day at 8:00 AM UTC at CTAO North, and 1:00 PM UTC at CTAO South.

Any other programming language or logging data source.

Log files shall be stored in one of the folder structures described above.

Note:

In the On-Site Data Centre in La Palma ACADA-AE-LOG-I-060 and ACADA-AE-LOG-I-070 can be realized, for example for LSTN-01(LST1), for Python, as[1]:

/fefs/onsite/data/R1/LSTN-01/monitoring/camera/logs/2020/09/03/controler1_2020-09-03.1.log

controler1_2020-09-03.2.log

…

Controler1_2020-09-03.34.log

…

controler2_2020-09-03.1.log

And for the FRAM into:

/fefs/onsite/data/R1/FRAM/monitoring/dome/logs/2020/09/03/domecontroller_2020-09-03.1.log

…

domecontroller_2020-09-03.16.log

ACADA-AE-LOG-I-075. Offline logging data source. For servers with no access to the computer cluster, log files shall be compliant with the current document and transferred to a shared directory once per hour.

[1] The component names in the example log files are illustrative and do not correspond to real component names.

3.3.2 Logging Data Model and Format

This section specifies the Logging data model and format relevant for this ICD. This data category is classified as R1/MON in the CTA data model [RD1].

ACADA-AE-LOG-I-080. Log data model. Any log inserted by the Array Elements into ACADA must respect the data model specified in Figure 2 and Table 1. In case an optional value is not present, it shall be replaced by the character "-".

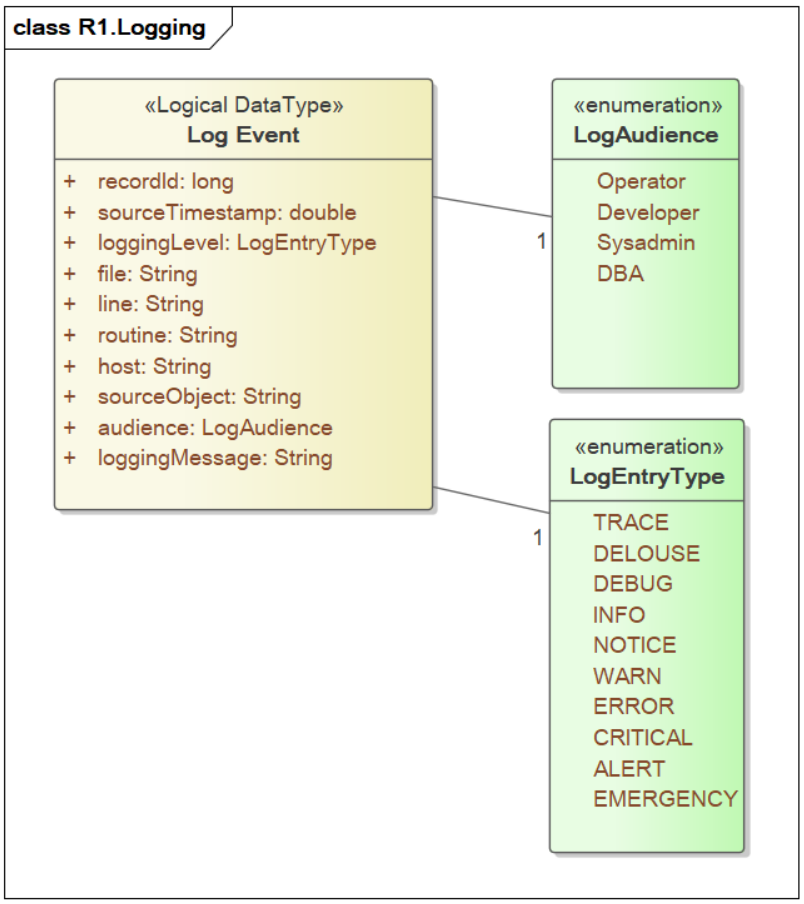

Figure 2: R1/Logging Data Model. Note that recordId is a unique identifier generated by ACADA after the log is ingested, and therefore not part of this ICD.

Table 1: R1/Logging Data Model Description

Name	Type	Description
	LogEvent	
sourceTimestamp	String/double	UTC timestamp in the format of %Y-%m-%dT %H:%M:%S.000, where Y is year, is month, d is day, H is hour, M is minute, S is second. For example: "2021-02-05T12:49:50.250". Internally represented as a double. (1)
loggingLevel	LogEntryType	The level of the log entry (see LogEntryType description).

file *(optional)*	String	The identification of the source file e.g. "/home/vagrant/ACS/motor/controller.cpp".
line *(optional)*	String	The line number in the source code where the log entry was submitted.
routine *(optional)*	String	name of the subroutine (function) where the log entry was submitted from, e.g. Activator::Init or just init()."
sourceObject	String	Name of the process from which the log entry is generated.
audience	LogAudience	The audience of this log
message	String	The message of the log (2)
LogEntryType		
TRACE	ENUM value	Trace logs are generated whenever a function is entered. And are used to report calls to a function. These are the lowest level logs, normally using during development to track bugs, but excluded from final products.
DELOUSE	ENUM value	The highest level of detail for debugging the system.
DEBUG	ENUM value	Debug logs are used only while debugging the system.
INFO	ENUM value	Info log level is used to publish information of interest during the normal operation of the system.
NOTICE	ENUM value	Notice logs are useful for logging normal, but significant activity of the system, for example startup or shutdown of individual services. They denote important situations in the system, but not necessarily error/fault conditions.
WARN	ENUM value	Warning logs are used to report to conditions that are not errors but that could lead to errors/problems
ERROR	ENUM value	Error logs denote error conditions.
CRITICAL	ENUM value	Critical logs denote an Alarm condition that shall be reported to operators through HMI
ALERT	ENUM value	Alert logs denote an Alarm condition that shall be reported to operators through HMI. This denotes a problem more important than Critical
EMERGENCY	ENUM value	Emergency logs denote an Alarm condition of the highest priority
Log Audience		
Operator	ENUM value	Operator of the array, using the HMI
Developer	ENUM value	The developer or maintainer of the module generating this log
Sysadmin	ENUM value	The on-site ICT administrator,
DBA	ENUM value	DBA is used for logs to database administrators
Notes: (1) Internally, ACADA will store the timestamps in TAI seconds since 1970-01-01T00:00:00.0. (2) The upper limit for each "message" text is 100 KByte.		

3.3.3 Logging Message Standards

ACADA-AE-LOG-I-090 Log Message Language. Any log message registered in ACADA by the Array Elements shall be written in English.

ACADA-AE-LOG-I-100 Meaningful Log Messages. Any log message registered in ACADA by the Array Elements shall include meaningful information which is relevant for the log audience.

Note: The logging guidelines presented in the Appendix III provide information for the personnel responsible of implementing the log messages.

3.3.4 Logging Configuration Settings

ACADA-AE-LOG-I-110 ACS software modules logging configuration. The Array Configuration System of ACADA shall contains parameter settings of the logging level-based filtering and storage that is handled via ACS.

ACADA-AE-LOG-I-120 OPC UA software modules logging configuration. The Array Configuration System of ACADA shall contains parameter settings of the logging level-based filtering and storage that is handled via OPC UA.

ACADA-AE-LOG-I-130 Low-level software modules logging configuration. Every software module generating low-level logs shall include a mandatory configuration item in the configuration file of the low-level software, or alternatively, a standalone configuration file. Such configuration item or file shall permit any maintainer to change the logging level filtering without needing to change the source code or recompilation.

Note: For standalone configuration files, the usage of JSON is recommended.

4 Appendix I: Level B and C requirements applicable to this interface

This interface addresses the following requirements, repeated here from Jama [AD1], the ACADA Level-B Requirement Specification document [AD2], and the Level-C Requirement Specification document for the ACADA Monitoring and Logging Systems [AD3] for convenience:

- **B-ONSITE-0830 Logging**. The actions of the System must be logged via OES. Logging levels must be configurable and follow the defined standards.
- **B-ACADA-2240 Logging.** ACADA shall provide automatic logging of all Errors, Warnings, Actions and Alarms.
- **B-ACADA-2245 Log Preservation.** ACADA shall pass all logging information down to a configurable level to the TOSS for long term preservation and access for engineering purposes.
- **C-ACADA-MON-LOG-010 Logging Collector.** LOG shall allow the collection of the following log entries:
    - Any aggregate log produced by SW elements;
    - Software logs of the observation scripts.
- **C-ACADA-MON-LOG-015 Logging Collector max data rate.** The maximum data rate in input for the Logging Collector shall be:1000 Mbps
- **C-ACADA-MON-LOG-050 Logging Prioritization**. LOG [ACADA logging system] shall categorize log entries in term of priority. This shall be performed using a Log Entry Level attribute.
- **C-ACADA-MON-LOG-060 Log Entry Level**. LOG [ACADA logging system] shall categorize log entries in terms of relevance:
    - Trace: Trace logs are generated whenever a function is entered. And are used to report calls to a function.
    - Debug: Debug logs are used only while debugging the system.
    - Info: Info log level is used to publish information of interest during the normal operation of the system.
    - Notice: Notice logs are useful for logging normal, but significant activity of the system, for example startup or shutdown of individual services. They denote important situations in the system, but not necessarily error/fault conditions.
    - Warning: Warning logs are used to report to conditions that are not errors but that could lead to errors/problems.
    - Error: Error logs denote error conditions.
    - Critical: Critical logs denote an Alarm condition that shall be reported to operators through HMI
    - Alert: Alert logs denote an Alarm condition that shall be reported to operators through HMI. This denotes a problem more important than Critical.
    - Emergency: Emergency logs denote an Alarm condition of the highest

priority

- **C-ACADA-MON-LOG-065**. **LOG data Timestamp.** MON shall be able to associate the timestamp to logging information. The timestamp specifies the exact time when the information was produced. The time is encoded in ISO 8601 format with a precision to one millisecond. The time is specified in UTC, with the following format:

  %Y-%m-%dT%H:%M:%S.000 ,

  where Y is year, is month, d is day, H is hour, M is minute, S is second.

5 Appendix II: Logging Data Model to Software Infrastructure API mapping.

5.1 ACS C++ logging API [RD4]

5.1.1 Adaptive Communication Environment ACE) Logging

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

- 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. 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 cannot be set separately, because type implies priority, and vice-versa.

- 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.

- ACE's logging mechanism is extensible, allowing for custom callbacks to be registered with an ACE_Log_Msg object. These callbacks (implementations of an 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.

- 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).

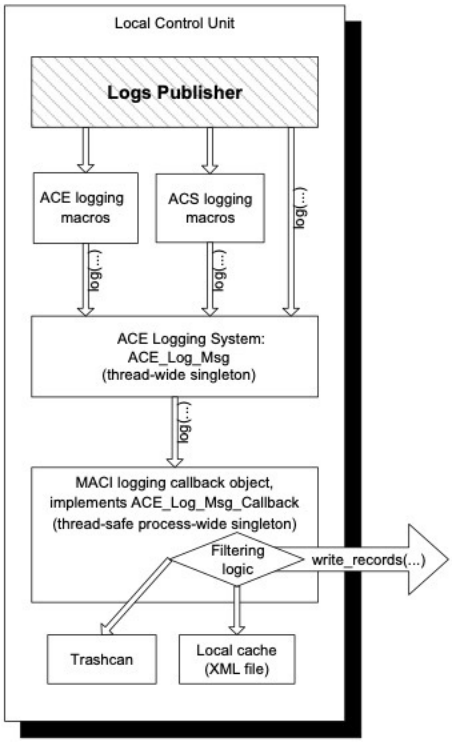

Figure 3: Architecture of the ACE Logging framework. The figure gives an overview of ACE Logging framework. The ACE Logging System gets log entries that can be generic or specific (using ACS logging macros). It submits them to an object implementing ACE_Log_Msg_Callback that provides the filtering and 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 the 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.

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.

5.1.2 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.

5.1.2.1 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>
```

5.1.2.2 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>
```

5.1.2.3 Submitting a Variables'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.

5.1.2.4 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...
```

5.1.2.5 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.

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

Note: This section describes what ACS provides as part of its API, but we want to note here that in CTA, for the time being, we do not use "Array" and "Antenna" identifier for logs. Nevertheless, the "Antenna" field could be used as a holder for CTA Array Elements unique identifier)

5.1.3.1 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, log-
Array, logAntenna)

#define LOG_With_ANTENNA_CONTEXT(logPriority, logRoutine, logMessage,
logArray, logAntenna)

#define LOG_TO_AUDIENCE(logPriority, logRoutine, logMessage, logAudience)
```

5.1.3.2 Example

```
#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,"array0
1","Antenna01");

     LOG_WITH_ANTENNA_CONTEXT(LM_WARNING,"main","LOG_WITH_ANTENNA_CONTEXT
","array01","Antenna01");

     LOG_TO_AUDIENCE(LM_WARNING,"main","LOG_TO_AUDIENCE",log_audience::OP
ERATOR);

     client.logout();

     return 0;
}
```

5.2 ACS Java logging API [RD4]

5.2.1 JSDK Java Logging API

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

- 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.

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

- 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. For more details, read about the Component-Container model in "Java Component Tutorial". Both loggers and the handlers are organized in a hierarchical namespace so that children may inherit some properties from their parents.

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

- 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.

5.2.2 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.

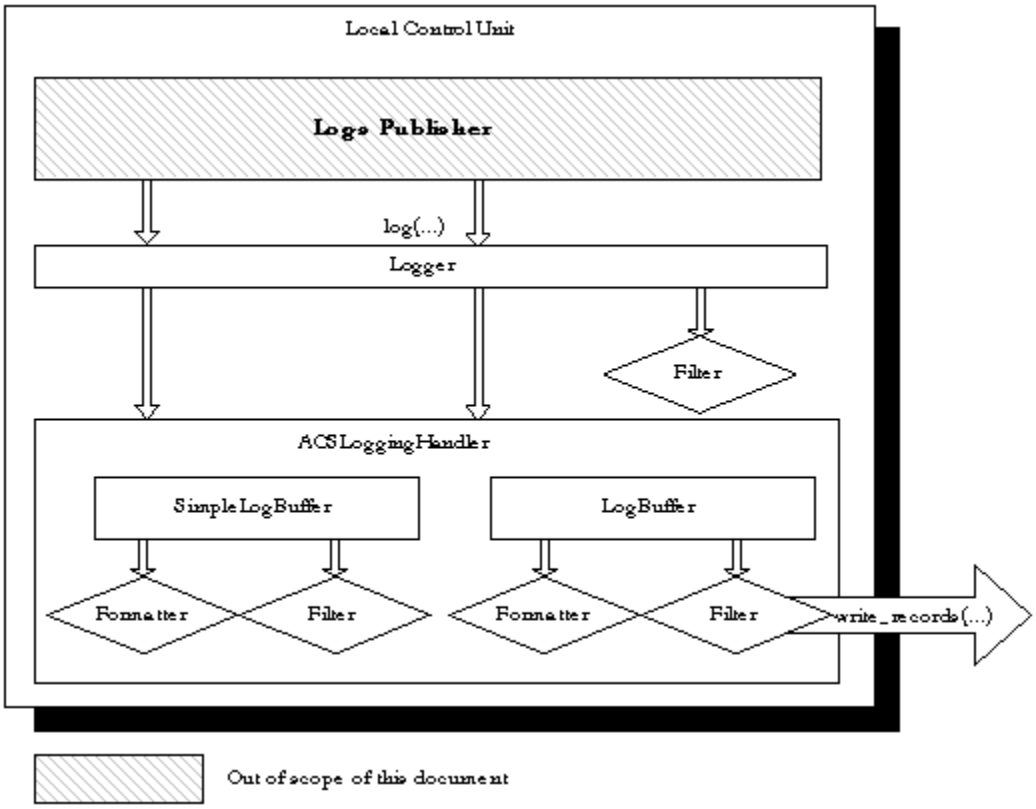

Figure 4: Architecture of the ACS Java Logging framework. The Logger object gets log entries that it submits to a handler that provides the formatting and the caching capabilities of the framework. Filtering is done both at the logger and at the handle.

5.2.3 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().getLoggerForAppli-
cation(clientName, true); // the last parameters enables or disables re-
mote logging
```

- For a component:

```
import java.util.logging.Logger;

import alma.acs.container.ContainerServices;

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

5.2.4 *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.

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

Note: This section describes what ACS provides as part of its API, but we want to note here that in CTA, for the time being, we do not use “Array” and “Antenna” identifier for logs. Nevertheless, the “Antenna” field could be used as a holder for CTA Array Elements unique identifier).

5.2.5.1 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 of 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);
```

5.2.5.2 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().getLog-
ger();

			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 an-
tenna", 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”);
		}
	}

}
```

5.2.6 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:

Table 2: Java log levels.

ACS Level (ACE Level)	**ACS Logging Priority**	**Java API Level**	**Java Logging Priority**
TRACE	12	FINEST (FINER)	3400
DELOUSE	2	FINER	400
DEBUG	3	FINE (CONFIG)	700
INFO	4	INFO	800
NOTICE	5	INFO	801
WARN	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

5.2.7 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.

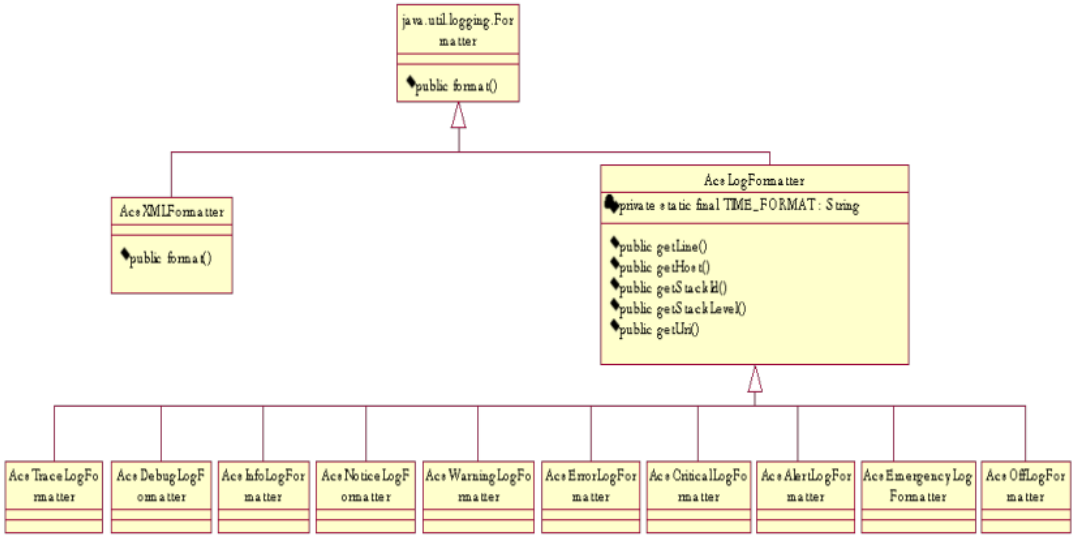

Figure 5: Class Diagram of ACS Formatters. The AcsLogFormatter object defines methods for getting the properties needed for formatting a string into an XML string. The AcsXMLFormatter calls any of the customized formatter.

5.3 ACS Python logging API [RD4]

5.3.1 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.

5.3.2 Short Logging Example

The following consists of a trivial Python logging example. The *acspyexmpl* CVS module is literally loaded with logging usage(s) and it is highly recommend reading it or 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")
```

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

Note: This section describes what ACS provides as part of its API, but we want to note here that in CTA, for the time being, we do not use "Array" and "Antenna" identifier for logs. Nevertheless, the "Antenna" field could be used as a holder for CTA Array Elements unique identifier).

5.3.3.1 API

New method in *Acspy.Common.Log.Logger*:

```
logNotSoTypeSafe(self, priority, msg, audience=None, array=None, an-
tenna=None);
```

5.3.3.2 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 an-
tenna", NO_AUDIENCE, "Array01", "Antenna01")

simpleClient.disconnect()
```

5.4 Mapping of logging model to the OPC UA:

See Section 3.3.1.

5.5 Mapping of logging model to low-level logs

See Section 3.3.1.

6 Appendix III: Logging guidelines

This appendix describes the guidelines to be used when creating the software logs handled by this ICD.

- **Longs should be meaningful by themselves**:
  - Always anticipate that there are emergency situations where the only thing to understand what occurred is the log file.
  - Reading the log itself should provide enough context to the reader.
  - If possible, add remediation information in the log message.
  - Don't add a log message that depends on a previous message's content. (The reason is that those previous messages might not appear if they are logged in a different category or level. Or worse, they can appear in a different place (or way before or also after, in a multi-threaded or asynchronous context.)
- **Don't Log Too Much or Too Little.**
  - Too much logging and it will be hard to get any value from it. Too little logging and we risk to not be able to troubleshoot problems.
  - Unfortunately, there is no magic rule when coding to know what to log – it is important the developers to understand the right balance of the logging information.
- **Every message in code should be an individual.** If one searched for an error message (via e.g. grep) it should not appear everywhere, or even twice.
- **No terminal colours.** Creates a mess and some people cannot see colours**.**
- **Think of Your Audience:** adapt your language to the allocated target audience. Understand that, for example, the product developer knows the internals of the program, thus their log messages can be much more complex than if the log message is to be addressed to the operator.
- **Logging Purposes.**
  - *Troubleshooting:* Most extended usage of logging, allows recording *post-mortem* details of an occurred incident.
  - *Auditing*: Capture significant events that matter to the management, legal, or security people. These are statements that describe usually what users of the system are doing (like who signed-in, who edited that, etc…).
  - *Profiling*: as logs are timestamped to the millisecond level, it can become a good tool to profile sections of a program, for instance by logging the start and end of an operation for later profiling usage.
  - *Statistics*: if you log each time a certain event happens (like a certain kind of error or event) you can compute interesting statistics about the running program. This will allow to eventually hook to the alarm system to detect too many errors in a row.
- **Don't Log Sensitive Information.** Make sure you never log:
  - Passwords and other credentials.
  - Session identifiers Information the user has opted out of.
  - Authorization tokens.
  - Personal Identifiable Information, such as personal names.