A WIDAR Correlator for the Expansion Very Large Array (EVLA):

Technical, Functional, and Performance Discussion Document

EVLA Correlator

Alarm Handling and Logging

NRC-EVLA Memo# 022

Sonja Vrcic, 26 January 2005

ABSTRACT

In the absence of the high-level system design that would define alarm handling and logging in the EVLA System as a whole, there is a need to define a common logging and error handling functionality for all the sub-systems of the EVLA correlator. This memo provides a brief overview of the following: logging and alarm handling in ALMA Common Software, the functionality supported by the current version of low-level EVLA CMIBs software (provided by Bruce Rowen) and Java Logging API.

The ALMA Common Software (ACS) is based on CORBA and, therefore, cannot be used on the EVLA correlator. However, assuming that the higher levels of EVLA software will use the ACS, the correlator should implement a similar, if not the same, logging and alarm handling scheme as ALMA.

Introduction

Every properly built system should be able to log events that occur in it so that the user of the system can gain an accurate understanding of the system's state, especially in the case of failure.

In addition to unusual conditions and failures, the system should log significant events that can be used for troubleshooting and debugging. For example, every component should log when it is created and destroyed.

During debugging, every entering and leaving of an important function could be logged to determine whether the function was called at all and with what parameters. This information is also useful for profiling and system tuning.

Every subsystem should log all received commands, including invalid and rejected commands.

In a distributed system, it is desirable to centralize logging information. A central logging file (repository) enables software tools and operators to examine the sequence of events that led to a certain event or that were caused by a certain event.

Another important issue is consistency of log entries. If all subsystems implement the same formatting rules, tools such as automated log parsers may be built to filter and sort log messages in order to facilitate monitoring and troubleshooting. On the contrary, if each subsystem defines its own formatting rules and priority levels, troubleshooting and maintenance may be very difficult and error prone.

In general, events that are reported / logged can be divided in three categories:

1. **Alarms** – caused by events that require immediate attention, they usually report a failure or a condition that may cause a failure (e.g. overheating of a component).

2. **Errors** – are caused by a badly specified or invalid command. Errors must be reported to the originator of the command and stored in the central log repository. The system administrator / operator may be informed, especially if errors occur repeatedly. The message sent to the originator, as a direct response to the received command, will follow rules defined for the specific interface, and may differ in format and content from the logged message.

3. **Logs** – are informational messages that are generated to keep track of the sequence of events and system state. May be used for debugging, troubleshooting and system tuning.

Ideally, the format and structure of all three categories is the same. The type and priority (severity) indicator may be used to determine the importance of the logged message, so that alarms become a special (high-priority) type of log messages.

ALMA

Requirements for the logging subsystem in ALMA are defined as follows[1]:

1. Logging system must provide an easy to use programming interface to the application developer.

2. Every log has an associated priority and type.

3. Every log is equipped with a timestamp accurate to a deci-millisecond (100's of ns).

4. The logging system 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 the timestamp.

5. The logging system should allow for a filtering, so that logs 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 are generated, and transmitted over the network to the central log on demand or when the local buffer reaches a predetermined size.

All the above listed requirements apply for the EVLA system.

Logging and alarm handling are implemented as a part of the ALMA Common Software (ACS), which is based on the following systems:

1. CORBA Telecom Log Service for centralizing all entries generated throughout the system.

2. CORBA Notification Service for distributing log entries to interested clients when the entries are submitted in the centralized logger.

3. The mechanism for generating, formatting, filtering and coaching log entries:

    - Java suppliers of logs use Java Logging API

    - C++ suppliers of logs use ACE Logging framework.

    - Other suppliers of logs, e.g. a Python application, can use the stand alone ACS Log Server which provides generic functionality.

[1] From the document ALMA Logging and Archiving, Rev 1.3

For alarm handling, ACS will probably adopt LASER Alarm Handler developed for CERN.

ACS Log Type and Priority

The document "ALMA Logging and Archiving" defines log entry types and priorities. **Table 1** below lists the ALMA log entry types. Each log type is assigned a default priority as indicated in the table.

Priority level is as an integer in the range 1 to 16. 0 indicates default priority.

Table 1 ACS Log types and priorities

Type	Default priority level
Trace	2
Debug	3
Info	4
Notice	5
Warning	6
Error	8
Critical	9
Alert	10
Emergency	11

LASER Alarm Handling System

LASER (LHC Alarm Service), an Alarm Handler System developed for CERN, will, most probably, become part of ALMA Common Software (ACS). If EVLA is going to re use the same or similar system, alarms generated by the EVLA correlator should contain information that is used/required by LASER.

LASER expects a fault state to be defined by a triplet:

- Fault Family – a collection of items that exhibit the same problems (system name)
- Fault Member – an instance of Fault Family (identifier)
- Fault Code – problem description. Problems are identified by a number (code) and a corresponding text that describes the problem.

In addition to description and code, LASER Alarm Handler expects a log message to specify a Fault Status, which is defined as follows:

- **Active** – alarm is active; sub-system (element) is in fault state.
- **Terminate** – sub-system (element) has returned in non-fault (normal) state.
- **Instant** – non-binary event (i.e. event where on-off concept is not applicable).
- **Change** – parameters have changed, but sub-system (element) is still in fault state.

Since the fault status may be used by any alarm handling system; the EVLA correlator alarms should indicate fault status.

Java Logging

It seems that large part of EVLA software will be written in Java. The Logging API is part of the standard Java development environment (JSE).

The Java Logging API offers both static and dynamic configuration control:

- Static control enables field service staff to set up a particular configuration and then re-launch the application with the new logging settings.
- Dynamic control allows for updates to the logging configuration within a currently running program. The APIs also allow for logging to be enabled or disabled for different functional areas of the system.

The Level class defines a set of standard logging levels that can be used to control logging output. Enabling logging at a given level also enables logging at all higher levels. Log level objects encapsulate an integer value, with higher values indicating higher priorities. The Level class defines seven standard log levels:

- SEVERE (1000)- serious failure
- WARNING (900)- potential problem
- INFO (800)- informational message
- CONFIG (700)- static configuration messages
- FINE (500)- tracing information
- FINER (400)- a fairly detailed tracing message
- FINEST (300)- a highly detailed tracing message

In addition, there is a level OFF that can be used to turn off logging, and a level ALL that can be used to enable logging of all messages. Values in the parenthesis indicate the integer value assigned to a severity level. A generated log message does not necessarily include the integer value.

An application may define additional logging levels.

Java Logging API provides Logger and Handler objects. Applications make logging calls on Logger objects, which create LogRecords. LogRecord is then passed to Handler object(s) for publication. Both Loggers and Handlers may use logging Levels and Filters to decide if they are interested in a particular LogRecord. A Handler can (optionally) use a Formatter to format the message (e.g. in XML) before publishing it to an I/O stream.

Java Logging API provides the following Handlers:

- StreamHandler: Writes formatted records to an OutputStream.
- ConsoleHandler: Writes formatted records to System.err
- FileHandler: Writes formatted records either to a single file, or to a set of rotating log files.
- SocketHandler: Writes formatted records to remote TCP ports.
- MemoryHandler: Buffers log records in memory.

An application may define additional Handlers.

Example of the XML formatted log:

```
<log>
  <record>
    <date>2005-01-19T16:24:17</date>
    <millis>1106180657444</millis>
    <sequence>4</sequence>
    <logger>JavaLoggingExample</logger>
    <level>FINER</level>
    <class>JavaLoggingExample</class>
    <method>testLog</method>
    <thread>10</thread>
    <message>ENTRY</message>
  </record>
```

```
</log>
```

Obviously, Java provides infrastructure that meats almost all requirements listed at the beginning of the document. The types and priority levels are not the same as defined in ACS, however, additional levels may be defined as listed in Table 2 on page 11.

UNIX/LINUX

The Linux kernel has an established "severity level" for error messages within the kernel and operating system. For now, the plan is that CMIB[2] software will be using this error reporting/logging system for the low level (device driver) parts of the system since it is the only thing readily available when executing code within the kernel. Generally, the only errors here are associated with memory access faults or other basically fatal problems. Either the hardware is working or is broken.

The Unix/Linux error priority levels are established as follows[3]:

```
KERN_EMERG    "<0>"  /* system is unusable               */
KERN_ALERT    "<1>"  /* action must be taken immediately */
KERN_CRIT     "<2>"  /* critical conditions              */
KERN_ERR      "<3>"  /* error conditions                 */
KERN_WARNING  "<4>"  /* warning conditions               */
KERN_NOTICE   "<5>"  /* normal but significant condition */
KERN_INFO     "<6>"  /* informational                    */
KERN_DEBUG    "<7>"  /* debug-level messages             */
```

The syslog system has the ability to filter these error messages based on a fixed threshold i.e. threshold = 4 means only warning <4> or better <0...3> messages are logged.

Typically, error levels 4 and worse would by default be propagated out of the module handlers to a (local or remote) master error task that makes further decisions on where these messages are sent. Implementation of the module handlers error reporting involves the user (either a human or the MCCC) specifying both an error message threshold and error message pipe (destination) with a reasonable set of defaults implied until some external system specifies otherwise.

Example:

As an example, consider a person doing module testing on a bench. The communications with a module occurs over three named pipes (data streams) using XML: the "write" pipe, "read" pipe and "error" pipe (stdin, stdout, and stderr).

[2] Correlator Management Interface Board – PC-104 board located on each correlator board.
[3] From manual section 2 of syslog.

The test person who wishes to see all errors and all the available diagnostic details (level 7 and below) have to send an XML string that specifies "set error report level to 7 and stream these errors out through stdout (all messages at the current threshold level will always go out to stderr):

```
<baselineBoard>
      <recirculator>
            <errorLevel = "7" />
            <errorPipe = "stdout" />
      </recirculator>
</baselineBoard>
```

The ability to set thresholds and direct where error messages are sent is already built into the CMIB software.

Decentralized Approach To Alarm Handling and Logging

Without a defined common logging and error handling strategy every element of the EVLA system will have slightly different approach to error handling and logging.

One solution is to allow every sub-system to define its own rules for error handling and logging that best match with the development environment and type of application.

In such system every log / alarm message could specify the location (URL) where more information about the event can be found. This would allow each sub-system to independently define the content and format of alarm / log messages.

The alarm handling system (that will eventually be added to the system) would either be able to handle various alarm messages based on the source of the message or would implement a pre-processor to add a wrapper to received alarm/log messages in order to create an illusion of a common programming style.

Advantage of such approach is that each sub-system could use a “native” format and priority scheme for alarm/log messages. Also, decisions related to alarm handling and logging would be postponed for same later time when more resources will be available.

Disadvantages of such approach are:

- Complexity - error-handling application would be rather complex and operators would need to learn error handling and logging scheme for all sub-systems.
- Runtime penalty - in the case that error / alarm handler have to add a wrapper to each received message.

- Maintenance – When any of the sub-systems is updated there may be a need to update wrappers. Personnel responsible for maintenance of the system would have to understand error-handling scheme for all sub-systems.

Conclusions

There is a need to define a common strategy for handling alarms and logs in the EVLA system. Without a defined structuring scheme and error handling strategy every subsystem will define its own rules and the project will end up in a mess of error definitions.

In the absence of the system design that would define overall alarm handling and logging strategy there is a need to define alarm handling and logging strategy that will be used by all the elements of the correlator, namely: CMIBs, Backend, MCCC and CPCC.

The following should be common to all correlator sub-systems, and eventually to all elements of the EVLA System:

- XML Schema (content and format) of the log/error message.
- Allocation of error messages / codes.
- Types / Priority levels.
- Handling strategy.

Proposal

1. Since the correlator (CMIBs, MCCC, CPCC and Backend) does not use CORBA, the ACS implementation cannot be reused in the correlator; however, the correlator should use the same or similar format for the log messages and the same priority scheme as ALMA.
2. All sub-systems should use the same format for alarms and logs, in other words, all sub-systems should use the same XML Schema. XML Schema indicates which elements are mandatory and which are optional.
3. The same structuring scheme should apply for code and description of all error/log messages. An integer value (code) should be assigned to each log / error message even if a sub-system does not specify integer values in generated alarms and logs.

4. Code and description of the alarm and log messages should be stored in the centralized repository. The repository may be a single file or a set of files, where each file contains definitions for a sub-system or an application. Due to a large number of messages, keeping the list of all alarm/log messages in a single file may be impractical, however, there should be the master alarm/log definition file that defines a code range for each sub-system.

5. All sub-systems should use the same priority scheme. The ACS priority scheme seems to be a logical choice.

6. A log / error message must clearly specify the source of the message (i.e. the sub-system that generated the message).

7. The operator should be allowed to set logging level for each sub-system (element) independently.

8. Each sub-system may (and probably should) provide a local logging repository. However, there should exist a central logging repository that contains logs / alarms for all the sub-systems. The EVLA may adopt the logging scheme defined for ALMA, where alarms are transmitted immediately, while logs are stored locally and transmitted only when a predefined threshold has been crossed. The threshold may be defined either as the maximum number of locally stored messages, the maximum size of the local logging file or as a timer. The functionality related to the centralized logging and alarm handling may be added later when the overall system design is defined.

9. The MCCC will monitor the state of all correlator components and locally log received alarms and errors. In the absence of the central EVLA log repository, the MCCC log file may be used as a central repository for the correlator sub-systems.

10. In the ALMA system, every computer system (node in the network) periodically transmits the list of active alarms. In the absence of the active alarms a hart-beat message is transmitted. The Alarm System may generate an alarm if source (subsystem) stops transmitting list of active alarms.

    This would require MCCC (and any other sub-system that monitors alarms generated by other systems) to distinguish between a new and a re-transmitted alarm. Since MCCC maintains status of all components, that should not be a problem. Re-transmitted alarms should not be logged on the MCCC.

    All the sub-systems of the EVLA correlator will continuously transmit hart-beat message, whether or not the list of active alarms will be part of the message is to be defined. Such functionality can be added / removed depending on the overall system design.

11. Should executor listen to CMIB reports? Or is the MCCC going to report a digest of the correlator state? MCCC will listen to CMIB alarm / status reports in order

to react to changes. However, the EMCS, in order to implement its own tools, may want to be able to listen to all alarm messages. To be defined.

Priority levels

Table 2 lists ACS priority levels and indicates how UNIX/LINUX and JAVA priority levels compare with ACS. Conversion from the UNIX priority level to the ACS priority level is rather straightforward, however log/alarm handler will have to ignore integer value assigned to the log type.

In order to comply with ACS, Java applications need to define new priority levels. Logs generated by Java applications should specify integer values, to allow alarm / logging handler to convert Java defined priority levels into ACS.

N.B. Java to ACS conversion shown in **Table 2** is as specified in the document "ALMA Logging and Archiving".

Table 2 ACS, UNIX and Java priority levels

ACS Priority level		**UNIX/LINUX Priority Level**		**JAVA Priority Level**	
Trace	2	KERN_DEBUG	7	FINEST	400
Debug	3	KERN_DEBUG	7	FINE	700
Info	4	KERN_INFO	6	INFO	800
Notice	5	KERN_NOTICE	5	INFO	800
Warning	6	KERN_WARNING	4	WARNING	900
Error	8	KERN_ERR	3	WARNING	901
Critical	9	KERN_CRIT	2	WARNING	902
Alert	10	KERN_ALERT	1	WARNING	903
Emergency	11	KERN_EMERG	0	SEVERE	1000

How to handle allocation of codes and descriptions

A file that contains log / alarm message strings / codes may become large, it may be more practical to allow each sub-system to maintain its own list of alarm & log messages and codes. However, in either case, we may end up with a chaotic collection of error messages with a lot of redundancy.

Also, we may run into situations where there is inconsistency between the message file and the source code. This may especially prove to be a problem during development, while everything is changing.

The file that contains the message string/code definitions must be under CVS control. When updated, it must be promptly checked in, so that other developers always see the latest version. In the absence of the centralized authority we have to rely on the discipline of the designers.

References

[1] ALMA Logging and Archiving, Klemen Zagar, Institute Jozef Stefan, 23-07-2004

[2] ALMA - ACS Error System, Bogdan Jeram et al., ESO, 17-12-2003

[3] LASER Project Vision, Caledrini, Polivka, 31.1.2001, http://proj-laser.web.cern.ch/proj-laser/

[4] Java Logging API – on-line documentation:
http://java.sun.com/j2se/1.5.0/docs/guide/logging/index.html

Appendix

ALMA Error System: ASI Alarm Message - Schema

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
      XML schema definition for source alarm messages
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="ASI-message">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="source-name" type="xsd:string" maxOccurs="1"   minOccurs="1"/>
        <xsd:element name="source-hostname" type="xsd:string" maxOccurs="1" minOccurs="1"/>
        <xsd:element name="source-timestamp" type="timestamp" maxOccurs="1" minOccurs="1"/>
        <xsd:element ref="fault-states" maxOccurs="1" minOccurs="1"/>
      </xsd:sequence>
      <xsd:attributeGroup ref="ASI-message-attributes"/>
        </xsd:complexType>
  </xsd:element>

  <xsd:element name="fault-states">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="fault-state" maxOccurs="unbounded" minOccurs="0"/>
      </xsd:sequence>
        </xsd:complexType>
  </xsd:element>

  <xsd:element name="fault-state">
        <xsd:complexType>
      <xsd:all>
        <xsd:element name="descriptor" type="descriptor-type"/>
        <xsd:element name="user-properties" type="properties"/>
        <xsd:element name="user-timestamp" type="timestamp"/>
      </xsd:all>
      <xsd:attributeGroup ref="fault-state-attributes"/>
        </xsd:complexType>
  </xsd:element>

  <xsd:simpleType name="descriptor-type">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="ACTIVE|TERMINATE|CHANGE|INSTANT"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="properties">
      <xsd:sequence>
        <xsd:element ref="property" maxOccurs="unbounded" minOccurs="0"/>
      </xsd:sequence>
```

```
    </xsd:complexType>

    <xsd:element name="property">
            <xsd:complexType>
           <xsd:attributeGroup ref="property-attributes"/>
            </xsd:complexType>
    </xsd:element>

    <xsd:attributeGroup name="property-attributes">
       <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="value" type="xsd:string"/>
    </xsd:attributeGroup>

    <xsd:attributeGroup name="fault-state-attributes">
            <xsd:attribute name="family" type="xsd:string"/>
            <xsd:attribute name="member" type="xsd:string"/>
            <xsd:attribute name="code" type="xsd:integer"/>
    </xsd:attributeGroup>

    <xsd:attributeGroup name="ASI-message-attributes">
            <xsd:attribute name="backup" type="xsd:boolean"/>
            <xsd:attribute name="version" type="xsd:string"/>
    </xsd:attributeGroup>

    <xsd:complexType name="timestamp">
          <xsd:attributeGroup ref="timestamp-attributes"/>
    </xsd:complexType>

    <xsd:attributeGroup name="timestamp-attributes">
       <xsd:attribute name="seconds" type="xsd:long"/>
            <xsd:attribute name="microseconds" type="xsd:long"/>
    </xsd:attributeGroup>

</xsd:schema>
```