ESCS logging subsystem

author: Marco Bartolini

16 maggio 2013

draft number: 3

IRA-INAF Medicina Radiotelescopes Via Fiorentina 3508/B 40059 Medicina BO Italia **T** +390516965811 **F** +390516965810
mbartolini@med.ira.inaf.it http://www.med.ira.inaf.it

Index


ISTITUTO NAZIONALE DI ASTROFISICA
INAF
NATIONAL INSTITUTE FOR ASTROPHYSICS

1. Logging

Logging is a fundamental component in a complex software system and ESCS is no exception. Many times in software development, a good project phase is not enough to know how the system will behave in every possible condition and every software hides some subtle bug or misbehavior which simply cannot be detected at code or compile time. Thus the need for logging what happens at runtime, logging is essential for:

- Development. Logging at the DEBUG level will enable the developer to inspect every single event in the system during its development in the effort of coding error free software.
- Post failure diagnostics. Logging will enable the diagnostic of the root cause of a runtime error and successive code modification.
- System Monitoring. Logging is one effective way to monitor system health at runtime and to give users and administrator an immediate feedback of what is happening under the hood.

1.1 Logging in ESCS

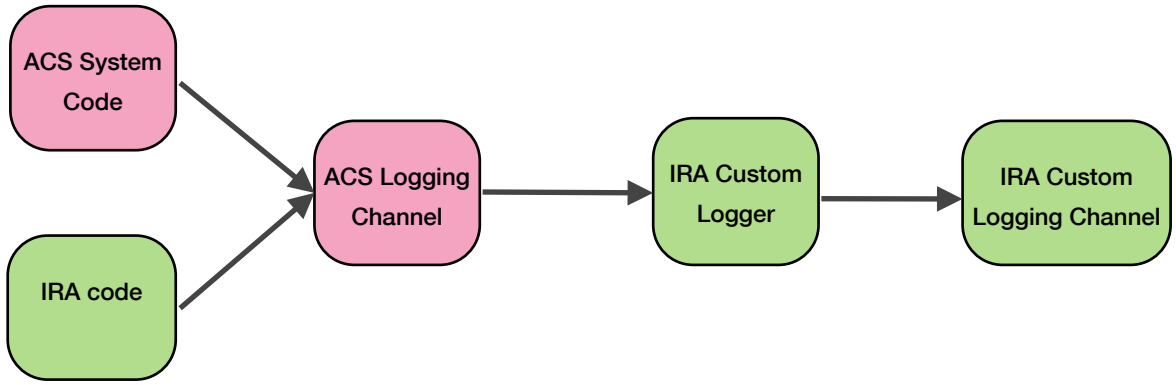
The ACS software system already contains a logging module so why do we need to develop a custom software component? First of all the ACS logging system as of version 8.2 is confused and spread between different versions and conventions and it is not easy to understand, it also does not ease the life of developers keeping them away from its usage. A second reason is that we need somehow to differentiate between the ACS system log messages and the log messages produced within the code we have written at IRA, at least for debugging purposes. A final reason is that we want to automate the logging setup at ACS start writing server components which will log only our custom messages.

2. Project structure

Many software modules have been developed in order to enable a custom logging system, and we wanna first have an overview of what happens at a higher level of abstraction.

In the diagram below green blocks represent the software components we have developed while pink ones represent what was present in the original ACS system. As you can see we first publish every log event to the ACS Logging Channel, mixing IRA and ACS events. This enables us to use ACS code in an effective way and ACS tools can be used to take in exam also our custom logging messages. Every log record is then captured and filtered by the IRA custom logger which implements its own

logics (i.e. write to file ...) while republishing our custom events to a separate channel for further elaboration and presentation to the user.



2.1 Custom Logging macros

2.1.1 Affected files

- Common/Libraries/IRALibrary/include/CustomLoggerUtils.h
- Common/Libraries/IRALibrary/src/CustomLoggerUtils.cpp

2.1.2 Architecture

We first have to look at where log messages are generated. Where we used to have *ACS_LOG* macros we redefined our own *CUSTOM_LOG* macros to be used in IRA produced code. The effect of the macro is that of adding a Key->Value data pair in the log message XML structure identifying our IRA generated events. Macros are defined in *Common/Libraries/IRALibrary/include/CustomLoggerUtils.h* and are:

- **CUSTOM_LOG** replaces the ACS **LOG** macro with identical signature.
- **CUSTOM_STATIC_LOG** is used for logging from a static context.
- **CUSTOM_EXCPT_LOG** is used for logging an exception event.

A typical usage in our code will be something like:

```
#include <IRA> //or only #include <CustomLoggerUtils.h>

//...
/** MyComponent.cpp - log message with ALERT level **/
//...
```

```
CUSTOM_LOG(LM_FULL_INFO, "MyModule::MyClass::MyMethod", (LM_ALERT, "My log
   message"));
```

Which will result in a log message like this to be published on the acs logging channel:

```
<Alert TimeStamp="2012-06-12T08:38:01.334" Routine="MyModule::MyClass::MyMethod"
   SourceObject="MANAGEMENT/CustomLogger"><![CDATA[My Log message]]><Data
   Name="source"><![CDATA[custom]]></Data></Alert>
```

Where the underlined xml text is what has been added by our macro with respect to the ACS logging macro.

Note that we can still use ACS defined log levels at this point, that's because custom logging events are created as particular ACS log events with some added data. This permits us to publish custom events on the ACS logging channel as defined at ACS startup and will result in minor impact on already written IRA code base, that will have to be changed only in the macro names.

2.2 CustomLogger ACS Component

Most of the custom logging logic resides in the CustomLoggerImpl ACS component and we will examine it starting from its idl definitions and xml configuration.

2.2.1 Affected files

- Common/Interfaces/ManagmentInterfaces/idl/CustomLogger.midl
- Common/Interfaces/ManagmentInterfaces/idl/ManagementDefinitions.midl
- Common/Servers/CustomLogger/include/CustomLoggerImpl.h
- Common/Servers/CustomLogger/include/expat_log_parsing.h
- Common/Servers/CustomLogger/src/CustomLoggerImpl.cpp
- Common/Servers/CustomLogger/src/expat_log_parsing.cpp
- Common/Servers/CustomLogger/config/CDB/schemas/CustomLogger.xsd
- /system/configuration/CDB/alma/MANAGEMENT/CustomLogger/CustomLogger.xml

2.2.2 Component configuration

Static parameters are defined at the component level and stored in the CDB as xml files, the schema specifies some important attributes to be defined:

```
<xs:attribute name="DefaultACSLogDir" type="xs:string" use="required" />
<xs:attribute name="DefaultACSLogFile" type="xs:string" use="required" />
<xs:attribute name="DefaultCustomLogDir" type="xs:string" use="required" />
<xs:attribute name="DefaultCustomLogFile" type="xs:string" use="required" />
<xs:attribute name="LogMaxAgeMillis" type="xs:long" use="required" />
```

Where the default directories must be defined as absolute paths to directories where custom log files will be saved. The filenames must be specified in the corresponding xml attributes. *LogMaxAgeMillis* specifies how long each record will be stored in memory before being broadcast to the logger, this behaviour permits to sort the notifications received from different components at different times before writing to file.

2.2.3 IDL definitions

The component is defined in the *Management* subsystem and the relative definitions can be found in *Common/Interfaces/ManagmentInterfaces/idl/CustomLogger.midl* - *ManagementDefinitions.midl* in particular, in the definitions we redefine the log levels based on a new *LogLevel* enumeration which will substitute the ACS naming scheme in our log records:

```
        /**
         * Log level definitions for ACS custom logging utilization
         */
    enum LogLevel{
           C_TRACE,
           C_DEBUG,
           C_INFO,
           C_NOTICE,
           C_WARNING,
           C_ERROR,
           C_CRITICAL,
           C_ALERT,
           C_EMERGENCY
};

      ACS_ENUM(LogLevel);
```

While this is how the server interface is defined:

```
        /**
         * Component which automatizes the logging functionalities in ACS.
```

```
        * This component is designed to intercept log messages and direct those to
          3 different outputs:
        * 1) Custom logging events, identified by the extra data source=custom,
             are written to a file in an abbreviated form only including the
             timestamp, the log level and the log message.
        * 2) All logging events are written to a sytem log file as xml ACS
             LogRecords, these include all ACS inormations.
        * 3) Custom logging events are further redirected to a CUSTOM LOGGING
             CHANNEL for client immediate notification.
        */
       interface CustomLogger: ACS::CharacteristicComponent
       {
               /**
                * Filename of the custom logging file where only custom events
    will be written.
                */
               readonly attribute ACS::ROstring filename;
               /**
                * Number of custom logging events received
                */
               readonly attribute ACS::ROlong nevents;
               /**
                * True if logging functionalities are active
                */
               readonly attribute ROTBoolean isLogging;
               /**
                * Minimum log level for filtered log messages
                */
               readonly attribute ROLogLevel minLevel;
               /**
                * Maximum log level for filtered log messages
                */
               readonly attribute ROLogLevel maxLevel;
               /**
                * Close the actual log files and open two new ones for logging.
                * Tries to create the necessary directory.
                * @param base_path_log: the directory name for the custom log file
                * @param base_path_full_log: the directory name for the full log
    file
                * @param filename_log: the file name for the custom log file
                * @param filename_full_log: the file name for the full log file
                */
               void setLogfile(in string base_path_log,
                               in string base_path_full_log,
                               in string filename_log,
                               in string filename_full_log) raises
                               (ManagementErrors::ManagementErrorsEx);
               /**
                * Emits a custom logging message. Used mainly for debugging and
    testing purposes.
                * @param msg: the logging message
                * @param level: the logging level
                */
               void emitLog(in string msg, in LogLevel level);
               /**
                * Emits a custom logging message from a static context. Used
    mainly for debugging and testing purposes.
                * @param msg: the logging message
                * @param level: the logging level
                */
               void emitStaticLog(in string msg, in LogLevel level);
               /**
```

```
                 * Emits a new Exception with two levels backtrace and logs it.
                 * Used for debug and testing.
                 */
                void emitExceptionLog();
                /**
                 * Close the actual log files and stops logging. Flushes all
                   buffers.
                 */
                void closeLogfile() raises (ManagementErrors::ManagementErrorsEx);
                /**
                 * Set the minimum logging level to be included in custom log files
                 * @param level: the logging level
                 */
                void setMinLevel(in LogLevel level);
                /**
                 * Set the maximum level to be included in custom log files
                 * @param level: the logging level
                 */
                void setMaxLevel(in LogLevel level);
                /**
                 * Flushes the log events queued by the LoggingProxy and writes to
                 * custom log files
                 */
                void flush();
        };
```

2.2.4 The Server

Server component logics are defined in *Common/Servers/CustomLogger/*

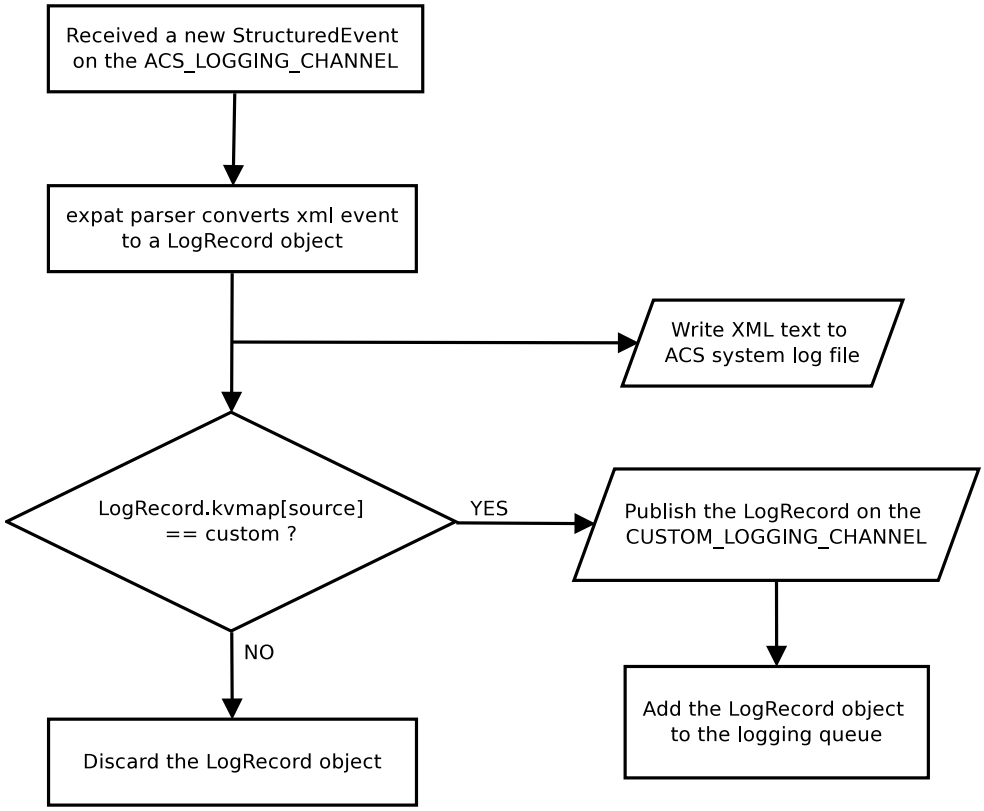
The *CustomLoggerImpl* basically subscribes to the ACS logging channel using a *StructuredPushConsumer* and receives all the *StructuredEvents* published on the channel. Note that at this point we receive both ACS and CUSTOM log records on the same channel. Every event is than treated as an xml string and parsed using the *expat* libraries using the functions defined in *expat_log_parsing.h* .

A new *LogRecord* object is thus created according to the RAII (Resource Acquisition Is Initialization) pattern by which we store LogRecords using boost::shared_ptr objects upon creation, which relieves the developer from memory management issues.

```
/**
 * Class used to represent a log record and the necessary information
 * to be stored during the xml parsing. These two have been packed for conveniently
 * make use of expat parsing primitives which accept one only structure as user
   data.
 * Attributes marked as _ are to be intended as XML accessories.
 * @TODO LogRecord informations should be decoupled from xml parsing structures.
 */
class LogRecord
{
public:
    /**
```

```
        * Constructor
        */
       LogRecord();
       virtual ~LogRecord();
       /**
        * The log level of this record.
        */
       Management::LogLevel log_level;
       std::string log_level_name, message, process_name, _element, xml_text,
     _data_name;
       /**
        * the timestamp of the logging event, extracted from the ACSLogRecord.
        */
       ACS::Time timestamp;
       /**
        * Key-Value couples of extra data tied to the log record.
        */
       KVMap kwargs;
       bool _initialized, _parsing_message_cdata, _parsing_data_cdata, _finished;
       int _depth;
       /**
        * Adds a Key-Value pair to the log record.
        * @param key the key string.
        * @param val the value string.
        */
       void add_data(std::string key, std::string val);
       /**
        * Reads a Key-Value pair from the record.
        * @param key the key string .
        * @return the associated value if key is found, NULL otherwise.
        */
       std::string get_data(std::string key);
   };
```

The LogRecord is than handled by the server which writes the ACS system log file and forwards the LogRecord to the LoggingQueue and to the CUSTOM_LOGGING_CHANNEL. At this point we can note that each event is written to the ACS system log file with all of its informations, this is intended for administration purposes so that ACS system administrators and developers have a place where every detail is stored for debugging and management purposes. This file would be overkill for end users, that's why CUSTOM events are defined so that we can filter what is propagated up to the end user's view.



Custom log messages are recognized by the **filter** method of *CustomLoggerImpl* class which searches for the key->value pair *"source = custom"* in the LogRecord's kwmap.

```
    /**
    * Filter the log records.
    * This filter shuold return true for all the messages produced by custom logging
    * functions and false for every ACS system log record.
    * @return True if the log record is a custom one.
    */
         virtual bool filter(LogRecord& log_record);
```

CUSTOM_LOGGING_CHANNEL is an ACS notification channel to which everyone can subscribe in order to receive the events in an unsorted and rough way, messages are sorted according to their timestamp only in the logging queue while previously we have no guarantee on the reception order.

```
/**
  * Structure used to transmit Logging events through the CUSTOM LOGGING CHANNEL
  * It's similar to an ACS Log Record but has been defined for our purposes.
  */
```

```
typedef struct CustomLoggingData
{
   ACS::Time timeStamp;
    LogLevel level;
    string msg;
 } TCustomLoggingData;

 const string CUSTOM_LOGGING_CHANNEL_NAME = "CUSTOM_LOGGING_CHANNEL";
```

2.2.5 The logging queue

LogRecord generated by IRA logging macros are added into a LoggingQueue which is defined in *expat_log_parsing.h* as:

```
typedef std::priority_queue<LogRecord_sp, std::vector<LogRecord_sp>,
   LogRecordComparator> LogRecordQueue;
```

Here LogRecords are ordered according to their *timestamp* attribute. A thread defined in the *CustomLogWriterThread* class consumes the queue accessing periodically at the head and checking the age of each log record. LogRecords older than a configurable age are popped from the queue and written to the custom log file. This technique permits to force a partial ordering of the log records within a certain amount of delayed reception which can be caused by the distributed architecture of the control system and by the different settings of components and containers. The maximum age after which logs are dumped to file is configured in */system/configuration/CDB/alma/MANAGEMENT/CustomLogger/CustomLogger.xml* together with other default parameters such as the log file names and paths.

2.2.6 Component lifecycle

We try to summarize the component lifecycle in the following list:

1. CustomLogger is loaded within the LoggerContainer
2. Upon initialization:
   - 2.1. subscribe to the ACS_LOGGING_CHANNEL
   - 2.2. initialize the expat XML parser
   - 2.3. Create the log writer thread
   - 2.4. create the CUSTOM_LOGGING_CHANNEL for output
   - 2.5. open the default log files
3. Upon execution:
   - 3.1. starts the log writer thread
4. On each StructuredEvent received on the ACS_LOGGING_CHANNEL

4.1.convert the event into a LogRecord object using the XML parser
4.2.write the xml log record to the system log file
4.3.filter the log file, and if it's the case:
*4.3.1.*publish the LogRecord to the CUSTOM_LOGGING_CHANNEL
*4.3.2.*add the LogRecord to the logging queue
*4.4.*each time interval the log writer thread
*4.4.1.*access the head of the logging queue:
*4.4.1.1.*if the first element is older than a configured amount of time write the log info to file and pop it from the queue, passing to the next
*5.*When log file changes:
5.1. Publish a last LogRecord telling that the log file is being changed
5.2. Flushes the content of the logging queue to file
5.3. Close the old log files and open the new ones

2.2.7 Log Files formats

In a working environment the generated log files will look like the following ones: the first one is the generated system log file (defaults to */archive/events/acs.log*) where you can see a Trace message generated by ACS code and a Debug message generated using the IRA macros.

```
<Trace TimeStamp="2012-06-28T14:31:37.067" File="acsncSupplierImpl.cpp" Line="459"
   Routine="Supplier::subscription_change" Host="hannibal.med.ira.inaf.it" P
rocess="LoggerContainer" Thread="ORBTask" Context="" SourceObject="LoggerContainer-
   GL"></Trace>

<Debug TimeStamp="2012-06-28T14:31:49.203" File="CustomLoggerImpl.cpp" Line="241"
   Routine="CustomLoggerImpl::emit_log" Host="hannibal.med.ira.inaf.it" Proce
ss="LoggerContainer" Thread="ORBTask" Context="" SourceObject="MANAGEMENT/
   CustomLogger"><![CDATA[our debug message]]><Data Name="source"><![CDATA[custom]]
   ></Data></Debug>
```

the second one is the custom log file (defaults to */archive/logs/station.log*) generated at the corresponding time. This log file will be presented to the user and contains only the information unrelated to the underlying ACS logics, originated by IRA developed software. Note that the Trace event has been discarded and the log records have a different format according to:

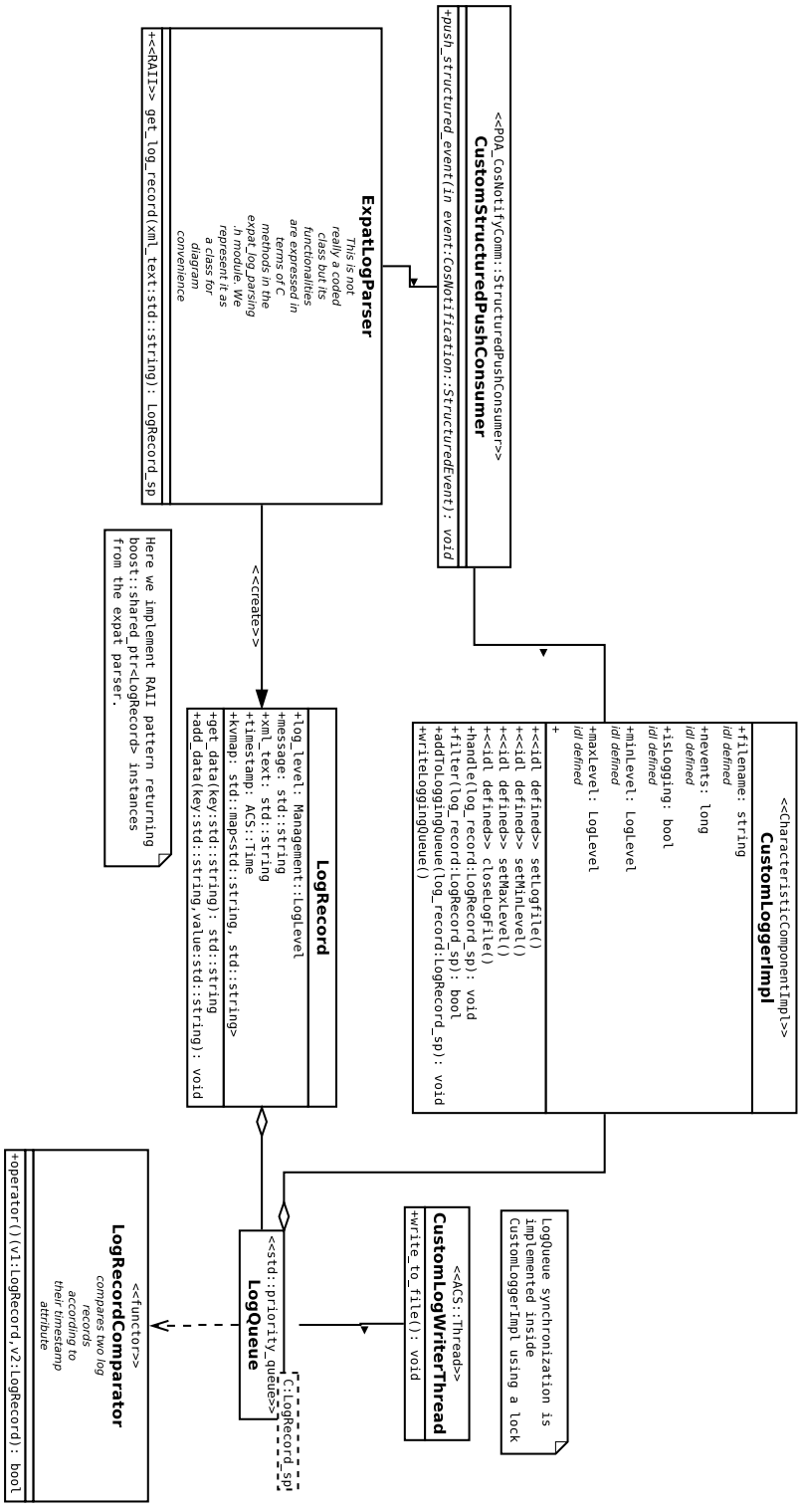
YEAR-DOY-HH:MM:SS:mmm LOGLEVEL THE LOG MESSAGE

where the timestamp and the log level are the first two words of each line separated by a whitespace, while the log message comprehends all of the following words, 0 or more.

```
2012-178-12:53:28.198 Info addSubscription - CustomLoggingData
```

```
2012-180-14:31:49.203 Debug our debug message
2012-180-14:32:01.013 Notice Main - log event nr: 0
```

2.2.8 CustomLogger UML class diagram



2.3 Python Logging

An important feature for our logging module is being able to collect log events generated by ACS clients written in the python programming language. An **IRAPy** python package has been created within the IRA libraries in *Common/Libraries/IRALibrary/src/IRAPy* to contain the custom IRA libraries developed for ACS

2.3.1 Affected files

- Common/Libraries/IRALibrary/src/IRAPy/__init__.py
- Common/Libraries/IRALibrary/src/IRAPy/customlogging.py

2.3.2 Custom Python Logger

A new logger class **CustomLogger** is defined in the *customlogging* module which inherits from **Acs.Common.Log.Logger** and redefines its **log** method adding the extra field *"source=custom"* in the LogRecord structure.

A new method **getLogger** is defined in the module which returns the specified logger. Using this logger with its logging method results in custom log records being generated and correctly filtered by the CustomLogger component, as suggested by the module execution method, which can be used for testing purposes:

```
if __name__ == '__main__':
    logger = getLogger("IRA_CUSTOM_LOGGER")
    logger.logDebug("Custom DEBUG message")
    logger.logWarning("Custom WARNING message")
    logger.logInfo("Custom INFO message")
    logger.logAlert("Custom ALERT message")
    logger.logCritical("Custom CRITICAL message")
```

2.3.2 Proposed ACS patch

For this technique to work we would need to modify the Acspy package according to the following patch:

```
--- ACS/LGPL/CommonSoftware/acspy/src/Acspy/Common/ACSHandler.py        2010-03-27
   17:46:49.000000000 +0100
+++ ACS/LGPL/CommonSoftware/acspy/src/Acspy/Common/ACSHandler_patched.py
   2012-04-19 11:20:59.000000000 +0200
@@ -280,6 +280,10 @@

         # Put remaining keyword arguments into NVPairSeq
         data = []
+        #Patch by Marco Bartolini - bartolini@ira.inaf.it
```

```
+         if hasattr(record, "data"):
+             for _k, _v in record.data.iteritems():
+                 data.append(ACSLog.NVPair(_k, _v))

          if 'priority' in record.__dict__:
              # The more exotic log functions have priority keyword arguments
```

The patch has been proposed to the ACS mailing list but it has not been taken into account.

2.3.3 Default python logger usage

Not being able to use the previous patch we opted for a second solution, creating a default logger instance in the IRAPy package initialization and giving it a standard name against which the CustomLogger filter function will match positively. Thus using the custom logging capabilities within our own python code will result in a code like this:

```
from IRAPy import logger
#import IRAPy

logger.logInfo("our custom info message")
#IRAPy.logger.logInfo("our custom info message")
```

which will generate the two following log messages in the system and custom log files:

```
<Info TimeStamp="2012-06-28T14:32:24.055" File="customlogging" Line="36"
   Routine="Unknown" Host="hannibal.med.ira.inaf.it" Process="IRA_CUSTOM_LOGGER"
   Thread="MainThread" Context="" SourceObject="IRA_CUSTOM_LOGGER"><![CDATA[Main -
   our custom info message]]></Info>
```

and

```
2012-180-14:32:24.055 Info Main - our custom info message
```

3. Logging Client

In order to immediately visualize the log messages generated by our custom logging system and published on the CUSTOM_LOGGING_CHANNEL a GUI client has been implemented which subscribes to the channel and presents the records to the user.

3.1 Affected files

- Common/Clients/CustomLoggingClient/src/loggingDisplay
- Common/Clients/CustomLoggingClient/src/_gui_customLoggingClient.py
- Common/Libraries/IRALibrary/src/IRAPy/bsqueue.py

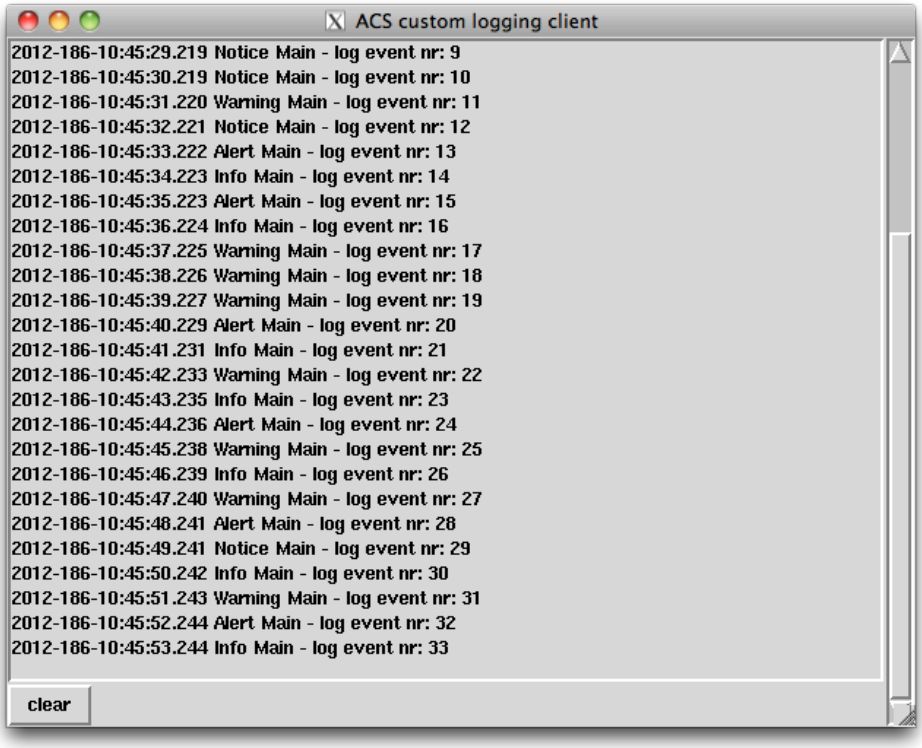
3.2 Usage

The application consists of a python process which implements a consumer of the CUSTOM_LOGGING_CHANNEL notification channel and displays the received log records in a Tkinter window. The GUI is really simple and does not offer many functionalities beyond the realtime log visualization.

The application can be launched as:

```
user@host:/path$ loggingDisplay
```

which will open the following window where the log events will be visualized:



4. Logging utilization guidelines

The custom logging system should be used whenever the developer want some information to be raised from the code up to the custom logging channel and the custom log file. These log records will be made available to the end users of the control system, so the relating channels should be used only in circumstances where informations relevant to the end users must be presented.

Log levels defined in our modules represent an identical hierarchy as those defined by ACS logging system, which are described in full detail in the document **Logging and Archiving** that can be found on ESO website at http://www.eso.org/projects/alma/develop/acs/OnlineDocs/Logging_and_Archiving.pdf . In its revision 1.36 , dated 2007-07-30 , log levels are described at pag. 17 - 18. We report the levels here for sake of completeness:

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.

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.

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.

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.

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.

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. Alerts are used for reporting errors that must be solved immediately.

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. Alerts are used for reporting errors that must be solved immediately.

4.1 Log Message Format

The custom log file will be formatted as explained in 2.2.7 but we still have to define some rules to be used while filling the message part of the log record. In order to obtain some uniformity in the log informations, developers should try to adopt the following convention:

- Messages are all lowercase.
- Never mention the log level itself in the log message. i.e. in an error message there's no need to say "*communication error*" , this information is already contained in the log level. Developer should instead try to explain what caused the error, i.e. "*connection reset by peer*".
- Data can be included in the message body within square brackets at the end of the message, i.e. "*antenna pointing at the source [az=44.5, el=18]*" . Data must be named parameters, separated by comma.
- Square brackets should be used only for data.
- As a general rule, messages are more effective when presented in the form: subject + verb + complement.