FERMILAB-CONF-10-705-CD

International Conference on Computing in High Energy and Nuclear Physics (CHEP 2010) IOP Publishing
Journal of Physics: Conference Series **331** (2011) 022017 doi:10.1088/1742-6596/331/2/022017

The Message Logging System for NOνA Experiment

Qiming Lu, J. B. Kowalkowski, and K. A. Biery

Computing Division, Fermi National Accelerator Laboratory
P.O.Box 500, Batavia, Illinois 60510, U.S.

E-mail: qlu@fnal.gov, jbk@fnal.gov, biery@fnal.gov

Abstract. The message logging system provides the infrastructure for all of the distributed processes in the data acquisition (DAQ) to report status messages of various severities in a consistent manner to a central location, as well as providing the tools for displaying and archiving the messages. The message logging system has been developed over a decade, and has been run successfully on CDF and CMS experiments. The most recent work to the message logging system is to build it as a stand-alone package with the name MessageFacility which works for any generic framework or applications, with NOνA as the first driving user. System designs and architectures, as well as the efforts of making it a generic library will be discussed. We also present new features that have been added.

1. Introduction

The basic functionality of a message logging system is to let user code report error or status messages. Additional features allow for archiving of the messages so that they can be later examined for problem diagnosis and handling. Moreover, a sound message logging system should also provide a uniform and sensible logging behavior, in terms of information output and formatting, while still keeping the low overhead. A certain degree of flexibility in customizing the logging behavior is as well important, as has been demanded in real applications. In modern high-energy experiments, the message logging system is one of the important components in both online and offline systems. It provides the functionalities to create, publish, display and archive status messages in a consistent manner across the different experiments and different groups within an experiment. It is an invaluable tool for diagnosing and handling problems in the system during the development process as well as in the production phase.

The message logger package we discuss in the paper originated from the ErrorLogger package in ZOOM (Fermi Physics Class Libraries Task Force project, named ZOOM). It was successfully used in CDF, D0, and other experiments at Fermi lab. The first major rework of the ErrorLogger package happened during its adoption by CMS and integration into the CMS source tree and CMS framework [2]. The package continues to be maintained and modified to fit new user needs. With the LHC coming online in early 2010, the MessageLogger continues to play a role in new experiments.

The most recent development of the MessageLogger package has been to make it a generic and stand alone package that can be easily integrated into any system or experiment. The source code was again extracted from the CMS source tree and given a new name of MessageFacility. NOνA (NuMI Off-axis ν_e Appearance experiment) is the first user and driving customer for the generic MessageFacility package as it comes online in 2011.

Operated by Fermi Research Alliance, LLC under Contract No. De-AC02-07CH11359 with the United States Department of Energy.

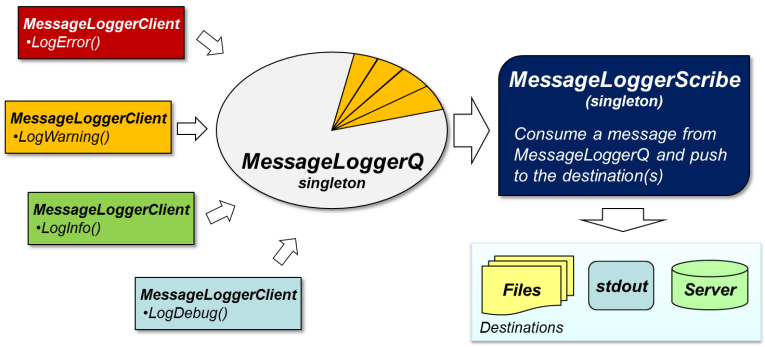

Figure 1. Flow chart of the message logging in a multi-thread environment. The MessageLoggerQ is a single presence single consumer queue. It accepts data from multiple producers, stores in a circular buffer, and supplies the data to the single consumer, which is a single presence of MessageLoggerScribe. Such design can avoid conflicts and prevent interlacing of message information from different threads, meanwhile minimizing the impact to the performance of main user application.

The remainder of this paper is organized as follows. Section 2 briefly discusses the underlying system design and architectures of Message Facility. Our new and most recent development to the package, as well as its first application in the NOνA experiment are described in Section 3. Finally, Section 4 gives the conclusions.

2. System design and architecture

The purpose of the MessageFacility package is to allow code in algorithm modules, service libraries, and other framework components to send messages to a unified message logging and statistics facility. The package captures and coordinates messages originating in multiple sources into a specified set of destinations. Meanwhile, it also provides means of controlling multiple output destinations, message limitations, and other behavior by a filtering using the configuration subsystem.

2.1. Issuing Messages

In order to issue messages, the user code needs to have the service instantiated properly. One of the following functions can be used in the user code to issue a message:

```
LogError   ("category") << a << b << ... << z;
LogWarning ("category") << a << b << ... << z;
LogInfo    ("category") << a << b << ... << z;
```

The functions represent three available levels of "severity" of the message. The string literal `category` is used to specify what this message is about, i.e., the type of the message. A message header is assembled with the category, severities, time and processing context, which is included in each message. The message body is all the info added from subsequent strings, ints, doubles, or any object that has stream insertion operator.

2.2. Message Handling

The effect of a user issuing a `LogInfo()` (for example) is the formation of an error object and the sending of it to the message queue. This error object is later consumed by the

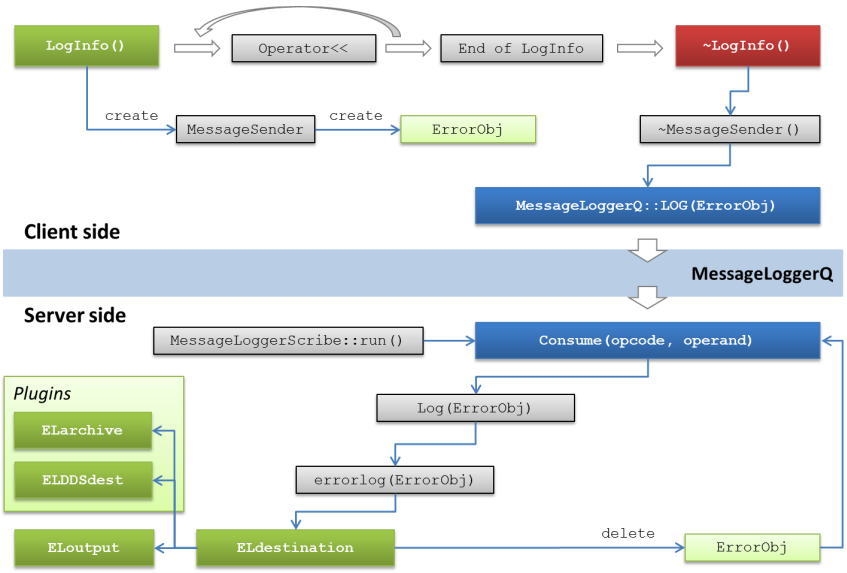

Figure 2. Flow chart of the execution path when `LogInfo()` is called in a multi-thread environment. The single presence of the message server residues in its own thread. It interacts with one or multiple clients through the MessageLoggerQ. Both the server side and the client side are in one process.

MessageLoggerScribe. This scribe can be configured to filter error objects and will dispatch the accepted messages to one or more destinations. This section will outline the steps involved.

The user code may be running in one or more threads, each of which might issue messages. The issuer drops the message to the message queue then returns immediately. We call these the client side. A single entity, the MessageLoggerScribe, picks up on these messages from the queue and forwards them to the logger one at a time; this prevents interlacing of message information from different threads. It also avoids the scribe being the bottleneck of the system. We call the thread running MessageLoggerScribe the server side. Figure 1 shows the diagram of the message service in multi-threaded environment. The single presence of MessageLoggerQ is the boundary between client side and server side. All user programatic interaction is on the client side, but the configuration (driven by the .cfg file) is dealt with on the server side.

The client log function uses RAII to generate and send out error object – it is a functor under the hood. In its constructor a temporary instance of MessageSender is created. The purpose of the MessageSender instance is to generate an ErrorObj on the heap, populate it with message header and message body, and finally send it to the server upon the destruction of the MessageSender instance. The interaction of MessageSender with the server side consists of two steps. First, it uses the MessageDrop instance to supply the module and run/event context. Here, the MessageDrop is a thread-specific singleton, whose purpose is to convey framework information, provided by a ContextProvider class through callback functions, to the point-of-invocation where a message is issued, as the functions issuing messages may not naturally have access to the module description or event id, etc. After the injection, the module and processing context go into the ErrorObj. Second, it invokes the static LOG method of MessageLoggerQ, pushing the pointer to the ErrorObj to the queue and returns.

At the server side, the MessageLoggerScribe continually consumes commands from the queue, as shown in the lower half of Figure 2. When an ErrorObj is received from the queue, the category string is parsed. The ErrorLog() function is then called for each category given in the ErrorObj. The purpose of ErrorLog() is to ship the ErrorObj to every destination. Each destination (normally ELoutput) will apply the limits and thresholds, format the message, add header information, and output the message. Finally, the completion of ErrorLog() deletes the ErrorObj – completing the promise made when it was passed responsibility for this heap-resident object.

3. Message Facility for NOνA

Effort has been made to continue maintenance of the Message Logger libraries. A decision was made to separate the message logging system from the CMS framework and build it as a generic stand alone package.

3.1. Generic Message Logging System

For the CMS MessageLogger, the framework takes the responsibility of starting the service and maintaining its lifetime. When used as a stand alone library, the MessageFacility must provide an interface to start and configure its service. Internally the message service is held in a singleton to avoid multiple instantiations within one process. The sequence of starting the service is wrapped into a free function with arguments specifying whether to start the service in a single-thread mode, or multi-thread mode, as well as detailed configuration information for the service to configure itself. A major concern for the MessageFacility being a stand alone package was the lifetime of the service. An intending design decision was to support the message logging during the lifetime of a process. It is, however, not permissible to log messages in the destructor of a static object.

In order to achieve this, the lifetime of the service presence has been designed to have three stages (modes): offline mode (pre-configuration), configuring mode (in-configuring), and fully operational mode (post-configuration). Clear boundaries between stages have been defined to instantiate a proper light-weighted or full service at each of the stages. When the main thread returns and exits, the static object holding message service presence is destroyed. In its destructor, it issues a `FLUSH` and a `EXIT_LOGGING` commands to instruct the MessageLoggerScribe to shut down the facility. All messages issued after this point are therefore ignored. The presence keeps waiting for the MessageLoggerScribe thread to finish logging all remaining messages in the queue and join to destroy itself.

3.2. Distributed Message Logging and Facilities

The data acquisition (DAQ) system of NOνA experiment consists of approximately 252 data concentrate modules (DCMs) running embedded Linux on a PowerPC platform. Along with the timing distribution units, buffer farm nodes, and permanent data loggers, the NOνA online system is a combination of a large number of distributed processes and nodes. One imminent requirement for the message logging subsystem is to provide the infrastructure for all of the distributed processes in the DAQ system to report status messages to a central location. There is also a need for tools to display and archive messages. Similar functionalities are provided by generic message logging systems such as CMLOG [1] and rsyslog [3] etc. In addition, MessageFacility also provides provides QoS control of the transportation, and API interfaces for future developments based on the message server.

In the design of MessageFacility, each logging destination is an instance of class inherited from the ELdestination base class, which is a virtual class defining the interface to a logging destination. Concrete classes derived from provide overriding methods for destination output needs. The system includes a destination that writes to files, called ELoutput, as shown in Figure 2. The design allows for new destination types to be added to the MessageFacility package. As per the requirements for distributed message logging, a new extension of ELdestination for handling message logging through the network interface has been developed. The added remote-logging destination is a mid-layer between MessageFacility and lower-level network transportation interface. To achieve reliability and quality-of-service-capable network transmission, it was decided to use one of the Data Distribution Service (DDS) implementations from PrismTech called OpenSplice DDS. DDS is a network middleware that implements a publish/subscribe model for sending and receiving data, events, and commands among the nodes. Nodes that are producing information (publishers) create “topics” (e.g., temperature, location,

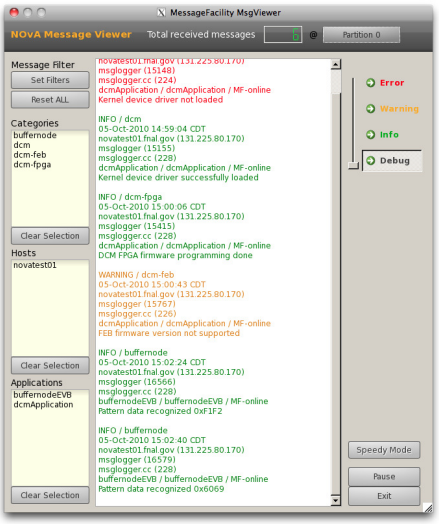

Figure 3. A screenshot of running msgviewer GUI application. The msgviewer is designed as a portable application that can be executed on any participant nodes for monitoring and inspecting the message flow on network. It also provides interfaces for message filtering based on the severity, message categories, hostname, and application name.

pressure) and publish "samples", while DDS takes care of delivering the sample to all subscribers that declare an interest in that topic. In the MessageFacility, the hostname, process id, and along with the severity of the message are defined as the DDS topic for messages. Thus, each distributed process has its own topic to publish samples (messages). Correspondingly, the system can have any number of active listeners subscribing to that topic. It adds great flexibility to the message logging system because it allows to have any number of message receivers listening to a subset of topics that are interested to them, on any participant nodes without interfering with each other. Furthermore, the implementation of OpenSplice DDS provides a high performance, real-time publishing subscribe messaging infrastructure. In our performance test, the peer-to-peer message throughputs in a ethernet test stand can achieve over 40,000 messages per second reliably with 256 bytes of message loads.

The newly developed ELDDSDest which uses OpenSplice DDS provides the solution for the need of distributed message logging. As a generic package, MessageFacility cannot have a dependency on OpenSplice libraries. For the balance between the functionality and the portability, a plugin manager has been implemented in the core MessageFacility library to allow additional functionalities with predefined interfaces, such as the DDS distributed message logging facility, or other special destinations that might be introduced in the future, to be loaded at the runtime. The core MessageFacility library shipped only has the facility of logging to files and standard streams, while the remote-logging facility is provided in the extension package.

By providing distributed message logging facility to the MessageFacility package, it is now an inter-process message logging and receiving facility. More information, such as the host name of the issuing machine, ip address, process id and application name, need to be extracted and added to the message header to make distinguish between message issuers.

In addition to distributed message logging, certain facilities are also needed for receiving, viewing, and archiving the messages issued by network peers. *msgserver* is an application that archives received messages, and *msgviewer* is a portable GUI application to inspect and view the message flow on the network from any participant node. As shown in Figure 3, the tool also provides message filtering tools based on the message severity level, category, application name and the host name of the issuing machine, to help capture the most wanted messages.

3.3. Hierarchical Configuration Model with FHiCL Language

MessageFacility is highly flexible and configurable in customizing the behavior of message logging. The main purpose of the configuration is to provide effective and flexible ways for

users to tune the message filtering and throttling down to a fine level, which requires a robust configuration model and a capable, human-friendly configuration language for that.

In MessageFacility, message filtering and throttling are based on the message severity and category by adjusting the following parameters: (a) *Threshold*, where a message with a severity level lower than a given threshold shall be ignored; (b) *Limit*, for a message category the logger ignores messages after some number (the "limit") have been encountered (limit of zero will disable reporting that category of messages); (c) *Timespan*, meaning if no occurrences of that type of message are seen in some number of seconds (the "timespan"), then the count toward that limit is to be reset.

MessageFacility uses FHiCL language for interpreting user written configurations. FHiCL (Fermilab Hierarchical Configuration Language), developed by the CET group in Computing Division at the Fermilab, is a JSON-like (JavaScript Object Notation) language which defines a series of *name* and *value* pairs, mainly to be used in system configuration. The language is well capable of describing hierarchical and heterogenous structures, while keeping the grammar simple and human readable.

4. Conclusions

MessageFacility as a stand alone distributed message reporting and central logging library has been successfully deployed and used in the NOνA experiment on various architectures and platforms ranging from x86 servers to embedded PowerPC with limited memory and CPU resources, since early in the development phase of the online DAQ system. The package is proven to be a reliable and flexible tool being used in wide range of applications including error reporting and handling, daily run logging, and debugging in the development and integration tests, etc.

More recently, MessageFacility has been adopted as an underlying library for a generic framework project called ART developed by the CET group in the Computing Division at Fermilab. Potential users of the ART framework, including Mu2e and NOνA offline, will also benefit from the message logging package shipped with ART.

For the near future, code maintenance and bug fixing are the main tasks for the MessageFacility package. Major design changes will be focused on two items: the first is the possibility of dropping support of the single thread mode, instead running the service in multithread by default; the second is to consider eliminating or blurring the boundary between the client side and the server side of the message service, as the concept of "Message Service" is now vague and not necessary for the stand alone package. Both would be expected to make the library smaller and more robust.

Acknowledgments

The MessageLogger package and its predecessor ErrorLogger was developed and maintained by Mark Fischler, Walter Brown, et. al. I thank them for all the discussion we had that helped me understand the design and details of the MessageLogger package. I also thank Marc Paterno, Jim Kowalkowski, Mark Fischler for discussions on the design of the new configuration model. Last but not least, I acknowledge Kurt Biery and all NOνA online DAQ software team fellows for inputs and feedbacks from field tests and daily use of the MessageFacility package in the NOνA online system.

References

[1] Jie C, Walt A, Matt B, Danjin W and William W III 1997 *Proc. of ICALEPCS* (Beijing) p 358
[2] *https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuideMessageLogger*, Retrieved on 05/12/2011
[3] *https://www.rsyslog.com*, Retrieved on 05/12/2011