A REMOTE TRACING FACILITY FOR DISTRIBUTED SYSTEMS

F. Ehm, A. Dworak, CERN, Geneva, Switzerland

Abstract

Today, CERN's control system is built upon a large number of C++ and Java services producing log events. In such a largely distributed environment these log messages are essential for problem recognition and tracing. Tracing is therefore vital for operation as understanding an issue in a subsystem means analysing log events in an efficient and fast manner. At present 3150 device servers are deployed on 1600 diskless frontends and they send their log messages via the network to an in-house developed central server which, in turn, saves them to files. However, this solution is not able to provide several highly desired features and has performance limitations which led to the development of a new solution. The new distributed tracing facility fulfils these requirements by taking advantage of the Streaming Text Oriented Messaging Protocol (STOMP) and ActiveMQ as the transport layer. The system not only allows storing critical log events centrally in files or in a database but also allows other clients (e.g. graphical interfaces) to read the same events concurrently by using the provided Java API. Thanks to the ActiveMQ broker technology the system can easily be extended to clients implemented in other languages and it is highly scalable in terms of performance. Long running tests have shown that the system can handle up to 10.000messages/second.

INTRODUCTION

The Controls Middleware (CMW) project was initiated at CERN [1] in 2006 to provide a unified communication middleware for operating the particle accelerator infrastructure. This includes communication with servers that directly operate hardware sensors and actuators. It enables for example operators in the CERN Control Centre (CCC) to control equipment remotely via in-house developed Graphical User Interfaces (GUI).

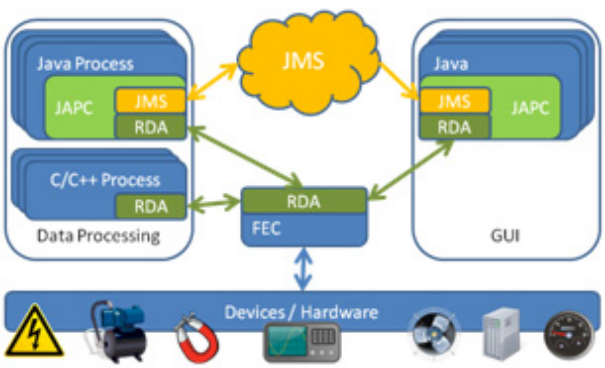

Figure 1: A simplified architectural overview on CERN's control system middleware components.

As illustrated in Figure 1 the CMW middleware is formed by a CORBA-based solution (RDA libraries) and a JMS based messaging infrastructure. Both cover similar use cases but they target at different system environments. JMS is used for high level middleware services and RDA for low level services running on hardware where very low message latency is demanded. The RDA library for example, is required by around 4000 Front End Servers (FES) running on 1600 Front End Computers (FEC) to allow getting/setting of hardware parameter values remotely.

Tracking potential issues in the middleware communication library layer is not an easy task as the involved services are distributed among many machines. In fact, there are two main problems:

- FECs are diskless systems running a real time LynxOS [2] operating system. Because there is no storage media it is not possible to write log messages to a local file.
- For other services which do have local disk storage it is difficult to correlate log events among them because files are distributed on various machines. Hence, accessing the data is a time consuming manual operation and slows down problem analysis.

Therefore, in the early days of CMW the idea of having a remote tracing facility was initiated and implemented. It allows collecting log events coming from RDA servers and from a subset of the middleware services.

THE CURRENT SYSTEM

In the current system log events are generated by the FECs and sent via UDP to a Java based central log server in plain text format. This server in turn converts the messages and writes them to a size-based rotating file.

Next to this, the CMW Admin GUI connects to the server and displays the log events received. It also allows setting a new log level via the RDA library remotely.

It is important to mention in this context that the services on FECs are hard real time processes which should be affected as little as possible by any other activity running on the same machine. Therefore, the communication between the CMW modules and the central log server is based on UDP as it is non-blocking.

Since the initial deployment this setup has been very useful for problem analysis and tracing. In particular the fact that a GUI enables a fast and easy information access makes it crucial for operation.

However, the demands have changed over time and the following problems have been identified:

- When setting a new log level in the CMW module all connected CMW Admin GUIs receive messages from the new level.

- Messages are lost due to a single threaded message acceptor in the central server.
- All messages go to one size based rotating file. When one source sends many events there is the risk that messages from other sources will be dropped.
- More messages need to be handled by the system since more services than originally planned are deployed.
- The C++ library only allows sending data to a remote host but not to a local file or to console output.

As a result of these problems it was decided to review and to implement a new central log event service.

REQUIREMENTS

As a first step towards a future solution the following requirements and limitations were defined:

- The new system must support log events from Java and C++ sources. Other languages may be supported.
- For C++ programs a lightweight library with little dependencies providing an easy API for storing events locally and sending them remotely should replace the old one.
- Operation overhead should be as low as possible and sufficient monitoring information must be provided.
- TCP for reliable and UDP for unreliable transport should be supported.
- Backward compatibility to legacy sources must be guaranteed.
- Support for precise timestamp information like microseconds and time zone where applicable.
- The log viewer must be provided as a standalone application, as well as a module which can be integrated into existing Java based diagnostic tools.
- Users should be able to read log messages remotely from a graphical user interface or via a console command *online* and *offline*. Online means that events are passed immediately to the client whereas offline means that the client has to actively poll for new events.

Restrictions

It should be mentioned that restrictions for the new system have also been defined:

- No operation critical system should depend on the distributed log event service.
- Messages may be lost due to inherent characteristics of the UDP transport layer.
- Only services running unattended are targeted to be used. This excludes highly dynamic user programs such as GUI's or console commands.

EXISTING MARKET SOLUTIONS

Collecting and storing log events from processes running on a Linux/Unix system is a well explored area and standardized for example via the widely used *syslog* protocol [3]. Here, processes send messages to a service running locally. The default for most Linux derivatives is *syslogd* which stores these events to local disk or forwards them to a remote syslog service via UDP. Other implementations like *rsyslogd*, *msyslogd* or *syslog-ng* exist and provide more features than syslogd. In addition, there are commercial and non-commercial tools to do analysis on the collected data. They allow an easier access via graphical interfaces (e.g. *Chainsaw* [4]) and partly provide extensive search and trending capabilities (e.g. *LogZilla* [5] and *Splunk* [6]).

Why Can We Not Take Existing Solutions?

As for the acquisition layer some of the present syslog implementations cover parts of the requirements but the main problem resides within the protocol itself. It contains limitations such as the missing support for microseconds and the maximum payload length of 2048 Bytes. Apart from this, none of the present syslog libraries supports setting the log level remotely

As for the C++ logging library there are many solutions available. However, most of them are in an early phase of development, do not support fully the requirements, are not stable enough or their dependencies are not supported on the current FECs.

Most of the existing GUI applications which are able to read events online are language dependent and connect directly to the log source. However, the new system requires reading log events from sources implemented in various languages and forbids direct connection.

Products like LogZilla or Splunk operate offline and provide fully featured tools for analysis and charting. However, they are proprietary solutions and do not allow to be integrated into existing in-house developed tools.

THE NEW ARCHITECTURE

Because there is no existing solution which satisfies the previously listed requirements a new central log event service has been implemented in Java and C++. As Figure 2 shows the *Converters* are responsible for accepting incoming log events, converting them into an internal common message format and sending them to the messaging service where they again can be read out by a *LogReader* or by a simple console reader.

Additionally, selected data may be stored permanently into a database or into local files.

The Log Sources

A program which requires sending data to the central service has two possibilities to do so: the first is the Linux standard syslog protocol and the second the newly developed *CMWlog*. Both can be used from C++ and Java. The main difference is that CMWlog supports microseconds and time zone information as well as TCP and UDP. It facilitates the STOMP [7] message frame and hence, is clear text based. STOMP is an open source protocol chosen for many messaging solutions [8] from various vendors and has gained great popularity in recent

years. Because it is so largely supported it was chosen for this project.

To allow Java based systems without the introduction of a large dependency set a new extension for the widely used Log4J [9] framework has been written.

The New C++ Log Library

CMWlog is a simple but full-featured logging library for C++ programs which - to a reasonable extent - resembles the Log4j Java library, providing flexible logging to files, to network, or to other destinations. It was developed to replace the old CMW logging library supports several platforms such as Linux and Windows but also old PowerPCs running LynxOS.

The library is fully configurable from the code or via configuration files. Like the previous version it supports setting the effective log level remotely. At the same time CMWlog is written to be thread-safe and it represents small memory footprint and resource consumption which is extremely important for previously mentioned old systems. Another important feature is the non-blocking method calls from the running user thread and proper behaviour in case of I/O errors not to affect time critical user's code execution.

The library utilizes concepts like *Loggers* and the *Appenders* [9]. These allow a high flexibility when channelling log events to different destinations such as standard output, files, or sending to a remote host. If required, the modular architecture of the library facilitates to be extended to other output destinations.

The Converters

A Converter is protocol-specific and hence accepts log messages only from particular remote sources. Internally, it is implemented using the Apache Camel [10] open source routing framework which features a very rich set of functionalities helping to reduce the development massively. It handles socket and thread operations and passes the received content to the user code. Here, the data is then turned into an internal common log message and subsequently sent off to the broker via TCP.

Currently, there are three converters. The first receives messages following the RFC5424 syslog format. The second reads messages coming from the new CMWLog log sources and the last one ensures backward compatibility with old legacy sources.

It is very simple to add new converters and thus to extend the usage to other log protocols. A concrete example for this is to accept also binary objects coming from Java programs using the Log4J or SL4J [11] log library.

To ease the operation monitoring metrics as well as management features such as blacklisting of log sources are exposed via the JMX [12] interface.

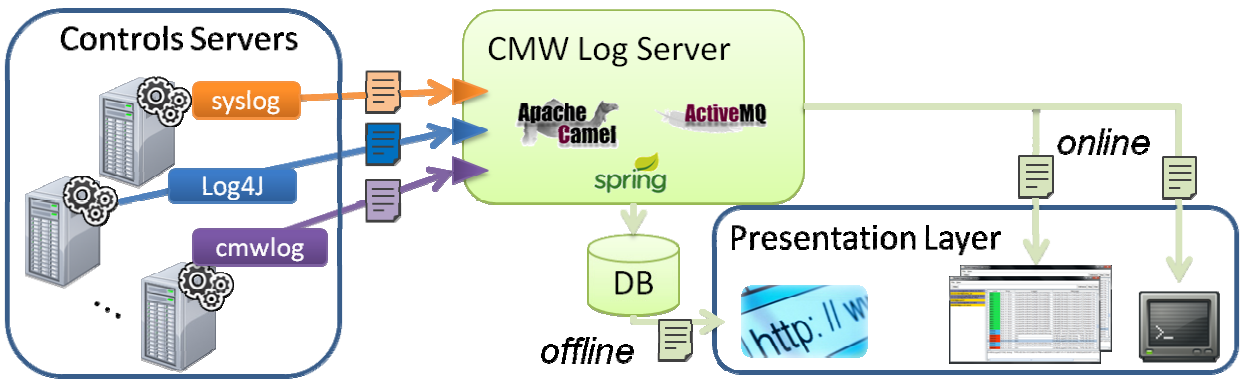

Figure 2: Architectural Overview of the new CMW tracing service.

Distribution of the Data

After a Converter have accepted and converted a log event it is then sent via JMS to a central messaging bus provided by the Apache ActiveMQ [13] product. ActiveMQ supports failover and clustering mechanisms, and has been used in the CERN controls system since 2005. As it has proven to be very reliable it was a natural choice for distributing log events to a potentially large set of client applications.

Data arriving at the broker is organized internally in topics. That is, each log source is identified globally by a unique identifier which maps in the broker to one specific topic. Clients may then read messages from one or more log sources by subscribing to one or more topics.

Storing Log Events Permanently

In order to fulfil the requirement of storing log events into a permanent storage two independent modules have been added to the system (see Figure 2). The *FileWriter* writes into size-based rotating files so disk space is not exceeded. At the same time the *DbWriter* pushes the data into a database (i.e Oracle or MySQL). In order to reduce storage usage, project specific policies on how long the data need to be kept have been put in place. For the CMW project, for example, data with warning and error severity are stored for one month only.

The Log Viewer

To finally view log events online and in real-time a graphical user interface (GUI) has been created. It displays the loggers in a tree structure and allows fine control of the desired levels. The messages are displayed in a table using a different colour for each severity.

The interface can run as a standalone application or as an integrated component within other GUIs. Moreover when connected to an RDA server it can remotely change the log level of the source easing on-the-fly problem tracing.

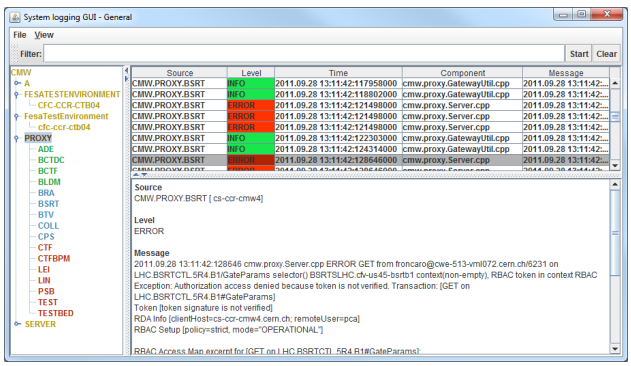

Figure 3: Example of the CMW log event viewer.

In addition, data can be viewed offline via an oracle driven web interface which provides sophisticated data filtering and selection tools. In case of an incident like a system or service crash this information can be very useful for later analysis. Another example is comparing trace messages produced by different sources at a given time with messages from another time. This gives the possibility to identify relationships between incidents.

PERFORMANCE TESTING

The new system has been deployed and tested in the very early stages of development. The focus was put onto the messaging broker as it handles most of the workload. For this, an ActiveMQ 5.4 message broker was installed on a 16 Core HP G6 server with 8GByte Memory. In order to simulate future workload two producers where set up on different machines sending in total 10.000 log events per second at constant rate via 1000 connections (TCP) to the broker. Each of these events contained a payload of 500 Bytes.

On the reading side 15 Java client applications where started each subscribing to a subset of the published data. Because one message is multiplied by the number of subscribers the effective outgoing message rate resulted in 30.000 messages per second. These performance numbers are slightly above the estimated future load to allow a better dimensioning of the system so bursts are handled safely, as well.

This test setup ran for 48 hours without any problems showing that ActiveMQ handles reliably the simulated load.

EXTENDING TO OTHER FIELDS

Because of its flexibility and simple use the new system has been investigated for transporting and storing important deployment and configuration feedback messages coming from kernel modules running on FECs. This feedback channel is currently missing but highly demanded. Developers have no means to verify the correct installation or reconfiguration of those modules.

Therefore, feedback messages were tested to be sent to the CMW tracing system and subsequently inserted into the database. This was successful and confirmed that the proposed system is a mature solution.

The same channel could possibly also be used for normal Java or C++ programs services. Information about the version, installation time and configuration could be collected centrally in a database for further use like viewing deployment history.

SUMMARY

The new tracing system fulfils all requirements and has been tested for performance and stability. It is designed to serve multiple reading applications for many equipment or service experts without putting additional load onto the log source and it offers to store the log events in a database or size-based rotating files. Users have the possibility to read messages online via a graphical user interface which was integrated into existing operation and diagnostic tools in the CERN control system. As data is also stored in a database the usage of existing market solutions for analysis is possible.

Because it allows being easily extended to other log protocols like Log4J or SNMP it offers an interesting solution to a larger field of activity.

REFERENCES

[1] CERN, www.cern.ch
[2] LynxOS, A real time operating system, http://www.lynuxworks.com
[3] The syslog standard, IETF, http://tools.ietf.org/html/rfc5424
[4] Chainsaw, Apache Log4J LogViewer, http://logging.apache.org
[5] LogZilla, http://www.logzilla.pro
[6] Splunk, http://www.splunk.com
[7] STOMP, Streaming Text Oriented Messaging Protocol, http://stomp.github.com
[8] Enterprise Messaging Solutions Technical Evaluation, Lionel Cons, Massimo Paladin, CERN
[9] Apache Log4J, A Log Library for Java programs, http://logging.apache.org/log4j
[10] Apache Camel, An enterprise routing framework, http://camel.apache.org
[11] SLF4J, Simple Logging Facade for Java, http://www.slf4j.org
[12] JMX, Java Management Extension, Oracle : http://www.oracle.com
[13] ActiveMQ: http://activemq.apache.org