Available on CMS information server

CMS CR -2012/124

The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

31 May 2012 (v2, 07 June 2012)

Distributed error and alarm processing in the CMS data acquisition system

G Bauer[6], U Behrens[1], M Bowen[2], J Branson[4], S Bukowiec[2], S Cittolin[4], J A Coarasa[2], C Deldicque[2], M Dobson[2], A Dupont[2], S Erhan[3], A Flossdorf[1], D Gigi[2], F Glege[2], R Gomez-Reino[2], C Hartl[2], J Hegeman[2,a], A Holzner[4], Y L Hwong[2], L Masetti[2], F Meijers[2], E Meschi[2], R K Mommsen[5], V O'Dell[5], L Orsini[2], C Paus[6], A Petrucci[2], M Pieri[4], G Polese[2], A Racz[2], O Raginel[6], H Sakulin[2], M Sani[4], C Schwick[2], D Shpakov[5], M Simon [2], A C Spataru[2], K Sumorok[6]

Abstract

The error and alarm system for the data acquisition of the Compact Muon Solenoid (CMS) at CERN was successfully used for the physics runs at Large Hadron Collider (LHC) during first three years of activities. Error and alarm processing entails the notification, collection, storing and visualization of all exceptional conditions occurring in the highly distributed CMS online system using a uniform scheme. Alerts and reports are shown online by web application facilities that map them to graphical models of the system as defined by the user. A persistency service keeps a history of all exceptions occurred, allowing subsequent retrieval of user defined time windows of events for later playback or analysis. This paper describes the architecture and the technologies used and deals with operational aspects during the first years of LHC operation. In particular we focus on performance, stability, and integration with the CMS subdetectors.

Presented at *CHEP 2012: International Conference on Computing in High Energy and Nuclear Physics*

[1] DESY, Hamburg, Germany

[2] CERN, Geneva, Switzerland

[3] University of California, Los Angeles, Los Angeles, California, USA

[4] University of California, San Diego, San Diego, California, USA

[5] FNAL, Chicago, Illinois, USA

[6] Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

[a] Now at Princeton University, Princeton University, New Jersey,USA

Distributed error and alarm processing in the CMS data acquisition system

G Bauer[6], U Behrens[1], M Bowen[2], J Branson[4], S Bukowiec[2], S Cittolin[4], J A Coarasa[2], C Deldicque[2], M Dobson[2], A Dupont[2], S Erhan[3], A Flossdorf[1], D Gigi[2], F Glege[2], R Gomez-Reino[2], C Hartl[2], J Hegeman[2,a], A Holzner[4], Y L Hwong[2], L Masetti[2], F Meijers[2], E Meschi[2], R K Mommsen[5], V O'Dell[5], L Orsini[2], C Paus[6], A Petrucci[2], M Pieri[4], G Polese[2], A Racz[2], O Raginel[6], H Sakulin[2], M Sani[4], C Schwick[2], D Shpakov[5], M Simon[2], A C Spataru[2], K Sumorok[6]

[1] DESY, Hamburg, Germany; [2] CERN, Geneva, Switzerland; [3] University of California, Los Angeles, Los Angeles, California, USA;[4] University of California, San Diego, San Diego, California, USA; [5] FNAL, Chicago, Illinois, USA; [6] Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

[a] Now at Princeton University, Princeton University, New Jersey,USA

E-mail: Andrea.Petrucci@cern.ch

Abstract. The error and alarm system for the data acquisition of the Compact Muon Solenoid (CMS) at CERN was successfully used for the physics runs at Large Hadron Collider (LHC) during first three years of activities. Error and alarm processing entails the notification, collection, storing and visualization of all exceptional conditions occurring in the highly distributed CMS online system using a uniform scheme. Alerts and reports are shown on-line by web application facilities that map them to graphical models of the system as defined by the user. A persistency service keeps a history of all exceptions occurred, allowing subsequent retrieval of user defined time windows of events for later playback or analysis. This paper describes the architecture and the technologies used and deals with operational aspects during the first years of LHC operation. In particular we focus on performance, stability, and integration with the CMS sub-detectors.

1. Introduction

The Compact Muon Solenoid (CMS) [1] is a general-purpose particle detector at the Large Hadron Collider (LHC) [2] at CERN in Geneva, Switzerland. The CMS Data Acquisition (DAQ) [3] is responsible to build and filter events from about 600 data sources at a maximum trigger rate of 100 kHz. The error and alarm system has to provide the facilities to retrieve, process, store, and display errors and alarms occurred during the operation of the CMS experiment. The DAQ is composed of few thousand hosts [4][5] and of O(20000) interdependent applications that need to be monitored in near real-time. One of the critical success factors for an error and alarm management system is scalability, several dimensions of scaling requirements [6][7][8] (e.g. numerical and geographical) have been considered to cover all aspects of the system design.

The main purpose of the CMS alarm and error system is to alert the operator when an abnormal operation occurs, but also to keep history of all past problems for post mortem analysis. The presented design is based on four powerful concepts: only notify important conditions, notify in time, respond, and provide guidance. The suggested infrastructure fits these needs by providing a set of expandable and reusable solutions allowing use of the alarming system for development, test and operation scenarios.

2. Architecture and design

The architecture is based on the XDAQ Monitoring & Alarming Service (XMAS) [9], which provides several plug-ins specialized for specific tasks and services to support and implement a fully scalable distributed monitoring and alarming system. It is a service-oriented architecture, in which a 3-tier structured collection of communicating components cooperates to perform the monitoring, error and alarm management tasks. The XDAQ middleware provides the universal application connectivity and makes applications and services inter-communicating [10] [11].

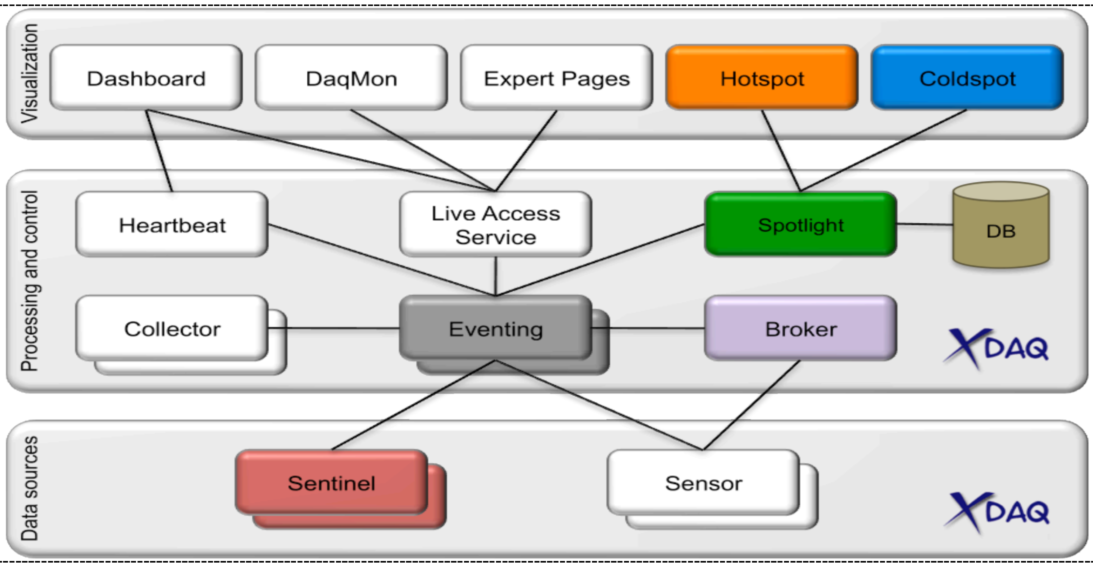

Figure 1. Diagram is showing the XMAS components used for reporting (Sentinel), collecting (Eventing), recording (Spotlight), and display (Hotspot and Coldspot) of errors and alarms.

As shown in **Figure 1**, the system builds upon a scalable publisher-subscriber [12] service consisting of a pool of eventing applications orchestrated by a load balancer application (broker).

When a DAQ application detects an anomaly it acts as data producers through sentinel services to publish an error or an alarm to the eventing service. The spotlight service is responsible for collecting, processing, and storing events. It subscribes to the eventing service specifying categories of error and alarm events it wants to receive. The spotlight supports two databases technologies: Oracle [13] for CMS production system and sqlite [14] for testing and small setups. The visualization components are called Hotspot and Coldspot, both applications are developed using Adobe Flash builder [15]. The Hotspot is used to visualize the error and alarm occurring in real-time using a graphical web interface. This application maps all occurring events and displays them according to a user-defined model of the system. The Coldspot allows querying spotlight through Query By Example [16] using a time window; this tool is used for post-mortem analysis.

As shown in **Figure 2**, two different report scenarios can be identified: applications that detect persistent deviations from the normal system behaviour report errors and deviations may also be transient, meaning an alarm is fired and eventually revoked when the asserted condition is resolved.

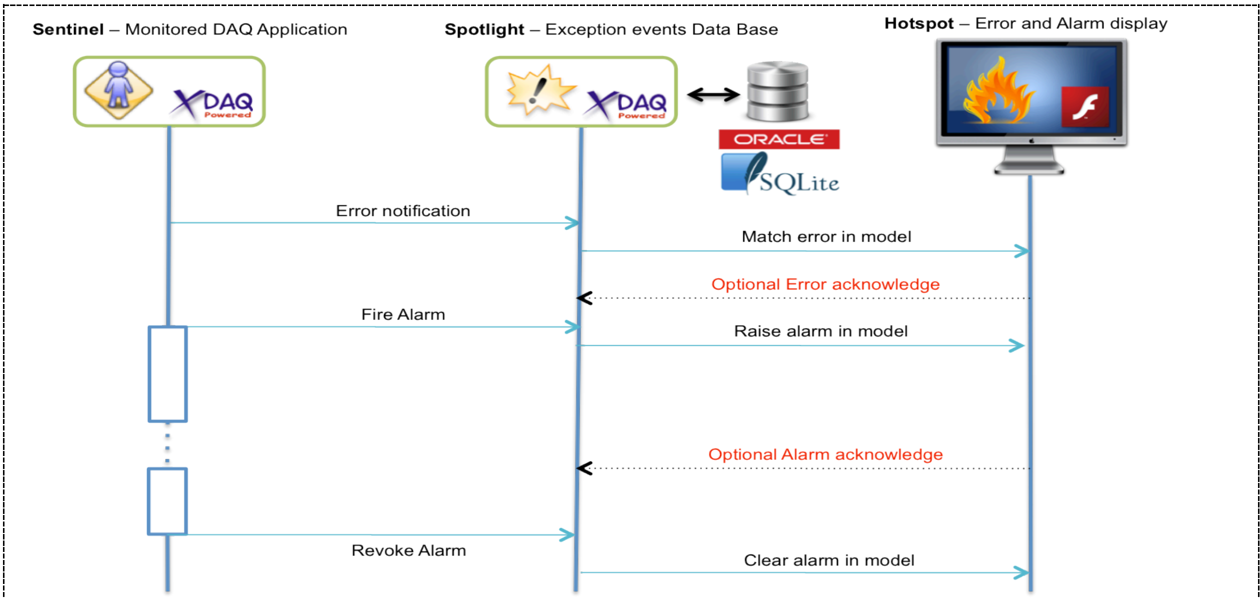

Figure 2. Error and alarm interaction diagram.

All services are re-locatable and run independently of each other without a need for external control. Communication among services is established through a rendezvous mechanism with the help of discovery services facilities [17].

3. Error handling

The XDAQ environment offers developers programmatic tools to deal with errors as they occur at runtime. A uniform approach is used to handle exceptions in every application.

```
// XCEPT_RETHROW macro
try {
    //....
} catch (std::exception& e) {
    XCEPT_RETHROW( xoap::exception::Exception,"Text message",e);
}

// XCEPT_RAISE macro
XCEPT_RAISE (xdaq::exception::Exception, "Text message");

// XCEPT_ASSERT macro

XCEPT_ASSERT (contextURL != "", xdaq::exception::BadCommand, "Text message");

//  XCEPT_DECLARE macro
// Create a new exception and use it as a variable called VAR
XCEPT_DECLARE (b2in::nub::exception::Exception, e , "Text message");


//  XCEPT_DECLARE_NESTED macro
// Create a new exception from a previous one and use it as a variable called VAR
try
{
    //....
}
catch ( b2in::nub::exception::Exception & e )
{
        XCEPT_DECLARE_NESTED (b2in::nub::exception::Exception, ex , "Text message",e);
}
```

Figure 3. Special macro instructions to throw an exception.

The exception handling mechanism available to C++ programs provides the basic ways of dealing with exceptional condition within an application throw "*try/catch*" and "*throw*" statements. The "*throw*" statement is replaced by special macro instructions (see **Figure 3**) that allow automatically building exception objects with run-time and compile time information. The mechanism is further enhanced to provide an additional "*re-throw*" statement that allows to build the full sequence of subsequent "*throw*" of exceptions namely the exception stack trace.

In this schema all errors and alarms are specified according to uniform approach by defining a C++ class that extends the standard "*xcept::Exception*", as shown in **Figure 4**. A "*xcept::Exception*" object encapsulates information specific to an error occurring in the distributed environment, e.g. the stack, subsystem, pc, application identifier etc. An exception can be thrown several and propagate to the function callers till the last function invoked that is willing to catch this exception and performs some recovery or reporting actions.

```
#ifndef _jobcontrol_exception_Exception_h_
#define _jobcontrol_exception_Exception_h_

#include "xcept/Exception.h"

namespace jobcontrol {
   namespace exception {
      class Exception: public xcept::Exception
      {
         public:
         Exception( std::string name, std::string message, std::string module, int line, std::string function ):
            xcept::Exception(name, message, module, line, function)
         {}
         Exception( std::string name, std::string message, std::string module, int line, std::string function, xcept::Exception & e ):
            xcept::Exception(name, message, module, line, function, e)
         {}
      };
   }
}
#endif
```

Figure 4. Declaration of an error for jobcontrol application.

3.1. Sending error messages

Reporting to external clients is eventually performed by calling the "*notifiyQualified*" method from within any application. **Figure 5** shows how to catch an error and append the exception trace to a newly created exception that will be reported.

```
try
{
     ... some operation ...
}
catch (toolbox::exception::Exception& exceptionObj)
{
    std::stringstream msg;
    msg << "An error has happened that cannot be handled by the application";
    XCEPT_DECLARE_NESTED(xmas::sensor::exception::Exception, errorObj, msg.str(), exceptionObj);
    this->notifyQualified("fatal", errorObj);
    return;
}
```

Figure 5. Example how to send an error after catching an exception.

3.2. Alarm messages

Alarm objects are treated similarly to ordinary errors with the exception that alarms can be transient and therefore revoked by the originator within a given period of time. This means that any instance of "*xcept::Exception*" object or a derived class can be used to represent an alarm, as shown in **Figure 6**.

```
#ifndef _sentinel_tester_exception_TestAlarm_h_
#define _sentinel_tester_exception_TestAlarm_h_

#include "sentinel/exception/Exception.h"

namespace sentinel {
   namespace tester {
      namespace exception {
         class TestAlarm: public sentinel::exception::Exception
         {
            public:
            TestAlarm( std::string name, std::string message, std::string module, int line, std::string function ):
               sentinel::exception::Exception(name, message, module, line, function)
            {}

            TestAlarm( std::string name, std::string message, std::string module, int line, std::string function,
                  xcept::Exception& e ):
               sentinel::exception::Exception(name, message, module, line, function, e)
            {}
         };
      }
   }
}
#endif
```

Figure 6. Example of an alarm for an XDAQ application.

Such an alarm object can be given to the Sentinel application component within an XDAQ executive container (context). Sentinel then takes care of routing this alarm through a publisher/subscriber component to the Spotlight error server that persistently stores the alarm and serves it to other clients. Alarms are resent periodically by the Sentinel application component until the alarm is revoked.

```
#include "sentinel/tester/exception/TestAlarm.h"
...
xdata::InfoSpace* is = xdata::getInfoSpaceFactory()->get("urn:xdaq-sentinel:alarms");
std::stringstream msg;
msg << "A test alarm has been emitted";
XCEPT_DECLARE(sentinel::tester::exception::TestAlarm, ex, msg.str());
sentinel::utils::Alarm * alarm = new sentinel::utils::Alarm("warning",ex, this);
is->fireItemAvailable ("test-alarm", alarm);
 xdata::InfoSpace* is = xdata::getInfoSpaceFactory()->get("urn:xdaq-sentinel:alarms");
 sentinel::utils::Alarm * alarm =  dynamic_cast<sentinel::utils::Alarm*>(is->find("test-alarm"));
 is->fireItemRevoked("test-alarm", this);
 delete alarm;
```

Figure 7. Example how to emit and revoke an alarm object.

Figure 7 shown how to emit and revoke an alarm object. Before enabling an alarm the first action is to retrieve an infospace called "alarms". This infospace remembers all fired alarms locally and is used to re-emit the alarms. It is the communication interface between XDAQ applications and the Sentinel subsystem for alarms. Using *"XCEPT_DECLARE"* macro an instance of the previously defined alarm class is created. Next the alarm that derives from an "*xcept::Exception*" and therefore represents an error object is embedded in a "*sentinel::utils::Alarm*" object. It is possible to provide the severity (warning, error, and fatal) of the alarm as the first parameter. The second parameter is the error object to be embedded and the third parameter points to the XDAQ application that is the creator of this alarm. Then the alarm is fired into the infospace by giving it an unique name, test-alarm in our example. Revoking an alarm works by retrieving the previously raised alarm from the infospace by the alarm's name ("test-alarm"in our example) and the ownership of the alarm stays with the application. Thus, the application has to take care of deleting the alarm object, either by keeping a reference to it or by retrieving it from the infospace. After you have retrieved the alarm, it is revoked from the infospace. That will tell *Sentinel* to emit a *revoke* message to the spotlight server and to stop sending the alarm over and over.

3.3. Communications

All error and alarm objects are serialized into SOAP messages for communication over the network according to the XSD schema as defined in **Figure 8** that describes the XML content to be embedded in a message. A binary serialization according the same semantic schema is also supported when high performance and scalability are required.

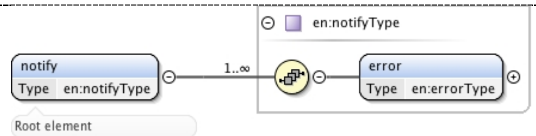

Figure 8. XML schema for **notify** SOAP message.

As shown at the top of **Figure 8** the ***notify*** message contains one or more errors, the ***error*** schema has attributes and elements to describe a generic error (see **Figure 9**) and can include one or more nested errors. The most important fields for the schema are the following:

- *identifier* is an identification of the exception class (e.g. jobcontrol::exception::JobCrash);
- *notifier* is the originator of this error message;
- *dateTime* is the time when the error occurred;
- *severity* is the level of error gravity (e.g. warning, fatal, and error);
- *message* is a textual description of the error details;
- *tag* is a special parameter. It carries application-level, end-to-end agreed information that allows to efficiently display the information on the user interface side;
- *uniqueid* is universally unique identifier (UUID [22]) for identifying an instance of an error.

The Error schema contains the *any* tag that permits the extension of the schema with user specific error information allowing the possibility to reuse the generic part of the schema maintaining the core error information. For the CMS data acquisition system the standard error has been extended with an additional schema: ***ApplicationErrorRecordGroup*** for XDAQ application software. This schema (see **Figure 10**) defines eight additional fields:

- *class* is the type or name of the described application (the fully qualified classname, e.g. ns1::ns2::MyClass);
- *context* is an URL of the applications container (e.g. http://HOST:PORT);
- *group* is the comma separated list of groups to which an application belongs;
- *id* is a numeric identifier for the application, for XDAQ the "lid" that identifies the application within its executive container;
- *instance* is the object instance of the application class;
- *service* is a textual identifier of the service that this application provides;
- *uuid* is the uniform unique identifier of this application instance;
- *zone* is the zone to which this application belongs, i.e. the partition identifier when multi-partitions are used.

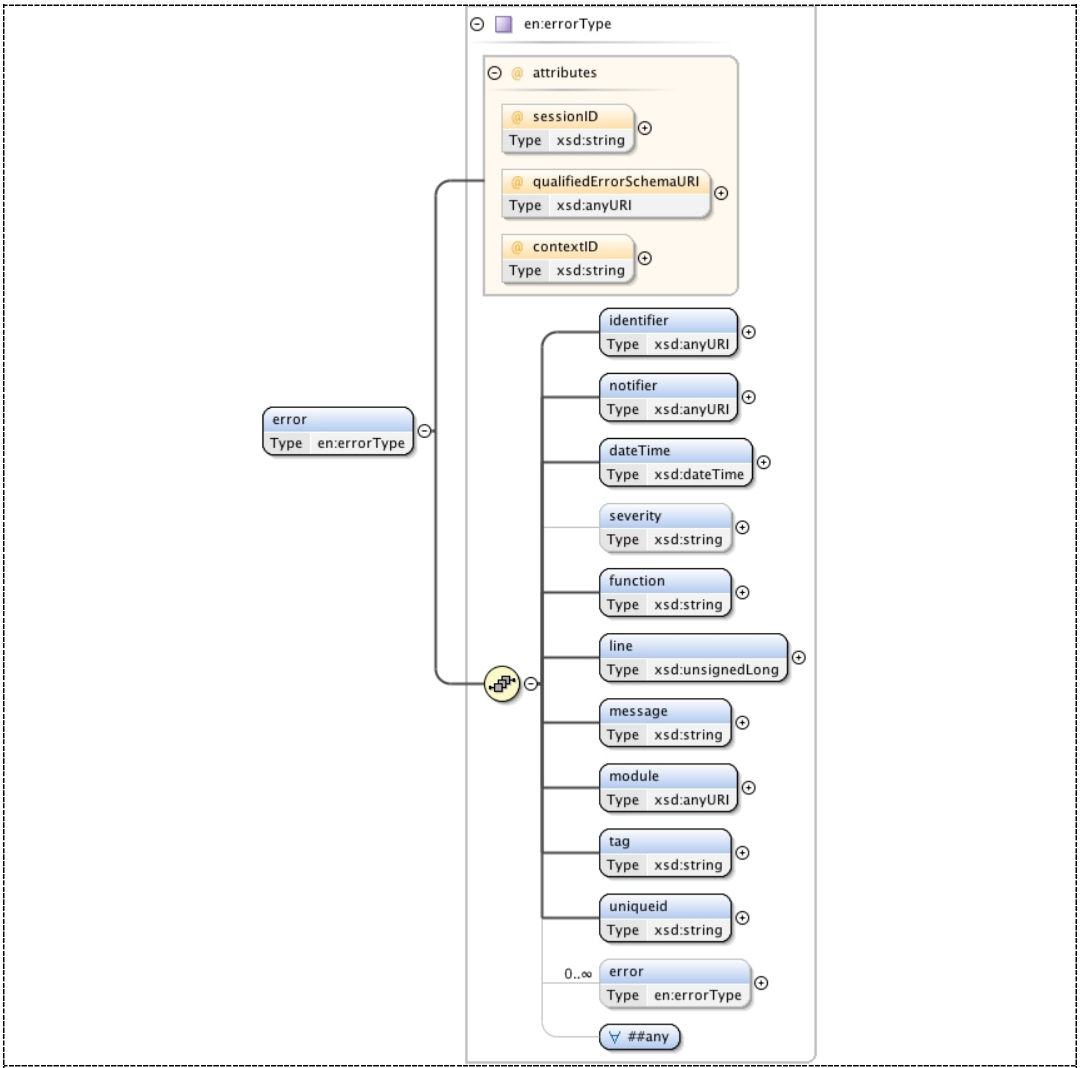

Figure 9. XML schema for describing the ***error*** type in notify message (c.f. **Figure 8**).

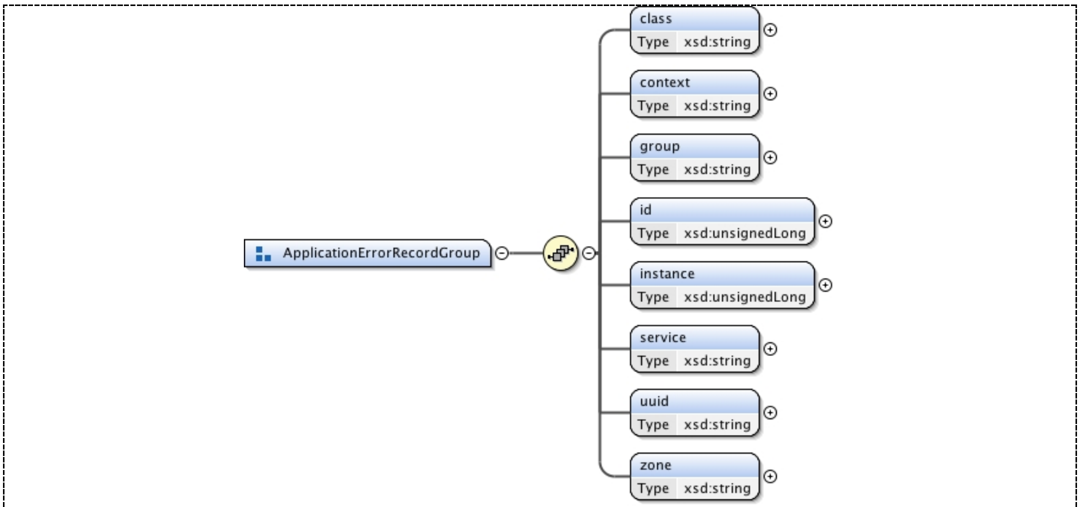

Figure 10. XML schema for ***ApplicationErrorRecordGroup*** SOAP message.

4. Visualization
In the CMS error and alarm system there are two different visualization scenarios to support the operation of the CMS experiment: an online display and a post-mortem display. The CMS operators use the online display during data taking to detect deviations from the normal system behaviour. The CMS experts are able to analyze past errors using the post-mortem display.

4.1. Online display model and graphics
The system model provides an abstract view of the system. Several perspectives can be used to represent different views of the system. A view can be defined as a recursive hierarchical structure to fulfill the system required. In order to match errors reported by the running system with the abstract model a number of filters are defined. Filters are associated with views by means of special nodes; named "filter nodes". These nodes define regular expressions matching the attributes from the error report.

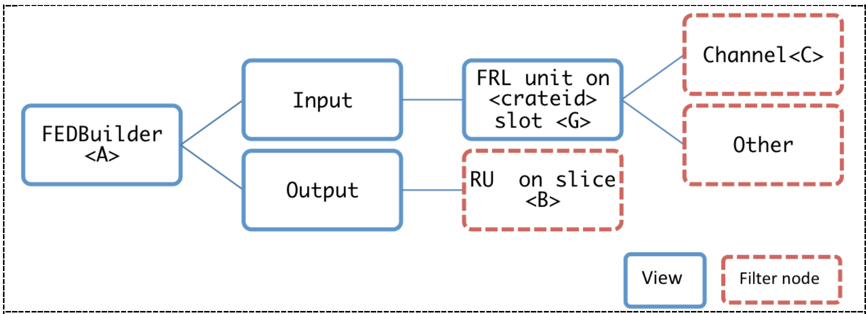

Figure 11. FEDBuilder class model. Blue boxes are different views instead the dashed red boxes are filter nodes used by connected views.

Figure 11 shows the FEDBuilder class tree model structure. The FEDBuilder refers to a set of applications that are logically grouped together. In this model the association between fedbuilder <A> and <crateid> can change in each configuration of the DAQ system. Therefore a model is derived from the configuration database. The following attributes (see **Table 1**) must be defined to intercept errors from all applications participating in the FEDBuilder task, they are used to intercept and count the exceptions and map them to the system tree model.

Table 1. Fedbuilder attributes.

Name	Group	Tag
Channel <C>	unit_frlcrate, crate_<crateid>	geoslot_<G>, channel_<C>
RU on slice <S>	unit_ru,fb_<A>, slice_<S>	
Other	unit_frlcrate, crate_<crateid>	

Hotspot is an Adobe Flex [18] application used to view and retrieve the error and alarm data stored and retrieved by Spotlight. Hotspot displays errors and alarms according the CMS data acquisition abstract model system. Errors and alarms are associated to elements of the system model and displayed corresponding to their severity levels. The tool offers different views of the model such as tree navigation, heat maps, and tables. **Figure 12** shows the main view of Hotspot, the left-most panel of the main view represents the model tree that categorizes errors into nodes within the tree. The nodes are categorized under labels such as 'Fedbuilders', 'FEDs', 'Slices', 'Infrastructure', and so on. These categories are represented as tiles on the right-most panel in the Heatmap. Each tile on the heatmap displays the name of the category it represents, its associated icon and, if existing, any exceptions associated with it.

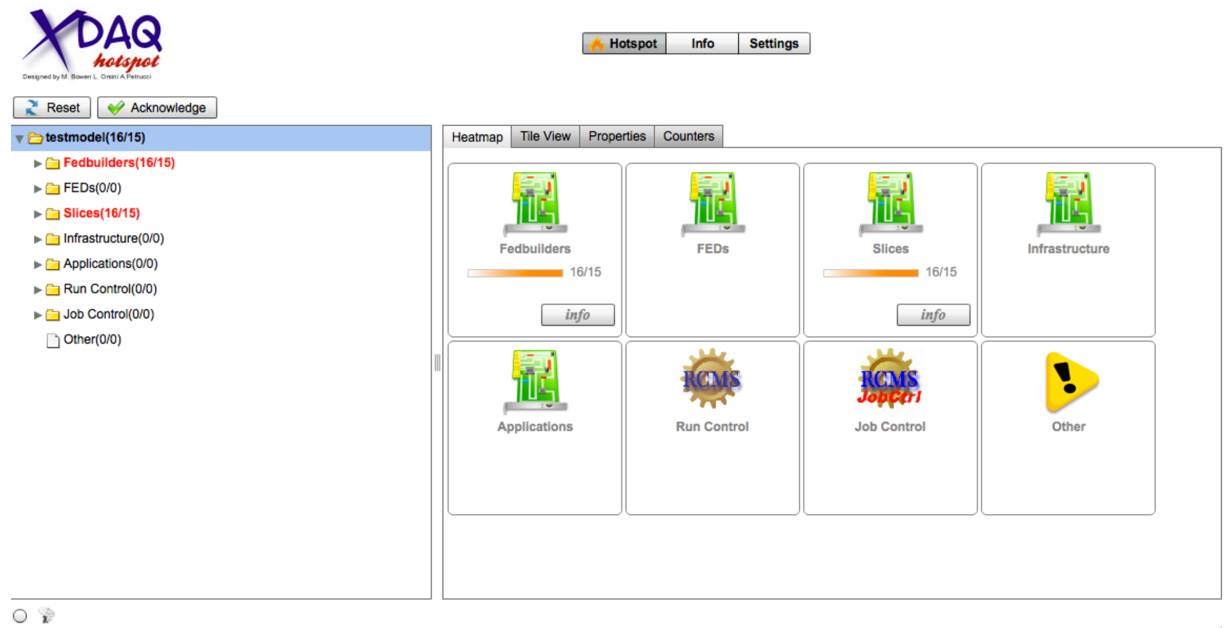

Figure 12. Main view of the Hotspot application.

The coloured bar represents the severity level of the most-severe exception contained, in our example there are Fedbuilder error exceptions and the bar is orange. Double-clicking on a heatmap tile brings the heatmap to focus on the nodes and subcategories within it allowing the user to navigate and refine his criteria for viewing exceptions. The user is also able to select elements on the model tree to accomplish the same effect. Selecting an element on the model tree and clicking "Acknowledge" allows the user to acknowledge (dismiss) all the exceptions categorized within that element. The "Reset" button removes all the exceptions being displayed on the screen and triggers a reloading of the model definition XML file – resetting the application to a state as if it had just been launched. In **Figure 12**, a fedbuilder::exception::DataCorruptionDetected error matches two different views: the "Fedbuilders" and the "Slices" perspectives.

Clicking 'Info' on a heatmap tile or clicking the 'Info' tab navigates to view the selected exceptions and their properties. **Figure 13** displays the exceptions selected under the previous Hotspot main view. This view allows the user to selectively acknowledge individual exceptions and identify their structure (as relating to parent and child exceptions), and their associated properties such as their severity, notifying application, error message etc.

4.2. Post-mortem analysis and graphics

Coldspot is an Adobe Flex application providing a post-mortem view of exception data stored by spotlight. The main feature is to retrieve the error through Query By Example using a time window.

Figure 14 shows the "Query" tab of the coldspot, this view allows the user to selectively choose each parameter to perform the query. The time window in which errors should be searched for can be specified by the controls at the top of the tab, by clicking the "submit" button the query is sent to the spotlight. The "display" tab displays the exceptions retrieved from the spotlight application with the same interface look and feel of the Hotspot "Info" tag (see **Figure 13**).

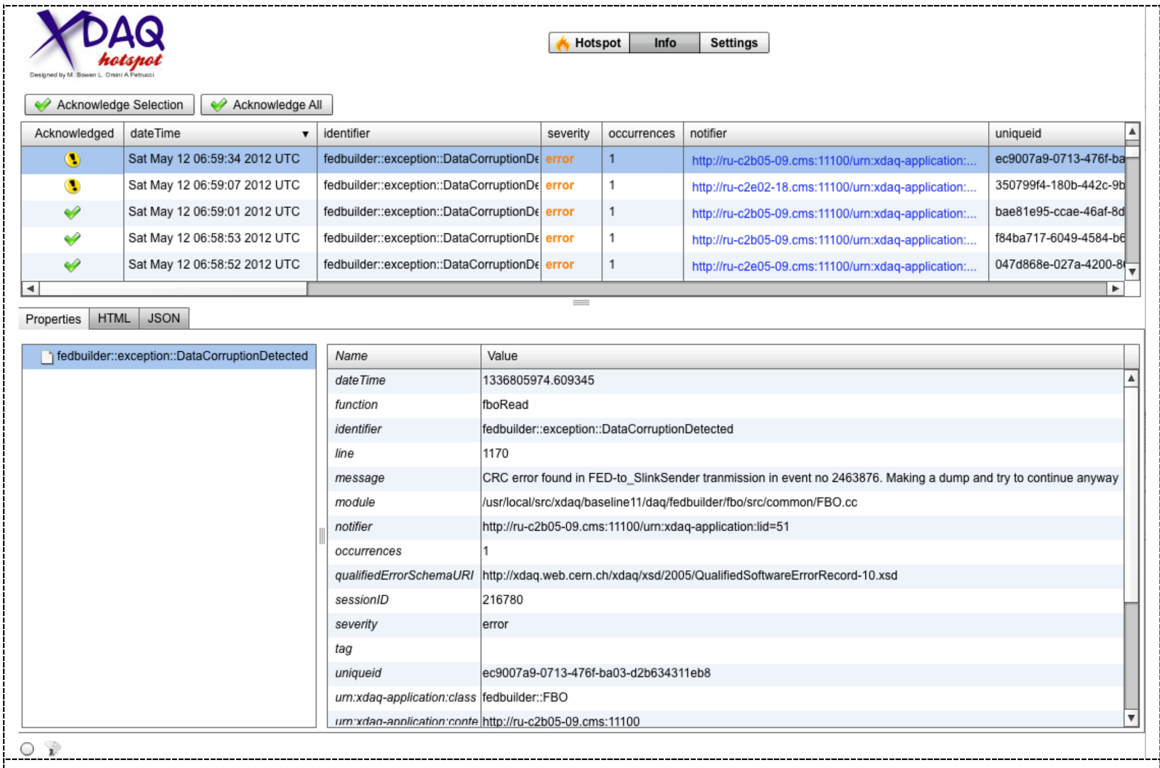

Figure 13. Hotspot application "Info" tab.

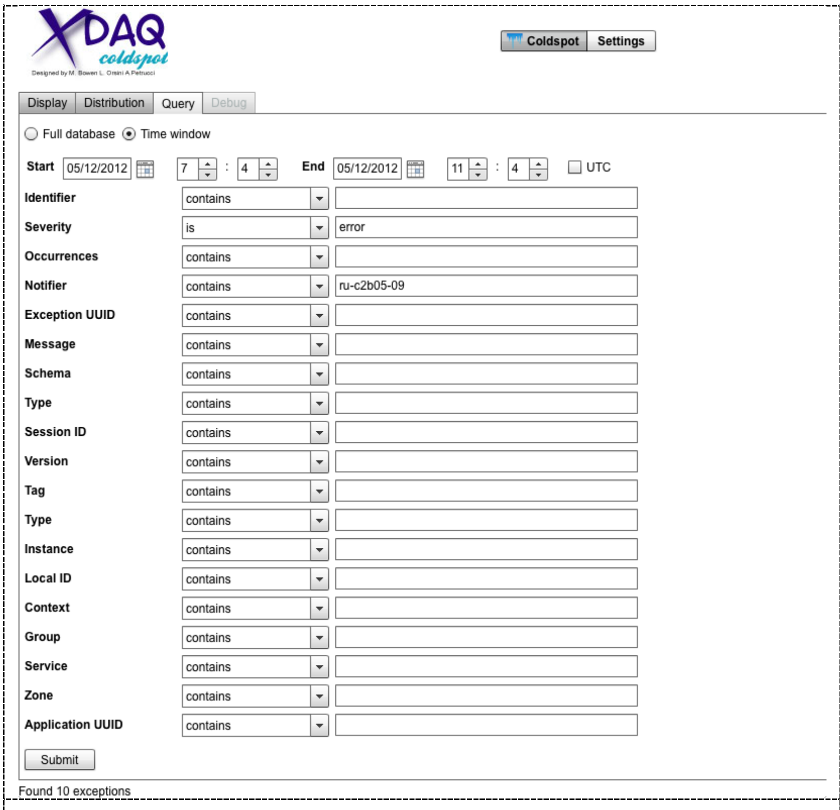

Figure 14. Coldspot application "Query" tab.

5. Persistency

The spotlight application is responsible for the storage and retrieval of error and alarm data. Three spotlight applications have been implemented to support different requirements:

- *Spotlight2g*, utilizes a database called SQLite to store the exceptions. SQLite is a in-process library that implements a self-contained, server-less, zero-configuration, and transactional SQL database engine. It is a very lightweight database, however it is unable to efficiently handle large datasets. To compensate for this, an archival system is required to record the data split across multiple database files. This was an early implementation for initial development as a proof of concept and now it is used for testing and small setups;
- *SpotlightOCCI*, making use of Oracle's OCCI API, is used to store this exception data in an Oracle database. The Oracle database has no upper-limit to the amount of data it can store and it has been shown to cope and operate smoothly with hundreds of thousands of records of exception data. The Oracle database is also a very widely used and robust RDBMS with thorough documentation, these features combining to make it a more suitable technology for recording the exceptions than the SQLite. This is used in the CMS production system.
- *SpotlightTT,* utilized an in-memory database called Oracle TimesTen [19]. This database is designed for low latency, high-volume data, event and transaction management. Because TimesTen's data is stored entirely in memory, no disk I/O operations are required when processing queries. Since memory access is much quicker than disk access, TimesTen is often used in situations that demand a fast and predictable response time, such as in telecommunications systems and financial transaction services. TimesTen supports standard ANSI SQL (in addition to its own commands for setup and administration) and can be accessed through standard ODBC and JDBC, or through the provided Oracle APIs (such as OCCI). TimesTen's primary limitation is the memory capacity available on the host machine. Because all TimesTen's data is held in memory, the amount being stored cannot be greater than the memory space made available by the operating system.

6. Benchmarks

In the CMS error and alarm system the latency for reporting an error from user application to spotlight it is less then one second with the current DAQ system. The scalability of the system is achieved with increasing the number of spotlight applications. The maximum performance for the spotlight application depends on the database capabilities. SpotlightOCCI has been measured to be capable to perform at approximately 600 Hz (insertions per second). TimesTen is an in-memory implementation of Oracle database, and stores database data in a machine's RAM as opposed to hard disk for quicker access. By caching the data in memory, data can be accessed and modified more quickly. Using a TimesTen memory cache, spotlightTT was able to perform approximately at 2kHz. Spotlight2g, for comparison, stored approximately 1kHz insertions per second.

7. Integration with CMS sub-detectors

The CMS sub-detectors use XDAQ framework to develop the online software needed for the data taking, and are responsible to bring data from their sub-detector front-end system to front-end drivers. Different teams are involved in developing the sub-systems software and alternative approaches are taken for the same problem. One example is the error reporting: storing errors in a local disk, collecting logs in a central place etc. The central DAQ group aims to standardize the error and alarm management for all CMS online systems. To succeed it is not enough to provide a common software but a common infrastructure is also needed, to cope with this problem the XDAQ as a Service (XaaS) has been designed based on the software as a service concept [20].

XaaS is a full set of interoperable services that provide standard functionalities for use in the XDAQ environment. A XDAQ zone defines the scope of a distributed XDAQ application and all processes are organized into and searchable groups known as zones. Each zone has its own error database, monitoring data types, and error report system model. An independent dedicated XaaS is available for each different XDAQ zone. Each CMS sub-detectors should have a XaaS infrastructure maintained by the central DAQ team and a central XaaS system could collect error and alarm from all the CMS experiment as shown in **Figure 15**.

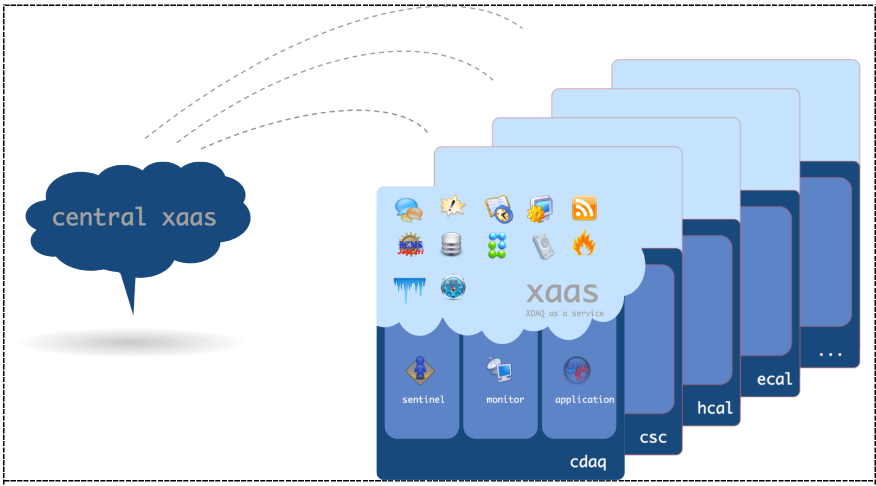

Figure 15. Central XaaS.

8. Summary

The error and alarm system has been implemented and is currently used in an operational environment for the central DAQ and almost all CMS sub-detectors. This software product line [21] is the result of several years of development and has proven its fitness for operation in the last three years of LHC operation. This paper summarized key requirements and outlined the resulting architecture (error handling, visualization, persistency, and integration) of the Distributed error and alarm processing in the CMS data acquisition system.

Acknowledgements

This work was supported in part by the DOE and NSF (USA) and the Marie Curie Program.

References

[1] The CMS Collaboration, The Compact Muon Solenoid Technical Proposal, CERN/LHCC94-38 (1994)

[2] The LHC Study Group, The Large Hadron Collider Conceptual Design Report, CERN/AC95-05 (1995).

[3] The CMS Collaboration, The Trigger and Data Aquisition project, CERN/LHCC 2002-26, 15 December 2002.

[4] Antchev G et al 2001, The Data Acquisition System for the CMS Experiment at LHC *in Proc. 7th Intl. Conf. Adv. Tech. and Particle Phys. Villa Olmo, Como, Italy (Oct. 15-19, 2001)* World Scientific Publishers (ISBN 981-238-180-5)

[5] Bauer G et al 2008 *CMS DAQ Event Builder Based on Gigabit Ethernet* IEEE Trans. Nucl. Sci. **55(1)** 198-202

[6] Gutleber J, Murray S, Orsini L 2003 Towards a homogeneous architecture for high-energy physics data acquisition systems *Elsevier Comp. Phys. Comm.* **153(2)** 155-163

[7] Parnas D L 1979 Designing Software for Ease of Extension and Contraction, *IEEE Trans. Softw. Eng* **SE-5(2)** 128-137

[8] Grama A Y, Gupta A and Kumar V 1993 *Isoefficiency: measuring the scalability of parallel algorithms and architectures*, IEEE Par. & Distr. Tech.: Systems & Applications **1(3)** 12-21

[9] G Bauer *et al* 2010, Monitoring the CMS Data Acquisition System, *J. Phys.: Conf. Ser.* **219** 022042

[10] Parnas D L 1979 Designing Software for Ease of Extension and Contraction, *IEEE Trans. Softw. Eng* **SE-5(2)** 128-137

[11] Nierstrasz O, Gibbs S and Tsichritzis D 1992 *Component-oriented software development* Comm. ACM **35(9)** 160-164

[12] Birman, K. and Joseph, T., Exploiting virtual synchrony in distributed systems *in Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87)*, 1987. pp. 123-138.

[13] Oracle database, http://www.oracle.com/us/products/database/index.html?ssSourceSiteId=otnen

[14] SQLite database, http://www.sqlite.org/docs.html

[15] Adobe Flash Builder, http://www.adobe.com/products/flash-builder.html

[16] M. Zloof. Query by Example. AFIPS, 44, 1975.

[17] Guttman E, Perkins C, Vaizades J and Day M 1999 *Sevice Location Protocol Version 2* Internet RFC http://www.ietf.org/rfc/rfc2608.txt

[18] Adobe Flex, http://www.adobe.com/products/flex.html

[19] Oracle TimesTen In-Memory Database, http://www.oracle.com/us/products/database/timesten-066524.html

[20] *Software As A Service: Strategic Backgrounder*. Washington, D.C.: Software & Information Industry Association. 28 February 2001

[21] Clements P, Northrop L 2002 *Software Product Lines* Addison-Wesley (ISBN 0-201-70332-7)

[22] Universally unique identifier, http://en.wikipedia.org/wiki/Universally_unique_identifier