An assessment of a model for error processing in the CMS Data Acquisition System

2010 J. Phys.: Conf. Ser. 219 022039

(http://iopscience.iop.org/1742-6596/219/2/022039)

Journal of Physics: Conference Series **219** (2010) 022039 doi:10.1088/1742-6596/219/2/022039

An Assessment of a Model for Error Processing in the CMS Data Acquisition System

S Dustdar[1], J Gutleber[2], R Moser[1,2] and L Orsini[2]

[1] Technical University of Vienna, Karlsplatz 13, 1040 Vienna, Austria

[2] CERN, 1211 Gevena 23, Switzerland

E-mail: dustdar@infosys.tuwien.ac.at, johannes.gutleber@cern.ch, roland.moser@cern.ch, luciano.orsini@cern.ch

Abstract. The CMS Data Acquisition System consists of O(20000) interdependent services. A system providing exception and application-specific monitoring data is essential for the operation of such a cluster. Due to the number of involved services the amount of monitoring data is higher than a human operator can handle efficiently. Thus moving the expert-knowledge for error analysis from the operator to a dedicated system is a natural choice. This reduces the number of notifications to the operator for simpler visualization and provides meaningful error cause descriptions and suggestions for possible countermeasures. This paper discusses an architecture of a workflow-based hierarchical error analysis system based on Guardians for the CMS Data Acquisition System. Guardians provide a common interface for error analysis of a specific service or subsystem. To provide effective and complete error analysis, the requirements regarding information sources, monitoring and configuration, are analyzed. Formats for common notification types are defined and a generic Guardian based on Event-Condition-Action rules is presented as a proof-of-concept.

1. Introduction

The Compact Muon Solenoid (CMS) experiment at the CERN LHC pp collider has to cope with an interaction rate of 40 MHz. Since no purely software-based distributed system may digest the total detector data of 1 MByte for a single event every 25 ns, pre-selection is performed in custom built, pipelined processors that reside close to the detectors.

The resulting data rate of 100 kHz is processed by the CMS data acquisition system [6] that consists of O(20000) interdependent services. It follows a service-oriented architecture (SOA) [1][8] where each service provides a SOAP control interface [10]. High-level data acquisition applications have been implemented using the XDAQ framework [7]. The CMS data acquisition system also provides low-level monitoring and alarming information through the XDAQ monitoring and alarming system (XMAS) [2] infrastructure that is based on a scalable and distributed publish/subscribe eventing system [3] and currently handles O(100000) notifications per second.

This paper will present the architecture for a dedicated error processing system to reduce the number of notifications to the operator for simpler visualization and to provide meaningful error cause descriptions and suggestions for possible countermeasures.

2. Gap Analysis

The CMS data acquisition system provides monitoring and alarming information but no facilities that analyze this information to derive high-level interpretations. Such an error processing system compares the actual with the nominal state of the monitored system. *Configuration information* defines the nominal state; *Run-time information* describes the actual state, which is provided through XMAS but lacks state information and integration with legacy services. As the system continues to be developed, error processing algorithms require continues adaptation. To ease this task the algorithms shall be formulated independent of communication protocol and format.

3. Technologies

Continuing with a service based approach and taking the previously mentioned requirements and constraints into account, we implemented an error processing system with Web Workflows. Web Workflows combine *business processes* with the Web by encapsulating a workflow behind a SOAP Web Service with a defined interface. They allow separation of protocols and formats handled by the Workflow engine and the definition of error processing algorithms as Workflows.

Major business process management software vendors provide their own Web Workflow engine implementations, for example Oracle BPEL process manager and IBM WebSphere Process Manager [17]. We chose the ActiveBPEL workflow engine [4] as it implements protocol interoperability (SOAP over HTTP) with the existing monitoring system out of the box and can be extended with new communication protocols and data formats without modifying Workflows. It provides a standards compliant workflow editor and depends on a limited number of software packages (Tomcat and Java) that are already used in the CMS experiment.

4. Run-time and Configuration Information

Run-time information represents the actual condition of the running system and can be categorized as shown in Figure 1:

- **State information** contains information about the actual state of services. With hierarchical states as defined in ASAP [5] (Figure 2) we can impose general states for visualization and error processing and allow flexibility by refinement of states when necessary for control. For example a service can define a custom sub-state *open.running.discard* to indicate that it is operational but discarding incoming data.
- **Error information** describes exceptions, which could not be handled locally by services. It embeds a complete exception trace for debugging. In addition custom properties can be added at each level of the exception trace to provide further information for error processing in an automated fashion.
- **Service information** contains dynamic data ranging from statistics to configuration data not known a priori. It is freely definable and usually specific to applications.

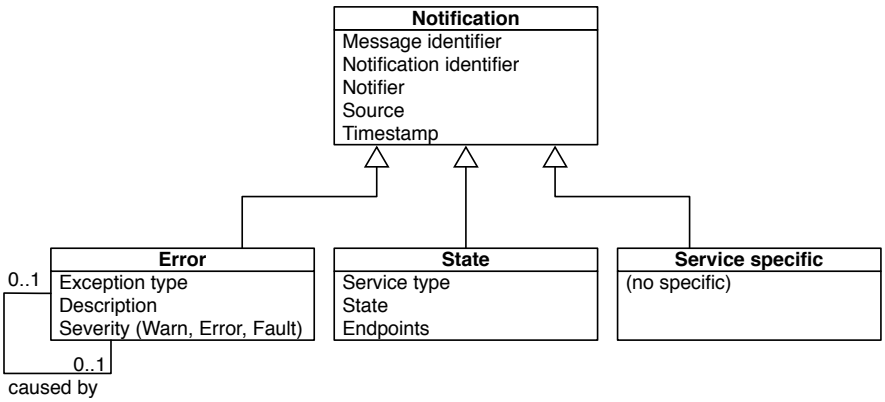

Figure 1 Run-time information types (notifications) and their primary properties.

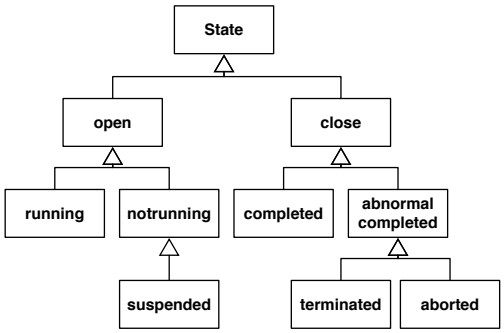

Figure 2 Hierarchical states allowing refinement and generalization.

Configuration information represents the nominal condition of the running system. It can be categorized in hardware and software information. Hardware information describes the setup of hosts, devices and networks. Software information specifies applications, services and communication endpoints.

5. Error Processing Architecture

A high-level error processing system is responsible to detect the cause of errors on startup and during operation of the monitored system. Therefore it analyzes differences between actual and nominal status of the system. The general architecture of our error processing system is depicted in Figure 3. The data layer contains services of the monitored system, which may emit data into the monitoring and alarming system. The logic layer contains the monitoring and error processing system and the visualization layer contains the graphical user interface the operator interacts with.

The *error processing system* contains two kinds of services, an *Error Processor* and *Guardians*. In our system the **Error Processor** is an intermediate, which subscribes to the monitoring and alarming system and asynchronously receives all error notifications generated by the services in the data layer. Subsequently these notifications are forwarded to error processing components, called *Guardians*.

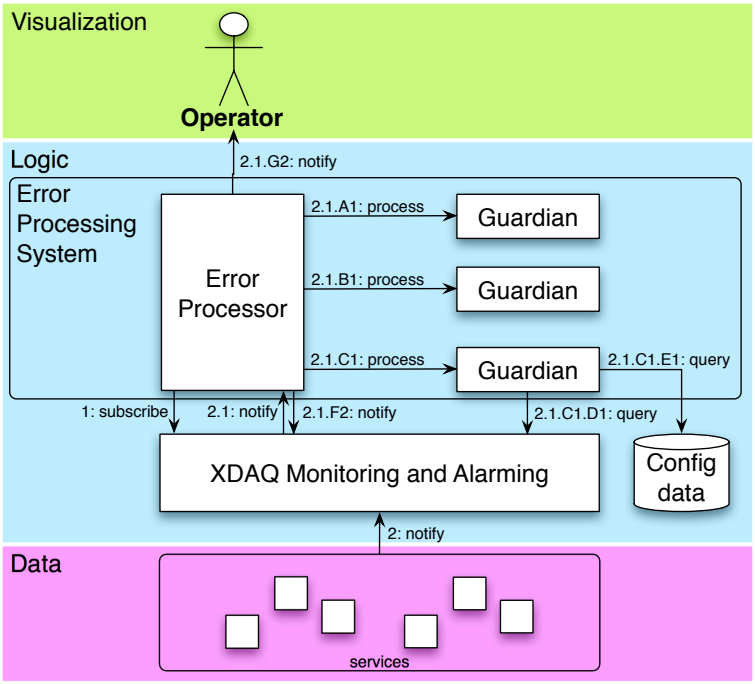

Figure 3 UML collaboration diagram of the error processing system and related services.

Guardians are logically ordered in a hierarchy as depicted in Figure 4 and contain expert knowledge about specific services or subsystems. A *Guardian* is only interested in a specific subset of notifications and thus provides a filter expression to reduce the number of notifications from the *Error Processor*. The low-level *Guardians* observe specific services whereas the higher ones observe groups of services. In case a *Guardian* cannot identify the cause of an error directly it may emit an exception, which is passed to a higher-level *Guardian*. Error processing should always be done on the lowest possible layer without incorporating knowledge about other subsystems or services. This keeps the higher-level *Guardians* abstract and confined to their respective group of applications. In case a *Guardian* could identify the cause of an error it may emit a notification to the operator.

All *Guardians* provide the same SOAP interface and as such may be implemented in any language. This allows integration with already existing rule-based systems or custom error processing code in case a generic *Guardian* is insufficient. The request message to the *Guardians* contains a list of error notifications and a list of URLs of monitoring data servers, which may be queried for more information. The response message contains operator notifications if an error cause could be identified or a derived error notification. Additionally it encloses a list of matched notification identifiers.

Finally the *Error Processor* forwards operator notifications to the operator, error notifications to XMAS and informs the operator to redefine the rules for unmatched notifications based on their unique identifier. Subsequently the derived error notifications are asynchronously sent from XMAS to the *Error Processor*, which will forward these notifications to another *Guardian*, effectively achieving a logical data flow as depicted in Figure 4.

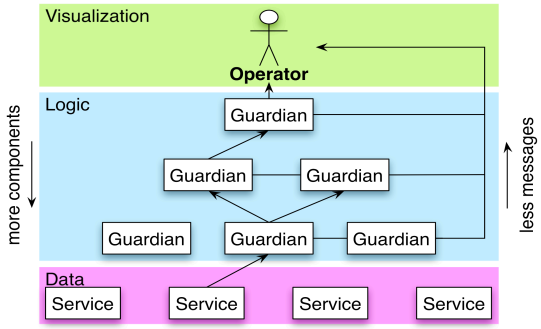

Figure 4 Logical data flow for error notifications (arrows in the middle) and operator notifications (arrows on the right).

We chose to implement error processing using BPEL as it already provides powerful languages for filtering (XPath) [12] and querying (XQuery) [11] XML data. Using those features we implemented a generic *Guardian*, which processes *Event-Condition-Action* (ECA) rules [9]. A rule that checks the diskUsage of our computers is shown in Figure 5. This is an example of a rule which is not triggered by an error notification but triggered periodically and checks service-specific information.

```
<eca xmlns:tns="http://xdaq.web.cern.ch/xdaq/wsdl/2008/guardianeca-10.wsdl">
 <source type='flashlist' name='diskInfo'>urn:xdaq-flashlist:diskInfo</source>
 <rule>
  <condition>/*/source/diskInfo/table/rows[ diskUsage/rows[xs:double(usePercent/text())>90] ] </condition>
  <action>
   <inform>
    <message>free disk space below 10 percent</message>
    <services source="condition">/*/rows/context</services>
   </inform>
  </action>
 </rule>
</eca>
```

Figure 5 ECA rule for generic Guardian detecting low disk space.

A set of rules is specific to one *Guardian*. A low-level *Guardian* for example defines a set of rules to process errors emitted by a specific service type, effectively leading to disjoint sets of rules for different *Guardians*. Higher-level *Guardians* define rules to match notifications derived by low-level *Guardians* only, leading to the hierarchical error processing depicted in Figure 4.

6. Enhancements

During evaluation of existing workflow engines we identified some shortcomings of BPEL and missing components necessary for integration with our system:

- BPEL workflows can only be triggered through SOAP messages and not through timers or even more complex rules.
- ActiveBPEL natively supports only SOAP based protocols.
- BPEL does not support to model an organizational perspective [15] and mapping of services to invoke activities must be modeled explicitly.

To overcome those shortcomings we implemented several additional services. Their interactions are depicted in Figure 6.

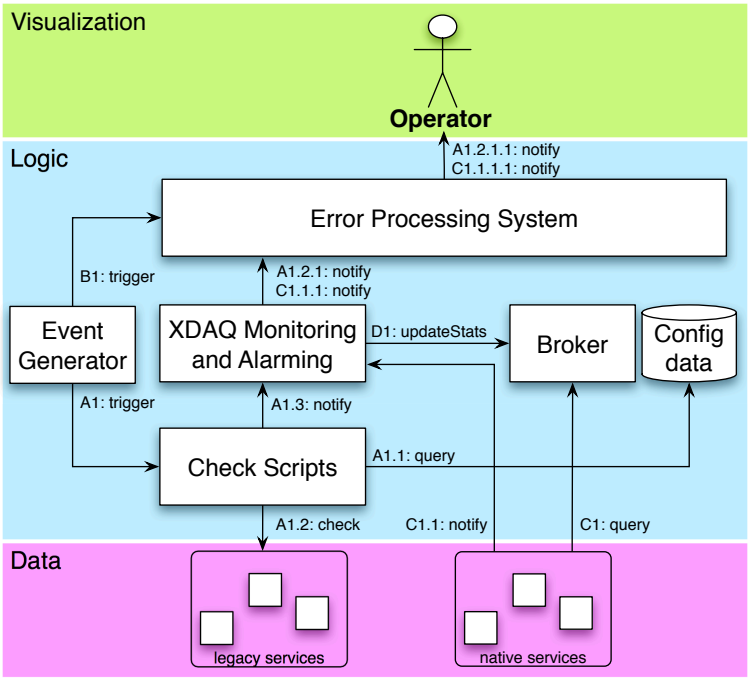

Figure 6 UML collaboration diagram for auxiliary components.

The **Event Generator** is a service, which sends SOAP messages based on predefined rules. Rules may match on workflow engine, timing and external user events. This allows periodic triggering of workflows and avoids ever running workflows, both concepts that are not supported by BPEL natively. The rule in Figure 7 shows a timing event (*MinuteTimer:trigger*) emitted once every 60 seconds. The timer is started based on the internal start event that is emitted as soon as the servlet engine in which the *Event Generator* is running is started. The second rule presented in Figure 8 starts a workflow which checks if all discovery services daemons [14] in our cluster are running and fully functional. The rule specifies that specific SOAP request message to be sent to a web service based on the previously mentioned timer event. This allows calling web services with without enforcing a specific interface on them.

```
<?xml version='1.0'?>
<netflow:event xmlns:netflow="http://xdaq.web.cern.ch/xdaq/xsd/2006/netflow-event-10">
 <netflow:component activated="true" changeable="false" class="ch.cern.cms.wf.event.Timer">
  <netflow:item name="name">MinuteTimer</netflow:item>
  <netflow:item name="type">timer</netflow:item>
  <netflow:item name="description">Timer for executing scripts once per minute </netflow:item>
  <netflow:item name="maxinstances">1</netflow:item>
  <netflow:item name="period">PT60S</netflow:item>
 </netflow:component>

 <netflow:bind xpath="/netflow:event[@name='internal' and @command='start']">
  <netflow:event name="MinuteTimer" command="start">
   <!-- contains SOAP message to send out if component supports that -->
  </netflow:event>
 </netflow:bind>
 <netflow:bind xpath="/netflow:event[@name='internal' and @command='stop']">
  <netflow:event name="MinuteTimer" command="stop">
   <!-- contains SOAP message to send out if component supports that -->
  </netflow:event>
 </netflow:bind>
</netflow:event>
```

Figure 7 Event Generator rule of a timer emitting an event once per minute.

```
<?xml version='1.0'?>
<netflow:event xmlns:netflow="http://xdaq.web.cern.ch/xdaq/xsd/2006/netflow-event-10">
 <netflow:component activated="false" changeable="true" class="ch.cern.cms.wf.event.Workflow">
  <netflow:item name="name">slpcheck</netflow:item>
  <netflow:item name="type">workflow</netflow:item>
  <netflow:item name="description">Script for checking if SLP daemons</netflow:item>
  <netflow:item name="maxinstances">1</netflow:item>
 </netflow:component>

 <!-- Event Bindings between internal components -->
 <netflow:bind xpath="/netflow:event[@name='MinuteTimer' and @command='trigger']">
  <netflow:event name="slpcheck" command="start">
   <ns1:StartServiceRequest xmlns:ns1="http://xdaq.web.cern.ch/xdaq/wsdl/2007/wfcheck-10.wsdl"/>
  </netflow:event>
 </netflow:bind>

 <!-- External (User) emitted events -->
 <netflow:emittable from='user' to='slpcheck' description='activate'>
  <netflow:event name="slpcheck" command="activate"/>
 </netflow:emittable>
 <netflow:emittable from='user' to='slpcheck' description='deactivate'>
  <netflow:event name="slpcheck" command="deactivate"/>
 </netflow:emittable>
</netflow:event>
```

Figure 8 Event Generator rule for triggering a web service (workflow) based on a timer event.

The **Broker** is a component for dynamically allocating resources and services according to *Quality of Service* (QoS) requests. It works with models for different scenarios. For example, the model for the monitoring system implements a load balancer for periodically allocating monitoring services to O(20000) services. This model itself relies on monitoring information, e.g. CPU load, to provide a

scalable monitoring infrastructure. Another example where a model would be useful is the assignment of services to hosts based on QoS attributes instead of statically assigning services to hosts. This can provide improved fault-tolerance and better resource usage in the data acquisition cluster. It will also simplify our workflows, as they will not need the informational perspective to model the organizational one [16].

Integration: As not all services publish directly into XMAS we added custom workflow checking scripts, which query the states of those services over SSH and publish their information into XMAS through SOAP messages. In addition some services use a custom, binary protocol for performance reasons.

Although WSDL allows defining interfaces independent of transport protocols, the ActiveBPEL engine only supports SOAP over HTTP as a protocol by default. ActiveBPEL solves this problem by providing InvokationHandlers, which translate between internal workflow engine data representation (XML) and custom formats and protocols and therefore allowing seamless integration with our system at hand.

7. Summary

This paper summarizes requirements and pitfalls during design and implementation of a generic error processing system using the CMS experiment as a case study. The presented error processing architecture relies on Workflow and Web Service technologies, which allow seamless integration into the existing environment. We implemented a generic workflow-based *Guardian*, which performs error processing based on ECA rules.

Tests of the error processing system were performed in the production environment of the CMS data acquisition system. In particular ECA rules have been defined for commonly encountered errors, such as failing service location protocol (SLP) servers and domain name resolution (DNS) servers. The error causes have been identified indirectly from error notifications emitted by data acquisition applications.

We observed that error notifications in our system can be classified in regards to the *number of originators* and the *number of notifications per originator*. A *Guardian* will handle those kinds of errors in the following ways:

- A *transient error* emitted by *one originator* leads to a single error notification. It will be matched by one specific rule in a low-level *Guardian* and will directly or indirectly (through a higher-level *Guardian*) emit one operator notification.
- A *transient error* emitted by *multiple originators* leads to multiple error notifications. It will be matched by one specific rule in a low-level *Guardian* and will directly or indirectly emit one operator notification.
- A *permanent error* emitted by *one or multiple originators* leads to multiple error notifications sent repeatedly. It will be matched by one specific rule in a low-level *Guardian* and will directly or indirectly emit the same operator notification repeatedly.

Our tests have shown that error notifications from multiple originators dominate the number of notifications. Our error processing system can handle these errors and reduces the number of notifications by the number of originators. This shows that the presented architecture is an adequate approach to analyze errors found in the CMS data acquisition system.

The low-level *Guardians* split the system into disjoint parts and thus scale to the number of services found in our system. Scalability is however limited by the *Error Processor*, which needs to forward all error notifications between XMAS and the *Guardians*. Additional measurements in the XDAQ framework revealed a performance bottleneck induced by the overhead of the SOAP protocol, which limits the throughput to 200 messages per second.

Planned improvements to the current system include porting XMAS to a binary protocol to reduce the protocol overhead. *Guardians* shall subscribe directly to XMAS to improve scalability of the error processing system. This requires extending the subscription mechanism to support complex filter expressions taking notification properties into account.

Due to the standardized notification formats, integration with other existing monitoring systems is feasible and would allow extending the scope of error processing beyond the core data acquisition applications. In addition providing a standardized interface for Guardians will allow us to take advantage by integrating distributed business rule engines [13] and already existing error processing components in the future.

References

[1] Booth D et al 2004 Web Service Architecture http://www.w3.org/TR/ws-arch
[2] Bauer G et al 2009 Monitoring the CMS Data Acquisition System *Proc. International Conference on Computing in High Energy and Nuclear Physics in Journal of Physics: Conference Series*.
[3] Box D et al 2006 Web Services Eventing (WS-Eventing) http://www.w3.org/Submission/WS-Eventing/
[4] ActiveBPEL Engine – official homepage http://www.activevos.com/community-open-source.php
[5] Fuller J, Krishnan M, Swenson K, Ricker J 2005 Asynchronous Service Access Protocol (ASAP) Version 1.0 http://www.oasis-open.org/committees/asap/
[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] Gutleber J, Moser R, Orsini L 2007 *Data Acquisition in High Energy Physics* Proc. Astronomical Data Analysis Software and Systems (ADASS) XVII, **394** 47.
[8] CERN 2002 *Data Acquisition & High-Level Trigger, Technical Design Report* CMS TDR 6.2, LHCC 2002-26 (ISBN 92-9083-111-4).
[9] Chen L, Li M, Cao J, Wang Y 2005, An ECA Rule-based Workflow Design Tool for Shanghai Grid, *2005 IEEE International Conference on Services Computing* **1** 325-328.
[10] Gutleber J et al 2005 HyperDAQ Where Data Acquisition Meets the Web *Proc. 10th Intl. Conf. Accel. and L. Exp. Phys. Control Sys.* (Geneva, Switzerland, 10-14 October 2005).
[11] Scott B et al 2007 XQuery 1.0: An XML Query Language http://www.w3.org/TR/xquery/
[12] Berglund A et al 2007 XML Path Language (XPath) 2.0 http://www.w3.org/TR/xpath20/
[13] Nagl C, Rosenberg F, Dustdar S 2006 VIDRE - A Distributed Service Oriented Business Rule Engine based on RuleML *Proc. 10th IEEE International Enterprise Distributed Object Computing Conference*. EDOC 2006: 35-44.
[14] Guttman E, Perkins C, Vaizades J and Day M 1999 Service Location Protocol Version 2 Internet RFC http://www.ietf.org/rfc/rfc2608.txt
[15] Russell N, ter Hofstede A, Edmond D, van der Aalst W 2004 *Workflow Data Patterns.*
[16] Zur Muehlen M 2004 *Workflow-based Process Controlling: Foundation, Design, and Application of Workflow-driven Process Information Systems*, Logos, http://books.google.com/books?id=EpgxaWJwkFQC
[17] Mendling J, Zdun U 2006 Experiences in Enhancing Existing BPM Tools with BPEL Import and Export *BPM 2006, LNCS 4102*: 348-357.