Journal of Physics: Conference Series **119** (2008) 022017 doi:10.1088/1742-6596/119/2/022017

The ALICE-LHC Online Data Quality Monitoring Framework

Sylvain Chapeland, Filimon Roukoutakis
CERN Physics Department, CH-1211, Geneva 23, Switzerland
E-mail: Filimon.Roukoutakis@cern.ch

Abstract. ALICE is one of the experiments under installation at CERN Large Hadron Collider, dedicated to the study of heavy-ion collisions. The final ALICE data acquisition system has been installed and is being used for the testing and commissioning of detectors. Data Quality Monitoring (DQM) is an important aspect of the online procedures for a HEP experiment. In this presentation we overview the architecture, implementation and usage experience of ALICE's AMORE (Automatic MOnitoRing Environment), a distributed application aimed to collect, analyze, visualize and store monitoring data in a large, experiment wide scale. AMORE is interfaced to the DAQ software framework (DATE) and follows the publish-subscribe paradigm where a large number of batch processes execute detector-specific analysis on raw data samples and publish monitoring results on specialized servers. Clients connected to these servers have the ability to correlate, further analyze and visualize the monitoring data. Provision is taken to archive the most important results so that historic plots can be produced.

1. Introduction

1.1. The ALICE experiment

ALICE (A Large Ion Collider Experiment) [1, 2, 3] is the LHC (Large Hadron Collider) experiment dedicated to the study of heavy-ion collisions at CERN. It will focus on the study of quark-gluon plasma formation signatures. It consists of several detectors of different types and is designed to cope with very high particle multiplicities (dN_{ch}/dy up to 8000). Commissioning of detectors is progressing in the course of 2007 in the underground experimental pit at the Swiss-French border. The detectors along with the required support services are expected to be operational by LHC startup.

1.2. The Data Acquisition Environment

Data Quality Monitoring (DQM) is an important aspect of every High-Energy Physics experiment. Especially in the era of LHC where the detectors are extremely complicated devices it is evident that a feedback on the quality of the data that are actually recorded for offline analysis is of great importance. From the DQM point of view, the Data Acquisition (DAQ) system is comprised from a set of nodes where partial and complete event building takes place. DQM aims at intercepting the raw data flow in these nodes and extracting physics information online.

DATE (Data Acquisition and Test Environment) [4, 5] is a software framework that has been developed to coherently drive the operation of the ALICE DAQ [6]. DATE is composed of

packages that perform the different functionalities needed by the DAQ system. These include low-level functionalities such as memory handling, process synchronization and interprocess communication, and higher-level functionalities like electronics readout, event building, data recording, runcontrol, information logging, error handling. DATE utilizes the DIM (Distributed Information Management) [7] system for interprocess communication between different nodes, the SMI++ (State Management Interface) [8] system for process control, MySQL database management system [9] for all the configuration databases and Tcl/Tk [10] libraries for GUI implementation. DATE also relies on standard Unix facilities like pipes and TCP/IP. The operating system of choice for the DAQ environment is Scientific Linux CERN 4 [11]. Further information is provided in the presentation titled "Commissioning of the ALICE Data-Acquisition System" [12], presented in this conference.

DATE provides a low-level monitoring package which forms the basis of any high-level monitoring framework for ALICE. It exposes a uniform Application Programming Interface (API) for accessing on-line raw data on DAQ nodes as well as data written in files. It gives the possibility of selecting the event sampling strategy for on-line streams in order to balance the needed computing resources.

1.3. Interactive Data Quality Monitoring: MOOD

MOOD (Monitor Of Online Data) [13, 14, 15] is the project aimed at serving the interactive DQM needs of ALICE. It is written in C++ and makes heavy use of the ROOT framework [16, 17, 18]. ROOT is used to provide the GUI and the analysis tools such as histograms and graphs. The DATE monitoring library provides the needed interface to the DAQ. MOOD has a plugable structure. The executable is a ROOT GUI application in which classes containing the detector specific functionalities are instantiated at runtime. These functionalities include the desired visual layout and the detector specific analysis on the raw data. For a detailed description of the application architecture and capabilities cf. [15]. MOOD served as the base to form the requirements of the ALICE online data quality monitoring framework, described hereafter.

2. Fundamental design requirements

The complexity of LHC experiments like ALICE imposes some fundamental requirements on the design of a modern automatic DQM framework. Following the "lovely" naming convention of frameworks adopted by ALICE DAQ (DATE, AFFAIR, MOOD), we named the new framework AMORE (Automatic MOnitoRing Environment). Gaining from the experience of MOOD, the following major design decisions were initially set:

- AMORE shall be a distributed application following the "Observer" design pattern [19], also known as publish-subscribe paradigm, in which a large number of information producers and consumers can be connected in a scalable way. This requirement cannot be fullfilled with a simple server-client paradigm. The information producers have access to raw data from the DAQ network as well as published data of other producers. The information consumers can subscribe to producers, post-process and visualize the data.
- AMORE shall have no source dependence on detector code. This is accomplished by heavy usage of C++ reflection. In computer science, reflection is the process by which a computer program of the appropriate type can be modified in the process of being executed, in a manner that depends on abstract features of its code and its runtime behavior. Figuratively speaking, it is then said that the program has the ability to"observe" and possibly to modify its own structure and behavior. The programming paradigm driven by reflection is called reflective programming. Typically, reflection refers to runtime or dynamic reflection, though some programming languages support compile time or static reflection. It is most common in

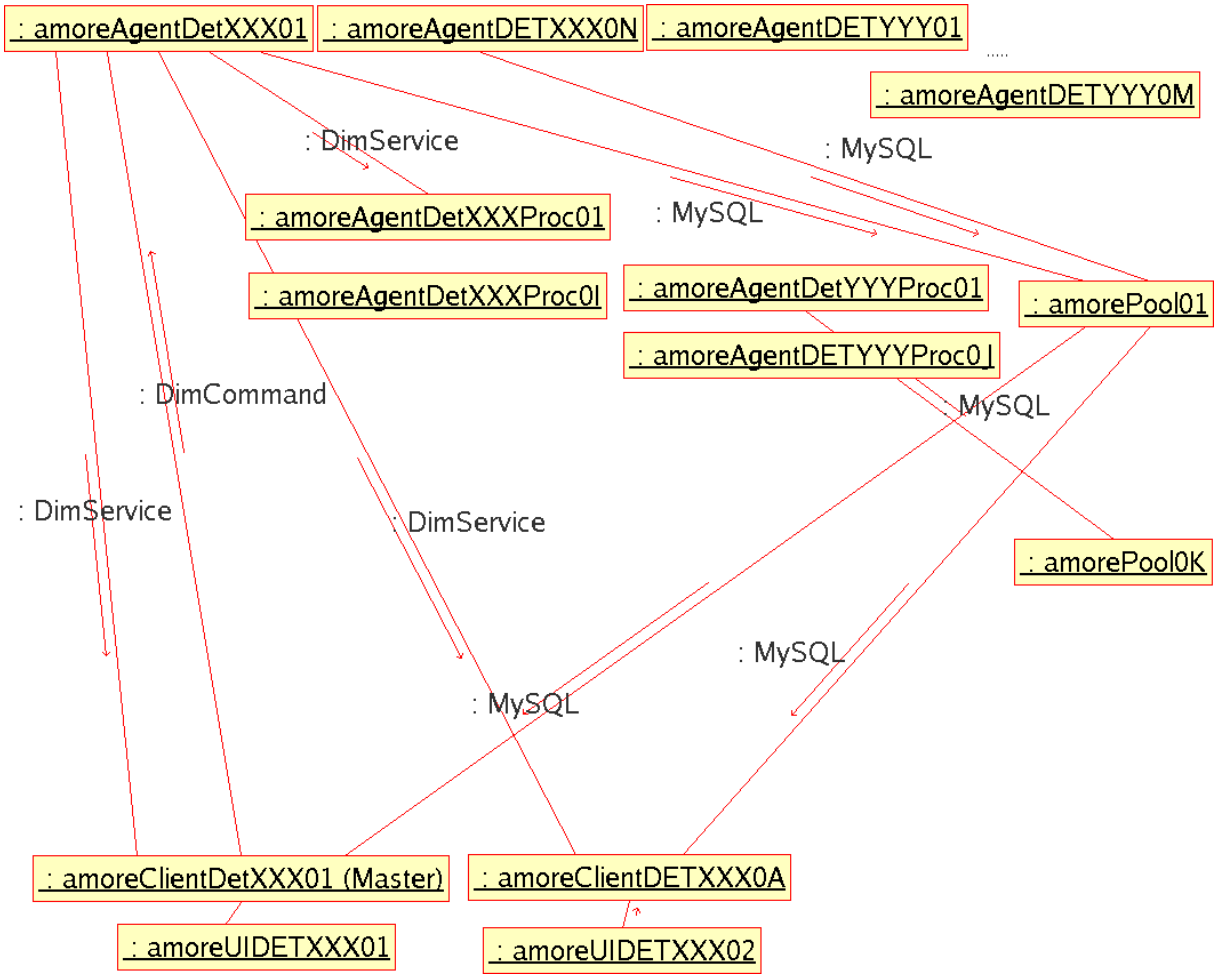

Figure 1. AMORE UML "communication" diagram

high-level virtual machine programming languages like Smalltalk, and less common in lower-level programming languages like C. In C++/ROOT it can be accomplished by building source code dictionaries at compile-time.

- AMORE shall use the same interprocess communication and control mechanisms as the ALICE DAQ, namely DIM and SMI++.
- AMORE publishers and subscribers shall use intermediate pools for data exchange, forming in essence a classical three-tier design. The original implementation shall use MySQL servers for this purpose, although provision shall be made that the framework is not bound to a specific implementation.
- All configuration shall be handled via MySQL databases.

The UML (Unified Modelling Language) [20] "communication" diagram of the AMORE processes is presented in Fig. 1. On the topmost level amoreAgents run as batch processes. They have access to raw data from the DAQ dataflow nodes. Their task is to decode and transform the raw data into physical quantities stored in the form of MonitorObjects. For the moment it suffices to consider MonitorObjects as histograms with additional housekeeping information that allow proper and coherent handling by the framework. Detailed description of the MonitorObject class hierarchy will be given in a later section. AmoreAgents have the ability

to dispatch an instance of their MonitorObject content to an amorePool. Each amoreAgent can connect to one and only one amorePool for this purpose. In essence amoreAgents publish their results on amorePools. On the opposite side, amoreClient processes subscribe to MonitorObjects on any amoreAgent. This allows amoreClients to receive regular updates on the content of the subscribed MonitorObjects. As it can be seen from the diagram, there is no direct MonitorObject transfer between amoreAgents and amoreClients. All such transactions occur via the amorePools which are implemented as MySQL servers. The framework also provides the possibility of non-first level amoreAgents that only have access to MonitorObjects published by other amoreAgents. They can perform post-processing and publish their results. The publication and subscription involves serialization of the object on the publisher side, the actual transport over the network and deserialization on the subscriber side. The serialization is handled by the facilities provided by ROOT, while the network transport, since it involves communication with the pool, is specific to the implementation of the latter. The current implementation will be discussed in a later section.

Communication between any processes for the purpose of command or information exchange is done through DIM. Any batch process in the framework acts as a DIM server which can receive commands from any DIM client implementation. In practice, the concept of a master client per subsystem will be introduced. Only the master client will be granted privileges to send commands to the amoreAgents of the subsystem. The rest of the clients for this subsystem will be simple observers, only able to subscribe to desired MonitorObjects.

The amoreAgent processes are implemented as Finite State Machines (FSM). This will facilitate the future integration with DAQ/ECS via the SMI++ framework. The amoreClient processes are modular. There exist the actual amoreClient backend which is responsible for the subscription handling and contains an abstract interface for any client application and a higher level layer which is responsible for the visualization and utilizes the abstract interface of the amoreClient.

3. The polymorphism and operation of publishers and subscribers

An amoreAgent is in reality a finite state machine, executing custom user code in a well defined time sequence depending on the DAQ environment status. For example, a tight event loop is executed during normal data taking while special functions are called at start and end of a run. To accomplish the execution of user code, the agent finite state machine subsequently calls member functions of an abstract based class, called PublisherModule, effectively implementing the object oriented version of the "Template Method" design pattern [19]. User code implements the pure virtual functions of this class in derived classes and a specific part of the agent, called Publisher calls these derived member functions. A fundamental requirement for AMORE was that no user source code dependency should exist. This is accomplished by usage of C++ reflection capabilities provided by ROOT. A dictionary for the user source code is created at compile time. It is then possible to create an instance of the user-specific class at runtime by using its name. In C++ pseudocode what is done is essentially:

```
PublisherModule* pointer=gROOT->GetClass(''MyPublisherModule'')->New();
```

where gROOT is the interface to the ROOT dictionary used to provide reflection capabilities to the system. The same approach is followed throughout the framework where large parts of user-defined code are to be executed. For this purpose, specialized abstract base classes exist for a subscriber and for a generic GUI client.

4. amorePool implementation via an RDBMS

A number of contemporary DQM frameworks were studied at the design phase of AMORE. Most of them follow the same three tier design with custom made pools for the exchange of data

between publishers and subscribers. We observed that the design and implementation of this pool is the most challenging part in this type of applications. Indeed, the publishing part can be considered as a simple number-cruncher running a well defined finite state machine and having minimal interaction with the environment, except from the input of source data and the output of the results in the form of higher-level objects. On the other end, the interactive clients that act as subscribers have a more complicated structure since they are a combination of a backend responsible for the communication with the pool and a frontend responsible for the interactivity, but they are neither time nor mission critical components of the system. The pools should be considered the most critical component for several reasons:

- They must handle the incoming traffic of several publishers and timely distribute the requested data to the subscribers in a manner invisible to both end peers, ie as like the traffic was direct from the publisher to the subscriber.
- They must be able to handle connections and disconnections of subscribers or random failures of publishers.
- The information flow must be optimized in the sense that subscribers should always receive data shortly after their publication and not for example just before they get republished.

By reviewing these requirements it was made clear that these are also the major challenges a Relational Database Management System (RDBMS) has to cope with. Therefore, it seemed natural to make an effort to utilize such a system as a pool. The RDBMS is used to handle the dataflow, while the subscriber notification is accomplished through the usage of DIM. The RDBMS of choice was MySQL for two reasons:

- It is an open-source, free for academic purposes program.
- It has been successfully used for handling all the configuration and logging aspects of DATE and the ALICE DAQ environment in general.

Each amoreAgent is considered a distinct entity in the framework, possesing its own unique identity in the system. Therefore, it makes sense to mirror this uniqueness by demanding that each amoreAgent is represented by a table in a database. This choice is also compatible with the specific implementation details of the used MySQL storage engine, INNODB. Since the tables are disk-based, with the table by default occupying one file, it is in general recommended to keep each table small. Therefore, the option to use a unique table for all the agents would not be optimal. This last solution would also require to include in the key index of the table the agent name which would put an additional overhead. Finally, by dividing the tables into different files, we take advantage of the operating system concurrency capabilities in I/O.

Each row of the agent table is associated with a MonitorObject that is published by the Agent at any given moment. The table contains four fields at the moment. The "moname" field is a character string containing the full name of a MonitorObject and is unique within the agent and the table and is used as a key in both contexts. The "updatetime" field contains the timestamp of the last dispatching of the specific MonitorObject from the agent to the pool. The field "data" contains the binary serialized data that correspond to this MonitorObject. The field "source" contains the DAQ node used as a source of monitoring data.

As with the DATE software environment, MySQL is used to define the configuration of the framework, namely the initial conditions for each agent, such as the operating node, the associated pool and the datasource. Finally, we plan to use one or more tables as needed for the storage of reference MonitorObjects.

5. MonitorObject class hierarchy

It is common practice for contemporary DQM frameworks to define a custom class hierarchy of objects to hold the higher-than-raw-data-level information that these frameworks are designed

to handle. Although, due to the statistical nature of information a DQM framework handles, most of the objects are some form of histogram, it is practical to define an abstraction that can unify the various types of histograms and other less utilized but still usefull data types behind a well defined, common interface. Needless to say, the hiding of the actual implementation behind a neutral interface is a well known Object Oriented design practice.

Most of the existing frameworks follow a pure Object-Oriented approach for the definition of such a class hierarchy. There exists a base class, often not abstract, with all the possible member functions that derived classes could use. The derived classes do not cover all the possible underlying data types but rather the ones that are mostly expected to be used. We decided to experiment with a slightly different approach. Our base class, named MonitorObject, is purely abstract [1] and holds at the moment a very lightweight interface which is common for all the derived classes. For example there are functions to retrieve the identity of the MonitorObject or to Reset it. The derived classes are mostly template-based except for the cases this would not make sense. There exist for example a MonitorObjectScalar<ScalarType> template class that can be instantiated -and actually is- for all the C++ fundamental data types. The same holds for various types of histograms. One, two and three dimensional histograms, one and two dimensional histogram profiles are all template-based classes which are instantiated using the relavant ROOT datatypes. It is the derived classes themselves that define the actual API and not the base class. This means that for example user code cannot Fill(x, y, z) a one-dimensional histogram without getting a compile-time error. Each derived class defines its own set of mathematical operations and -most importantly- quality checks that make sense for the specific type of objects. It is our goal that all the "histogram"-like classes form distinct vector spaces over the field of MonitorObjectScalars. If this approach proves to be successfull, many common practice operations could have an intuitive meaning for a physicist. For example, a common operation in DQM frameworks is the "collation" or "gathering" of histograms representing the same physical quantity from different sources with the purpose of acquiring one object with higher statistics. In our approach this operation could be so intuitive as the numerical addition of MonitorObjects. Another example, the asymmetry of two histograms could easily be expressed and calculated in user code by an operation as simple as

$$hasymmetry = (h1 - h2)/(h1 + h2)$$

allowing statistical correlations between similar data originating from different sources.

In order to meet requirements related to the successfull utilization of amorePools by the online applications responsible for calibration, namely "Detector Algorithms", it was decided, in addition to the above strictly defined class hierarchy which is functionally equivalent to the ones utilized in similar DQM frameworks currently under development for the LHC, to incorporate the ability to store arbitrary ROOT objects on dqmPools, encapsulated in a generic MonitorObjectTObject class. This class holds a pointer to the base ROOT class TObject in which any external derived object can be attached.

The ability to load different Publisher modules with varied sets of MonitorObjects at runtime, combined with the wide range of supported MonitorObject data types, already offers great flexibility to users and developers alike. Despite that, our plans are to incorporate to AMORE a special MonitorObject encapsulating a ROOT TTree class. A TTree is a highly versatile data structure capable of holding n-tuples of arbitrary data types with powerfull querying mechanisms, forming in essence a kind of Object Oriented Database. This addition will generalize a custom DQM application which uses this scheme, already created for the largest ALICE detector, TPC. Such an addition is expected to be useful to detector experts since TTrees support the creation of arbitrary histograms on the fly by querying the contained data.

[1] In the current implementation the base class is actually derived from a concrete ROOT class for several practical reasons, but this interface is hidden in the actual MonitorObject API

6. The publication and subscription mechanism

AMORE has been designed to handle several thousands of MonitorObjects in an experiment-wide scale. It is expected that some agents will contain up to $O(10^4)$ MonitorObjects of various types, while the number of agents that is planned to be deployed on DAQ nodes is $50 - 100$. We believe that it is neither robust nor elegant to delegate the handling of so many dynamically allocated structures to the end user. Therefore, on the publisher side, an entity following the "Factory" design pattern [19] should be responsible for the handling of the MonitorObjects throughout the lifetime of the agent. On the subscriber side, a counterpart should exist that does not register new MonitorObjects in the environment but rather provides efficient and robust access to the published ones.

The publishing interface follows the general form

```
void Publish(MonitorObject<Type>*&, ``unique name'', ...)
```

where the first argument is a reference to a pointer of a templated typedef strictly defined by the framework as described in a previous section. The end-user is only required to provide an uninitialized pointer variable as a first argument. The second argument is a character string representing the MonitorObjects unique name. It is evident that throughout the system there exist a unique identification for every MonitorObject in the form of a pair (amoreAgent name, MonitorObject name). Equivalently and/or depending on the context we can define a fully qualified pathname such as /amore/<amoreAgent name>/<MonitorObject name>. Finally, a variable number of arguments follow, not in the sense of C variadic functions but in the sense of several overloaded Publish function definitions with different number of arguments. The correct overloaded function is deduced as usual at compile-time from the type of the MonitorObject and from the number and type of the arguments.

The backend of the publishing includes dynamic allocation of the proper MonitorObject type and registration of the pointer to an associative container that maps the MonitorObject name to its pointer. The MonitorObject is owned by the framework once "published" and the user has no responsibility for freeing the memory used by it. At this point it should be mentioned that the word "Publish" does not mean automatic publication of the object to the pools, but rather the user code intention to make this specific object potentially available to the system. The actual publication happens in well defined time intervals if there are subscribers requesting this specific object. Notification to the subscribers is accomplished through DIM facilities. The update period can be based on elapsed time or number of events and will vary per agent according to the specific monitoring needs.

The reciprocal to the publication, statically-typed subscription mechanism is implemented by the family of functions with the general form

```
void Subscribe(MonitorObject<Type>*&, ``unique name'')
```

A weakly typed version of the subscription mechanism also exists, namely

```
void Subscribe(``unique name'')
```

In this case, as there is no direct pointer handle, user code queries the Subscription backend executing in C++ pseudocode the following operation

```
MonitorObject<Type>* pointer=
dynamic_cast<MonitorObject<Type>*>(gSubscriber->GetMonitorObject(``unique name''))
```

where gSubscriber is the interface to the subscriber backend.

7. Deployment and user code development model

AMORE build system is based on GNU autotools [21]. From the build system point of view, AMORE is essentially a class library comprised of several libraries built by libtool and a small collection of executables, either binaries or UNIX scripts. It is distributed as a binary RPM [22], which is the format of choice for all the software that is installed on ALICE DAQ machines. After installation, the following (conceptual) directory structure exists:

```
/<prefix> -\
            -bin -\
                   (The AMORE batch and interactive (GUI) executables
                    and setup/configuration scripts)
            -include -\
                       amore -\
                               (The AMORE API)
            -lib -\
                   (The AMORE libraries, shared and static versions)
```

AMORE is essentially a framework to allow coherent execution of detector-specific DQM-related code in the DAQ context. It follows that user libraries are integral part of the complete system. For this reason, a source code kit is distributed to detector developers that contains templated examples and appropriate Makefiles to build the libraries that can be automatically loaded by AMORE. There are 4 subdirectories at the moment and each directory contains a Makefile that has as default target the creation of a shared library with a name of the form libAmore<DET><dirname>.so, where DET is the 3-digit alphanumeric code of the detector, defined as an environment variable or in the Makefiles and dirname one of the directory names described hereafter. The top level Makefile supports all the usual targets like clean in order to cleanup the build, dist in order to create a tarball distribution and rpm, to create a source/binary RPM. The directory named common serves the purpose of providing a library with all the common functions that may be needed by the other modules like -but not limited to- decoding, mapping and common definitions. The directory publishers contains all the modules that could be loaded on an agent in order to be used by the Publisher FSM and the directory subscriber the same for the Subscriber FSM. Finally, the ui directory contains all the modules that create the desired GUI layouts. The build system takes care of creating the appropriate source code dictionaries so that the contained classes can be instantiated by AMORE without their explicit declaration. The convention followed is that user code lives under amore::DET::<dirname> namespace to avoid name clashes between code from different detectors, in case this situation arises in future versions of AMORE, that is, modules from different detectors are handled by a single process. Except for the usage of the MonitorObjects and the publication/subscription API of AMORE, these libraries can link to the ALICE offline framework, AliROOT and -statically- to any other required library.

8. Visualization

In our view, AMORE is really defined as a high processing throughput, low overhead fabric of batch processes having as input low level raw data and as output high level physics observables in the form of MonitorObjects. Based on these observables, automated batch process can take decisions on the quality of the data. Moreover, in principle it would suffice to expose the amoreClient API thus enabling user code access to MonitorObjects updates so that custom GUI clients can be built. Our experience from MOOD usage has however shown that detector groups can actually benefit a lot from a simple, well designed and generic GUI client that works out-of-the-box, well integrated with the DQM framework. Therefore, it was decided to provide a MOOD-like GUI application, utilizing the well-tested ROOT GUI capabilities. For the moment,

user code is responsible for the handling of GUI elements and the manual update of the screen after the retrieval of MonitorObject updates. Our plan for the following months is to provide a fully managed environment where MonitorObjects are associated with custom GUI widgets that offer additional to the standard ROOT functionality, for example context menus that are used to manage MonitorObjects directly on agent side by using the amoreClient backend. In addition, we plan to provide a graphical "browser" of the active MonitorObjects of all the agents. Finally, there are plans to use components of the ALICE Event Visualization Environment (AliEVE) for the AMORE GUI client.

9. Data Quality Assessment

It is usual practice to introduce special classes and -in our opinion- rather elaborate schemes in order to assess the data quality and issue alarms to the shift crew. We opted for a different, rather minimalistic approach which fits nicely with the overall architecture and is simple and elegant but yet powerful and generic enough for an LHC experiment. In our design, there is no special notion of either an alarm or a specific quality check framework. All the quality checks are simply either member or friend functions of the MonitorObjects they can be applied to. The result of the quality check itself is in any case a quantity which is directly translatable to a MonitorObject. For example the success or failure of a test can be represented by an integer MonitorObject with two distinct possible values or -if a kind of fuzzy logic is required to be present in the outcome- more than two values. It is trivial to define in this sense a MonitorObjectStatus class which contains a C++ enumeration with the following fields: kGreen, kYellow, kOrange, kRed, kGray or equivalently kOK, kWarning, kError, kFatal, kUnknown, all simple integral numbers. But if this is a MonitorObject it follows that it can be published just like any other "ordinary" MonitorObject. Then arbitrary number of amore processes can subscribe to multiple MonitorObjects like this and readily obtain the status of the associated quality checks. All kinds of actions are possible once the status is known in a subscribed process. Part of the screen can obtain a different color, a message can be logged, ECS can be notified, additional quality checks can be triggered. If needed, additional metadata can be part of the MonitorObjectStatus class. The most obvious one is a container of references to associated MonitorObjects. This can be implemented as an associative array of strings to MonitorObject pointers if the associated objects lie in the same process or more generally an associative array of strings to metadata used to indirectly access MonitorObjects in a relational database.

10. Conclusions

AMORE has been released in Summer 2007. The goal is that some of the detector groups will have AMORE modules ready for deployment and testing in the ALICE cosmic data challenge scheduled for October 2007. This will be the first large scale test of the system under realistic conditions.

References

[1] The ALICE Collaboration 1995 ALICE Technical Proposal for A Large Ion Collider Experiment at the CERN LHC Tech. Rep. CERN/LHCC/95-71, LHCC/P3 CERN
[2] The ALICE Collaboration 1995 ALICE: Physics Performance Report, Volume I Tech. Rep. CERN/LHCC/95-71, LHCC/P3 CERN
[3] The ALICE Collaboration 2005 ALICE: Physics Performance Report, Volume II Tech. Rep. CERN/LHCC/95-71, LHCC/P3 CERN
[4] The ALICE DAQ Project website URL `http://ph-dep-aid.web.cern.ch/ph-dep-aid/`
[5] The ALICE DAQ Project 2006 *ALICE DAQ and ECS User's Guide* release 5.0
[6] The ALICE Collaboration 2004 ALICE Technical Design Report of Trigger, Data Acquisition, High Level Trigger and Control System Tech. Rep. CERN-LHC-2003-062, ALICE TDR 10 CERN
[7] C Gaspar et al 2000 *International Conference on Computing in High Energy and Nuclear Physics* (Padova, Italy)

[8] C Gaspar et al 1997 *Proc. 10th IEEE NPSS Real Time Conference* (Beaune, France)
[9] The MySQL website URL `http://mysql.org`
[10] The Tcl/Tk website URL `http://www.tcl.tk/`
[11] The Scientific Linux CERN website URL `http://linux.web.cern.ch/linux/`
[12] S Chapeland et al 2007 *International Conference on Computing in High Energy and Nuclear Physics* (Victoria, BC, Canada)
[13] The MOOD website URL `http://ph-dep-aid.web.cern.ch/ph-dep-aid/MOOD`
[14] Cobanoglu O, Ozok F and Vande-Vyvre P 2005 *Proc. 14th IEEE NPSS Real Time Conference* (Stockholm, Sweden)
[15] Cobanoglu O, Chapeland S and Roukoutakis F 2006 *Proc. 15th IEEE NPSS Real Time Conference* (Batavia, IL, USA)
[16] R Brun et al 1997 *Nuclear Instruments and Methods in Physics Research* **A389** 81–86
[17] R Brun et al 2006 *ROOT Users Guide* CERN release 5.14
[18] The ROOT website URL `http://root.cern.ch`
[19] Gamma E, Helm R, Johnson R and Vlissides J M 1994 *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley) ISBN 0-201-63361-2
[20] The UML website URL `http://www.uml.org/`
[21] The GNU website URL `http://www.gnu.org/`
[22] The RPM website URL `http://www.rpm.org/`