Orthos, an alarm system for the ALICE DAQ operations

Sylvain Chapeland\textsuperscript{1}, Franco Carena\textsuperscript{1}, Wisla Carena\textsuperscript{1}, Vasco Chibante Barroso\textsuperscript{1}, Filippo Costa\textsuperscript{1}, Ervin Denes\textsuperscript{2}, Roberto Divia\textsuperscript{1}, Ulrich Fuchs\textsuperscript{1}, Alexandru Grigore\textsuperscript{1,3}, Giuseppe Simonetti\textsuperscript{1,4}, Csaba Soos\textsuperscript{1}, Adriana Telesca\textsuperscript{1}, Pierre Vande Vyvre\textsuperscript{1} and Barthelemy von Haller\textsuperscript{1}, for the ALICE collaboration

\textsuperscript{1} CERN, European Organization for Nuclear Research, Switzerland.
\textsuperscript{2} Institute for Particle and Nuclear Physics, Wigner Research Center, Budapest, Hungary
\textsuperscript{3} Polytechnic University of Bucharest, Romania
\textsuperscript{4} Dipartimento Interateneo di Fisica ´M. Merlin´, Bari, Italy

E-mail: Sylvain.Chapeland@cern.ch

Abstract. ALICE (A Large Ion Collider Experiment) is the heavy-ion detector studying the physics of strongly interacting matter and the quark-gluon plasma at the CERN LHC (Large Hadron Collider). The DAQ (Data Acquisition System) facilities handle the data flow from the detectors electronics up to the mass storage. The DAQ system is based on a large farm of commodity hardware consisting of more than 600 devices (Linux PCs, storage, network switches), and controls hundreds of distributed hardware and software components interacting together. This paper presents Orthos, the alarm system used to detect, log, report, and follow-up abnormal situations on the DAQ machines at the experimental area. The main objective of this package is to integrate alarm detection and notification mechanisms with a full-featured issues tracker, in order to prioritize, assign, and fix system failures optimally. This tool relies on a database repository with a logic engine, SQL interfaces to inject or query metrics, and dynamic web pages for user interaction. We describe the system architecture, the technologies used for the implementation, and the integration with existing monitoring tools.

1. Introduction

1.1. The ALICE experiment

ALICE (A Large Ion Collider Experiment) \cite{1}\cite{2}, is the heavy-ion detector designed to study the physics of strongly interacting matter and the quark-gluon plasma at the CERN LHC (Large Hadron Collider). It primarily targets heavy-ion lead-lead collisions (Pb-Pb), but it also has a substantial physics program with proton-proton (pp) and proton-ion (pA) collisions. The experiment has been designed to cope with the highest particle multiplicities expected for Pb-Pb reactions. The detector includes high resolution tracking (silicon detectors, large time-projection chamber), particle identification, and triggering elements. It features two large magnets, a main solenoid and a dipole on the Muon arm. ALICE consists of 18 sub-detectors, being able to take data independently (standalone operation) or in global partitions (set of sub-detectors running together). In 2011, ALICE collected $2.10^9$ pp events and $150.10^8$ Pb-Pb interactions, which amounts respectively to 1.7 PB and 0.8 PB of data recorded.
Several online systems work together in order to operate the detectors. The Experiment Control System (ECS) is the state-machine software in charge of synchronizing and controlling all the online components: the Data-Acquisition system (DAQ), the Detector Control System (DCS), the Central Trigger Processor (CTP) and the High-Level Trigger (HLT).

1.2. The ALICE Data-Acquisition system
The ALICE DAQ [3][4] handles the data flow from the sub-detector electronics to the archiving on tape, as seen on Figure 1. A first layer of computers, the Local Data Concentrators (LDCs), reads out the event fragments from the optical Detector Data Links (DDLs). Up to 12 DDLs can be connected to the same LDC, and several LDCs may be needed to collect the data from a single sub-detector. The event fragments aggregated in sub-events are then transferred to a second layer of computers, the Global Data Collectors (GDCs), in charge of performing the event building. The same GDC receives all the fragments of a given event, and assembles them in a full event, which is then recorded to a transient storage (TDS) before being migrated to tape (PDS). Each uninterrupted data taking period is called a run, ranging from few minutes to many hours, with the same hardware and software configuration. There can be several runs in parallel, with different sub-detectors running together or standalone.

Figure 1. The ALICE DAQ architecture
The DATE software is the distributed system managing the DAQ data flow. It handles the configuration and control of the distributed processes running on the DAQ nodes. The DAQ farm also hosts the ECS, and a number of central services not directly part of the data flow, like the Alice Configuration Tool (ACT), the experiment logbook, the Data Quality Monitoring (DQM), and the operator consoles in the ALICE control room.

This paper focuses on the operation of the DAQ facilities, which currently consist of 480 PCs (servers and desktops), 423 RORCs (the PCI readout cards receiving data from detectors through optical link), 79 disk arrays, and 43 other devices (switches, power distribution units, etc). The other online systems (DCS, CTP, and HLT) run on dedicated machines managed by other teams.

2. Needs for an alarm system
The ALICE DAQ system uses hundreds of hardware and software components susceptible to fail. Abnormal situations have to be detected, advertised, and fixed rapidly in order to maximize the experiment data taking time. The ALICE DAQ has been running in production since 2007, and numerous hardware or software interventions were unavoidable.

2.1. Initial DAQ operation work-flow
This section describes how the ALICE DAQ was operated until 2012. Information about errors and failures was available from various sources, and handled mostly manually. The main sources of information are Lemon [7] (a system monitoring tool, developed and used by CERN IT services), DATE infoLogger (the central service collecting log messages from most DAQ software components), and custom tools to monitor e.g. health of the disk arrays, detector event size, or back-pressure on the detector data links. Issues may be reported by the experiment shifter (following an error messages in the logs or a custom display), by the DAQ on-call person (after an incident has been relayed to him), sometimes directly by e-mail on the DAQ support mailing list (e.g. in case of high temperature), and sometimes not at all (i.e. the information is available, stored or displayed, but no notification mechanism is in place). Depending on the errors, the corresponding DAQ experts would then take action (immediately or later, based on notification latency and situation urgency), follow-up issues and report actions (e-mail, experiment logbook or DAQ logbook). Operation documentation (both for shifter and experts) is available in a Wiki, where the various components are described with their related possible issues and corresponding fixes.

2.2. Optimization of operations
Although it is possible to operate and maintain the DAQ facilities in these conditions, we sought to optimize the support load and reaction time. Indeed, the heterogeneous sources of information and intervention work-flows sometimes hinder effective upkeep. We hence decided to use a common tool to detect, log, report, and follow up abnormal situations in the DAQ system.

Such tool improves issues tracking and solving at the experimental area, hence minimizing DAQ down time. It also eases operation for shifters and reduces support load on experts. In particular, it improves the status visibility of the DAQ 'background' components, i.e. all the services not directly visible but which are used by the main DAQ control flow (as opposed to those actively handled by the DAQ control flow and for which failure feedback is already extensively available).

No tool was readily available to fulfill our needs, so we developed one, named Orthos and presented in this paper. Instead of replacing all the information systems already in place (some of the information sources being very specific, custom sensors are anyway unavoidable in our working environment), we only needed the “glue” to connect together all components and feed their output in a common work-flow:
Orthos provides a unified interface to raise alarms from the various DAQ components, and means for the DAQ actors (operators and experts) to handle them. It brings together information and action flows.

Orthos does not make (all) measurements itself, but it is rather informed by new or existing monitoring metrics.

Orthos is intended to be a standalone product, which reports issues of hardware and software operated by the ALICE DAQ team: DAQ counting room facilities, DAQ, ECS, DQM software. This list is the primary target of Orthos field of action, but can be extended as needed. All systems look identical as far as Orthos is concerned; all interactions have to be done through the same common interface, without need of specific knowledge of the external systems. Orthos provides a service which collects and processes monitoring data in a central repository, and user interfaces to access problem reports remotely.

Orthos does not replace performance monitoring and usual metrics plotting, features which are still covered by the fabric monitoring tool in use (although there is sometimes a slight overlap, as most monitoring frameworks do include some level of exception monitoring and notification). Orthos is rather a layer on top of all sources of monitoring information to make sure that appropriate actions are taken when needed, and following uniform procedures independently of the source reporting an abnormal situation.

3. Design and implementation

3.1. Definitions

In the rest of this document, the following definitions apply:

• Alarm: an event to be detected and followed-up by Orthos. It corresponds to any abnormal situation, harmless or critical. It may (or not) require a preventive or corrective action, immediately or at a later point in time. It might also be a simple event notification, to be aware of if something else happens. The term “alarm” in this document does not necessarily imply the emergency connotation usually associated to it. It is used everywhere for simplicity, instead of the more accurate “abnormal situation” phrasing.

• Metric: Boolean value defining an alarm condition status (1=active, 0=inactive). It is the result of an external measurement of some environmental conditions which summarizes if there is an abnormal situation or not.

• Alarm classes and instances: a given kind of alarm is globally defined as an alarm class, which describes a general behaviour (alarm handling and follow-up actions). At run time, multiple occurrences of the same alarm class can be raised. Each of them is an alarm instance, and is uniquely identified by its class, a source (where it comes from, i.e. node/device name), and an optional key (which refines the context in which it happens). For example, the alarm class 'disk space low' would have as key the 'disk partition name', and an alarm instance could be the tuple (disk space low, aldaqpc123, /tmp). Successive measurements (in time) of the same tuple (class, source, key) trigger status updates of the same alarm instance.

3.2. Alarm work-flow

Figure 2 shows a typical alarm work-flow. Orthos corresponds to the dashed rectangle. In addition to 'human actors' here using the Graphical User Interface (GUI), there may also be 'CPU actors', for automation of some corrective actions (e.g. a process which would be notified when /tmp is full and clean it automatically), but such components have not been implemented yet.
The logic engine processes incoming data, stores them in the repository, and creates corresponding tickets. The repository stores the alarm history and current alarm instances states. It also keeps track of the user reactions and follow-up.

Orthos inputs metrics, as defined in 3.1, i.e. only Boolean values: threshold filtering (or other kind of logic to detect an alarm state) is handled by the sensor providing metrics. Typically, for alarms associated to continuous values, an alarm condition is detected by a measurement reaching a threshold and situation is back to normal when the measurement goes below this (or another) threshold.

Figure 3 defines the life cycle of an alarm notification as a state machine. The initial status in a normal situation is 'inactive'. When the alarm condition is detected, the logic goes to state 'active'. The operator should then acknowledge the alarm so that it moves to state 'acknowledged'. It goes back to initial state when the alarm condition is not met any more. The status 'gone' is used when the alarm disappears before acknowledgment. Operator has to acknowledge such states to clear the alarm and to ensure that transient errors do not go unnoticed. Acknowledgment of an alarm should be done only by the relevant operator (i.e. the actor(s) assigned to fix it).

When an alarm is raised, an issue tracking ticket can be created (depending on alarm type, but it is the case for most of them), and uniquely identified, so that further action logs can be attached to this event. The ticket is not created immediately when the alarm is raised. The choice is given to the operator acknowledging the alarm to create a ticket (or to refresh it when an open ticket already exists.
for this alarm instance), in order to do a first level filtering of alarms. Then Orthos takes care of the corresponding actions accounting and ticket follow-up. This functionality is defined as the tracker. Each ticket has a status (ignore, pending, action taken, closed), and some actions to update it (reassign, change state). Also, it takes into account the importance of the event, availability of assignees, expected reaction time for the alarm. The issue tracking entry is closed by the assignee when the necessary actions have been taken (or automatically when the alarm disappears depending on the type of alarm). Follow-up comments and updates in action log can be appended at any time. A new ticket should not be opened for an alarm if one exists already (e.g. for alarms going on/off repetitively). The issue ticket, once created, follows his own life-cycle (not the same as the alarm life cycle).

The main control panel displays the (new or past) alarms based on filter settings (alarm and ticket status/parameters), and allows to acknowledge them and access the interface to modify alarm and ticket status (e.g. log actions made to recover, or add comments).

The repository keeps the history of recent measurements, but eventually aggregates them to track state changes only.

3.3. Interfaces

The following software interfaces are available to interact with Orthos:

- An interface to define the configuration of each alarm: this is how we define which built-in metrics are measured, specifying alarm classes and associated keys, plus other parameters like thresholds and sampling time. This interface is not used for the injection of measurements from external monitoring tools.
- An interface to inject metrics: this is the procedure to insert boolean measurements in Orthos from external (possibly remote) components. Each measurement injected includes the following fields: alarm class name (e.g. ‘disk space low’), instance key (e.g. ‘/tmp’), source (e.g. ‘pc123’), status (alarm off or alarm on), timestamp, severity (optional, e.g. warning or error), comment (a textual description giving details on the measurement, e.g. ‘partition 96% full’). The (class name, source, instance key) combination uniquely identifies a given event, and 'status' reports the current situation. Other parameters are kept for information. Some metrics can be defined with the special type 'No contact', in which case the logic engine raises the alarm on the absence of a measurement after a given timeout. This is useful to detect services down.
- An interface to notify alarms: this callback interface avoids polling on the repository for new events. It pushes alarms to subscribers. It is an internal mechanism only for the time being, but it will be extended to publish on demand events to external components (e.g. for a daemon responsible to send email or SMS for urgent matters).
- An interface to query alarms: this interface provides access to the repository information (alarm status, ticket log, etc).
- An interface to report actions taken on alarms: this interface allows acknowledging alarms and updating the tracker issues.

In addition, a dedicated graphical interface implements the interaction with users. It provides (or plans to provide) the following features:

- Display and browse alarms and metrics states
- Group alarm by similar sources, by type of alarms, hierarchical view of sources (equipment – detector – host)
- Show alarms of a given source (PC123 => disk full + wrong software)
- Bulk operations on a set of alarms (all alarms of a given type, or from a given source)
- User authentication (to log who does what, and to restrict alarm processing to target audience)
- Follow-up of alarms
- Take actions, update tickets, assign issues
- Direct access to knowledge base (what to do when an alarm is raised)
- Possibly ignore some alarms repeating for a known reason, while waiting for the fix
- Easy access from anywhere (possibly remote operations for on-call support)

An example of Orthos display is shown in Figure 4 below, with one test alarm already acknowledged, and a new pending alarm (not yet acknowledged) for disk space usage on /tmp.

**Figure 4. Example screenshot of Orthos user interface**

### 3.4. Metrics

We built a list of abnormal situations for which intervention is required, and grouped them together in families. This results in the short list of alarm classes shown in Table 1, which covers most of the exceptions we had to deal with in production until now.

**Table 1. List of Orthos alarm classes**

| Alarm class name       | Description                                                                 | Example                                                                                     |
|------------------------|-----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Disk space low         | the free space (bytes or inodes) on a disk partition is low                  | /tmp 95% full on a PC                                                                      |
| Process dead           | a required process is dead                                                   | central log daemon missing                                                                |
| Temperature high       | equipment temperature is too high                                           | 35C ambient temperature in the rack                                                        |
| Event size out of limits| the average event size of a detector is too small or too big                | Detector X produces events twice bigger than expected because of wrong pedestal settings  |
| Device hardware error  | There is a hardware device error.                                           | Disk broken in a disk array                                                                |
| Device conditions out of range | Some running parameter of the device is out of range.                   | Current delivered by a power distribution unit too high                                    |
| Host down              | A machine is not running                                                    | Database machine is off                                                                   |
| Wrong software installed | RPM package not installed, wrong version, or installed where it should not. | Not the latest DATE version is installed                                                   |
| Wrong software configuration | A software system setting is wrong                                      | Firewall settings not as expected                                                         |
### 3.5. Implementation

Effort has been put on simplicity of the implementation in order to minimize development time, and validate the initial concept. We tried to apply a maximum of existing tools and technologies already used in our group. This results in a robust (no down time in production over the past 6 months) and scalable (350 nodes monitored and 100.10^6 measurements without significant system load) notification and follow-up system, taking advantage of the existing components in place.

The core of the repository is based on a MySQL [8] database server. Measurements are inserted in a table. The logic engine runs as a trigger procedure upon reception of new metrics, and updates the corresponding alarm states in a separate table. No additional process is needed: the logic runs directly in the MySQL server. The GUI interacts with the repository by means of SQL select queries and procedure calls to update alarm states.

The tracker feature was initially though to be implemented as another SQL logic engine in the database. However, as we started to use JIRA [9] for DAQ development and operation reports, we also implemented the Orthos tracker in JIRA to benefit from its extended list of features, and have all the issues (i.e. those created by Orthos, and those created manually) handled in the same way.

The GUI interface is written in PHP [10] and generates HTML code with CSS for the layout, and published by an Apache web server [11].

PHP interface to JIRA is done using Representational State Transfer (REST), i.e. HTTPS/JSON POST/GET methods, to gain full read/write access to the tickets. Authentication and corresponding access rights are granted using Shibboleth [12] and the CERN central single-sign-on service [13]. Fine-grain permissions are easily defined by creating and populating corresponding E-groups [14], which are list of users created on the CERN central services, and for which properties (e.g. check if a user belongs to a given group) are available through SOAP Web Services.

A distributed sensor is installed on all DAQ nodes for some basic measurements. It is indeed sometimes more convenient to re-implement some trivial system measurements (e.g. ‘disk full;) to avoid being dependant on the existing monitoring tools in use. We plan to migrate from Lemon to another monitoring framework, like Zabbix [15], and prefer to have a stable source of measurements during this transition phase.

A C++ class provides easy sensor implementation for custom measurements, with built-in configuration and monitoring loop. A C API provides simple metric injection function to insert health reports from existing running processes (e.g. DQM).

The corresponding Orthos binaries are distributed as separate RPM [16] packages (devel, shared, www, repository, and sensor). Documentation on alarm classes and corresponding issue-fixing procedures are available in the DAQ wiki (implemented with TikiWiki [17]), and information directly linked from the human interface for easier access.

### 4. Conclusion

Orthos provides means to consistently gather from various sources and follow-up with common procedures abnormal situations in the ALICE DAQ facilities. The proof of concept was demonstrated to work fine with a reduced set of alarm classes to begin with, and the system has been put in
production at the experimental area. The implementation reuses a number of external tools and components which were already in place, allowing limiting the development effort and reducing expected code maintenance. We will now extend the framework to integrate all measurements needed to cover the majority of alarm cases which have to be dealt with during the ALICE DAQ operation.

**References**

[1] ALICE Collaboration 1995 Technical Proposal, CERN-LHCC-1995-71.
[2] ALICE Collaboration 2008 The ALICE experiment at the CERN LHC, JINST 3 S08002
[3] ALICE Collaboration 2004 The Technical Design Report of the Trigger, Data-Acquisition, High Level Trigger, and Control System, CERN-LHCC-2003-062
[4] Altini V, Carena F, Carena W, Chapeland S, Chibante Barroso V, Costa F, Divià R, Fuchs U, Makhlyueva I, Roukoutakis F, Schossmaier K, Soós C, Vande Vyvre P and von Haller B 2009 Commissioning the ALICE Experiment, Proc. CHEP 2009 (Prague, CZ: JPCS)
[5] http://cern.ch/lemon
[6] http://mysql.com/
[7] http://www.atlassian.com/JIRA
[8] http://php.net
[9] http://httpd.apache.org/
[10] http://shibboleth.net/
[11] Ormancey E 2007 CERN Single Sign On solution, Proc. CHEP 2007 (Victoria, CA: JPCS)
[12] https://espace.cern.ch/e-groups-help/
[13] http://www.zabbix.com/
[14] http://rpm.org/
[15] Tiki Wiki CMS Groupware, http://info.tiki.org