Experiences and evolutions of the ALICE DAQ Detector Algorithms framework

Sylvain Chapeland¹, Franco Carena¹, Wisla Carena¹, Vasco Chibante Barroso¹, Filippo Costa¹, Ervin Denes², Roberto Divia¹, Ulrich Fuchs¹, Alexandru Grigore¹,³, Giuseppe Simonetti¹,⁴, Csaba Soos¹, Adriana Telesca¹, Pierre Vande Vyvre¹ and Barthelemy von Haller¹, for the ALICE collaboration

¹ CERN, European Organization for Nuclear Research, Switzerland.
² Institute for Particle and Nuclear Physics, Wigner Research Center, Budapest, Hungary
³ Polytechnic University of Bucharest, Romania
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 18 ALICE sub-detectors are regularly calibrated in order to achieve most accurate physics measurements. Some of these procedures are done online in the DAQ (Data Acquisition System) so that calibration results can be directly used for detector electronics configuration before physics data taking, at run time for online event monitoring, and offline for data analysis. A framework was designed to collect statistics and compute calibration parameters, and has been used in production since 2008. This paper focuses on the recent features developed to benefit from the multi-cores architecture of CPUs, and to optimize the processing power available for the calibration tasks. It involves some C++ base classes to effectively implement detector specific code, with independent processing of events in parallel threads and aggregation of partial results. The Detector Algorithm (DA) framework provides utility interfaces for handling of input and output (configuration, monitored physics data, results, logging), and self-documentation of the produced executable. New algorithms are created quickly by inheritance of base functionality and implementation of few ad-hoc virtual members, while the framework features are kept expandable thanks to the isolation of the detector calibration code. The DA control system also handles unexpected processes behaviour, logs execution status, and collects performance statistics.

1. Introduction

For the reader’s convenience, this introduction reuses material presented in previous work to summarize the context of the ALICE online calibration framework. More details can be found in the cited references.

1.1. The ALICE experiment

ALICE (A Large Ion Collider Experiment) [1][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).
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 \(2 \times 10^9\) pp events and \(150 \times 10^6\) Pb-Pb interactions, which amounts respectively to 1.7 PB and 0.8 PB of data recorded.

1.2. The ALICE Data-Acquisition system

The ALICE Data-Acquisition system (DAQ) [3][4] handles the data flow from the sub-detector electronics to the archiving on tape, as seen on Figure 1.

![ALICE DAQ Architecture Diagram](image)

**Figure 1. The ALICE DAQ architecture**

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.

1.3. Calibration procedures
The 18 sub-detectors require specific calibration tasks to be performed regularly in order to achieve the most accurate physics measurements. These systems are indeed sensitive to configuration settings, mechanical geometry, environmental conditions changes, components aging and sensors defects. The corresponding set of procedures to calibrate the sub-detectors involves events analysis in a wide range of experimental conditions. These calibration tasks may be done either in dedicated runs, or in parallel to physics data taking. Typical examples of calibrations include pedestal and gain computation, dead and noisy channels mapping, etc.

The calibration results produced may be needed to configure the detector electronics for data taking, for example to produce zero-suppressed data or to mask noisy channels, in order to reduce the data volume. Therefore, these results should be available right after the calibration data-taking procedure, in order to reconfigure the detector accordingly for the next physics run. In addition, the results are also used offline for the events reconstruction. Both usages of the results involve a drastic timing constraint on the way they are produced. It would be too heavy to make the full calibration analysis offline (a first pass over the data would be needed to produce calibration results), and sometimes too late (for calibrations required very frequently, or for which results are needed for the detector configuration). Only the most complex calibration data analysis should be done offline.

Therefore, a dedicated framework has been designed and implemented to achieve as much as possible the detector calibration directly online, and to address the heterogeneous requirements specific to each calibration task.

1.4. The detector algorithms
The ALICE online calibration framework is used to implement and run a set of detector algorithms (DAs), which are calibration tasks running online. DAs are provided by the sub-detector teams, using the global framework to develop detector-specific calibration procedures.

The DA framework architecture is described in more details in a previous CHEP paper [5]. Each DA grabs detector data (physics or calibration events) and produces results online. These results can be reused directly online (e.g. to configure the detector electronics or give quality feedback to the DQM system), or shipped offline (to be post-processed and used in event reconstruction).

A DA consists of some specific detector code (to analyze events and produce results according to a given calibration task), using a support library to interact with the external components (read configuration, grab events, log messages, export results, deal with the control system commands). There are two types of DAs: the MON DAs (monitoring DAs, those which subscribe to events on the fly at runtime), and the LDC DAs (those which analyse at end of run a locally recorded data file).

The detector code and makefiles are hosted by the ALICE Offline framework, ALIROOT [6], so that the algorithms can reuse all detector specific classes and libraries, e.g. to decode the data stream or reconstruct events.

2. Experience with the DA framework
The DA framework has been running in production since 2007. This section reviews the different aspects related to their operation.

2.1. Release and deployment
There are at the moment 50 different DAs regularly used online at the ALICE experiment. Between 2007 and 2012, 328 DA packages upgrades have been done in order to add new features or fix bugs. This high demand from detector teams has put to the test the release procedures and build facilities. To detect a maximum of problems before production, strict procedures have been put in place: upon request, a specific DA version is independently validated by the DAQ team on a set of reference data files. If results are consistent and DA process behaviour is fine (calls to interfaces successful, no error or runtime crash, no obvious memory leak, etc), it is then installed in production. Some scripts have been developed to quickly build on demand requested revisions of a DA and deploy them. This is mostly automated. Only the consistency check of the log output requires a manual intervention, in order to verify that no unexpected message appears in the text output. The creation of the binary packages is done on a dedicated server with standard installation matching the environment in production.

2.2. Testing
A central test facility with all dependencies installed has been setup for users, to help ensuring that DAs are adequately tested before validation and deployment requests. Such central system helps fixing issues, which are more difficult to reproduce when users develop, build and install DA code on their own machines (which may have library versions discrepancies or other differences in the build environment). It also removes some workload on users, who do not need to setup and maintain a private test system (which reproduced by 18 sub-detectors, and even more DA developers, would be costly in hardware and manpower resources needed). Our central test facility in the lab is crucial to understand and fix problems found in production. It has always been possible to reproduce issues found at the experimental area by replaying the same data (or more generally a subset of it) as the actual data stream which caused problems. Dedicated scripts have been written to retrieve from permanent data storage and concatenate in a single local file all the sub-events of a sub-detector in a given run to help reproducing errors. The development workflow is summarized in Figure 2.

![Figure 2. DA development flow](image)

2.3. Static build
DAs are built as static executables to ensure stability and independency from the runtime environment (which may evolve due to more volatile components, like the DQM which follows many short-term evolutions to answer immediate needs). Indeed, we wanted to avoid unpredictable effects due to the release of upgraded versions of shared components, like ROOT used both by DAs and the DQM. We had to rebuild DAs because of changes in a statically linked dependency only a couple of times,
following some changes in the event raw data format. We plan to remove such limitation in the future and allow forward-compatibility, so that newer versions of event header would be transported the same way, without change needed in the monitoring library. There is of course a cost in term of runtime memory used by a static executable when several concurrent processes could instead use a shared version. However, we run a limited number of DA processes (1-4) on each node, and we were therefore not affected by memory shortage, even if the footprint of these libraries easily reaches a few hundred megabytes. On the other hand, a big advantage is the possibility to deploy and run on the same set of hosts DAs requiring different versions of ROOT or ALIROOT, without having to maintain different versions of these packages (which is then error prone if machines are configured differently). Until now, the positive effects of having a static executable have clearly outbalanced the limited number of inconveniences we faced, so we will continue with this choice.

The MON DAs run on a dozen shared hosts, and the LDC DAs run on the detector readout nodes. The MON DAs are statically allocated to a given host, which may be shared among different DAs. It is configurable, and can easily be changed at runtime in case of hardware problem, or if a DA takes so much resource that it is better to run it standalone on a dedicated machine to avoid affecting other DAs. Again, the static package eased the deployment or move of DA at runtime, as it requires a single RPM and prevents possible conflicts in dependencies.

2.4. Computing resources usage

In the first approach, resources used by DAs were not controlled. DAs are initially mono-threaded executables, so we distributed them in a way that a CPU is fully allocated to each of them. The machines have 16GB of memory, which is more than enough to run at least 4 of them. MON DAs run on independent nodes separated from the main DAQ data stream, so it does not limit the DAQ throughput in case of heavy load. The event subscribing policy commonly used is “get as many events as possible”. If the DA can not consume all events, they are just dropped when monitoring buffers get full. Another policy exists to “get all events and apply backpressure on the DAQ stream if needed”: this is in practice applied to make sure a DA can retrieve some rare but indispensable events when it needs them. For example, this is the case for the Start Of Data event (in which some detectors add some electronics or configuration settings in the payload), or the TPC laser events (one hundred events generated every few minutes). In these particular circumstances, “force monitor all” has no impact on the DAQ bandwidth given the limited amount of events involved, which fit in the monitoring buffer. As far as memory resources are concerned, we faced several times memory leaks in the detector code. This can cause some bad side effects, especially in long runs, like machines swapping or even getting unreachable. We have therefore added some built-in monitoring and set some configurable limits on DA memory usage in the process launching and controlling DAs.

2.5. Interfaces

As regards interfaces of DAs to the outside world, the designed DA information exchange mechanisms worked well and fulfilled the needs. Runtime parameters have been stored in configuration files in a central online database (implemented with MySQL, see section II-A-5 of [7]). These settings can be edited directly by detector experts to apply changes to the DA behaviour without needing recompilation (e.g. to define thresholds depending on the beam type, or to turn on or off some features). This central storage is also used to store persistent results which need to be kept, or measurements which need to be carried over from one run to the next. Injection of the DA output messages to the central DAQ log facilities allowed experts to remotely diagnose issues and check DA behaviour. The export mechanism to the Offline condition database sustained the load, and stored transiently 300000 files (over 500TB) up to now, with a SFTP server and MySQL indexing table. These transient files are hosted on the File Exchange Server (FXS, see section 2.3 of [5]), which accumulated data over time as shown in Figure 3.
2.6. Control
On the control part, some tuning was needed to set the appropriate timeout values when DA takes too much time to exit after it is requested to terminate. Indeed, some of the calibration tasks require some post-processing of data at end of run, and it may need intensive computing to review the data accumulated and produce results. After some iterations, the timeout values have individually been adjusted in the DAQ runControl and ECS to make sure DAs are given enough time in normal conditions to complete, yet abnormally blocked processes are effectively killed. These timeout parameters were originally external (and quite static settings), but another more flexible timeout layer (editable at runtime) has been added in the launcher process.

3. Evolutions
All in all, in the past 5 years, the originally designed framework allowed to fulfil all calibration needs. Some further enhancements have however been implemented or planned.

3.1. Control and bookkeeping
To improve the runtime control of DA processes, a new ‘launcher’ process has been implemented to better track what the DAs do at runtime. It is an executable written in C.

The launcher process launches the DAs and follows their execution. There is one such launcher per DA. First, the launcher checks whether or not the DA should be active in the current run type (physics, calibration, etc), and starts it if needed. This is defined in a centrally managed configuration file. Additional configuration parameters include the DA executable path, the detector it belongs to, the data source to be subscribed (i.e. which detector sub-events it should collect). The runtime limits are also set in this way: the maximum core size and memory consumption allowed and the time after which the DA process is killed if it does not exit properly at the end of the run. The launcher then instantiates a SMI [8] control state machine needed to interact with the runControl: it advertises the DA status (running, error) and receives commands to stop or abort the DA. The launcher spawns the DA as a child process, and ensures it keeps running. It collects exit status, and reports via the state machines when the DA stops.
The launcher also integrates bookkeeping features. It redirects the DA process standard and error outputs to the DAQ infoLogger system, which collects centrally all DAQ processes reports. This information is made available at runtime in the operating consoles, and is archived for later use and queries. The logs are also accessible directly from the experiment logbook, via a dedicated link for each DA. For a given run number, the detector expert can then easily retrieve remotely the DA messages with this interface. Moreover, the launcher also stores in the experiment logbook execution statistics for every DA which has been launched. At the moment, the MySQL table holding this information keeps track of the run number, the DA name, the detector name, the time at which the DA was started and when it completed, its exit status (one of ok, error, crashed, killed), the time it took to exit after the end of run request, if it went to timeout or not, and the amount of CPU time it used. An example view is seen in Figure 4.

These records can then be queried to detect if a DA has recurrent problems, and in which circumstances. By correlating these data with the run conditions, it is for example possible to detect a DA which always fails in runs longer than 2 hours or with more than few million events. Performance trends and reports are then easy to produce. Some of the DA statistics are however not available directly from the launcher. We plan to add directly in the DA the possibility to store in the logbook statistics on the number of events collected and processed.

The launcher process described above is now in production at the experimental area, and has already been useful for accounting tasks and remote log access. It provides a close loop control, and fine granularity on timeouts and resource allocation. It has improved our monitoring and reporting capabilities concerning DAs.

3.2. Parallelization
In order to benefit from the multi-cores architecture of CPUs, and to optimize the processing power available for the calibration tasks, we have worked out a new version of the DA framework (DAv2) to ease the development of multithreaded DAs.

In the first version of the DA framework, routines were provided to interface the DA with external components (in particular, all data I/O operations). However, the implementation of the main event processing loop was left to the user (but easy to populate from a base skeleton). As a consequence, there was also some code duplication (e.g. event subscription and control loop) from one DA to the other, and new features (e.g. report statistics on the number of events processed by all DA) had to be implemented on a case by case basis and independently in each DA.

The new framework is intended to provide parallelism and an enhanced set of features in the DA runtime engine. It provides the main loop and all other common facilities. It involves some C++ base classes to effectively implement detector specific code, with independent processing of events in parallel threads and aggregation of partial results. The workflow has been designed to transparently (for the user) allow processing of events in parallel. In this new version, users do not need to care about the control flow and event distribution, it is only needed to implement a few methods of some base classes. As the DA runtime engine is implemented in the framework, user will benefit from future evolutions without changes in their code.

![Figure 5. DA data and control flow](image)

A DA is made at runtime of several C++ objects: one DAio instance, taking part of initialization, configuration, and result collection of the DA, and one (or several) DAprocessor instances, receiving the runtime events and processing them in parallel threads. There is also one DAinfo object, providing DA information and documentation. The workflow is show in Figure 5.

To implement a DA, it is needed to implement 3 classes to be derived from the framework DAinfo, DAio, and DAprocessor base classes. They need also to be registered with predefined macros, so that the framework knows how to instantiate the user classes.

The class inheriting from the DAinfo class provides information on the DA: name, detector, version, etc. The data members of the base class define the DA behavior at runtime: maximum number of events to process, how many DAprocessor instances can be started in parallel (or 1 if it is not wished to have parallel handling of the events), and the settings for filtering on event types and trigger
classes to be used. The DAinfo member variables hence fully describe what the framework engine will do at runtime and how it will organize the data flow within the DA.

A user class inheriting from the DAio class provides the input/output functionality for the DA, and all features that do not run in parallel. A single instance of this class is implemented at runtime. The constructor of this class implements the DA configuration (e.g. read configuration from database file). A processResults() method is called after the main event processing loop, receiving as parameter an array of DAprocessor objects. These DAprocessor objects are not active any more (their thread completed), but they hold the partial results which now may be combined. Final results may then be exported to the DAQ storage. The base DAio class provides all the I/O routines which where available in the DA-framework v1: database access, File Exchange Server access, runtime environment, etc.

Finally, a user class inheriting from the DAprocessor class provides the individual event processing functions. Several objects are created and run in separate threads. Methods to be implemented are:

- a processEvent() method, called every time a new event is distributed to the thread.
- a publishIntermediateResults() method, called (optionally) from time to time during the run in order to publish intermediate results (e.g. to the DQM framework)
- a prepareResults() method, called at the end of the main event loop, when the DA is entering the post-processing phase, where the final results for this thread may be generated. The DAio->processResults() is called only after all DAprocessor instances have completed their prepareResults() method.

All the DAprocessor objects are associated to the same unique DAio instance. It allows them to access the shared configuration parameters that may be published in DAio. In the case it is not possible to produce partial results in the separate threads to aggregate them later, one can (from the DAprocessor instances) write to some shared data member in the DAio instance. In this case, the lock()/unlock() methods from DAio are used to ensure that a single DAprocessor accesses the resource. Note that it may affect parallelization performance, and should be done only if the lock/access/unlock time is much smaller than the event processing time.

In addition to the base class member methods, the DA framework provides access to a log object which may be used to output info/error messages.

As seen above, the full specification of a DA can be done with a handful of methods to be written by user. It can then run directly in the framework in parallel. Development time and later maintenance is greatly reduced. The framework takes care of the complex thread mechanisms and support functions.

The DAv2 framework is provided as a standalone C++ library. User code implementing the derived classes may of course rely on existing AliROOT code if necessary, provided that the features used are thread-safe where needed (i.e. mainly in the processEvent() method).

Preliminary tests have shown a very good scalability. When the corresponding event processing allows, full benefit can be taken from the inherent event processing in parallel. Such implementation works particularly well when events can be fully processed independently and partial results can easily be aggregated (e.g. by linear combination of histograms). Otherwise, the benefit depends on the ratio of the event processing which can be done in parallel, without the need to lock shared resources to store partial results.

The DAv2 framework allows using all cores available on a system to process detector events in parallel. If needed, one could go one step further, and distribute events on different hosts, to have even more threads in parallel. This could be implemented quite easily at the level of the monitoring library, by ensuring that hosts (identified for this mode of operation) subscribing to detector events receive different events (at the moment, they would possibly get copies of the same events). Aggregation of the results from the different hosts could then be done offline, in the software component which already collects and post-processes the various calibration results produced on different DAQ hosts for a given detector. However, we do not see the need of more computing power at the moment for calibration procedures. The DAv2 framework will gradually be deployed in production during 2012.
for specific calibration tasks which would benefit from processing more events, or for which a reduced latency is wished.

4. Conclusion
A framework for the implementation of calibration procedures in the ALICE online environment has been designed and used in production with success for the past 4 years. It contributed to an effective running of the detectors. Dedicated build and test environments have been setup to cope with the high demand of detector algorithm releases. Extended control and bookkeeping features have been added to the initial design for a better tracing of online calibration issues. Finally, an object-oriented framework supporting event processing in parallel was implemented to increase the computing power allocated to calibration tasks on multi-core architectures.

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] Vande Vyvre P et al 2010 Commissioning and initial experience with the ALICE on-line J. Phys.: Conf. Ser. 219 022022
[5] Chapeland S et al 2010 Online processing in the ALICEDAQ, the Detector Algorithms, J. Phys.: Conf. Ser. 219 022004
[6] AliROOT web site, http://aliceinfo.cern.ch/Offline/AliRoot/Manual.html
[7] Chapeland S et al 2009 The ALICE DAQ online databases, Proc. 16th IEEE-NPSS Realtime Conf. RT '09, 361-365 IEEE Xplore RTC.2009.5322155
[8] SMI++ - State Management Interface, http://smi.web.cern.ch/smi