Available on CMS information server **CMS IN 2005/XYZ**

CMS Internal Note

May 24, 2005
Version 0.1

Requirements and Design of the Physics and Data Quality Monitoring Infrastructure at CMS

C. Leonidopoulos and E. Meschi

CERN, Geneva, Switzerland

Abstract

This note describes the set of requirements and the architecture of the CMS Physics and Data Quality Monitoring infrastructure.

1 Introduction

The Physics and Data Quality Monitoring framework (DQM) aims at providing a homogeneous monitoring environment across various applications related to data taking at CMS. It must be fairly flexible and easily customizable so it can be used by different groups across the experiment. Examples of "customers" that can benefit from a common monitoring package are: the high-level trigger algorithms in the Filter Farm, subdetector groups that wish to supervise the operation of local DAQ systems, or even purely off-line batch jobs in a potentially "production validation" scheme.

The DQM organizes the information received by a number of monitoring producers and redirects it to monitoring-consuming clients according to their subscription requests (see Fig. **??** for a generic graphical representation of the producer-client relationship). On the producer side, it uses an implementation-neutral interface allowing direct insertion of monitoring statements in the reconstruction code. This interface can be accessed from standalone programs, or can be used from within *e.g.* ORCA "applications" and "modules". On the client side, an interface connects the received objects with a set of tools for evaluating the monitoring information, setting alarms and accessing a monitoring database for storing objects and editing log files.

2 General

The DQM is designed to receive sets of objects ("monitoring elements", or ME) from monitoring producers ("sources"), organize and redistribute them on a periodical basis. Sources are defined as individual nodes that have either direct access to or can process and produce information we are interested in. The creation and update of MEs at the source can be the result of processing input event data (event consumers) or input monitor elements (monitor consumers).

At the other end of this architecture are the monitoring information consumers ("clients") that receive a list with the available monitoring information ("monitorable") from all sources combined. The clients can subscribe to and receive periodic updates for any desired subset of the monitorable, in a classic implementation of the "publish-subscribe" service.

We introduce a hierarchical system of nodes that are responsible for the communication between sources and clients (*e.g.* subscription requests) and the actual monitoring transfer. These nodes serve as collectors for the sources and as monitoring servers for the clients, and will be called "collectors" in the rest of this document.

For small monitoring systems, a single collector should be sufficient (Figure **??**). For larger systems (*e.g.* the HLT Farm), one could envisage different levels of merging and elaboration. For example, in Figure **??** a first level of collectors is responsible for gathering all the monitoring information from multiple sources, effectively sharing the bandwidth and the workload. A second level of collectors is used to sort the monitoring already gathered according to content. A client could connect to one or more collectors, depending on the kind of information it is interested in subscribing. The collector "hierarchy", namely the full network of nodes that connects a source to the client and the exact configuration of its members, is static. It is never modified after startup. The configuration should clearly define the "coupling point" (*i.e.* collector) that every source and client can connect to. Whereas source and clients should be able to disconnect and reconnect at run-time (*i.e.* without affecting the stability of the rest of the system), the collector network must be always running if the monitoring service is meant to be uninterrupted.

A decision has been made that the DQM process should do the minimum necessary work at the source level. This is simply an attempt to minimize the interference with the "main" application running at the source node (*e.g.* analysis, calibration/alignment, trigger algorithm, etc). To that end, sources are connected to only one collector each (but a collector can connect to multiple sources). Clients do not have direct access to the sources. All source-client communication is done through the collector (or collectors).

All CPU-intensive tasks (*e.g.* comparison to reference monitoring element, display, database storage, etc.) should be attached at the client's side.

This designs aims at

- shielding the sources from connecting clients that could slow down the main application or threaten the stability of the source, and
- facilitating the quick transfer of the monitoring information from the sources to the collectors.

The production of the monitoring information is clearly separated from the collection and the processing. The

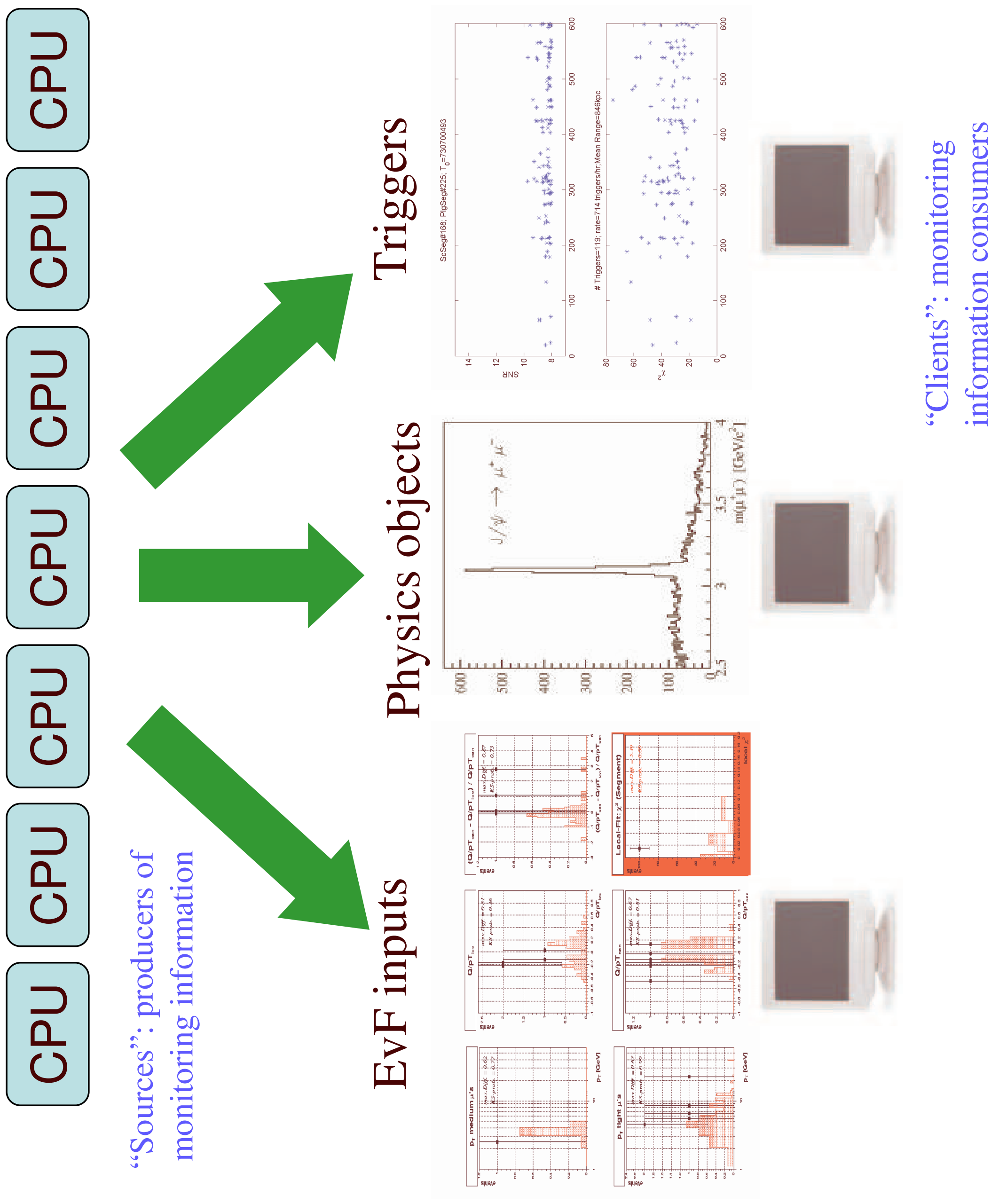
CPU
CPU
CPU
CPU
CPU
CPU
CPU
“Sources”: producers of monitoring information
EvF inputs
Physics objects
Triggers
“Clients”: monitoring information consumers

Figure 1: DQM is a generic monitoring framework serving different customers.

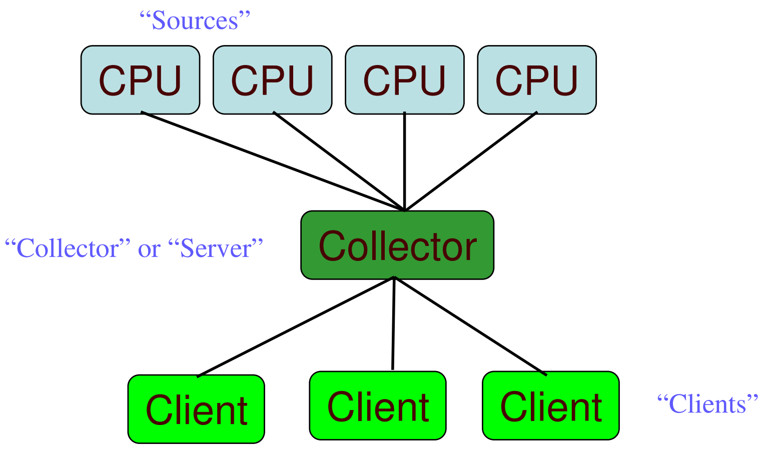

Figure 2: A simple monitoring system with a single collector.

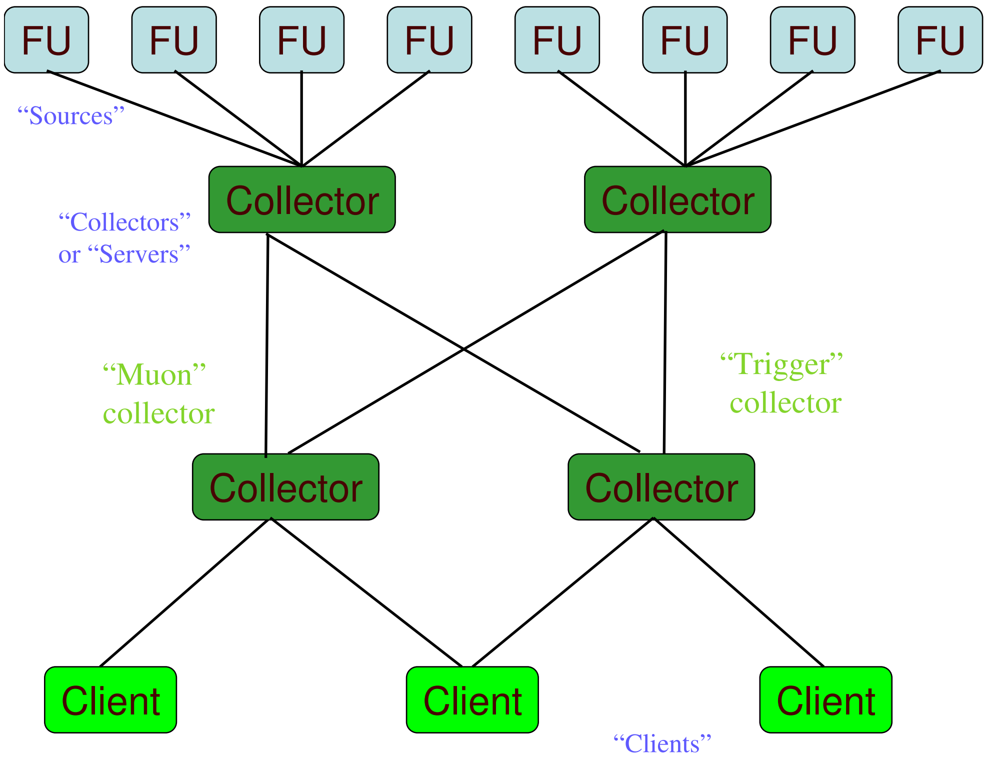

Figure 3: The high-level trigger farm as an example of a more complex monitoring system with two levels of collectors needed.

collectors act as the “middle man”: they are responsible for advertising the monitorable to different clients and serve monitoring requests.

In normal running mode, sources and clients are in an infinite monitoring loop:

- The source task list consists of
  - sending the full monitorable to the collector once (and updates, when necessary)
  - receiving (un)subscription requests (if any), and
  - sending the udpdated monitoring information to the collector
- The client task list consists of
  - receiving the monitorable from all sources when connecting (including updates, when necessary)
  - (un)subscribing to a(ny) subset of the available monitoring information
  - receiving the updated monitoring information, and
  - performing operations on MEs, setting alarms, accessing the monitoring database (see below)

3 Limitations

The nature of the collection and processing of the monitoring information is statistical by construction. In particular, the DQM

- is meant to help the experts identify problems that occur (and monitored) *over a period of time*.
- does *not* give access to particular events.
- does *not* guarantee that two clients will receive identical monitoring information.

The monitoring cycle[1)] may not be constant in time, especially for monitoring configurations in which the shipping time is comparable to the requested update period. Furthermore, the DQM should have the flexibility to dynamically adjust the monitoring rate, or even skip monitoring cycles, in order to minimize deadtime. The skipped monitoring packages are not considered lost information, as long as the exact fraction of the monitoring cycles received by the client is known.

4 Requirements

4.1 General

The DQM provides support for

- storing monitoring information into 1D-, 2D- and 3D-histograms, profiles, scalars (integer and real numbers) and string messages. It does *not* provide support for full events, raw data or other objects[2)] that cannot be accomodated in one of the above formats.
- abstract interfaces for booking, filling and accessing MEs (see Appendix, Sec. **??** for the rationale of this choice).
- directory structures, *i.e.* unix-like directories with virtually unlimited depth, from which the monitoring client can “pick and choose” (Fig. **??**).
- creation of root-tuples with the monitoring structure “on demand”.
- *dynamic* lists of sources and clients: individual sources and clients can be added or removed at run-time.

[1)] Defined as the time between two consecutive shipments of the updated MEs from the source to the collector (for the source), or from the collector to the client (for the client).

[2)] This is provided by an independent mechanism, not discussed here.

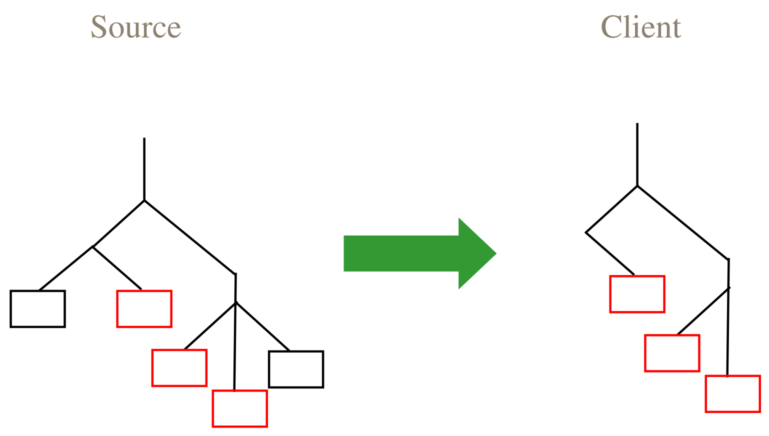

Figure 4: DQM support for directory structures and subscription “à la carte”.

4.2 Sources

- Support for *static* monitorables (*i.e.* monitorables defined at start-up).
- Support for *dynamic* monitorables: the sources should be able to update the list of available monitoring information at run-time and advertise it to the monitoring clients (through the collectors).
- The sources need not do more than booking and filling (and potentially deleting) MEs. The framework takes care of communicating this information to the client.
- Stability: the main application should not be affected if the connection with the collector is lost (*e.g.* the collector has crashed).
- Support for a “reset” switch specifying whether monitoring contents should be reset at the end of monitoring cycle. This flag (defined per ME) should be `true` for MEs that describe dynamic content (*e.g.* hit occupancy of a subdetector) and `false` for MEs that describe accumulating quantities (*e.g.* # of events processed, # of fatal errors caught, or rare counters).

4.3 Clients

- The clients should be able to receive monitoring information from multiple sources.
- Support for reception of static and dynamic monitorables from all sources (*i.e.* “initial” list of MEs at startup, and subsequent updates if necessary).
- Support for static monitoring subscription lists (*i.e.* automatic loading and transmission of a subscription list, as well as “edit” and “save” functionality).
- Support for dynamic monitoring subscription lists: the clients should be able to subscribe and unsubcribe at run-time “à la carte”.
- The clients should be able to request additional monitoring information (not listed in the initial monitorable). This would be equivalent to a “debug” command that requests the creation of an extra (temporary) set of MEs from one or more sources. At some subsequent time, the client is expected to issue another command cancelling the request (instructing the sources to cease producing the extra monitoring information, and therefore, lower the monitoring load).
- Support for a user interface that provides access to a set of tools (discussed in Sec. **??**).
- Stability: the main application should not be affected if the connection with the collector is lost (*e.g.* the collector has crashed).

4.4 Collectors

- Support for dynamic lists of sources and clients: A collector should be able to handle
  - disconnecting or dying nodes (sources and clients), as well as
  - new connections at run-time.

  Every time the number of active nodes changes, the collector must modify monitorable and subscription requests accordingly.

4.5 Client tools

As discussed in Sec. **??**, the majority of the operations or tasks involving MEs takes place on the client side. Here we define a set of tools used for these tasks, accessible only by clients.

4.5.1 Analysis tools for monitoring element operations

- "Soft" reset: operation that resets the contents of a ME for $t < t_0$ (the user should still have access to both $t < t_0$ and $t > t_0$ intervals). This functionality is useful only for MEs with the "reset" switch set to `false` (specified at the source, see Sec. **??**).
- "Accumulate": operation that sums the contents of a ME over many monitoring cycles. This functionality is useful only for MEs with the reset switch set to `true` (specified at the source).
- Collation: sum up MEs at client's side:
  $$C \equiv \sum_i h_i$$
  where h_i is the i^{th} ME and C the sum of all MEs.

  It is the client's responsibility to make sure that the resulting "summary" ME makes sense (*i.e.* collation of identical-in-form MEs).
- "Hard" reset: In addition to the "soft" reset functionality, the user has the option of permanently erasing the contents of ME that has been defined locally (and is therefore invisible to other nodes), *e.g.* an "accumulation"- or "collation"-type ME.
- Comparison to reference ME: *e.g.* Kolmogorov's test (giving the "% C.L." that a histogram and its reference are consistent with each other), "# of sigmas away from expected value", etc.

4.5.2 Alarm class

The overall "status" of a ME can, for display purposes, be summarized by a single discrete parameter. This is convenient for summary pages on a GUI that can give the overall status of a subdetector with a color-coded system. For example, the list of warning types (and associated color on the display) could include

- "New": a new alarm has appeared (red).
- "Acknowledged": an alarm has been acknowledged; action still pending (orange).
- "Addressed": the problem still remains, but experts have been notified or a plan has been made (yellow).
- "No warning": no problem exists (green).

The user is able to set, raise or lower alarms according to information extracted with the tools described in Sec. **??**. The alarm setting can be manual, automatic or hybrid. This aims at accomodating different situations, such as commissioning periods, routine problem-checking and cases where either a program or the user can make the call.

4.5.3 Monitoring database and archiving capabilities

Users should be able to store (and retrieve) custom sets of MEs. The user interface should provide

- Support for various sets of MEs: "current" (*e.g.* last hour, day, week) or "older" (*e.g.* last run, month or year).
- Option for storing objects on a server ("official") or locally.
- Access to "reference" sets (determined by trigger configuration: *i.e.* . luminosity, trigger streams, prescales, etc).
- Support for configuration lists for DQM clients (*e.g.* . default subscription lists, update frequencies, etc).
- Access to a "problem and solution" history. Quering flexibility is desired (*e.g.* per subdetector and/or per run, week or with any other "keyword").

4.5.4 Control structure and configuration

- The infrastructure supports the concept of a DQM configuration containing the list of DQM processes that must be configured for normal data taking.
- Each application can be started centrally and constantly reports its status to a supervising entity.
- Each application exposes a control interface similar to one used to control the DAQ system. This interface is used by the DAQ supervisor(s) to control the operation of the various processes. The interface can be customized to allow direct interaction with an individual DQM process to perform specific operation via a UI.

A hypothetical task list here could include

- configuring and starting a DQM process.
- displaying a report on the DQM process status.
- an interface à la "run control" for automated actions (*e.g.* stop- and start-run, reset, reconfigure).

4.5.5 Display and Graphical User Interface

A GUI gives users a quick way of accessing the UI methods discussed in this section. A non-exhaustive list includes

- Global display with individual tabs per group (giving a "one button" access to the state of all subdetectors: warnings, # of processed events, rates, etc).
- Option for static display or cycling through subscription list.
- Navigation capability: browse though monitoring structure and click on desired ME.
- Web-based display client.

The GUI would ideally be easily customizable, to allow different groups to tailor it to their needs. An investigation of different software packages is currently underway for selecting a suitable application development framework.

A Rationale for accessing monitoring elements via a neutral interface

The current implementation of the monitoring transfer mechanism from a node to another involves `ROOT` (classes `TSockets`, `TServerSocket` and `TMessages`). This is out of convenience, since users have expressed the desire to be able to save monitoring structures in root-tuples that they can access later. Since the "behind-the-scenes" mechanics is implemented with `ROOT`, one would be tempted to allow users to deal directly with `ROOT` objects. However, we have made the choice to hide the "true" format behind a transparent interface for both sources and clients, for a variety of reasons:

- An abstract interface does not bind the user access methods to a specific analysis framework and/or implementation. In our particular case, the transfer mechanism and the "true" ME format (`ROOT`) could change in the future, without breaking the source and client programs.
- Having an abstract interface that hides the raw monitoring data from the user is a good OO practice. The set of allowed operations on the monitoring objects should be defined by the (abstract) user interface, not the framework used for the implementation.
- Additional functionality can be added to the monitoring objects (*e.g.* alarms) without directly inheriting from `ROOT` classes.