Journal of Physics: Conference Series **396** (2012) 052050 doi:10.1088/1742-6596/396/5/052050

Agents and Daemons, automating Data Quality Monitoring operations

Luis I. Lopera, on behalf of the DQM Group

Los Andes University, Bogotá, Colombia

lilopera@cern.ch

Abstract. Since 2009 when the LHC came back to active service, the Data Quality Monitoring (DQM) team was faced with the need to homogenize and automate operations across all the different environments within which DQM is used. The main goal of automation is to reduce the operator intervention at the minimum possible level, especially in the area of DQM files management, where long-term archival presented the greatest challenges. Manually operated procedures cannot cope with the constant increase in luminosity, datasets and uptime of the CMS detector. Therefore a solid and reliable set of sophisticated scripts, the agents, has been designed since the beginning to manage all DQM-related workflows. This allows to fully exploiting all available resources in every condition, maximizing the performance and reducing the latency in making data available for validation and certification. The agents can be easily fine-tuned to adapt to current and future hardware constraints and proved to be flexible enough to include unforeseen features, like an ad-hoc quota management and a real time sound alarm system.

1. Introduction

In the IT operations world it is common for repetitive tasks to be automatic; furthermore, if such tasks require 24/7 action, it is more than logical to have them automated. One motivation is the cost incurred in hiring enough people with the right training and set of skills to cover the 24/7 schedule. Moreover, as proven in the manufacturing industry, repetitive tasks cause the worker to occasionally make mistakes due to boredom and other factors: these mistakes can reduce the system's ability to provide quick and accurate results. Designing programs that take care of the heavy load and relinquish the operator to an oversight position alleviate both problems: it reduces the number of qualified personnel required to look after the system and frees the operator of repetitive tasks, giving him/her the time to focus on improving the system, thus putting his/her skills to a better use.

"Agents and Daemons" (A&D) emerges as a highly modular solution oriented to automate the backstage operations that make Data Quality Monitoring (DQM) possible in CMS. It consists of a sophisticated collection of scripts; which ensures that DQM's data is presented, in a timely fashion, to the collaboration' shifters and subsystems' experts.

This paper will show the design ideas of the system, focusing in particular on CMS and DQM data workflows; a modular architecture is proposed for scalability, robustness and easy maintenance. Figure 1 shows a schematic view of the data flow, from the detector to data certification.

2. Understanding CMS's data flow[2]

Data in CMS is produced in different stages and with different purposes: on one side we have the detector data and all the post processing that is required to provide the input to the analyses; on the other, we have the simulation data used in conjunction with real data to reach valuable conclusions or to validate new releases of the CMS reconstruction framework, CMSSW. For the scope of this paper data processing is divided in two realms: online and offline. Online is all about detector readout and real time monitoring; offline is where post processing, simulation and validation take place.

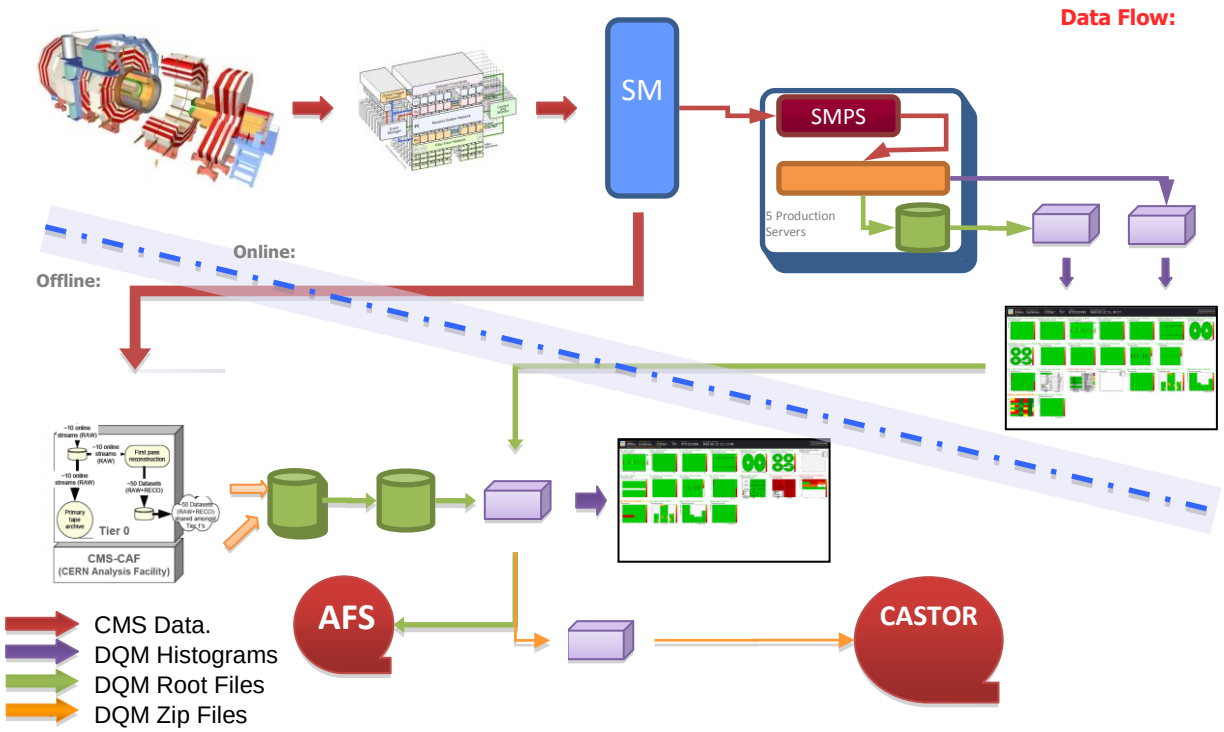

Figure 1. Graphic representation of CMS data and DQM data

2.1. Online Realm

Data in CMS starts at the detector readout, where the Level 1 Trigger (L1T) makes a first selection of individual sensor states and, based on established criteria, decides to record the event. Then this event is reconstructed in the filter farm where more complex objects are created and, based on these new objects and the raw information coming from the sensors, the event is either discarded or sent to post processing in Tier 0. This second decision state is called the High Level Trigger (HLT).

The L1T and HLT systems are controlled by the data acquisition system (DAQ) of CMS that provides the necessary infrastructure to process events at a rate of around 150 KHz out of which 300Hz are selected for physics analysis. The DAQ achieves this feat by running a large amount of HLT processes in parallel. Events that have been reconstructed and selected are then sent to the Storage Managers (SM), which is composed of 16 servers whose purpose is to build the stream of data (stream) to send to Tier-0 for further offline processing. Other streams can be produced for internal use, as for the DQM online-stream case.

A stream is defined in the HLT menu and it has 2 main characteristics: the list of trigger paths an event must fire in order to be chosen, and the list of reconstructed objects that are going to be sent with the event. There are two distinct classes of streams: calibration and physics. The former stores events that are fired by the calibration routines of the sub-detectors; these are valuable to assess the state of

the detector. The latter is the most important as it is the base for the physics analyses of the entire collaboration.

These two classes of streams are exclusive by design: therefore DQM needs to have its own stream definition with a suitable combination of the two types of events. Online DQM was entrusted with monitoring not only the data state of the physics events but also the detector status and performance.

2.2. Offline Realm

Here DQM becomes a step of the different scenarios that the collaboration runs to produce official data for all of its members to use in analysis and other activities. These scenarios are: express reconstruction, prompt reconstruction, re-reconstruction, CMSSW release validation (using real data and simulated data) and Monte Carlo production. These scenarios can be run at different LCG sites, and from the DQM point of view a mechanism has to be provided in order to collect all the different DQM results produced on each scenario.

3. From Event data to DQM data into the DQM GUI and beyond

The DQM step of the different data processing scenarios in CMS is a statistical analysis tool that uses ROOT as the backbone to produce its data [3]. In particular, DQM uses histograms to save the data into files that have to be registered in the DQM GUI index so that shifters or experts can use its powerful features to determine the quality of the data.

The GUI's index is the "database" where histograms are stored in an efficient way for quick readout[3]; this special format is used to allow multiple users extract substantial amounts of information in the form of histograms in hundreds of milliseconds. It is designed as a multiple-reader-single-writer database: this choice influenced the design of A&D as it sets the pace of the interaction between the files and the GUI.

3.1. Online Realm

In online DQM has several consumers running in parallel on a distributed system. These consumers are also controlled by DAQ, but their functionality is slightly different from the filter units: among other things, the output of each process is a DQM file per consumer containing histograms usually associated to one specific detector. The input is provided by connecting to one of the streams available in the SMs.

The DQM files are left on the server machines where the consumers run; due to legacy real time constraints, the need to rapidly and indistinctively terminate DAQ processes and the instability of the ever-evolving DQM analyzers' code, files were written out with partial results at fixed-time intervals. More recent and stable versions of the online consumer infrastructure [4], allow for the final file to be actually saved at the end of the run. The idea of partial files being saved is nevertheless still in practice to have a fall back solution in case of problems.

The first problem that A&D need to solve is how to distribute the files from the server machines to the two instances of the online GUI without overwriting files with the same name and selecting the correct version from a final root file or a temporary file. The second issue is how to transfer the files from the online network to the general-purpose network (GPN) so that they can be properly archived. Finally, due to hard disk space restrictions in the GUI servers, the index size has to be constrained and a sample delete strategy is required.

An additional requirement, derived from the experience collected from the continuous running of the experiment, suggested that audio alarms should sound when certain conditions are detected in a selected list of histograms in the GUI.

3.2. Offline Realm:

In the offline realm DQM has three official data collection servers: in these servers DQM files, produced in different LCG sites, need to be registered in order to be accessible by the GUI. Allowing the GUI to accept and save files posted by a POST method using the HTTP protocol solved this

problem. However the GUI does not automatically registers the files, as files for the same run/dataset may come with only partial results. For example, in the express scenario, results need to be shown before the entire run is processed: thus, partial files are sent from the harvesting process. This presented a great load on the server, where multiple files from the same run/dataset could appear within seconds from each other, wasting the time spent registering the first files as the new files would replace the existing information. So, a smart selection was needed. Also, under certain circumstances, the servers will receive different types of files belonging to the different scenarios: in this case an order of registration needed to be established to prioritize, e.g., express data files over Monte Carlo ones.

The offline servers have much better disks capacities than the online ones, so the index size is not an issue. Nevertheless, it soon became clear that the amount of DQM files in the system would cause a quick overflow of the disks. Therefore, one of the first issues is to deal with the version of the files, making sure that when more than one version is available for registration, only the latest one is registered to save time, and all the rest erased from the repository to save disk space.

Unfortunately the compress ratio between a DQM file and its equivalent size on the index is about seven to one: this means that the amount of files on disk grows seven times faster than the index size. The amount of root files being kept around, even if unique, had also to be controlled in order no to fill the disks with root files, subtracting precious space to the index.

Since the GUI provides access to the files stored in the repository for download, a balance was needed between free disk space and files kept around for user download. Also, some users preferred to access the DQM files through AFS, therefore, a strategy was also needed in order to keep a reasonable amount of files there since the space available is very limited.

Finally, all the root files needed to be sent to CASTOR for permanent storage on tape. But sending each individual file turns out to be inefficient in terms of tape usage and file allocation: therefore, they needed to be bunched together into zip archives and then sent for permanent storage.

4. Agents and Daemons: a modular solution

The main problem in the data flow is how to move the DQM files around in a safe way that provides a certain level of tracking and allows for robust manipulation.

The answer to the challenge consists of two parts: the first part was to derive specific tasks so that each A&D module would be in charge of exclusively that one. The second part was to sort them into processing chains so that a clear flow of the information was achieved and it was easy to assess the current status of the system.

4.1. What makes an agent and what makes a daemon

Due to the modularity of the system, the per-task division, and the condition to run permanently, the difference between an agent and a daemon relies on the Unix definition of daemon: a daemon is a process that is forever running and is detached from any control terminal. On the contrary, an agent is a process that runs one cycle of operation and then exits: therefore agents are usually run by a cron job.

As it is presented later in this section, most modules are daemons, and only two were designed as agents. This disparity comes from requirements like AFS credentials which were only safely renewable through the “acron” interface, or by function specification, where it came in the form of a short lived watch dog.

Python was the language of choice to implement the modules due to its widespread use in CMS and the facilities that it provides: readability in the form of compact code writing which makes code revision among peers a lot easier and its object oriented approach, which makes it a very powerful scripting and application development tool.

To standardize construction as much as possible, each A&D module consists of a combination of the following elements: a drop-box (input directory), a series of output directories and some configurable parameters set by command line or by configuration file. From the program structure

point of view, each module has an endless loop with a try…catch block that only quits on a KeyBoardInterrupt[1] exception and a sleep time at the end of the cycle.

In case of errors, the sleep time is enforced and the loop is attempted once more. This approach is valuable since it takes care of temporal problems, which, once solved, allow the system to resume operations without the need to restart its components. Other types of problems are handled internally by the module design. If the problems are in the module itself, the module undergoes a serious revision and an update is installed after thorough testing in a lab installation.

4.2. Tale tales, the Info files go to work

The average size of a typical DQM file is around 200MB: this makes it impractical to repeatedly move it around as it increases the chances of corrupting the file in unrecoverable ways; this also applies for the zip archives made out of compilation of several DQM root files. Also, there's the need of keeping track of some metadata that can be preserved forever and be used to keep track of which files have been processed by the system. The solution to both problems is the info file; it is a simple string representation of a python dictionary object containing the metadata information for the file it is associated to. In the case of the DQM root files the metadata .dqminfo file has the following structure:

```
{ 'check': 'VERIFY: Good to go',
  'class': 'offline_data',
  'dataset': '/Commissioning/Run2010B-Nov4ReReco_v1/DQM',
  'era': 'Run2010',
  'filepat':
'OfflineData/Run2010/Commissioning/0001473xx/DQM_V%04d_R000147390__Commiss
ioning__Run2010B-Nov4ReReco_v1__DQM.root',
  'import':
'/dqmdata/offline/uploads/0001/DQM_V0001_R000147390__Commissioning__Run201
0B-Nov4ReReco_v1__DQM.root',
  'md5sum': '1fb9a119768f0b0c50c9ced73ebc46ce',
  'origin':
'/dqmdata/offline/uploads/0001/DQM_V0001_R000147390__Commissioning__Run201
0B-Nov4ReReco_v1__DQM.root.origin',
  'path':
'OfflineData/Run2010/Commissioning/0001473xx/DQM_V0001_R000147390__Commiss
ioning__Run2010B-Nov4ReReco_v1__DQM.root',
  'primds': 'Commissioning',
  'procds': 'Run2010B-Nov4ReReco_v1',
  'runnr': 147390,
  'size': 64975210
  'tier': 'DQM',
  'time': 1289552517,
  'version': 1,
  'xpath': '/Full/Path/DQM_V0001_R000147390__Commissioning__Run2010B-
Nov4ReReco_v1__DQM.root',
  'zippath':
'OfflineData/Run2010/Commissioning/DQM_Offline_Run2010_Commissioning_R0001
473xx_S0002.zip',
  'zippat':
'OfflineData/Run2010/Commissioning/DQM_Offline_Run2010_Commissioning_R0001
473xx_S%04d.zip'
}
```

Figure 2. DQM Info file structure

In the case of the .zinfo files, used for the zip archives, the file contains the following elements:

```
{
  'frozen': 1290158144.5512209,
  'stime': 1290158995,
  'tries': 0,
  'vtime': 1290185504.1519611,
  'vtries': 1,
  'zactions': 1,
  'zmtime': 1289552524.626848,
  'zpath':
'OfflineData/Run2010/Commissioning/DQM_Offline_Run2010_Commissioning_R0001
473xx_S0002.zip'
}
```

Figure 3. .zinfo file structure.

These info files are created in the same location where their corresponding root files or zip files are located; they are then hard linked into the input or output folders of the different modules. In the case of the dqminfo file the information is created once and never updated throughout the chain: this makes this file suitable to be handled in parallel chains as long as no modification is made to the content to guaranty consistency. The zinfo file is modified by each client, forcing it to be part only of a serial chain. The current state of the A&D does not account for thread or multi-process safety for writing these files.

4.3. The processing chains

Once the modularity of each task is established and the mechanism to arrange the sequence of tasks is in place, all that is needed is to define the agents and arrange them in their proper order to complete the desired action.

Currently four processing chains are defined. The first one is the online chain, which takes care of collecting the files from the consumer servers and putting them in a shared location on NFS; from here the receive daemon takes over and assigns the correct version numbers to the files and puts them in the drop box of three Import daemons which will register the files in three independent instances of the GUI: two production and one backup server. Associated with this chain are the Sound Alarm daemon and the delete daemon. They have no interaction with the root files, but they do have interaction with the GUI index: therefore they follow the standard daemon design.

The next chain is the Online-Offline one. This chain involves the production Online GUI that is accessible from the GPN and its root file download interface; it simply downloads the new root files directly to the offline repository and creates its respective dqminfo files from fake metadata as some of the information is not available. The reason is that these files will skip registration but need to be zipped and later erased; therefore the fake metadata files are necessary.

The offline chain starts when the receive daemon picks the files from the upload directory, put them in the repository with their respective info files and finally send the info file to the drop box of the import daemon and the zip daemon. The import daemon registers the file into the index following the constraints mention before about versioning and priority. The subsequent version control daemon ensures that only the latest version for each file is kept. The root file control quota daemon enforces specific quotas, based on the data type, thus controlling the amount of disk space used by the repository. This chain has a helper daemon called the IndexMerge daemon: it only comes into play when massive updates to the index are required, in which case a series of smaller indexes are prepared offline and then are interlaced with the normal registration procedure to allow the update and the new information to work concurrently.

Finally the zipping chain starts when the receive daemon from the offline chain puts a dqminfo file in the zip agent dropbox. The zip agent will append the files to the proper archive until it reaches at

least 1.8GB in size. During this period the zinfo file is available for the next agent who will check if the file is ready for transfer, either because it has reached the 1.8GB size or because it has been without modification for more than a week. Once the file is ready to be moved, it is frozen by setting the timestamp in the frozen field of the zinfo file, and passed to the next agent which copies the file to CASTOR. Once the copy is complete the next agent ensures that the file has been properly copied to tape, and then it puts it in the dropbox of the clean agent that removes the archive from disk.

Table 1. Brief module description

A/D	Name	Description
A	visDQMAfsSync	Syncs partially the offline repository with a folder on AFS, run by acron.
D	visDQMCreateInfoDaemon	Keeps watching on a directory and creates dqminfo files for ROOT files that do not have them. Then it puts them in the zip agent drop box for archival.
D	visDQMDeleteDaemon	Deletes runs/datasets based on queue files from the GUI index.
D	visDQMImportDaemon	Adds ROOT files to the index.
D	visDQMIndexMergeDaemon	Not shown in the chains, is a helper daemon. It is useful to merge partial indexes with the main index, coordinating with the import daemon.
D	visDQMOnlineSyncDaemon	Downloads ROOT files from the online GUI into the offline root repository.
D	visDQMReceiveDaemon	It picks up files from the upload directory, checks the naming conventions, and creates dqminfo files.
D	visDQMRootFileQuotaControl	Keeps ROOT files' repository disk under control.
D	visDQMVerControlDaemon	Removes older version of ROOT files, only keeping the latest available.
D	visDQMZipCastorStager	Stages zip archives in castor for permanent storage
D	visDQMZipCastorVerifier	Verifies that the archives have been copied correctly.
D	visDQMZipDaemon	Creates zip archives with the ROOT files.
D	visDQMZipFreezeDaemon	Prevents new info being added to an archive, and marks it as ready for transfer to castor.
A	aliveCheck(2).sh	Ensures that all A&D are running.
D	fileCollector2.py	Collects files from producer machines an upload to the online receive daemon dropbox.
D	producerFileCleaner.py	It keeps the disk usage of a local repository of the producer machines under control.
A	daily-offline	Among other things it removes the zip archives from the zipped repository that have been successfully transferred to castor.

5. Identified Problems and Future updates

As expected, when the complexity of the system grew, a few problems where encountered. The most current and upstanding problem is a stopping issue that causes index corruption when the delete, merge and index daemons are stacked on top of each other waiting to carry out some work on the index. All write operations to the index are done by an independent executable that is called form the daemons: it is possible that at least one of the stacked daemons is able to fire a writing job over the index without the previous writing job actually finished as, by design, the writing jobs are not

interrupted when the daemon exits. As mentioned before, the index is designed for one single writer, but this is not fully enforced: the simultaneous write operation triggered by the stopping procedure could cause the index to overwrite the internal data pointer tables, making them invalid: as a result entire sections of data become unreachable by the GUI.

As new tools are being developed, it has become even more clear that there is a high level of code reuse throughout the A&D system: therefore a new architecture has been proposed where each module becomes a thread in a multithreaded program and all recurrent code is put in to a service framework, starting with a writer service to serialize write request to the index, avoiding the stopping issue described earlier.

Finally, the greatest advantage of the proposed architecture is that it can be configured using python configuration files, in the same way as the DQM GUI does, reducing the complexity of management scripts and allowing the customization of parameters, based on the environment, to be stored in one single place.

6. Conclusions

The A&D system has proven to be a reliable way to automate routine system operations. Furthermore the metadata info files mechanism has proven to be a reliable and successful way to keep track of what has been done; it also saves enough information that allows doing pre-selection before actually downloading full archives from CASTOR. It also has proven to be an easy way to distribute the system over multiple servers.

Corrupted indexes have been reconstructed with information available on disk and files stored in CASTOR: so far we have not suffered an unrecoverable data loss scenario.

The system architecture proved to be flexible enough to run in four different offline servers plus the online simply adapting the proper configuration files.

Operator intervention has been greatly reduced and, in case of problems, the system provides enough information to easily pin point them; once they are fixed, it resumes activity without any inconveniences.

The system has undergone major hardware upgrades twice and, in both occasions, operations were resumed within seconds of finalizing the copy of the repository: this meant very small downtime (measured in minutes) to move from the old server to the new one.

One of the great advantages of the system is that to maintain the running system or make new modules and modifications, only a good understanding of python and a good familiarity with file management at an operating system level are needed. No additional skills are required, e.g. database management or complicated replication models. This makes this an ideal tool when the human resource is scarse, and hard to come by.

References

[1] Python documentation http://docs.python.org/release/2.6.8/
[2] CMS Collaboration 1994 CERN/LHCC 94-38 *Technical proposal* (Geneva, Switzerland)
[3] CMS data quality monitoring web service
L Tuura *et al* 2010 *J. Phys.: Conf. Ser.* **219** 072055 doi:10.1088/1742-6596/219/7/072055
[4] CMS Online Data Quality Monitoring: Real-Time Event Processing Infrastructure Srecko Morovic for the CMS DQM Group