Tools to use heterogeneous Grid schedulers and storage system

Mattia Cinquilli[*†]
INFN and Università di Perugia
E-mail: mattia.cinquilli@pg.infn.it

Giuseppe Codispoti
INFN and Università di Bologna
E-mail: giuseppe.codispoti@bo.infn.it

Alessandra Fanfani
INFN and Università di Bologna
E-mail: alessandra.fanfani@bo.infn.it

Federica Fanzago
INFN Padova
E-mail: fanzago@pd.infn.it

Fabio Farina
INFN Milano
E-mail: fabio.farina@cern.ch

José Hernández
CIEMAT
E-mail: jose.hernandez@ciemat.es

Stefano Lacaprara
INFN Padova
E-mail: stefano.lacaprara@pd.infn.it

Hassen Riahi
INFN and Università di Perugia
E-mail: hassen.riahi@pg.infn.it

Daniele Spiga
CERN
E-mail: daniele.spiga@pg.infn.it

Eric Vaandering
FNAL
E-mail: ewv@fnal.gov

Stuart Wakefield
Imperial College London
E-mail: stuart.wakefield@imperial.ac.uk

The Grid approach provides a uniform access to a set of geographically distributed heterogeneous resources and services, enabling projects that would be impossible without massive computing power. Grid computing both provides the resources that allow scientists to run jobs over large amounts of data and also allows these data to be distributed all over the world, through the storage systems. Different storage projects have been developed and many protocols are being used to interact with them such as GsiFtp and RFIO. Moreover, during the last few years various Grid projects have developed different middleware such as gLite, CondorG, ARC, etc., and each one typically implements its own interface. For a user community that needs to work on the Grid, interoperability is a key concept. An efficient usage of the distributed resources is required to manage the heterogeneity of the involved resources. To handle different Grid interfaces and the resource heterogeneity in a really transparent way we have developed two modular tools: BossLite and Storage Element API. These deal respectively with different Grid schedulers and storage systems, by providing a uniform standard interface that hides the difference between the systems they interact with. BossLite transparently interacts with different Grids, working as a layer between an application and the middleware. Storage Element API implements and manages the operations that can be performed with different protocols used in the main Grid storage systems. Both the tools are already being used in production in the CMS computing software which uses the Grid services and resources for distributed analysis and Monte Carlo production. In this paper we describe their implementation, how they are used and give some result of performance measurements.

13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research, ACAT2010
February 22-27, 2010
Jaipur, India

[*]Speaker.
[†]Presented on behalf of the CMS collaboraton.

1. Introduction

Highly distributed systems are becoming a standard choice for implementing computing systems used in modern scientific research. The choice of a distributed system allows resource sharing, thereby reducing costs and delegating responsibilities to the individual institutes and participating national organizations. It also permits load balancing of the available resources through replication of selected data sample in different sites. A large community of scientists spread throughout the world is engaged in the CERN-based Large Hadron Collider project and it is therefore natural that a distributed computing model has been adopted for the processing and storage of experimental data. Distributed systems offer high performance and scalability, whilst assuring constant availability of the data through replication of data over many sites. The Grid middleware interconnects a wide variety of heterogeneous resources that have been selected independently at the various sites. Moreover, the funding and development of middleware software tends to be organised on a regional basis and this leads to a number of different implementations of middleware software stacks; the EGEE project [2] is responsible for Grid implementation and deployment in Europe; the pre-existing OSG project [3] connects facilities in the United States; other national projects, such as NorduGrid [4], are also deployed.

Even through these different middleware stacks have been designed to inter-operate, they typically use different applications and usually expose different interfaces. Thus any specific workload management and data management tool implemented by end-user organizations must be able to deal with these differences in order to gain access to distributed resources that are using different middleware. We have built a set of libraries that hide implementation details from end users enabling the access to both local and Grid resources through a heterogeneous environment. They provide:

- a generic interface to user job description language, job submission, tracking and handling provided by different Grid and batch systems
- a generic interface to storage resources through the standard access and transfer protocols.

With this approach, any resource implementation specific behavior is localized and any new solution can be easily implemented in a single place. High level tools can use the underlying resources in the most efficient way through a generic standard interface, while the intermediate level takes care of an optimal integration with a continuously evolving environment that is typical of the middleware.

2. Design strategy

The aim of the project was to provide flexible and friendly services to transparently use different Grid middleware stacks and storage systems. This led us to develop two libraries, BossLite and SEAPI, using a common architecture based on a set of classes and a collection of subroutines to be used within a high level application. This approach facilitates an easy extension, sharing and changing of code and data in a modular way. The modular approach is an important technique to compose the libraries from separate modules. Both libraries implement an application programming interface (API) that provides a standard front-end to an end-user application and easily handling of the specialized plug-ins that interact with the given system (middleware or storage). Each plug-in handles a specific system and contains the specific implementation of the methods referenced by the library interface. The libraries themselves guarantee interoperability even when implementing a new plug-in (corresponding to a new system) without having to change anything else within the libraries themselves. The API interface provides an abstraction of the library itself, separating external communication from internal operations. This also allows modifications of the library to be made without having any effect on the external entities that interact with it. Whilst the approach introduces a small overhead, it offers an important advantage in terms of maintainability by enforcing logical boundaries between modules with APIs and plug-ins.

An important requirement was to have a thread safe library since the applications that use these libraries work with threads in order to speed up the interaction with the middleware and storage systems. Above all, the BossLite and Storage Element API libraries work as light and tiny frameworks, which do not add much overhead and which are easy to use with respect to the advantages they offer.

3. Implementation of the scheduler: BossLite

BossLite is the Python library developed to interface the Workload Management tools to the Grid middleware and local batch systems. BossLite provides a generic abstract interface, enabling standard operations such as job submission, tracking, cancellation and output retrieval. Scheduler-specific interfaces are implemented in specific plug-ins, loaded at run-time. Presently, plug-ins are implemented for EGEE (gLite), OSG (CondorG) and NorduGrid (ARC) middleware stacks and LSF, Condor, PBS and SGE batch systems. BossLite relies on a database to track and log information into an entity-relationship schema. Information is logically remapped into Python objects that can be transparently used by higher level frameworks and tools.

BossLite must be highly efficient so as to not introduce a bottleneck in Grid operations and to provide safe operation in a multi-processing/multi-threaded environment. For that reason, the interaction with the database is performed through safe sessions and connection pools. A database session provides access to database operations through a generic abstract interface, enabling standard operations such as open/close connection, query, insert/update rows and so on. In the current implementation, two subclasses provide support for MySQL [6] and SQLite [7] databases. Support for other databases can be added by creating new subclasses. The usage of database connection pools allows a set of open connections to the database to be kept thus avoiding the continuous creation and closure of them for each interaction with the database. In the very same way, the Scheduler part is composed by a set of high level APIs, that implement the default behavior and connect the Scheduler and the Database part. The high level scheduler API depends on the Database high level API, allowing coherent update and handling of the information to be stored in the database. The BossLite architecture is showed in Figure 1.

3.1 Job description and bulk operations

The BossLite database structure reflects a typical high energy physics analysis use case, which usually requires access to a huge amount of data, typically split in many files. Since, in general,

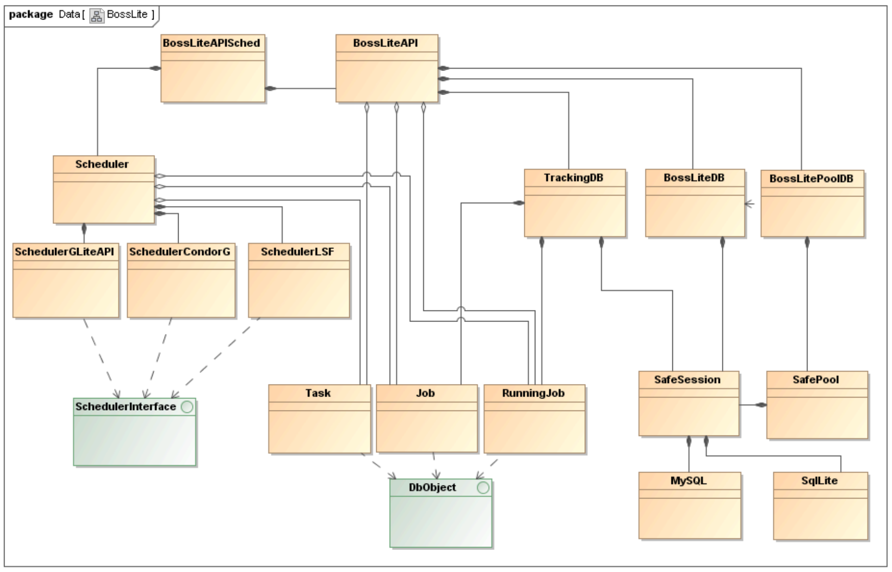

Figure 1: Schematic view of the BossLite Architecture.

the same algorithm has to run over the whole sample, the analysis task can be split into many jobs that may run in parallel. The only difference between them is given by the parameters identifying the portion of data to be accessed. This is a very general use case; the same procedure applies to Monte Carlo data simulation, where data can be produced by parallel jobs in small files and merged later. In general, jobs submitted to the Grid split and parallelize a huge processing task with the aim to reduce the total wall-clock time needed to get a result by using the distributed resources. For that reason the set of jobs realized for a given task share a huge common part (usually also in terms of input files) and are distinguished by small peculiarities. This allows the implementation of bulk operations at the middleware level, which translates in practice into compound jobs such as the DAG in CondorG and collection and parametric jobs in gLite.

The BossLite job description is designed to easily make use of this feature in the scheduler specific interface implementation. A top entity called task groups similar jobs, storing common information such as common job requirements, the dataset to be accessed/produced and so on. Jobs are then provided with a static part, characterizing the job itself by recording, for instance, arguments and file names, and a dynamic part. The latter stores runtime information such as scheduling timestamps, job status and destination. Since a job may fail for many reasons, there may be as many resubmissions as needed: they are stored in "running instances" in order to record the execution information of every submission for logging purposes. A job may have a running instance for each resubmission. A schematic view of the database structure is shown in Figure 2.

BossLite features can be better exploited by the high level tools using the described logic. The actual implementation is flexible enough to allow many other policies, such as collecting everything under a single task and mapping one-to-one jobs and their running instances.

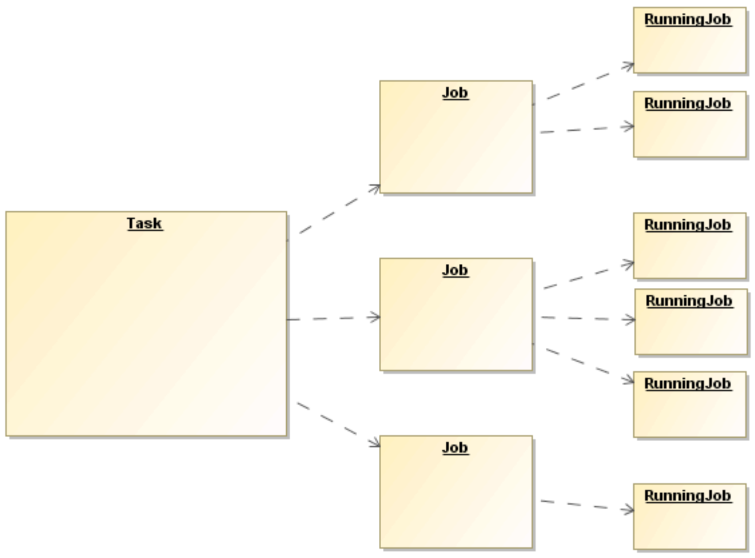

Figure 2: Task information mapping in the BossLite database.

4. Implementation of the Storage API

A Storage Element provides Grid interfaces to site storage systems, giving uniform access to storage resources through the support of different protocols and operations. There is not a particular correlation between the kind of storage system used by a site and the Grid middleware supported by the site: in a single middleware environment different storage systems can be used. Also, different protocols can be used to perform an operation, depending on the storage system and its interface, on the specific use case (because particular operations possible just with a well defined protocol) and on the site reliability.

SEAPI (Storage Element Application Program Interface) is a tool that has been developed to handle operations such as file transfers and removal on many storage facilities by using different protocols. It works as a framework which collects the specific requests and performs the operations, translating every single request into the command to be performed. The main operations that are currently being supported are:

- copying or moving a list of files from one storage to another one or from a local system to a storage (or viceversa); this includes automatic sub-directory creations on the destination storage;
- checking if a file or directory exists;
- getting the file or directory permissions;
- changing the file or directory permissions;
- getting the file size;
- deleting a file from the storage;

- creating directory;
- listing directory content.

For some protocol it is not possible to support all the functionalities listed above (eg: lcg-utils does not directly support permissions setting).
The main goal of the development has been the transparent interaction with many kinds of Storage Element, completely hiding the heterogeneity given by their interfaces and the functionality provided by different protocols. The main points considered during the development and that are being considered during current improvements are:

- to provide a complete transparency of usage by a common interface that links each operation that can be performed, independently of the protocol; this results in a tool that works with remote files as if they were local;
- the interaction between different protocols when executing operations that include two distinct storage resources (e.g.: copying data from a storage to another one);
- to handle with accuracy different kind of errors and failures, depending on the storage, the protocol and the operation being performed.

4.1 Architecture

To solve the above listed problems SEAPI is structured as a modular tool that takes into account the heterogeneity of the storage systems, protocols and errors. This approach allows to have an extensible tool with the generic concept of protocols and storage. The main architectural blocks are:

- **interface**: a generic entity that loads and translates a call depending on the protocol associated to the *storage* defined and the operation requested; it can be considered as an intermediate: the *client* (the entity that uses SEAPI) makes a uniform call through the *interface* which hides different protocol needs;
- **storage**: class referenced by the *client* which creates a storage object adding it to the *interface*, more *storage* objects can be created and passed to the *interface* to perform transfer operations between two storage elements; when more storages are provided to the interface, the requested operation is performed on the first given storage;
- **plug-ins**: each plug-in represents a different protocol and contains the right set of the protocol operation instructions; these are loaded dynamically when the *interface* and the *storage* need to initialize an operation;
- **error manager**: set of methods that look at the operation results (such as the output of the command execution and the related exit code) and that can recognize if there has been a failure or not, and which kind of failure happened; if so, the specific method prepares a report with the specific error message, a short explanation of it and the whole output of the failed operation;

- **exceptions**: groups and maps the errors into specific exceptions raised by the *plug-ins* with the details provided by the *error manager* method.

In Figure 3 an example of SEAPI flow when being used to perform an operation is shown. When

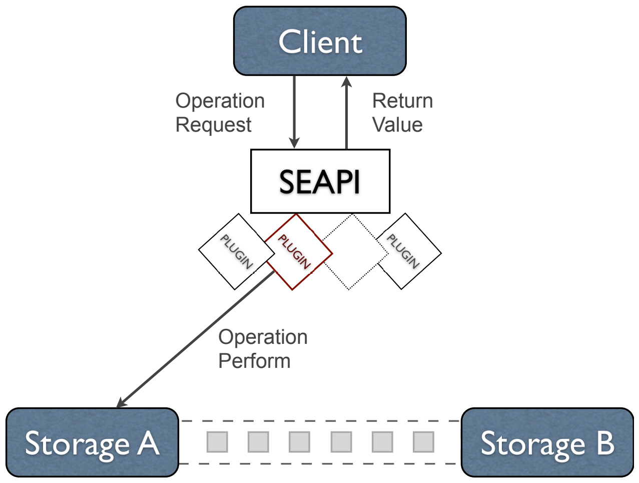

Figure 3: SEAPI example of usage

an operation request is made by a client, SEAPI has a common and generic interface which works as an intermediate that collects and translates the request. This interface chooses between different set of commands depending on which operation is requested and which protocols are involved looking at their compatibility. Then the interface itself initializes and dynamically loads at runtime the right plug-in which contains the right set of instructions. The selected plug-in performs the operation, by specifying the main generic options for each command associated to the operation and the options received as parameter by the client and by the interface (e.g.: virtual organization, operation timeout, specific protocol parameters, etc...).

SEAPI also manages the credentials, typically independent from the protocol but not from the plug-in. Two types of credentials are being supported: proxies generated by certificates and kerberos token.

5. Current status in CMS

The components described above have been integrated with the relevant tools in the CMS Workload Management infrastructure [10], namely the CMS Remote Analysis Builder (CRAB) [8] and ProdAgent (PA) [9]. These are CMS specific tools designed and developed to support end user distributed analysis and Monte Carlo data production respectively.

These tools have been developed with the aim of using the generic Grid tools to manage the distributed resources provided by the World-wide LHC Computing Grid (WLCG) [11] and the Open Science Grid (OSG). After the design and at the time of writing, NorduGrid middleware has begun to be supported by the CMS Collaboration. From early on, middleware interoperability was one of the main goals for the top level CMS tools (CRAB/PA).

5.1 BossLite usage in CMS

In the last 6 years at CERN many different Computing implementations have appeared and existing middleware has evolved. The interoperability provided by BossLite represents crucial development. CMS [12] is one of the most important HEP experiments with significant computing requirements. There is a clear need to test, evaluate and compare all possible production middlewares and services in order to develop, design and optimize the CMS Computing Model. Having the opportunity to use multiple interfaces to interact with heterogeneous resources and services, facilitates the CMS computing development and gives an important boost to comparisons of Grid middleware native tools. From the scheduler's point of view, BossLite allowed CRAB to transparently interact with gLite WMS [13], CondorG [14], NorduGrid, and also with local batch systems, such as LSF, PBS, and SGE. The support of local batch systems was crucial, since the CMS Computing Model includes a dedicated layer for low latency activities so as to gain access to all raw data: it combines flexible CPU resources with rapid access to the entire CMS dataset for fast and "hot" analyses running at the CERN Analysis Facility (CAF) [15].

Figure 4 shows the number of jobs submitted using the ProdAgent and CRAB job submission tools during the last year. We observed a peak of 200000 jobs per day; typically analysis jobs averaged 35000 jobs per day and production jobs 49000 jobs per day.

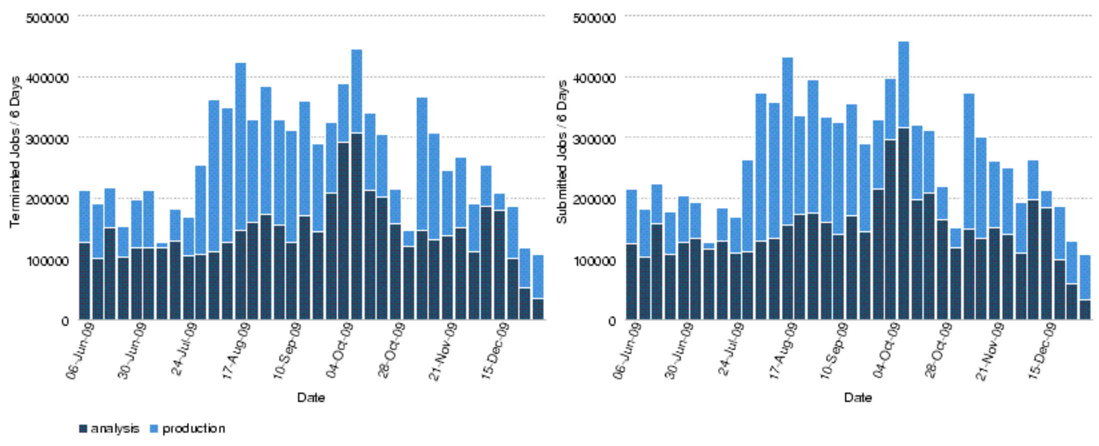

Figure 4: Number of analysis and production jobs per day from June to December 2009 (completed jobs on the left, submitted jobs on the right)

5.2 StorageAPI and CMS integration

The Storage API demonstrated to be important in the CMS analysis tool, mostly in the implementation of client-server sandbox transfers, stage out algorithm (remote and local) and inside the server architecture (interaction with the close storage). The server architecture adopts a modular software approach, made of independent components (Agents), implemented as daemons, communicating through an asynchronous and persistent message service based on a publish and subscribe model. One of the most important external components in the server architecture is represented by a Storage Element. Inside the SE are stored the input and output files of the user wokflows. The close Storage Element is used by the CRAB client to collect the user input sandbox (ISB) during job submission, and then to collect the job output sandbox (OSB) at the end of the execution. The default implementation is GridFTP-based [16], but other flavors are also supported, for instance rfio, used for the CERN Analysis Facility (LSF based) and dCache [17]. Given this heterogeneity, the benefit of the Storage API has been shown. At the client level the SEAPI library instantiates the proper protocol on the fly, based only on the server configuration. Within the server, the integration is done to enable the components to interact with the storage area: for cross check, file consistency and movement purposes. The final important benefit of such an intermediate layer is in the stage-out algorithm developed and integrated with CRAB. Here transparency allowed defining a light weight Python module implementing just the logic, with fallback and sanity checks, for the stage-out of the produced files. For this purpose the real benefits of the Storage API are the standardization of the errors reported, and the easy usage of different storage clients.

5.3 Overall Performance of CMS tools

In Figure 5 are shown the number of jobs submitted by CMS users through different middleware systems which have different interfaces and workflows. Most jobs are submitted through:

- *gLite*: which is the actual version of the EGEE project middleware;
- *CondorG*: the middleware version of the OSG project.

The reason why the *LSF* and *ARC* schedulers usage is low is because they were integrated in the official tools at a later time in the development cycle of our libraries. All the job submission and most of the related sandbox transfers are performed by BossLite and the Storage Element API respectively.

As explained and shown, it is important to have a common interface but it is also critical that the interface be rich enough to give different implementations enough flexibility to optimize for performance. So, it is important to compare the performance of the proposed tools with that of the underlying Grid systems. Concerning *gLite* (the most used scheduler) dedicated tests in CMS demonstrated that a single CRAB Server instance in multi user mode can reach 50000 jobs per day using 2 instantiations of WMS (the resource broker part of the *gLite* middleware). Figure 6 shows the incremental number of jobs a single CRAB server handled over two days. The jobs are grouped by status: the jobs in *Cleared* status shown in red color indicates that about 97.5% of the jobs have been processed and already archived by the server; the *Done* status groups the jobs that have not yet been processed by the server (about 1.1%). The rest of the jobs are still pending or running and being handled by the middleware, so the server has to wait for these. Then after these tests we concluded that at a rate of about 50000 jobs per day a single server accumulates a delay on processing about 1.1% of the jobs. This is an acceptable rate that can also be improved in the future by optimizing the applications code. Using the same scheduler a single ProdAgent instance reached around 30000 jobs per day. In both cases, BossLite and Storage Element API did not prove to be a bottleneck, while the middleware reached its limits before the CRAB and Prodagent tools. All the jobs referred to in Figure 6 and Figure 5 were submitted to a large number of sites: this is normal behavior for the CMS computing infrastructure, since official sites that are actually collaborating and sharing their resources are 100. Each site is independent and locally

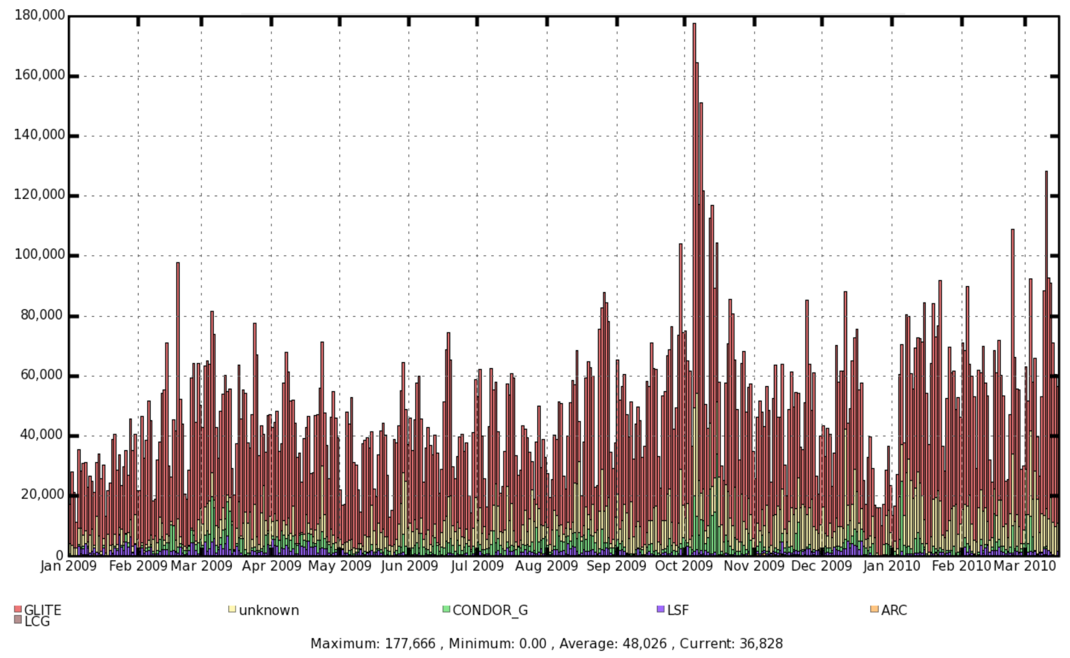

Figure 5: Distribution of the analysis jobs by schedulers, from January 2009 till March 2010.

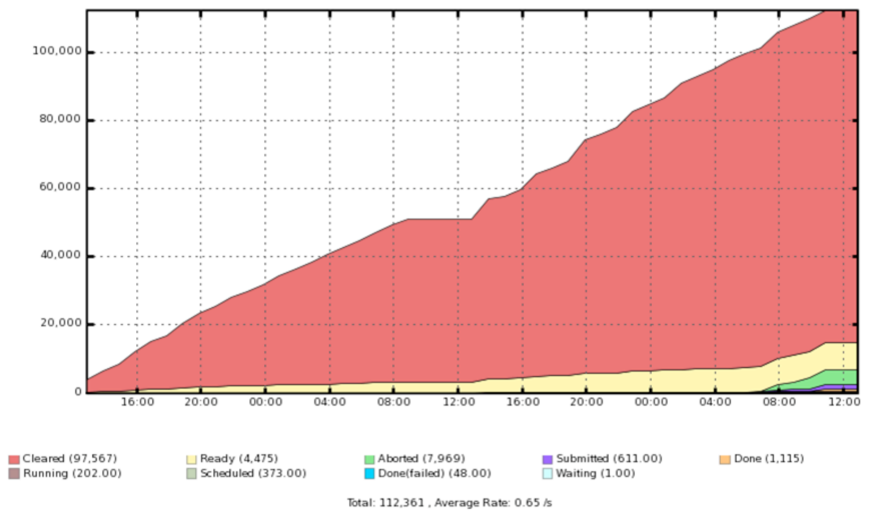

Figure 6: Crab Server and gLite middleware usage; number of jobs by status.

managed, also it has free choice on which storage element to maintain and support. This means that different sites support different storage independently from which middleware they are part of. Figure 7 shows the distribution of analysis jobs amongst all CMS sites. Given this environment the Storage Element API allowed us to improve the site usability from a storage point of view. Also, in the analysis workflow, the storage is a crucial point also because it is used not just to read the experiment data, but also to stage out the user analysis results. The number of user per day during

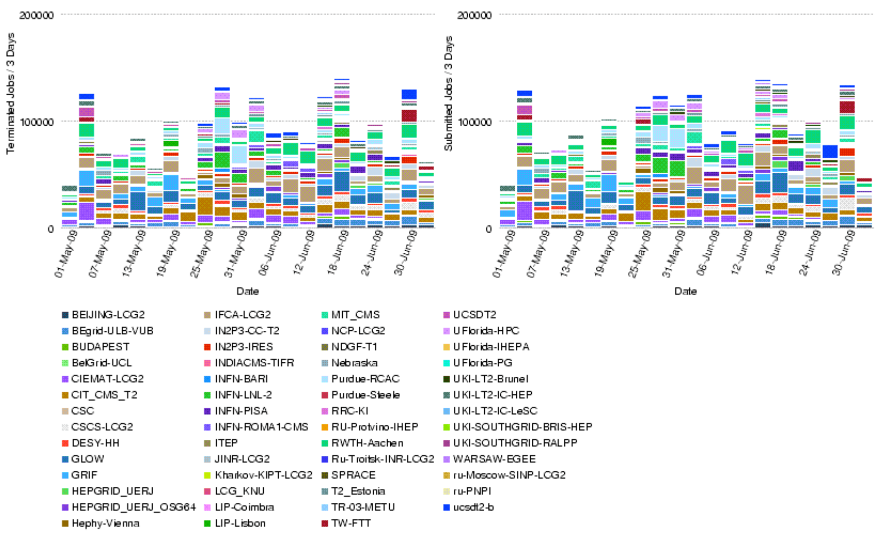

Figure 7: Jobs distribution between CMS Grid sites

2009 are shown in Figure 8.

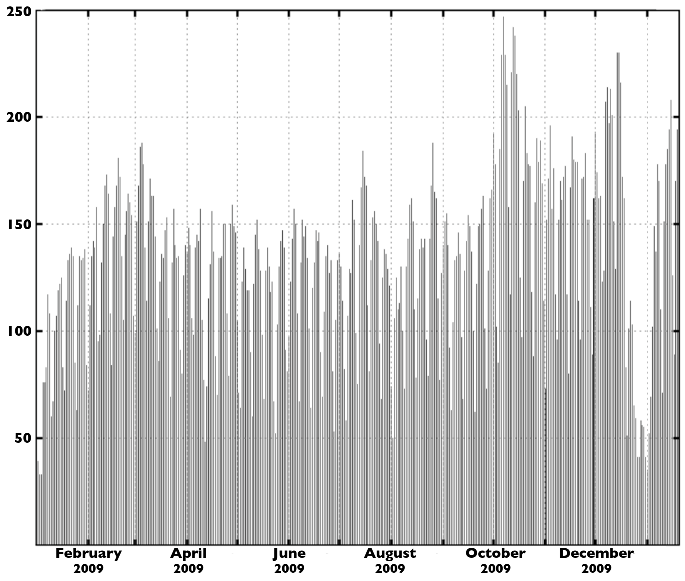

Figure 8: Users per day from June 2009 to January 2010

6. Conclusions

The CMS Workload Management tools use dedicated libraries to satisfy the data production and analysis needs and to optimize the infrastructure and resource usage, also by transparently handling heterogeneous systems. Interacting with several middleware, storage and batch systems requires to handle heterogeneous resources that may differ for implementation, software version or even specific configurations adopted by local sites. A design strategy with a common interface for the plug-ins that handle specific system implementation is solution that gives significant benefits to the Grid community and related tools.

A set of python libraries have been successfully developed to interface the Grid middleware, local batch systems (BossLite) and Grid storages systems (SEAPI). The developed libraries have been also deployed within the CMS Workload Management tools (CRAB and ProdAgent) since the start of 2008. The most crucial design metrics have been demonstrated both during the daily production usage and during specialized CMS challenges used to verify the readiness of the tools and the infrastructure itself. The last challenge was the Common Computing Readiness Challenge in May 2008 [18] where the CMS Grid sites were tested using realistic analysis workflows, and more importantly, having real users submitting a large number of analysis tasks reaching an unprecedented scale. The scale of jobs submitted meets the number of CMS jobs expected during data taking. In addition to maintenance, the main development effort will be spent improving the interface, especially for the Storage Element API. We will also optimize the error and exception management, which will be a key point for the usability of such layers. Given the obtained results, in the near feature BossLite will be fully integrated with the whole CMS Workload Management system. Our experience shows that these layers give very good results in terms of reliability and usability.

References

[1] LHC Homepage, http://cern.ch/lhc-new-homepage/

[2] Enabling Grids for E-sciencE (EGEE) Portal, http://www.eu-egee.org

[3] PORDES, R. et al., "New science on the Open Science Grid", Journal of Physics: Conference Series 125:012070 (2008).

[4] P. EEROLA et al., "The Nordugrid production grid infrastructure, status and plans", Proceeding of Fourth International Workshop on Grid Computing, 2003, ISBN: 0-7695-2026-X

[5] G. VAN ROSSUM et al., Python Language Website, http://www.python.org

[6] MySQL, http://www.mysql.com

[7] SQLite Home Page, http://www.sqlite.org

[8] G. CODISPOTI et al., "CRAB: A CMS application for distributed analysis", IEEE Trans.Nucl.Sci.56:2850-2858 (2009).

[9] D. EVANS et al., "The CMS Monte Carlo Production System: Development and Design", 18th Hadron Collider Physics Symposium 2007 (HCP 2007) 20-26 May 2007, La Biodola, Isola d'Elba, Italy. Published in Nuclear Physics B - Proceedings Supplements, Volumes 177-178 (2008) 285-286

[10] D. SPIGA for the CMS Collaboration, "CMS Workload Management", Nucl.Phys.Proc.Suppl. 172:141-144,2007

[11] LHC Computing Grid (LCG), Web Page, http://lcg.web.cern.ch/LCG/ and LCG Computing Grid - Technical Design Report, LCG-TDR-001 CERN/LHCC 2005-024, (2005)

[12] S. CHARTRCHYAN et al., "The CMS Experiment at CERN LHC", Jornal of Instrumentation, vol 3, pp.s08004 (2008)

[13] P. ANDREETTO et al., "The gLite Workload Management System", Proceedings of Computing in High Energy and Nuclear Physics (CHEP) 2007, Victoria, British Columbia (CA), Sep 2007.

[14] D. THAIN et al., "Distributed computing in practice: the Condor experience", Concurrency - Practice and Experience, Volume 17 (2005) 323-356

[15] CERN Analysis Facilty Web Site, https://twiki.cern.ch/twiki/bin/view/CMS/CAF

[16] W. ALLCOCK et al., "GridFTP Protocol Specification", GGF GridFTP Working Group Document, September 2002.

[17] G. BEHRMANN et al., "A distributed storage system with dCache", J.Phys.Conf.Ser.119:062014, 2008.

[18] D. BONACORSI, L. BAUERDICK, "CMS results in the Combined Computing Readiness Challenge (CCRC'08)", Nuclear Physics B, Proceedings Supplements, 11th Topical Seminar on Innovative Particle and Radiation Detectors (IPRD'08), Siena, Italy. 1-4 October 2008. vol. 197, pp. 99 - 108 (2008).