Available on CMS information server

CMS CR -2009/071

The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

12 May 2009

PhEDEx Data Service

Ricky Egeland, Tony Wildish, Chih-Hao Huang

Abstract

The PhEDEx Data Service provides access to information from the central PhEDEx database, as well as certificate-authenticated managerial operations such as requesting the transfer or deletion of data. The Data Service is integrated with the SiteDB service for fine-grained access control, providing a safe and secure environment for operations. A plug-in architecture allows server-side modules to be developed rapidly and easily by anyone familiar with the schema, and can automatically return the data in a variety of formats for use by different client technologies. Using HTTP access via the Data Service instead of direct database connections makes it possible to build monitoring web-pages with complex drill-down operations, suitable for debugging or presentation from many aspects. This will form the basis of the new PhEDEx website in the near future, as well as providing access to PhEDEx information and certificate-authenticated services for other CMS dataflow and workflow management tools such as CRAB, WMCore, DBS and the dashboard. A PhEDEx command-line client tool provides one-stop access to all the functions of the PhEDEx Data Service interactively, for use in simple scripts that do not access the service directly. The client tool provides certificate-authenticated access to managerial functions, so all the functions of the PhEDEx Data Service are available to it. The tool can be expanded by plug-ins which can combine or extend the client-side manipulation of data from the Data Service, providing a powerful environment for manipulating data within PhEDEx.

Presented at *Computing in High Energy Physics 2009 (CHEP09),21/3/2009,Prague,Czech Republic,15/05/2009*

PhEDEx Data Service

Ricky Egeland[a], **Tony Wildish**[b], **Chih-Hao Huang**[c]
[a]**University of Minnesota, Twin Cities**
[b]**Princeton University**
[c]**Fermi National Accelerator Laboratory**

Abstract. The PhEDEx Data Service provides access to information from the central PhEDEx database, as well as certificate-authenticated managerial operations such as requesting the transfer or deletion of data. The Data Service is integrated with the "SiteDB" service for fine-grained access control, providing a safe and secure environment for operations. A plug-in architecture allows server-side modules to be developed rapidly and easily by anyone familiar with the schema, and can automatically return the data in a variety of formats for use by different client technologies. Using HTTP access via the Data Service instead of direct database connections makes it possible to build monitoring web-pages with complex drill-down operations, suitable for debugging or presentation from many aspects. This will form the basis of the new PhEDEx website in the near future, as well as providing access to PhEDEx information and certificate-authenticated services for other CMS dataflow and workflow management tools such as CRAB, WMCore, DBS and the dashboard. A PhEDEx command-line client tool provides one-stop access to all the functions of the PhEDEx Data Service interactively, for use in simple scripts that do not access the service directly. The client tool provides certificate-authenticated access to managerial functions, so all the functions of the PhEDEx Data Service are available to it. The tool can be expanded by plug-ins which can combine or extend the client-side manipulation of data from the Data Service, providing a powerful environment for manipulating data within PhEDEx.

1. Introduction

In order to meet the data distribution requirements [1, 2, 3] of the CMS [4] experiment at the LHC, the Physics Experiment Data Export (PhEDEx) [5, 6, 7] project was designed to facilitate and manage global data transfers over the grid. Since its conception in 2004, it has evolved considerably and has been shown to be a robust, reliable, and scalable system which has guided over 62 million transfers of over 67 PB of data. PhEDEx provides a simple mechanism to request the transfer of thousands of files at a time, which then sets into motion an array of special-purpose software "agents" progressing the transfer state machine to the desired end — files on disk at the destination. Reliability is ensured by independently verifying each file transfer when it lands at the destination, robustness is achieved by intelligently backing off when large numbers of failures are encountered. Through numerous challenges and daily use for months, the transfer management layer of PhEDEx is not in question.

PhEDEx lives within a loosely-coupled ecosystem of CMS services, and must provide mechanisms for integration with them. The Dataset Bookkeeping Service (DBS) [8] provides a metadata catalog of CMS data to physicists as well as production and analysis systems, and it refers to PhEDEx to obtain the locations of the data. The CMS Remote Analysis Builder (CRAB) [9, 10] and production system (ProdAgent) [11] similarly need information on the location of data in order to function properly. The Tier-0 [12] production system is the starting point for CMS data, and it needs to invoke PhEDEx to initiate transfers to the Tier-1 centers which have the responsibility to store the data long-term. Finally, monitoring and web-management systems need a way to obtain data from the PhEDEx database for presentation, as well as to execute management actions such as initiating new transfers or deleting data.

In order to provide a single solution for these integration requirements a web-based data service was created, which interacts with the PhEDEx transfer management database (TMDB) to provide the necessary information and to initiate data placement actions. This paper outlines the design and initial usage patterns of the data service.

2. Motivation and Design Considerations

As mentioned in the introduction, the primary reason for developing the data service was to enable the integration of PhEDEx with other CMS Computing services. This section gives the motivation for choosing a web data service, as opposed to some other method. We will compare to two other options: a command-line interface (CLI) or a library (API), which could be packaged and deployed at the clients location.

2.1. Database Connections

Concurrent database connections are a limited resource. The TMDB resides on a 4-node Oracle Real Application Cluster (RAC), and for our application (PhEDEx) it is configured to allow 400 concurrent connections per node. The small size of our data results in Oracle choosing to place our application on only one node of the cluster, so we really only have 400 concurrent connections. The distributed PhEDEx agents, which do the actual transfer management work, currently take about 270 of these connections. A CLI or API integration solution would likely result in us running into the connection limit, which would negatively impact both the integration use-case and the transfer management workflow.

Furthermore, distributing the code in a CLI or API solution would require us to more widely distribute the database connection information. This increases management overhead, as the connection parameters (especially the passwords) are not static. It also increases the risk to inadvertent abuse to the system. There are no mechanisms available to throttle a misbehaving CLI- or API- using program, which is for example querying the same data repeatedly in a tight loop.

Providing a web data service mitigates both of these concerns. The web server will only use a small number of persistent database connections. The connection information need only be distributed to one additional location, the web server. Finally, the accesses to the database can be throttled, either through caching or host-based access limitations.

2.2. Interoperability

CMS Computing is done in a heterogeneous programming environment. Due to the popularity and usefulness of the web, developing a web data service automatically makes the data access "compatible" with almost any programming language or environment that can be imagined, as most have an HTTP library and XML parsers. We avoid having to support multiple languages.

2.3. Expanded Use

Providing our data as a web service makes our data easily available to more than just official CMS computing projects. The "entry cost" for using a web data service is significantly lower than using a client-side library, and as a result we expect more involvement in the processing of our data. One area where we stand to benefit from this approach is the independent development of monitoring applications using the data service. A monitoring application is a significant development investment, and it is difficult to satisfy all users at the same time, as they may wish to visualize the data in other ways. By providing easy access to the data itself, independent developers can provide their own monitoring solutions. Furthermore, because the data is served over the web, it is likely that these monitoring applications can also be served over the web, multiplying their usefulness.

2.4. Modern Web Monitoring Platform

Another motivation for the data service was to use it as a backend to an updated web-based management and monitoring platform. The existing monitoring and management tool is the PhEDEx website, which is a stand-alone piece of CGI code that dynamically serves static webpages. SQL which is used to generate the webpages is not shared by other components of PhEDEx, resulting in low code reuse. By building a web data service we hope to increase code reuse by building a monitoring and management platform on top of it, using modern asynchronous access techniques (Ajax) to provide a better user experience. For this purpose JSON (JavaScript Object Notation) is a notably efficient data transport format, as JSON objects can simply be evaluated in the web browser, eliminating a separate parsing step. This motivated us to enable the data service to provide multiple output formats for the same functions, XML being more common and familiar to many clients, and JSON being a more efficient format in a web browser.

2.5. Design Considerations

Having given the benefits to providing a web data service above, we should now mention one of the detriments: the increased importance of backwards compatibility. Once an API is provided via some URI, any change to that API or URI will break all existing clients immediately. A distributed library or CLI is not impacted by this in the same respect, as the client has the option to not upgrade. A web server is in a sense the single installation of a "library", and updating it impacts all the users simultaneously.

In a similar way, the importance of "getting it right the first time" is increased. A typo in a parameter name or an inconsistency from one function to the next has some inertia *not* to be resolved, because fixing it will break existing clients. Because of this fact, we set out beforehand to outline some principles to adhere to for the data service, which we check our code against continually before release. We develop the data service as if we were developing a library for future unknown and inexperienced developers.

2.5.1. An API call returns only one data structure An API call does one thing and one thing only. No option shall change the format of the returned data. This is to ensure that clients cannot be surprised by results and know what to expect.

2.5.2. Common entities have required attributes across all API calls Entities with unique IDs shall always have that ID as an attribute. Basic attributes (e.g. number of files in a block) will appear with that entity no matter what the context is. This is to allow for client-side correlation of results from separate calls.

2.5.3. Utilize hierarchy wherever possible Do not flatten results, even where it seems convenient. The full context of data entities should be a part of the result, and the client should not have to rely on the options to the call to successfully interpret the results.

2.5.4. Consistent call/response semantics An attribute which is filterable should have the same name in the response as in the input.

3. Design

3.1. Server

The data service, like the rest of PhEDEx, is written in Perl. It uses the `CGI.pm` library to handle HTTP requests and responses according to the CGI standard. It is deployed in the Apache httpd/mod_perl environment, which ensures that the server code is always compiled and in memory. Database connections are done using the `DBI` database interface library, with the `DBD::Oracle` library as the driver. The `Apache::DBI` module is used to transparently provide persistent database connections in the mod_perl environment.

The interface to the data service is defined through its URIs. Table 1 describes the URI structure. The root of the URI includes the HTTP protocol, the host server, and the root application path `/phedex/datasvc`. The path to a given API method contains the `FORMAT`, `INSTANCE`, and name of the `API`. `FORMAT` determines the format of the output data. XML, JSON (JavaScript Object Notation), and Perl (`Data::Dumper` format) output are available for all APIs. `INSTANCE` is an alias for the database instance to fetch the data from. PhEDEx is deployed on three separate database instances for use with different transfer activities. The API name is simply the name of the function which will return (or accept) data. Following this path are the `OPTIONS`, in the usual CGI query name-value format. The options allowed are dependent on the API the client is calling, but many APIs share the same kinds of options.

https://HOST/phedex/datasvc/FORMAT/INSTANCE/API?OPTIONS https://HOST/phedex/datasvc/doc/API	
`FORMAT`	output format; `xml`, `json` or `perl`
`INSTANCE`	database instance; `prod`, `debug`, `test`
`API`	method to use; `blockreplicas`, `subscribe`
`OPTIONS`	method options; `block=/X/Y/Z#123`
`doc`	output documentation about a method

Table 1. URI structure for the data service.

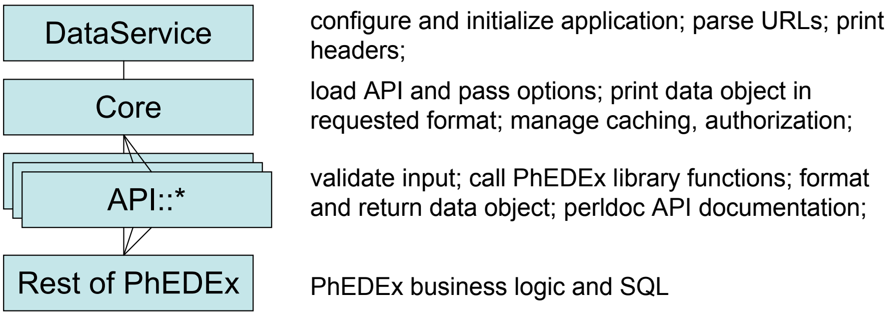
DataService
configure and initialize application; parse URLs; print headers;
Core
load API and pass options; print data object in requested format; manage caching, authorization;
API::*
validate input; call PhEDEx library functions; format and return data object; perldoc API documentation;
Rest of PhEDEx
PhEDEx business logic and SQL

Figure 1. Design of the PhEDEx data service.

Figure 1 shows how the data service code is organized. Each request to the server is handled by the `DataService` module, which is responsible for configuring and initializing the application, parsing request URIs, and printing the response headers. `DataService` passes the requested API to the `Core` module, which will load that API module (if it exists), pass it the client's options, and print the result data in the requested format. The `Core` module is also responsible for caching and interpreting client authorization.

Each API corresponds to a lightweight Perl module. A module need only provide one function, `invoke()`, which is passed the options provided by the user. `invoke()` returns an object containing the result of an API. A data service API module loads what other modules it needs to provide the result from the PhEDEx code library. The API module also provides the documentation for itself, in perldoc format, and may also define the `duration()` function, to specify how long data of that type may be cached, or the `need_auth()` function, to indicate that the API requires authentication.

An API module returns a data object in a special structure designed to be easily represented in XML. In this format, an XML element is represented by a hash reference, with each of its attribute-value pairs represented by the key-value pairs in that hash. An array of elements of the same type is represented by a reference to an array of hash references.

Each result object is wrapped by a parent element called `<phedex>` which contains attributes describing the request and response itself, such as a timestamp the request was made, the URI requested, and the time it took to serve the request. In this way, every saved response from the data service has some minimal identifying information, which is useful for debugging.

The wrapped result object is then sent to the formatting algorithm, which prints the object in the format the client requested. For XML, a simple recursive function prints the structure, while avoiding that the entire XML output be stored in memory.[1] For JSON, the `JSON::XS` library does the formatting with good performance. For Perl output, we use the `Data::Dumper` module to provide pretty-printed serialized Perl output. Other formats can easily be made available by expanding the `Formats` module.

Error handling is done using Perl's `die` built-in. An exception thrown anywhere in the API module is caught, wrapped in an `<error>` element, and sent as the response to the client in the format they requested. Currently all errors are returned with HTTP response code 200 (Success), in the future we plan to use HTTP response codes for failures where they apply.

Documentation for each API module is served by calling the `pod2html` program on an API module. Using perldoc documentation in the API module makes the documentation easy to maintain, as it is in the same file as the module providing the result logic. Documentation is served under the `/phedex/datasvc/doc` URI path. This root path provides general information about the data service and a hyperlinked list of available APIs. Requesting a URI such as `/phedex/datasvc/doc/blockreplicas` will display information about the `blockreplicas` API call, as written in the `BlockReplicas.pm` module.

3.2. Deployment

The data service is deployed on a load-balanced cluster in a reverse-proxy configuration. A load-balanced cluster of frontend nodes running Apache httpd are world-visible and receive all HTTP and HTTPS requests to the data service. For HTTPS requests, the frontend nodes provide the authentication layer of the data service, checking client X509 certificates against a list of accepted signing authorities. Authenticated client requests are then passed to a backend node with request headers set to mark them as authenticated. The backend nodes are not world visible, and are configured to only accept requests from the frontend nodes, both in the host machine firewall and in the server configuration.

The application software is deployed via Red Hat Package Modules (RPMs) built using the CMS package building toolset [13]. Everything from the application code to the Apache server is built or repackaged and stored in a CMS software repository. Deployment documentation must be up-to-date and able to be followed verbatim by a potentially novice cluster admin.

Three pairs of frontend/backend clusters exist in order to bring new versions from development into production. The *test* cluster is for development of the applications, and it

[1] An important reason not to use an XML library for this

is not world-visible. The *pre-production* cluster is for integration of a new version with the production environment. Finally the *production* cluster serves the production version of the service. Deployment packages which fail testing on the *test* or *pre-production* clusters are not put into production until the problems are resolved. The application must be stable and compatible with its dependent client projects before production release.

3.3. Security Module

API methods, particularly those which write data to the database, need to be protected by an authentication and authorization layer. As mentioned in section 3.2 the frontend nodes provide X509 certificate authentication to the backend nodes in a reverse-proxy configuration. A reusable `SecurityModule` component provides interpretation of the authentication variables found in the environment, as well as authorization determination. Based on the X509 distinguished name (treated as a unique identifier for a user), a lookup is done in CMS SiteDB [14] to determine what authorization roles the requesting client has, if any. Roles are mapped to a scope — which is either a CMS site or a group. The `SecurityModule` queries for this information and provides it to the API method.

Each API method which needs to be secured has an access list of which roles and scopes are allowed to invoke its functions. This list is stored in the application configuration file. For example, the service can be configured to only allow `subscribe` to be accessed by clients with the `T0 Operator` role. More fine-grained access is possible, for example only allowing `inject` to be called by the `Production Manager` of the site to which the client is asking to inject data.

3.4. Command-Line Client

As a complement to the data service, we provide a command-line client. The client, simply called `phedex`, uses the `LWP::UserAgent` HTTP library. Its basic functionality is to parse command-line options for each data service API method, build a request to send to the data service, and print the results in the user's requested format (XML, JSON, or Perl). It has a similar plug-in design as the data service itself, with a simple module describing each data service API method, and capable of parsing the options understood by that method. Furthermore, these modules can be extended to provide a "report" format, for providing human-readable output to the terminal screen. Users can also supply their own formatting modules to generate custom formats.

The `phedex` client also becomes a useful development and debugging tool when used with the `FakeAgent` module. This module is a superclass to `LWP::UserAgent` which fakes the server response by bypassing a HTTP network connection and instead directly invoking the data service response code. The authentication environment is also faked, as if it had been done by the frontend machine. This allows the developer to test the full chain of client-server code without the overhead of running the actual web server. The whole interaction runs in a single process, and makes testing with a debugger possible.

4. Performance

Two kinds of performance testing have been done on the data service: optimization testing and use-case testing. The optimization testing was fine-grained testing of choices in SQL, output formatting libraries, and caching strategies. We do not elaborate on the results of our optimization tests here.

The use-case which was tested was that of using the data service as a Data Location Service (DLS) [15]. The DLS concept has been in CMS for many years, and there is a long-standing project to abstract the DLS "backends" behind a uniform python library and command-line interface. At the time the data service was developed, the production DLS backend was the DBS, which would get information about data location via a (sometimes problematic) synchronization process with the PhEDEx database. The first production use of the data service was to become the direct source of data location information, bypassing the synchronization step with DBS.

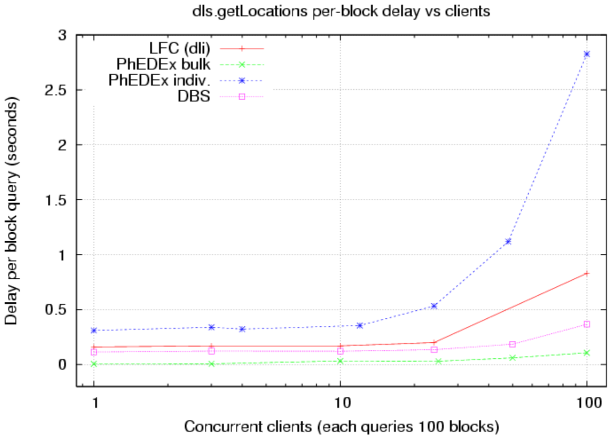

Figure 2. Parallel client performance test of DLS library with different backends.[16]

Figure 2 shows the results of a performance test of three DLS backends, and two separate access patterns for the PhEDEx data service. The test measures the response time to fetch the location of blocks (a block is a set of files used in CMS computing) as a function of the number of concurrent clients. Each client is fetching the location of 100 blocks. For the PhEDEx data service, two access patterns are measured: single-block queries where each client asks for the location of a random block 100 times in series, and bulk-block queries where each client asks for the locations of 100 random blocks in one request.

The results show that for single-block queries (which entail more client-server round-trips), the PhEDEx data service performs slightly worse than the DBS or LFC backends, until about 10 concurrent clients at which point the performance deteriorates rapidly. However, for bulk-block queries the per-block performance is much improved over the DBS an LFC backends, and scales well up to 100 concurrent clients.

The single-block queries are likely dominated by the fact that the server was configured to spawn at most 10 child processes to handle requests, and the many serial accesses increase the likelihood of true concurrent accesses. By tuning the number of server child processes to the limits of memory on the host cluster, we expect we can improve the scaling for single-block queries. However, the best method of access will always remain bulk-block queries, as it removes a number of latencies (client-server, server-database) in the chain.

5. Usage Patterns

We observed the usage of the data service over a period of three months beginning in December 2008 in order to understand how the service, which is globally accessible, is being used. We found an increasing number of requests over the period, from 81,000 requests in December 2008 to 156,000 in February 2009. This averages to about 5,200 requests per day, which is a light load as far as web services go, but we expect the load to increase as CRAB analysis ramps up and as the data service becomes integrated with the PhEDEx web site.

We also looked at which output formats were being requested. Surprisingly, Perl is the most requested format with 64% of all requests, followed by 36% for XML and only 0.03% for JSON. This tells us something interesting about the use of the data service. The “official” users of the data service (other CMS computing projects) are using the XML format. That the Perl format is more popular shows that the data service sees heavier use from non-official users, which we regard as a good trend. This means that users are finding interesting ways to use the data service, themselves providing solutions to problems, lightening the load on PhEDEx developers.[2]

The first official use-case for the data service was to enable PhEDEx to serve as the DLS (Data Location Service). CMS made this transition in June 2008, when the DLS client implemented a "PhEDEx backend" and was used by the CRAB analysis project. The transition to the PhEDEx backend was largely transparent to users and proved that the PhEDEx data service is a workable method of cross-project integration and data sharing. The next integration will be with the Tier-0 system, which at the time of this writing is nearly complete.

Finally, table 2 catalogs the projects using the data service, both official CMS projects and non-official, user community projects. Even at this early stage in the development we see an impressive diversity of projects using the data service, and hope to see that trend continue.

Project	Type	Requirements
DLS	CMS Computing	block replicas, file replicas, list of SEs
CRAB[a]	CMS Computing	block replicas, file replicas
DBS Discovery	CMS Computing	block replicas, % complete
Tier-0[+]	CMS Computing	make injections, subscriptions; block replicas
Dashboard[+]	CMS Computing	transfer statistics
PhEDEx Website[+]	CMS Computing	all data and management features
Nebraska Consistency Tools	User Contribution	block replicas, file replicas
Netvibes Dataset Monitor	User Contribution	block replicas
ItalianT2Tools	User Contribution	block replicas, file replicas

Table 2. A list of projects using the data service.

[a] CRAB uses the DLS library, so it is indirectly using the data service.

[+] Project integration with the data service was in development at the time of writing.

6. Conclusion

The PhEDEx data service satisfies cross-project integration requirements for CMS computing by providing a flexible method of data-sharing in various formats over HTTP, and provides a secure method of managing PhEDEx by exposing certificate-authenticated APIs over HTTPS. The service provides a platform for increased involvement from outside developers, allowing them to produce solutions to their problems without increasing the load on PhEDEx development. The service increases code reuse within the PhEDEx project, especially as it transitions to be the backend driving the PhEDEx monitoring and management web pages. We find that a web data service is a useful component to a distributed computing system, satisfying a diverse set of requirements in a single application framework.

[2] This is the optimistic view; alternatively it is possible that non-official users are simply abusing the service, making more requests than they need. We prefer to maintain some optimism about our users.

References

[1] CMS Collaboration 1996 The Compact Muon Solenoid computing technical proposal Tech. Rep. 1996-045 CERN/LHCC

[2] CMS Collaboration 2005 The CMS computing project technical design report Tech. Rep. 2005-023 CERN/LHCC

[3] CMS Collaboration 2004 The CMS computing model Tech. Rep. 2004-035 CERN/LHCC

[4] CMS Collaboration 1994 CMS technical proposal Tech. Rep. 1994-38 CERN/LHCC

[5] Rehn J, Barass T, Bonacorsi D, Hernandez J, Semeniouk I, Tuura L and Wu Y 2006 PhEDEx high-throughput data transfer management system *Computing in High Energy Physics 2006 (CHEP06)* (Mumbai, India)

[6] Tuura L, Bockelman B, Bonacorsi D, Egeland R, Feichtinger D, Metson S and Rehn J 2007 Scaling CMS data transfer system for LHC start-up *Computing in High Energy Physics 2007 (CHEP07)* (Victoria, Canada: IOP)

[7] Egeland R, Wildish T and Metson S 2008 Data transfer infrastructure for CMS data taking *XII Advanced Computing and Analysis Techniques in Physics Research* (Erice, Italy: Proceedings of Science)

[8] Afaq A, Dolgert A, Guo Y, Jones C, Kosyakov S, Kuznetsov V, Lueking L, Riley D and Sekhri V 2007 The CMS dataset bookkeeping service *Computing in High Energy Physics 2007 (CHEP07)* (Victoria, Canada: IOP)

[9] Codispoti G, Cinquilli M, Fanfani A, Fanzago F, Farina F, Lacaprara S, Miccio V, Spiga D and Vaandering E 2008 Distributed analysis with CRAB: the client-server architecture evolution and commissioning *XII Advanced Computing and Analysis Techniques in Physics Research* (Erice, Italy: Proceedings of Science)

[10] Spiga D 2009 Automatization of user analysis workflow in CMS *Computing in High Energy Physics 2009 (CHEP09)* (Prague, Czech Republic: IOP) in these proceedings

[11] Lingen F V, Evans D, Wakefield S and et al 2009 CMS production and processing system - design and experiences *Computing in High Energy Physics 2009 (CHEP09)* (Prague, Czech Republic: IOP) in these proceedings

[12] Evans D, Hufnagel D, Metson S, Mason D, Gowdy D, Foulkes S and Seanchan R 2008 The CMS Tier 0 *XII Advanced Computing and Analysis Techniques in Physics Research* (Erice, Italy: Proceedings of Science) pending publication

[13] Muzaffar S, Eulisse G, Lange D and Pfeiffer A 2009 CMS software build, release and distribution — large system optimization *Computing in High Energy Physics 2009 (CHEP09)* (Prague, Czech Republic: IOP) in these proceedings

[14] Metson S 2009 SiteDB: Marshalling the people and resources available in CMS *Computing in High Energy Physics 2009 (CHEP09)* (Prague, Czech Republic: IOP) in these proceedings

[15] Afaq A, Bockelman B, Peris A D, Egeland R, Fanfani A, Farina F, Kutznetsov V, Lueking L, Metson S, Sekhri V, Verlato M and Wildish T 2009 The evolution of the data location service in CMS *Iberian Grid Infrastructure Conference IBERGRID'2009* (Valencia, Spain) pending publication

[16] Performance study courtesy of A. D. Peris, plot used with permission