

Metadata Monitoring Requirements

Solveig Albrand, Dirk Düllman, Paul Millar¹

Abstract

Metadata is oft quoted as data about data. In any large distributed system, such as a Data Grid, the metadata service will consist of one or more databases and support libraries for querying these databases. These libraries may allow users to form abstract interactions with the stored information, converting these into standard SQL statements for the underlying databases.

Any such service would have many aspects of the service that different classes of user may desire to monitor. This document will build a catalogue of the different aspects and from them seeks to derive a list of requirements that a monitoring system needs to satisfy in order to furnish end-users with the information they need. It also gives a brief overview of existing systems.

Table of Contents

1.Metadata service architecture.....	2
2.Who might be asking the questions?.....	3
3.Why monitor?.....	4
4.What to monitor?.....	5
4.1.Hardware usage.....	5
4.2.Overall database usage.....	6
4.2.1.MySQL.....	6
4.2.2.PostgreSQL.....	7
4.3.database usage per user.....	8
4.4.Metadata server status.....	8
4.4.1.Stand alone servers.....	9
4.4.2.JMX and Tomcat.....	9
4.5.Contact information.....	9
4.6.Data consistency checks.....	9
4.7.Heartbeat monitoring.....	10
5.How should the information propagate?.....	10
6.Existing Information Systems.....	10
6.1.Oracle.....	10
6.1.1.Monitoring.....	10
6.1.2.Streams.....	11
6.1.3.Oracle Enterprise Manager (OEM).....	11
6.2.R-GMA.....	11
6.3.MDS.....	11
6.4.BDII.....	12
6.5.CondorView.....	12
6.6.Hawkeye.....	12
6.7.Glue schema.....	13
6.8.Nagios.....	13
6.9.Ganglia.....	14
6.10.LEMON.....	14
6.11.MonAlisa.....	14

¹ The author to which correspondence about this paper should be sent.

7.Existing Querying Systems.....	15
7.1.Direct Database monitoring.....	15
7.2.ApMon.....	15
7.3.LISA.....	16
7.4.GIP.....	16
7.5.GridICE.....	16
8.Conclusions.....	17

1. Metadata service architecture

The metadata service and the monitoring it requires, consists of the following logical architecture:

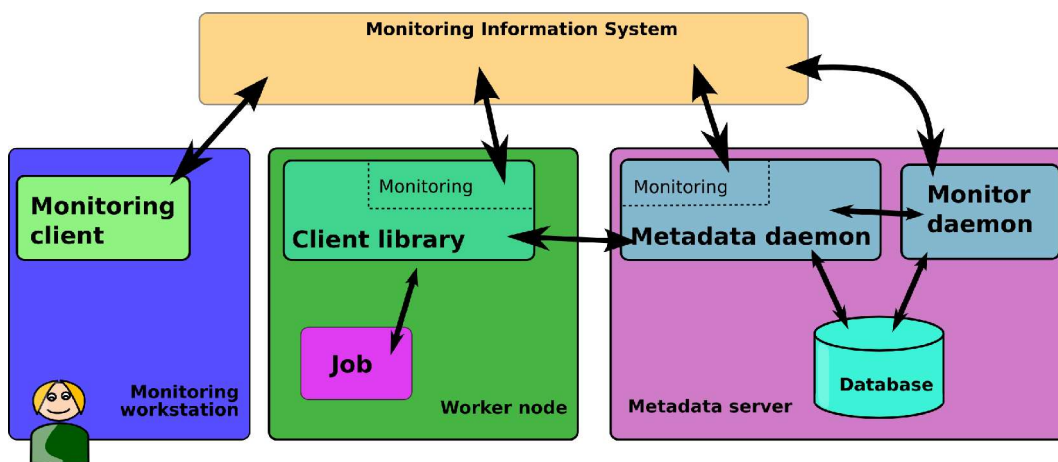


Fig. 1 basic components of a metadata service

Figure 1 shows a job as some executable running on a worker node with its corresponding access to a metadata service, with resulting update of the monitoring systems. We assume that by this stage all brokering, scheduling and local batch queuing has taken place and that the job is active.

There may be job-specific metadata activity prior to the job running. We assume that this activity is handled by standard Grid services. Therefore, any monitoring of this activity is outside the remit of this document.

The basic elements of Figure 1 are as follows:

- **Worker node**—the machine on which the job runs. It could be located at a Tier-0, Tier-1 or Tier-2 site.
- **Metadata server**—some machine on which a database engine and metadata server runs. Within a VO, this may be a single machine or multiple depending on the metadata service deployment plan.
- **Monitoring workstation**—any machine of someone who wants to investigate the performance of the metadata service.
- **Job**—some arbitrary executable that is running on the worker node through the normal Grid job-submission process.

- **Client library**—is the library the job uses to access the metadata service. It may “register” the job within the monitoring system.
- **Metadata daemon**—the daemon (or set of daemons) that provide access to the metadata service from worker nodes. This may be a service provided centrally, replicated down to Tier-1 sites or to both Tier-1 and Tier-2 sites. The monitoring component can record job-specific monitoring within the Monitoring Information System.
- **Database**—some storage back-end. Current proposed databases are Oracle and MySQL.
- **Monitoring Daemon**—Databases, such as Oracle, provide a decentralised monitoring infrastructure (described later). Other databases (such as MySQL) do not provide this facility. This daemon may also monitor performance of the metadata server.
- **Monitoring Information System**—this will carry information throughout the Grid in a decentralised fashion, allowing people to query the monitoring system for useful information.
- **Monitoring client**—software for querying the monitoring system and to discover trends and causes of problems.

It should be borne in mind that the components shown are abstract objects that may be instantiated in any number of actual daemons, libraries, etc. For some systems, the distinctions of different logical components might be realised within the same actual object. For example, we show a metadata server as a distinct object; however, some of the functionality (viz a viz one database compared to another) may exist within the database itself; or within the Oracle monitoring architecture, the database monitoring is handled by the database itself.

It is worth emphasising that there are three components from which one might wish to collect information: the database, the metadata daemon and the client code. In general, a job will cause activity within all three components. Collecting information about this activity and tying this information to the job that triggered the activity is the purpose of the monitoring system.

2. Who might be asking the questions?

Not everyone will want to know all aspects of the metadata service. The end-user physicist may have no desire to know fine detail of how particular queries have been executed by the underlying database; whereas, for someone working on optimising the metadata libraries this information may be crucial.

Moreover, different classes of user will have access to information from different subsets of all information systems. It is important to identify the different classes of end-user so that we can make sure the different information systems open to them are sufficient to acquire all information they need.

The following classes of end-user are identified:

- **sys-admin**—someone with a level of responsibility in maintaining the databases, for example in maintaining the machines up-time.
- **physicist**—someone running jobs on the Data Grid for analysis,
- **software engineer**—a programmer responsible for some part of the metadata service,
- **VO manager**—some member of a group who take overall charge of members of their virtual organisation.

The roles are not necessarily exclusive: although physicists might want to know only about their jobs, the same person might also have additional responsibilities. As a single person might adopt different roles under different circumstances; the presentation of information needs to be flexible enough to

provided sufficient information for the situation whilst hiding irrelevant information, unless that information is needed. Because we cannot know in advance who will be adopting which role, the information system (providing the information to the end-user) needs to allow for providing all possible information to all people, with the user-interface selecting the most relevant.

If the tools for enquiring the monitoring service are designed to provide sufficient information for the targeted end-user, then there should be a mechanism for providing additional information when a different role is adopted. This can be achieved by defining a number of views of the data: “my jobs” view for the physicist, “my site” view for the sys-admin, “my VO” for the VO manager and some advanced reporting for the software engineer. End users should be free to navigate between different views, as the same user may adopt different roles throughout his or her investigation.

3. Why monitor?

When looking at a metadata service (rather than just a metadata daemon) certain aspects of the service need to be monitored. Typical reasons for monitoring depend on under which role the end-user is working. The situation may become entangled when people adopt different roles over the course of their investigation.

A physicist might want to know what percentage of some job's wall-clock run time was due to metadata access. Incorrect usage of the metadata client-side library might result in queries taking unnecessarily long to return the data. Similarly, when a job is taking longer to complete than was anticipated, the monitoring system may give an idea how far the job is in completing (depending on the specific job) and also if the job is slow due to some performance issue with the metadata service.

A VO manager might want to monitor access to obtain a good estimate of current and future resource requirements. The monitoring system should provide some estimate of current usage, and that current hardware is able to serve metadata requests within an acceptable time-scale. By monitoring trends in both usage and server performance, one can anticipate when hardware will need upgrading.

A sys-admin might want to monitor at the hardware level that the storage is OK, that system settings are appropriate and the database is installed correctly. At a crude level, this could be found by monitoring gross hardware performance statistics (such as CPU usage, load average, disk IO), but monitoring database-specific performance statistics might gain insights into other performance tweaks.

The software engineer might want to look for bottlenecks and ways of optimising performance by looking for queries that particularly adversely affect the database. By resolving which clients were executing when the databases performance deteriorates, the engineer can isolate any such problem and fix it.

If databases are deployed to Tier-2, then the Grid infrastructure might not be able to expect sys-admins to be expert in tuning a database for optimal performance. It is not clear to what extent a deployed database can be “pre-tuned” to near-optimal values, to what extent per-server tuning will increase performance or even if this level of tuning is necessary.

Assuming that some level of tuning is required, then some “expert” (a software engineer, VO-manager, or someone else) might be needed to assist the Tier-2 sys-admin. This person, whom we assume is off-site, might need to know detailed performance and configuration of the database for them to suggest changes that would improve performance. Therefore, this information needs to be present within the monitoring system.

The metadata daemon may require similar fine tuning, if simply to determine in the correct fraction of the server's available memory is being used or to detecting broken configuration. The local sys-admin may not have sufficient time to learn the current characteristics of a well-functioning metadata daemon, characteristics that might be obvious to either the VO manager or the software engineer. Providing this information within a distributed information system allows those with specific knowledge to check a site installation and suggest ways of improving performance.

4. What to monitor?

When choosing what to monitor, it is tempting to monitor everything. This can cripple the service, so what is monitored must be chosen carefully. The following lists some aspects of the service that we might want to monitor. It is not meant to be a list of all things that should be monitored.

When monitoring, there are two levels of monitoring one might require. If observing a trend is needed for particular element under scrutiny, then the data needs to be stored somewhere. If the trend is needed over a long period, then hierarchical storage might be used: where data is stored for a short period sampled frequently, but for a longer duration at a lower sampling frequency.

Another monitoring option is to provide information about the current situation. The current configuration of some database component is potentially useful, but the historical trends of the parameter is of less use.

4.1. HARDWARE USAGE

When looking at service provision, one potential problem is with the server fabric and underlying OS. Various aspects of the server hardware should be monitored to detect problems as they arise, to see if the allocated hardware is sufficient for demand and to plan hardware upgrade as demand increases over time.

The following statistics should be monitored:

- **Bandwidth usage** (peak values, average usage); check that, during data transfers to maintain consistency, bandwidth is sufficient. Also, that query results aren't unduly delayed due to lack of bandwidth.
- **Disk usage**; monitoring trends in disk usage, estimating time until the disk is full, warning when disk usage reaches a certain percentage of available disk space.
- **CPU usage**; whether the CPU is working at full rate.
- **Swapping**; whether the machine is swapping due to lack of memory. This should never happen on a production system, but it should be checked.
- **Load average**; whether the machine has too many tasks vying for CPU usage.

Historical values for these variables might be needed to see, if when presented with a change in usage or performance, whether the hardware is sufficient for demand.

These values give a gross overview of any problem. One is likely to see an issue even if the problem lies elsewhere. For example, if some problematic client issues CPU-intensive database queries, the effect will be an obvious spike in the CPU usage statistics, yet the culprit lies outwith the server. This monitoring is useful to catch problems that more targeted monitoring might miss.

4.2. OVERALL DATABASE USAGE

Various non-user-specific statistics are available to be collected. Although not directly useful for user-based monitoring (such as monitoring the impact a user's query has on the database), these statistics allow the monitoring of service provision. If too many clients are using the service concurrently or some client code is causing high database load, these statistics should illustrate that the problem is manifest within the database itself.

The available data depends on the database in use and the most common databases are summarised below:

4.2.1. MySQL

The current state of a MySQL server is described by Server System Variables, Server Status Variables, storage engine status variables and current thread status variables. Independent of the storage engine, MySQL v4.1.14 has 250 variables whilst v5.0.13rc has 437 variables. The InnoDB engine has some 82 variables (depending on if lists are stored) and each thread has some 7 variables describing it. One could monitor in excess of 500 variables.

Storing values (to provide a history and demonstrate trends) would be useful for some values, for others only their current values are useful and some variables are unimportant. One might group the parameters into three categories: performance, configuration, unimportant.

The performance variables should be monitored over time, allowing comparison of how the server is affected by various loads.

Configuration values might require reviewing either when a server is brought into the group or when a problem comes to light. As configuration would not require storage, one can be more liberal with which variables one wishes to monitor, provided the bandwidth is sufficient.

SERVER SYSTEM VARIABLES

A system variable is a read-write variable. They can be either global, per session, or both and tend to affect how the current server behaves.

Global variables require elevated privileges to change, but session variables can be altered by the end-user. Session variables take their default value from the global setting when the session starts, but are unaffected by any subsequent change to a global variable: only new sessions will adopt the new global variable value.

The number of variables changes with different versions of MySQL, (v4.1.14 has 188, v5.0.13rc has 213). Some variables can only be set when starting the server. Those that can be changed whilst the server is running are dynamic system variables.

The server system variables may affect the performance of the server, but are unlikely to change often. The values can be monitored infrequently and storing historical values is unnecessary.

SERVER STATUS VARIABLES

Server Status Variables are read-only values that describe the current performance of the server. They can be obtained by querying the server ("SHOW STATUS"). Many of the values can be reset.

The list of available data increases with more recent versions of MySQL (v4.1.14 has 162, v5.0.13rc has 224). Some of these are storage engine specific (those starting "innodb_" for example), some are for features that are likely to be switched off (slave databases, for example) and some are likely not to be relevant to performance issues for MySQL servers as deployed in the Grid.

STORAGE ENGINES STATUS

MySQL supports multiple Storage Engines. The list of supported engines can be obtained (SHOW ENGINES:). The default engine is MyISAM, but a popular alternative is InnoDB.

There is language support for obtaining the status of the storage engine. Most engines do not currently (as of v5.0.13rc) support returning their current status but the InnoDB engine does, returning some 82 variables. These variables can be grouped into the following types:

- table and record locks held by each active transaction,
- lock waits of a transactions,
- semaphore waits of threads,
- pending file I/O requests,
- buffer pool statistics,
- purge and insert buffer merge activity of the main InnoDB thread.

Depending on how InnoDB is configured, and even if InnoDB is to be used, some or all of this information may be useful.

STORAGE ENGINE INDEX STATISTICS

MySQL uses table-based index statistics to estimate how many rows must be read for each ref access and to estimate how many row a partial join will produce.

The MyISAM storage engine generates the index statistics using one of two methods. One method considers all NULLs to be different (i.e. each NULL is in a different value group, each group having cardinality of 1) whilst the other method considers all NULL values to be the same (i.e. being part of the same value group). Unfortunately, there is no way to tell which method was used to generate statistics for a given table.

Depending on which style of joins end-users are using predominantly (either “<=>” or “=” in SQL), NULLs will either be considered different or the same. Generating the wrong style of indexes might adversely affect complex queries.

Other storage engines have only one method for collecting table statistics. Usually it is closer to the statistic that treats NULLs as equal.

THREAD INFORMATION

Information on the current thread pool is available through the command SHOW PROCESSLIST; This lists information about the current threads, including the user, database, time utilised and current state. It may be possible to extract the per-user CPU utilisation from the available information within the metadata server.

4.2.2. PostgreSQL

[Home page](#)

PostgreSQL is a popular open-source database that include some enterprise features missing from MySQL. It includes a statistics subsection that collects statistics from the active server processes. This subsystem can be switched on or off via the standard configuration file `postgresql.conf`.

Within the configuration file, the parameter `stats_start_collector` must be true for the statistics subsystem to be launched. Running this subsystem, in of itself, requires little overhead and is on by default. The parameters `stats_command_string`, `stats_block_level` and `stats_row_level` (row-level access statistics) control how much information is recorded. These options control the acquisition of command string, block-level and row-level statistics respectively.

By default, all three options are switched off, so only minimal statistics are collected. Switching these options on results in increased statistics, but also increased overhead. New server processes will start with the statistics gathering options from `postgresql.conf`, but which statistics are gathered can be altered during a session through the SET command. To prevent users hiding their activity from administrators, only superusers have suitable privileges to alter what activity is monitored.

To display the available data, PostgreSQL provides some 17 built-in views, summarising the gathered statistics. Each view provides a set of statistics on some group of object within the database. For example, the `pg_stat_database` view has one row per database, with columns for database OID, database name, number of active server processes connected to that database, number of transactions committed and rolled back in that database, total disk blocks read, and total buffer hits.

If the preconfigured views are not sufficient, it is possible to extract the statistics for an arbitrary collection of objects. Some 24 SQL functions are provided for directly querying the statistics subsection. These functions can be used to build SELECT statements that report on the current statistics.

4.3. DATABASE USAGE PER USER

Ideally, one should be able to measure the impact (on the server) of the end-users' activity; for example, by measuring the time the CPU spent executing the end-users' request. Obtaining an indication of the effect of a user on the database is difficult for two reasons.

First, the database “load” (however this is defined) may depend on some complex interaction between two or more end-users. A query that would normally take little time might take longer due to some concurrent or previous query affecting the database. These correlated effects would be almost impossible to detected.

Second, database connections are typically pooled within the database back-end, with a database user mapping to some specific role. There would be a many-to-many mapping between threads used by each of the end-users within the same database role. Extracting which threads were used (and the corresponding CPU utilisation per thread usage) by some specific end-user may be impossible.

However, within the metadata server it may be possible to monitor for certain “bad” activity (e.g. requesting a Cartesian product). These activities should be detected and logged against the end-user and software that generated that request.

4.4. METADATA SERVER STATUS

Often, metadata services are provided through some “wrapper” server that adds additional functionality to the storage database. The mapping between metadata concepts may be fairly straightforward or more complex, depending on the level of additional functionality the wrapper adds.

The metadata servers may be implemented as a stand-alone code or as an application running within some application container, such as Tomcat for Java applications. In both stand-alone and application container, the activity of the metadata server itself may be worth monitoring, either to discover bottlenecks in performance, or to watch for the server deviating from designated correct behaviour.

4.4.1. Stand alone servers

For stand-alone servers, this monitoring requires some specialised mechanism for providing monitoring information. The nature of this monitoring can be arbitrary complex, but must be specifically coded for that server. Servers, such as AMGA, fall into this category.

4.4.2. JMX and Tomcat

The JMX specification (see [specification](#) and [best practise](#)) describes how to control and monitor web-applications in a uniform fashion. Tomcat (since v5.0) supports the JMX MBean interface, allowing monitoring of potentially all objects within deployed web-applications.

The information available depends on the instantiated object's class, but all objects of the same class (ThreadPool, for example) have the same information. This allows monitoring of components of a deployed web-application without necessarily having detailed knowledge of that application.

4.5. CONTACT INFORMATION.

Although not directly related to the monitoring system, it is important that contact information be readily available. If a sys-admin needs to contact someone due to a disk becoming full, corrupt, or for some planned intervention, then the contact details of some VO-manager needs to be available. Likewise, when a machine stops working or is badly configured the VO-manager needs to know who to contact.

Without this information, the monitoring becomes much less useful as one cannot easily contact the person responsible for fixing the problem. Contact information for the sys-admin could propagate through the metadata monitoring system, but should be present in other systems.

4.6. DATA CONSISTENCY CHECKS

Metadata, as it is currently conceived, is stored within relational databases. There may be several tests one can conduct to check that the data stored in the database is correct. For example, one could check that every entry of a particular field has at least one entry in some auxiliary table, or that data is of a particular form.

This level of monitoring would check that any replication has happen in some consistent way and that data entered into the database matches some criteria for correctness.

There are several issues with this aspect of monitoring:

1. What are the correct rules? The validation rules are likely to change over which the period the service is available. How are these rules updated?
2. What action to take if a database fails the checks? Is a failing test indication of some failed replication and that the database should be either taken down (requests being routed to some backup service) or merely that someone should investigate.
3. How time-intensive are these tests? These tests might take a large amount of resources to conduct. How do we measure the impact of running these tests, how often should they be run?

Answering these questions is out with the scope of this document, but they are presented as topics for further investigation.

4.7. HEARTBEAT MONITORING

This is testing that certain services are up and running and generally returns a boolean value (running or not running), although one might extend this to be some measure of performance, with zero indicating the service is not running.

When monitoring a service, one might monitor at two levels: shallow and deep. A shallow heartbeat merely checks that it is possible to make some contact with the service. The deep heartbeat tests all components of the service, ideally testing that data can be read off of physical storage. Both tests should use the minimum resources to achieve their goals. Considering the metadata service as an example, a shallow heartbeat might simply connect to the web-interface whilst the deep heartbeat might do some database query that is quick, but for which database access is necessary.

One might want to heartbeat monitor the metadata server and the database itself. This would allow someone to distinguish between a misconfigured metadata server and database being down.

The shallow heartbeat might run more frequently, picking up gross problems rapidly whilst the deeper tests would run less frequently, determined by how much resources they use.

5. How should the information propagate?

Data about the metadata service is itself metadata. This data needs to be stored somewhere so it can be analysed. This is shown in Figure1 as the Monitoring Information System. The information system consists of two parts: a communications pathway and a storage system. Existing information systems may include both elements (such as R-GMA) or just the one (such as Hawkeye) and rely on external applications to store the information, such as the Round-Robin Database (RRD) system.

Certain information needs to be stored for an extended period. For other values, only the current values need be accessible. For the latter case, it is better to obtain the values directly from the end machine unless the frequency of requests exceeds the update frequency (on average). For most parameters, this is unlikely, so server-side caching is unnecessary and the information should be pulled when needed.

The following section gives a brief overview of the major Information Systems currently available.

6. Existing Information Systems

There are a number of existing Information systems: methods of transporting information between different servers.

6.1. ORACLE

[Home page](#)

Oracle is a company that provide proprietary database technology. In addition to its internal monitoring, two technologies are introduced here: streams and OEM.

6.1.1. Monitoring

Oracle databases expose their database statistics inside a special schema (equivalent to “databases” in other RDMS). These typically have tables like EVT% and SMP%. The values of this schema indicate current or historical values of monitored performance variables.

6.1.2. Streams

Streams allows multiple databases to be collected into a federation. Schema supported through streams have guarantees on update times and consistency across the database cohort. The streams implementation allows servers to report their monitoring values back to some central site.

6.1.3. Oracle Enterprise Manager (OEM)

The OEM is a standard method for allowing a DBA to control a federation of Oracle databases. It consists of the Oracle Management Server (OMS), Console and Oracle Intelligent Agent (OIA).

OMS is a middle tier that acts as a conduit for OIAs, similar to the Monitoring Information System in Fig.1. The Console connects to OMS to monitor and configure the databases.

The Console is graphical user-interface that can schedule jobs, events and monitor the database. The console is available for Windows, within a Unix xterm or via a web browser.

The OIA runs on the target database and executes jobs and events scheduled through the Console. It functions similarly to the database-monitoring role of the Monitoring Daemon in Figure 1.

6.2. R-GMA

[Home page](#)

R-GMA is a system for collecting and distributing information on wide-scale systems, such as Grid computing. The architecture consists of three components: *Consumers*, *Producers* and a directory or *Registry* service.

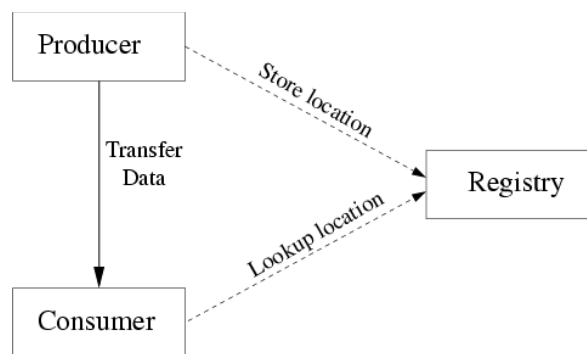


Fig. 2 Components of R-GMA system

Producers provide information, consumers read the information and both refer to the registry to register their existence (for producers) and to discover suitable producers (for consumer). Any code supplying or obtaining information from R-GMA need know nothing about the Registry: the Consumer and Producer libraries handle registry interactions behind the scenes.

The R-GMA system appears like one large relational database and can be queried as such. Monitoring values can be stored within R-GMA via custom Producers (or by using the generic GIN producer). These values can be extracted as an overview or as a set of current “live” values by an application acting as a Consumer.

6.3. MDS

[Home page](#)

With the Globus Toolkit (GT) v4.0 (GT4.0), the Monitoring and Discovery System (MDS) system provides both a legacy system and a web services (WS) based information system.

The legacy system is from previous Globus releases (GT2.x) and comprises of two components: GRIS and GIIS. These both use LDAP as the query mechanism. A GRIS server provides information whilst a GIIS server aggregates information from multiple GRIS servers.

Under GT4.0 the WS-based system comprising of an index service and a trigger service. Index services allow for accumulation, aggregation and querying of data. Trigger services allow for specific activity to take place under predefined circumstances.

The index server uses a registry similar to UDDI. Clients of the index service use WSRF resource property query and subscription/notification interfaces. Indexes servers can reference other index servers, allowing a natural data hierarchy with aggregation. Any information registered to the system has a specific lifetime and is automatically removed once it expires.

The trigger service collects information and compares that data against a predefined set of conditions. When the conditions are satisfied, a specific action takes place. For example, one could configure the trigger service to email a sysadmin when the available disk space drops below a certain threshold.

The Aggregator framework provides a uniform method of merging datasets, which is used in the index and trigger services. At the time of writing, the Aggregator in MDS (from GT4.0) includes information from Hawkeye, Ganglia, WS GRAM (job submission system in GT4.0) and Reliable File Transfer Service (RFT, from GT4.0).

Current deployment within LCG uses the legacy MDS model, with a GRIS for each site component and a site-wide GIIS to aggregate this information. Only information pertinent to the GLUE Schema v1.2 is collected, which is made available to the site BDII server.

6.4. BDII

[Home page](#)

In the current LCG deployment model, a single BDII server is present at each site. The BDII server polls the site's various GRISes (part of the legacy MDS system) and stores the information within a database.

The BDII server does not “push” this information further, but allows external servers to query its current contents. Top-level BDII servers collect the monitoring information from site BDII servers, obtaining a list of all such server from the GOC (Grid Operational Centre) database. The information these top-level BDII servers provide is used by LCG clients.

6.5. CONDORVIEW

[Docs, v6.0](#)

CondorView is a monitoring system for performance evaluation of Condor. CondorView consists of both a mechanism for collecting data (such as pool utilisation) from a Condor pool and tools to achieve visualisation of the data. The visualise of data is achieved by a dedicated application rather than making the information available on a web-page.

Information is contained as ClassAds. All elements within the cluster need be configured to send their data to the one CondorView server. This server then accepts requests for information from a CondorView client, which then visualises the data for the end-user.

6.6. HAWKEYE

[Home page](#)

Hawkeye is the monitoring system used within the Condor project. It uses elements of Condor and ClassAds to provide a flexible method of monitoring a cluster.

The system is modular. Many modules already exist (for example: CPU, number of users, load average, PBS queue status), but it is also possible to write new modules.

As modules are executed, they are expected to produce ClassAd attribute/value pairs. These are propagated aged through the normal ClassAds mechanism.

Modules are executed and output one or more ClassAd attribute/value pairs (one pair per line). Each ClassAd is terminated by a line with a dash character (-). There is support for numerical and string data and limited support for arrays.

It is possible that Hawkeye is a continuation of CondorView as there is considerable overlap between the two projects; but this isn't stated anywhere.

6.7. GLUE SCHEMA

[Home page](#)

The GLUE Information Model (aka GLUE Schema) is the result of a collaboration effort started in April 2002 by the EU-DataTAG and US-iVDGL projects. Current projects that are participating in this activity are EGEE, LCG, Grid3/OSG, Globus and NorduGrid. The aim is to define an information model and mapping to concrete schemas for representing Grid resources. These concrete schemas are expressed in LDAP, XML and in an abstract relational model.

The schema includes standard method for providing a site's current state in terms of Computing Elements (CEs), Storage Elements (SEs) and underlying batch systems. It contains some useful abstract concepts (such as a host), but does not state how any additional monitoring information should be integrated within the information model.

6.8. NAGIOS

[Home page](#)

Nagios is an open source host, service and network monitoring program. Monitoring is conducted by a central server that running several plugins. Each plugin returns the current values of monitored quantities. Monitored quantities can cause triggers, which result in some form of alert (email, SMS, ...). Tests can be cascaded, so a failure of some earlier test would prevent later tests that would automatically fail from being conducted. An example is if a network switch stops working, no tests would be conducted for hosts connected through that switch. In this case, a single alert would be generated.

Nagios can query remote sites four different methods:

- Use the **check_by_ssh** plugin. This uses the ssh command to log into the remote machine and run a test plugin on a remote machine.
- The **nrpe** addon: The nrpe daemon runs on the remote host and listens for request from the check_nrpe plugin. The check_nrpe plugin sends a request to the nrpe daemon to execute a plugin on the remote host and then passes the results back to the Nagios process. Any plugins that you want to execute on the remote host must be installed on the remote host beforehand.
- Use the **nrpep** or **nagios_statd** addons. These addons work in a similiar manner to the nrpe daemon.
- **check_snmp** using a snmp query mechanism.

6.9. GANGLIA

[Home page](#)

Ganglia is a system that allows easy accumulation of data, so hierarchies of different machines of the same class can be easily collated. In contrast to Nagios, its default deployment uses IP-multicast traffic to propagate information. This information is collated at all sites so all elements within a cluster will know all data about all elements within the cluster. Because of the multicast traffic, this is achieved without the corresponding $O(n^2)$ increase in traffic: each information source transmits once, but all listen and remember the information.

In addition to supporting multicast, Ganglia also support sending information through unicast traffic. With the addition of cluster aggregation (using the gmetad daemon), large geographically-diverse “grids” (in this context, groups of clusters) of computers can be monitored. Large scale monitoring is feasible whilst retaining the possibility of drilling down to individual machines.

6.10. LEMON

[Home page](#)

The Lemon (LHC Era Monitoring) monitoring architecture was developed at CERN as part of the Quattor system for wide-scale Linux server deployment. Despite running within the Quattor framework, it has no dependence on Quattor or LEAF.

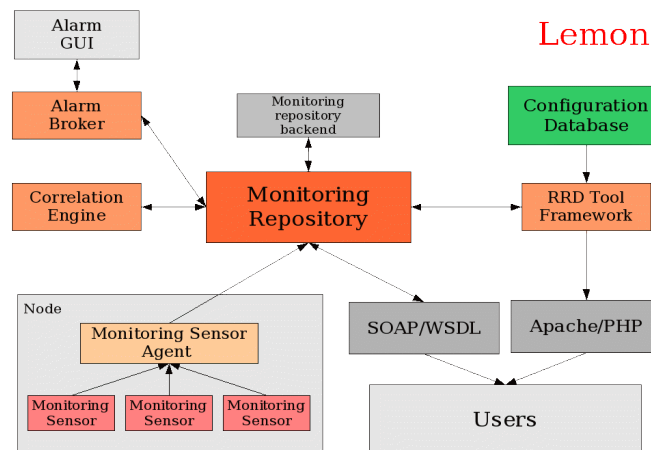


Fig. 3 The LEMON architecture

The LEMON architecture each node has one or more monitoring “sensors”. These sensors forward their information to a monitoring sensor agent, which caches the output and pushes the information up to the monitoring repository. The Monitoring repository can use either a database or flat-file back-end as storage, and can be queried via a SOAP interface. The elements of this architecture are shown in Figure 3.

6.11. MONALISA

[Home page](#)

Mon-Alisa is the monitoring infrastructure used within the CMS collaboration. It is based on the JINI design goals of providing dynamic service system that other agents can discover and utilise, if they require the information provided.

The system uses both the JINI infrastructure and SOAP for communication. With these, the system provides a self-describing global monitoring system with a set of loosely coupled higher level services.

The framework can integrate existing monitoring tools and procedures. Support already exists for integrating results from Ganglia, MRTG, LSF, PBS and user defined scripts.

7. Existing Querying Systems

Querying systems extract information from a local server and into it into some information system. This is an analogous function to the Monitoring Daemon in Figure 1. Some querying systems (most noticeably those providing direct monitoring) also include a front-end to display results.

7.1. DIRECT DATABASE MONITORING

Many systems exist that query underlying databases (usually MySQL) and provide that information, either directly (via some user interface) or by building some web interface. Whilst these may be useful to a sys-admin with direct access to the server, these are of limited use in this context as they do not inject the gathered information into some suitable information system.

The following is by no means an exhaustive list of available systems, which are unfortunately not of use in a distributed Grid infrastructure:

- [mtop](#) – Free package that shows which threads are utilising the CPU most.
- [MyTop](#) – Similar to mtop, shows thread usage in real-time.
- [MySQL Management](#) – commercial package with good support for monitoring but requiring direct access to database.
- [Moodss MySQL modules](#) – Modules to monitor a MySQL through a object-based spread-sheet. The moomps package provides a mechanism for relaying the information (acting as the Information System). Although moomps would allow multiple machines to be monitored, authentication and authorisation required for Grid-level support is not currently supported and the effort of deploying many moomps would be considerable.
- [Monit](#) – a stand-alone monitor, with a web interface, that can be used to monitor MySQL databases.
- [MySQLer](#) – a stand-alone monitor that generates web pages that describe a server's performance.

7.2. APMON

[Home page](#)

The ApMon system allows client code to submit arbitrary monitoring information to a near-by MonAlisa server. The communication is via UDP as shown in Figure 4.

This information will only update the configured MonAlisa Information System.

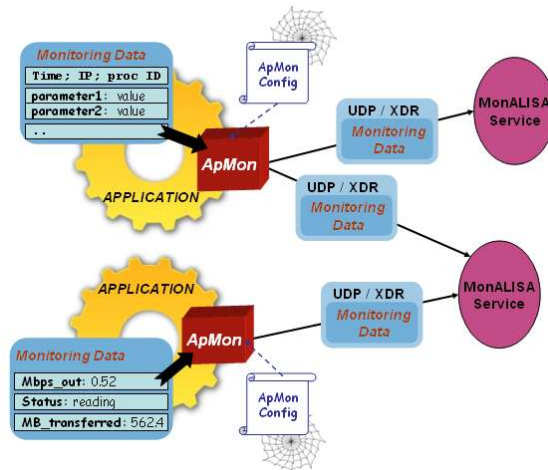


Fig. 4 The components of ApMon/Mona Lisa monitoring

7.3. LISA

[Home page](#)

LISA is a web-start application, so targeted at use with end-user machines through their browser.

LISA provides monitoring of hardware and OS and can upload this information to MonAlisa by acting as an ApMon agent. It can also participate in end-to-end network bandwidth measurement (e.g. IPERF), trigger actions when predefined conditions are met and can discover MonAlisa services.

7.4. GIP

[Home page](#)

The General Information Provider (GIP) is a mechanism to provide abstract information to an LDAP-based information system (for example MDS). The information provided may be external to the Glue schema.

GIP runs as a daemon on the system to be monitored. Incoming queries are replied to by cached information stored on the filesystem if this information is fresh enough. If the information is too old, then the corresponding plugin is rerun to generate fresh information. Plugins are written so they write the monitoring values to standard output (stdout), which the GIP daemon will redirect to the corresponding file.

Under the current deployment structure in LCG, GIP plugins are run to populate the MDS system and the R-GMA systems.

7.5. GRIDICE

[Home page](#)

GridICE is a distributed monitoring tool deployed on many (but, at time of writing, not all) sites throughout LCG. It aims to monitor many of the services provided by a Grid site primarily for VO-, GOC- and (to some extent) site-level consumption. GridICE propagates information via the MDS Information System, using an extension to the GLUE schema. It can aggregate information, provides different views of the available information and store data for retrospective and trend analysis. It does not provide a mechanism for integration into existing site-level monitoring systems, nor an easy mechanism for providing additional monitoring targets.

8. Conclusions

It is clear that vast amounts of information is available to those wishing to monitor a metadata service. These include monitoring the deployed database(s), metadata daemon(s), any application contains used, client-side monitoring and the underlying server fabric and OS.

Within a Grid context, there may be many services located in geographically diverse locations, with people requesting information who are as diversely located. These users may adopt different roles, depending on activity they are conducting. The information presented to any one user must be carefully chosen: showing irrelevant information to a user will hide the important data. However, the information must be navigable, as one cannot assume the role under which the user is view the data nor always which of the available data is import.

Given the vast amount of available information, care must be taken to limit the information collected. Moving this information about is a challenging problem, but one already solved by one of the existing Information Systems. R-GMA is one such system that lends itself to this form of monitoring, due to its design and targeted audience. Other systems, such as MonAlisa are also viable alternatives.

Two components that are currently lacking are a general user interface with which the end-user can navigate available data and a monitoring agent that is easy to deploy and can inject information into one (or more) information systems.

Existing monitoring systems, such as Nagios and Ganglia provide a rich front-end for navigating available data. The trigger feature of Nagios is also of use to sys-admins. Work is in progress at CERN to develop a monitoring front-end to LEMON that is similar to Oracle OEM-Console.

Work has started on the [MonAMI](#) project to develop a suitable Monitoring Daemon for collecting data that can then be passed to any number of information systems (LEMON, R-GMA, Ganglia, Nagios, ...) and be easily configured to site-specific requirements.