Central Data Warehouse for Grid Monitoring

Author: Ioan Gabriel Bucur

Supervisor: Wojciech Lapka

CERN openlab Summer Student Report

9/9/2011

Contents

Abstract

The purpose of this report is to describe the work performed at CERN during the openlab Summer Student Programme 2011, as part of the IT-GT-TOM section. The main goal of the project was to improve the functionality of the Central Data Warehouse at CERN by writing some stress tests to monitor its behaviour through a Nagios system, detecting deficiencies (slow response, missing data, etc), notifying the service managers in an automated way and implementing solutions to the detected deficiencies.

As a result of our work, new database structures and triggers have been created to compute the number of metric inserts into the Central Data Warehouse. Two new Nagios probes have been developed to make use of these database entities by detecting data flow anomalies and reporting them. In addition, by modifying an existing procedure to take into consideration the number of metric inserts, a better metric data flow has been ensured.

As a by-product, a purging mechanism has been developed to ensure automated cleaning of stale data. Last but not least, improvements made to Central Data Warehouse metric loading procedures have enabled a finer filtering of incoming metrics.

1 Introduction

1.1 The LHC and the WLCG

The **Large Hadron Collider (LHC)** is the largest and most powerful particle accelerator in the world. It consists of a ring of superconducting magnets, 27 kilometres in circumference, along with a number of particle accelerating structures. The gargantuan construction resides in a tunnel spanning the border between Switzerland and France at about 100 metres underground. [1] We can safely affirm that the LHC is home to some of the most important scientific experiments of the decade. The LHC will (hopefully) change our understanding of the Universe by answering big questions such as 'Why do particles have mass?', 'Do extra dimensions of space really exist?' or 'What is our Universe made of?' [2].

The LHC, through its six experiments (ATLAS, CMS, ALICE, LHCb, TOTEM and LHCf), generates somewhere around 15 petabytes (PB) of data per year consisting of HEP (High-Energy Physics) events. Each event represents a particle collision which consists of roughly 2MB of information. [3] Not only does all this data require an inordinate amount of storage space, but also it must be reconstructed, filtered, processed, classified and analysed. After the needed statistical quantities are extracted from the events, this real data must be in turn compared with expected data from simulations. [9]

To meet the enormous computational and storage need of the LHC experiments, the **Worldwide LHC Computing Grid (WLCG)** was launched in October 2008. This infrastructure was built by integrating thousands of computers and storage systems from more than 140 data centres and grid initiatives in 35 countries around the world. The WLCG is today the largest distributed or grid-based infrastructure, a collaborative computing environment on an unprecedented scale. [4]

The WLCG is made up of four layers or "tiers", each providing a specific set of services. Tier-0 is the **CERN Computer Centre**, which provides less than 20% of the computational capacity. However, all the information received from the LHC passes through this central hub, undergoing initial processing and storage. The data are distributed to eleven large Tier-1 sites with round-the-clock support for the grid and enough storage capacity for a large fraction of the data. These sites are located in Canada, Germany, Spain, France, Italy, the Nordic countries, Netherlands, the United Kingdom and the United States (two of them). The Tier-1 sites make data available to the over 160 Tier-2 sites, which are typically universities or other scientific institutes. Tier-2 sites have sufficient storing capacity and provide adequate computing resources for specific analysis tasks. Tier-3 sites represent individual scientists who access the WLCG facilities through local computing resources such as a university local cluster or even an individual PC. [4]

1.2 SAM

The **Service and Availability Monitoring framework (SAM)** is a grid monitoring and reporting system for large-scale production grids. It is built on top of open-source components such as Nagios, ActiveMQ and Django to provide a scalable and reliable monitoring infrastructure. WLCG uses SAM to monitor the availability and reliability of all resources provided by the computing centres collaborating in the project. [11]

SAM integrates many components, some off-the-shelf, some specifically designed for SAM, each with a well-defined functionality. **Nagios**, the heart of SAM, is used to execute monitoring tests. Through Nagios, tests are scheduled and

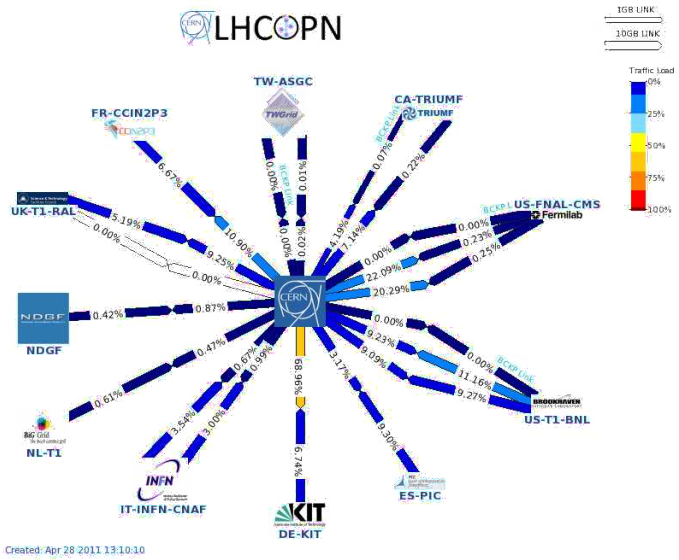

Figure 1: WLCG Tier-0/1 Structure, connected by the LHC Optical Private Network

their results are sent (via the messaging system) to the components that need them, such as the central Metric Store or the MyWLCG portal. **Apache ActiveMQ** is used as an integration framework, adding flexibility, reliability and scalability to the distributed SAM monitoring system. ActiveMQ Messaging serves to transport test results between components.

A set of databases is used for storing both configuration information - the **Aggregate Topology Provider (ATP)** and the **Metric Description Database (MDDB)** - and the test results produced by Nagios - the **Metric Result Store (MRS)**. ATP aggregates the grid topology from all authoritative sources and provides it to other components. Grid topology information includes projects (WLCG), grid infrastructures (EGI - European Grid Initiative, OSG - Open Science Group, NDGF - Nordic DataGrid Facility), sites, services, Virtual Organisations (VOs) [15] and their groupings, downtimes, as well as a history of all of the above. MDDB provides information concerning the metrics which are used to test the grid infrastructure: definitions, their properties, their grouping into profiles. Profiles represent combinations of metrics used for the computation of different availabilities and for the configuration of Nagios installations. Last but not least, MRS keeps the metric results for service end-points for the grid infrastructure, as well as their status changes. [13]

Another SAM component, the **Availability Calculation Engine (ACE)**, is used to process the raw test results into calculation metrics such as site and service availability and reliability. [8] Finally, the **MyWLCG portal** is the main visualisation tool, providing visualisations for both the test results and the availability calculations. MyWLCG presents a grid-aware view of the data collected by the Service Availability Monitoring framework.[11]

In Figure 3 we can observe a typical workflow scenario of the SAM framework. In the *Configuration Phase*, the Nagios Configuration System (NCG), which we will examine more closely in Section 2, receives topology data from ATP and profile data from MDDB. With this information, NCG builds the Nagios configuration in an automated way. After Nagios is configured, the system proceeds with the *Test Execution Phase*. In this phase, Nagios schedules and executes tasks against various grid services, either directly or through the Workload Management System (WMS). Test results are sent to the Messaging System. The *Storing Results Phase* follows. Profile, topology and test result information is all gathered in the Metric Results Store. This information is then forwarded to the MyWLCG portal for the final phase, the *Visualisation Phase*. The MyWLCG portal provides views of both current and historical test data.

SAM falls under the supervision of the GT (Grid Technologies) group of the IT department, which is responsible for maintaining and developing grid middleware: grid software components and the grid monitoring infrastructure. More specifically, SAM is coordinated by the Tools for Operation and Monitoring section of the GT group (IT-GT-TOM). The TOM section is responsible for designing, developing and running tools used by WLCG for the daily operation of its production infrastructure. [10]

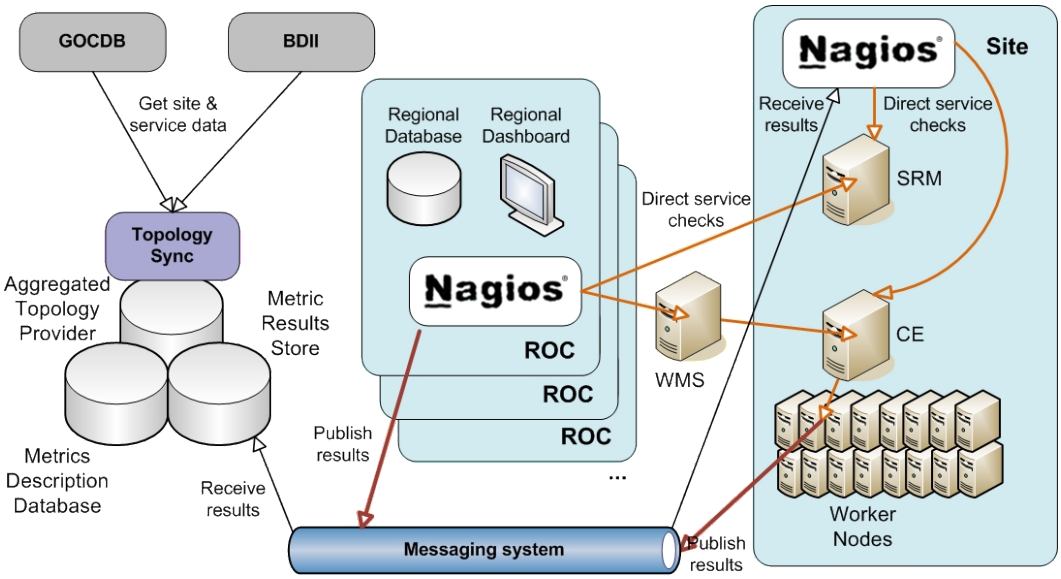

Figure 2: Overview of the SAM framework

2 SAM Architecture

In this section we will focus mainly on the components of SAM strongly related to the work presented in this report, namely the Nagios monitoring system and the central Metric Results Store. Whenever needed, further explications about other components as well as the relationships between SAM components will be provided.

2.1 Nagios

Nagios is a popular and powerful open source IT infrastructure monitoring software application. Although primarily a network monitoring system, Nagios can monitor any software element that has the ability to send collected data via a network. Such software elements are called *probes*, *plug-ins* or *checks* in Nagios terminology. Unlike many other monitoring tools, Nagios does not include any internal mechanisms for checking the status of hosts and services on your network, but relies solely on probes to do the work.

2.1.1 Nagios probes

A Nagios **probe** is basically a normal computer program (compiled executable or script) of varying complexity that that can be run from a command line and returns an integer value. Nagios will execute a probe whenever there is a need to check the status of a service or host. The probe does something to perform the check and then simply returns the results to Nagios. Nagios will process the results that it receives from the probe to determine the current status of hosts and services on your system and then takes any necessary actions (running event handlers, sending out notifications, etc). Theoretically, a Nagios probe could be written in any programming or scripting language. However, at CERN, four languages are preferred for probe development: C, Python, Perl or Bash (or more generally, shell scripting). [12]

The probes act as an abstraction layer between the monitoring logic present in the Nagios daemon process and the actual services and hosts that are being monitored. The upside is that you can monitor just about anything you can think of. If you can automate the process of checking something, you can monitor it with Nagios. The downside is the fact that Nagios has absolutely no idea what it is that you're monitoring. Only the probes themselves know exactly what they're monitoring and how to perform the actual checks. [5]

The only two conditions a probe must always respect (at least) is to exit with one of several possible return values and return at least one line of text output to STDOUT. Nagios allows four valid return codes for probes, each with its specific meaning. The output text that accompanies the return code should try to explain as clearly as possible (and in few lines if possible) what is the reason for the service returning that particular code. Starting with Nagios 3.x, probes can optionally print even multiple lines of text output.[6]

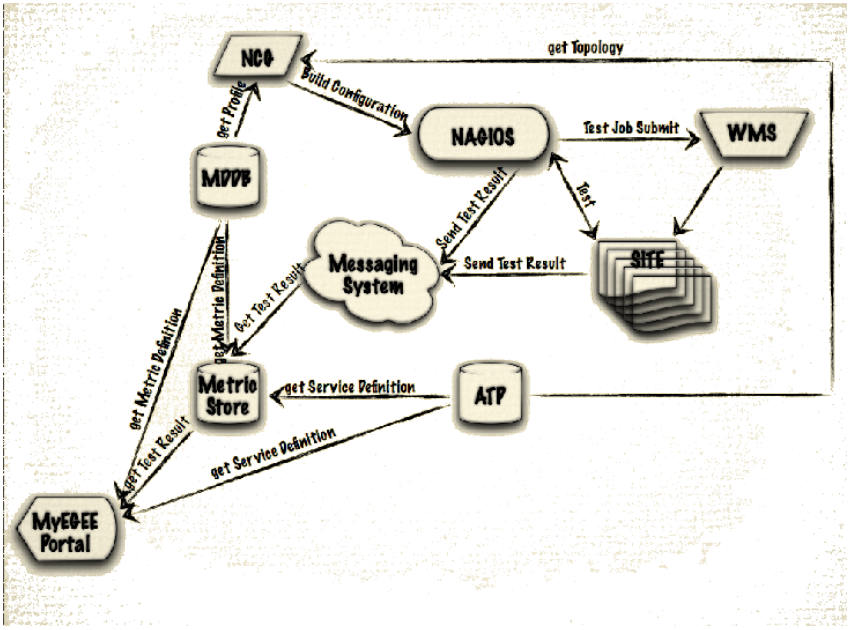

Figure 3: SAM Workflow Example

Return code	Service Status	Description
0	OK	The plug-in was able to check the service and it is functioning properly.
1	WARNING	The plug-in was able to check the service, but it returns a warning, usually because some metric or parameter is above or below a "warning" threshold.
2	CRITICAL	The plug-in was able to check the service, but it returns a critical warning, usually because some measured metric or parameter is above or below a "critical" threshold.
3	UNKNOWN	Invalid command line arguments were supplied to the plug-in or low-level failures internal to the plug-in (such as unable to fork, or open a tcp socket) prevented it from performing the specified operation. Higher-level errors (such as name resolution errors, socket timeouts, etc) are outside of the control of plug-ins and should generally not be reported as UNKNOWN states.

Table 1: Nagios Probe Valid Return Codes

Under normal circumstances, the Nagios system will receive one of the four valid codes from the probe. For each valid code, it displays the probe name, the return value coloured suggestively (OK = green, WARNING = yellow, CRITICAL = red, UNKNOWN = orange), along with one or multiple lines (starting from Nagios version 3.x) of probe text output from STDOUT. In unusual cases, when high-level errors occur, Nagios can receive other codes (such as 255 return by the Perl command "die"). In this case, Nagios considers the status of the host/service as CRITICAL. We can examine a sample Nagios display in Figure 4.

Probes may also return optional performance data that can be processed by external applications. If a probe returns performance data in its output, it must separate the performance data from the other text output using a pipe (—) symbol. Thus, the output format is: *"TEXT OUTPUT — OPTIONAL PERFDATA"*. In order to be readable, the performance data must be structured in a specific format. The expected format is: *'label'=value[UOM];[warn];[crit];[min];[max]*, where the warn(ing) and crit(ical) thresholds, the minimum and maximum values and the unit of measurement (UOM)

grid-monitoring.cern.ch	ch.cern.sam.MrsCheckDBInserts	CRITICAL	09-05-2011 20:18:30	0d 0h 0m 43s	1/3	MrsCheckDBInserts CRITICAL - Number of inserted messages is critically low: 0
	ch.cern.sam.MrsCheckDBInsertsDetailed	PENDING	N/A	0d 0h 1m 38s+	1/3	Service check scheduled for Mon Sep 5 20:48:44 CEST 2011
gridmsg106.cern.ch	msg-consume2db	PENDING	N/A	0d 0h 1m 38s+	1/3	Service check scheduled for Mon Sep 5 20:34:53 CEST 2011
gridops.cern.ch	SWAT	PENDING	N/A	0d 0h 1m 38s+	1/3	Service check scheduled for Mon Sep 5 20:49:53 CEST 2011
ops-monitor-dev.cern.ch	Nagios Host Summary	PENDING	N/A	0d 0h 1m 38s+	1/3	Service check scheduled for Mon Sep 5 20:36:02 CEST 2011
	Nagios Process	PENDING	N/A	0d 0h 1m 38s+	1/3	Service check scheduled for Mon Sep 5 20:51:02 CEST 2011
	Nagios Service Summary	OK	09-05-2011 20:18:15	0d 0h 0m 58s	1/3	26 Services: 8 OK, 1 WARNING, 3 CRITICAL, 0 UNKNOWN, 14 PENDING
	Nagios Web Interface	PENDING	N/A	0d 0h 1m 38s+	1/3	Service check scheduled for Mon Sep 5 20:52:11 CEST 2011
	hr.srce.CADist-Check	CRITICAL	09-05-2011 17:53:09	7d 3h 26m 4s	2/2	(Return code of 127 is out of bounds - plugin may be missing)
	hr.srce.CADist-GetFiles	CRITICAL	09-05-2011 17:38:09	7d 3h 11m 4s	3/3	(Return code of 127 is out of bounds - plugin may be missing)
	hr.srce.CertLifetime	OK	09-05-2011 19:17:08	196d 5h 21m 14s	1/2	CERT LIFETIME OK - Certificate will expire in 127.69 days (Jan 11 09:52:37 2012 GMT)
	org.egee.ImportGocdbDowntimes	OK	09-05-2011 19:40:07	0d 8h 39m 6s	1/3	OK - Downtimes successfully synchronized.
	org.nagios.DiskCheck	OK	09-05-2011 19:36:05	196d 5h 17m 40s	1/3	DISK OK - free space: / 22207 MB (65% inode=97%): /boot 67 MB (72% inode=99%): /dev/shm 1004 MB (100% inode=99%): /afs 8789 MB (100% inode=100%):
	org.nagios.MrsDirSize	OK	09-05-2011 19:37:50	196d 5h 15m 54s	1/3	OK - /var/spool/nagios2metricstore size: 16 KB
	org.nagios.NCGPidFile	OK	09-05-2011 20:14:49	0d 3h 4m 24s	1/3	FILE_AGE OK: /var/run/ncg/ncg.pid is 668 seconds old and 0 bytes
	org.nagios.NagiosCmdFile	OK	09-05-2011 20:05:54	25d 5h 29m 11s	1/3	OK - /var/nagios/rw/nagios.cmd
	org.nagios.ProcessCrond	OK	09-05-2011 20:07:44	40d 17h 54m 43s	1/3	PROCS OK: 1 process with command name 'crond'
	org.nagios.ProcessNpcd	OK	09-05-2011 20:05:56	44d 2h 29m 24s	1/3	PROCS OK: 1 process with command name 'npcd'
	org.nagiosexchange.LogFiles	WARNING	09-05-2011 20:09:49	7d 3h 9m 24s	1/1	(null)

Figure 4: Nagios monitoring display example

are optional.

PNP4Nagios is an external application used at CERN for processing performance data. PNP4Nagios is a Nagios add-on which analyses performance data and stores it automatically into Round Robin Databases (RRD). An example of PNP4Nagios output is illustrated in Figure 5. In this particular example, the total number of Nagios services in the last 4 hours is visualised. As we can see, in the last 4 hours, two new services were added.

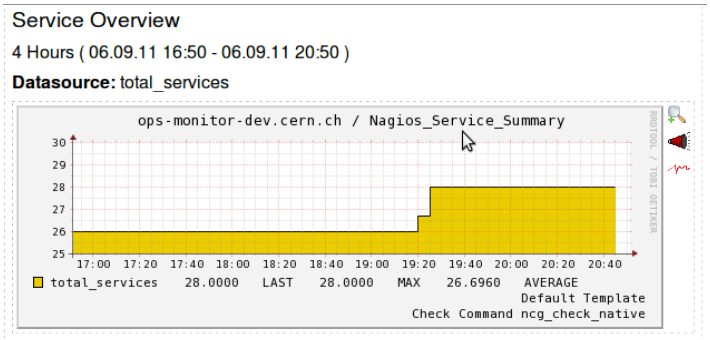

Figure 5: PNP4 Nagios Performance Data Visualisation Example

2.1.2 The Nagios Configuration Generator

As we have already mentioned in Section 1, there are two sources of configuration data for Nagios. The Aggregate Topology Provider provides the topology information, working out which sites and services should be tested, while the Metric Description Database provides the profile information, deciding which tests should be used to check a particular host. All this information is fed into the **Nagios Configuration Generator (NCG)**, which creates the appropriate Nagios configuration automatically. This is very useful since the configuration can be several thousand lines long. The administrators have a lot of control over the exact configuration that NCG generates so that it works well with their local setup.

NCG is a three phase configuration generator. The original target monitoring system for NCG was Nagios, but NCG now has more general application through a modular, extensible design. The three phases of NCG are as follows:

1. Information about all hosts and grid services associated with a named site is gathered (topology information gathering).

2. The topology is merged with data defining the probes used for gathering metrics from each type of grid service. After this merging a complete map of the site monitoring system is available.
3. The resulting output map is used to generate configuration files for a specific target monitoring tool (e.g. Nagios). This is the only phase which is dependent on the target tool.

Three files should be considered first when configuring Nagios using NCG: *Hash.pm, ncg.localdb and ncg.conf*. *Hash.pm* is where you configure the probes themselves. Here is an example configuration:

```
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{native} = "Nagios";
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{probe} = "demo/probe-example";
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{metricset} = "probe_example";
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{config}->{path} = $NCG::NCG_PROBES_PATH_GRIDMON;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{config}->{interval} = 5;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{config}->{timeout} = 30;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{config}->{retryInterval} = 3;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{config}->{maxCheckAttempts} = 3;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{flags}->NOHOSTNAME = 1;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{flags}->PNP = 1;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{parameter}->{'--warning} = 500;
$WLCG_SERVICE->{'ch.cern.demo_probe'}->{parameter}->{'--critical} = 200;
```

The variable $WLCG_SERVICE holds the probes (also called metrics in NCG terminology) and their configuration. The {config} values are very important since they define the way the probe is called by Nagios: how often, from which path. The {*config*} -> {*path*} value concatenated with {*probe*} value form a string that represents the full path of the probe (in this example it is "$NCG::NCG_PROBES_PATH_GRIDMON/demo/probe-example"). Through the {*flags*} key you can enable for example PNP4Nagios support for the performance data. Finally, the probes parameters are given arguments via the {*parameter*} key.

Every new probe must be added to a profile defined by a $WLCG_NODETYPE key, for instance:

```
$WLCG_NODETYPE->{roc}->{CE} = ['ch.cern.demo_probe', ...];
```

When we create a new profile for a probe such as `$WLCG_NODETYPE->{demo_profile}->{'demo_service'} = ['ch.cern.demo_probe'];` if the service name is not consistent with Grid Configuration Database (GOCDB) terminology, it must be explicitly specified as belonging to certain hosts in *ncg.localdb*. [14] These type of configuration lines are called static file rules:

```
ADD_HOST_SERVICE!grid-monitoring-probes.cern.ch!dummy_service
```

The new profile also needs to be added to the NCG configuration file (*ncg.conf*):

```
<NCG::LocalMetrics>
 ...
 <Hash> PROFILE = demo_profile </Hash>
 ...
</NCG::LocalMetrics>
```

After the configuration is written, the WLCG Nagios configuration generator, *ncg.pl* must be run. By default, the output is stored as a set of Nagios configuration files in the directory */etc/nagios/wlcg.d*. One must note that each time the site configuration changes (e.g. new services are added, hosts are removed), it is necessary to rerun *ncg.pl* and restart nagios (*/etc/init.d/nagios restart* or *service nagios reload*). Be aware that restarting the service might change key output for *backspace* from normal behaviour to printing ^?. To remove that behaviour, execute command `stty erase ^?`.

2.1.3 The Nagios::Plugin Module

`Nagios::Plugin` and its associated `Nagios::Plugin::*` modules are a family of Perl modules. The purpose of the collection is to help developers create plugins that conform the Nagios Plugin guidelines [6]. The `Nagios::Plugin` modules provides an object-oriented interface to the entire `Nagios::Plugin::*` collection, while `Nagios::Plugin::Functions`

provides a simpler functional interface to a useful subset of the available functionality [7]. In combination with the Nagios embedded Perl interpreter (ePN), the Nagios::Plugin makes Perl a very suitable scripting language for writing Nagios probes.

A Nagios::Plugin object is created by means of the *new* method / constructor: `Nagios::Plugin->new(usage => $usage_string, version => $VERSION, blurb => $blurb, extra => $extra, url => $url, license => $license, shortname => $shortname, plugin => basename $0, timeout => 15, );`

Defining arguments for your probe becomes simple with the *add_arg* method: `$plugin->add_arg( spec => "hello=s", help => "Hello string", required => 1, default => "Hello, world!" );` The *spec* argument is a regular Perl `Getopt::Long` argument specification. It consists of a series of one or more argument names for this argument (separated by "—"), suffixed with an "=¡type¿" indicator if the argument takes a value. Types include "=s" for a string argument, "=i" for an integer argument and "-f" for a float argument. Appending an "@" in the end indicates multiple such arguments are accepted. The following are some examples: `hello=s; hello|h=s; ports|port|p=i; exclude|X=s@; verbose|v+`.

The `nagios_exit(CODE, $message)` method permits exit with return code CODE and a standard Nagios message of the form "SHORTNAME CODE - $message", where *shortname* is defined in the `Nagios::Plugin` object constructor. Nagios::Plugins exports the return code constants *OK, WARNING, CRITICAL, UNKNOWN* by default. Performance data can be added easily via method *add_perfdata*, which has the following signature: `add_perfdata( label => $label, value => $value, uom => "kB", threshold => $threshold )`. The method may be called multiple times and it includes the performance data in the standard plug-in output messages printed by the various exit methods.

2.2 The Metric Results Store

The Metric Results Store is a centralized data warehouse at CERN based on Oracle. It stores a copy of each metric output from the EGI (European Grid Initiative) and OSG (Open Science Grid) services. This represents around 400 sites with 1,800 services being monitored and more than 600,000 metric results stored daily. The metric data is loaded via the Messaging Infrastructure to table METRICDATA_SPOOL. It then remains there for a number of minutes (currently 10 minutes). This happens so that we can ensure that we received data from all Nagioses and that the received data is ordered by test execution (column *check_time*). Every 3 minutes, an Oracle DBMS_scheduler job called SAM_MS_LOADDATA is started. This job calls the procedure *dataloader.loadmetricdata* from the package DATALOADER, which loads data from METRICDATA_SPOOL to the tables METRICDATA, METRICDATA_LATEST and STATUSCHANGE_SERVICE_PROFILE.

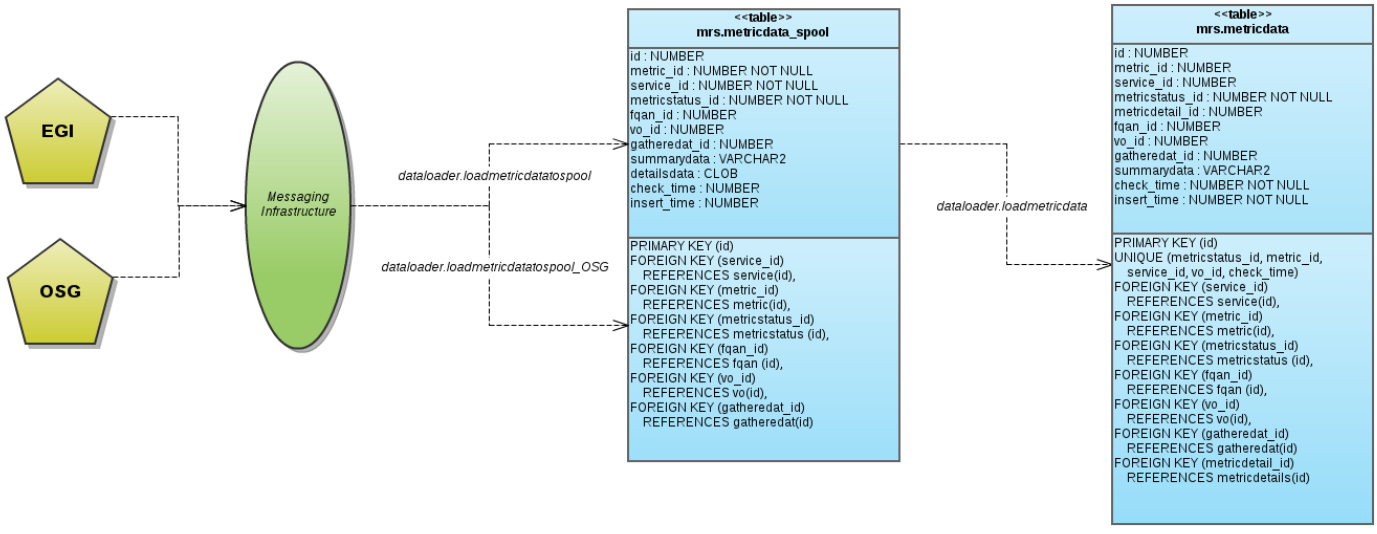

Figure 6: Metric Results Store Data Flow

3 Central Data Warehouse Tuning

Sometimes the metric data is "blocked" on the messaging broker (e.g. data consumers were down). In this case MRS will not receive any data from the message brokers for a period of time. We need to detect such a situation - data has not been received in the last "n" hours - and for this purpose we will use a Nagios probe. Also, there are certain situations when MRS receives more data that normally (e.g. when a consumer that was down is restarted). In this case, metric data loading should be delayed (Section 2.2 -> job SAM_MS_LOADDATA should wait) until the insertion rate returns to normal values. In other words, the system should wait until all the messages piled up on the broker are consumed.

3.1 Gathering Metric Results Store Statistics

In order to challenge the problem of abnormal data flow, we require supplementary procedures that can provide a quantitative overview of this parameter. We also require associated data structures to store these new measurements. Our proposed solution consists of computing the metric load (number of metrics inserted) into the table METRICDATA_SPOOL per hour and per minute. For this purpose, we implemented three new Oracle PL/SQL triggers called on each insertion to METRICDATA_SPOOL. Only the insertions of metrics that come from messaging are considered, so we exclude those which are marked as MISSING or REMOVED. These triggers affect accordingly three new data structures created in the MRS database. In this section, we will describe in details the new structures and triggers.

Table **METRICSTORE_CURR_LOAD_H** contains the hourly number of metrics inserted in METRICDATA_SPOOL. A new row is inserted each hour, uniqueness being ensured by the UNIQUE key constraint (rec_date, hour). Data is kept in this table only for one month (30 days), after which it is purged (via trigger METRICDATA_SPOOL_INSERT_H). The table has the following columns (see Figure 7):

- *id* NUMBER : holds the row / record identification number; always use the associated sequence **METRICSTORE_CURR_LOAD_H_SEQ** when inserting to ensure that each new row has a unique ID
- *rec_date* VARCHAR2 : a string keeping the date when the record was inserted in easy-to-read format DD-MON-YYYY (DEFAULT is current UTC date)
- *hour* NUMBER : this column stores the hour when the record was inserted into the table (DEFAULT is current UTC hour)
- *timestamp* TIMESTAMP : this columns stores the exact time of insertion in TIMESTAMP format (DEFAULT is current UTC timestamp)
- *number_of_records* NUMBER : this is the column that keeps the measurements regarding metric load per hour

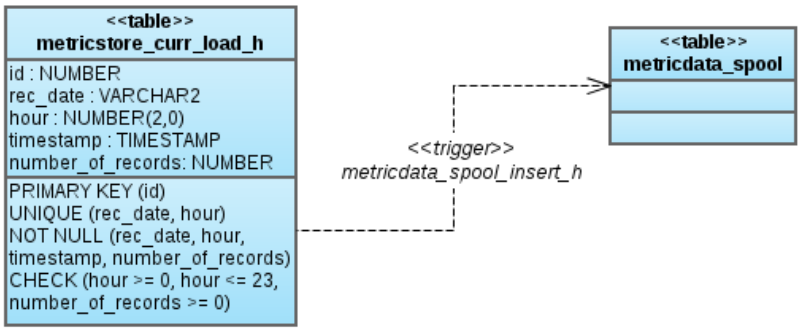

Figure 7: Table METRICSTORE_CURR_LOAD_H

Trigger **METRICDATA_SPOOL_INSERT_H** operates on METRICSTORE_CURR_LOAD_H after each new metric result is inserted into METRICDATA_SPOOL. The trigger functions according the flowchart in Figure 8.

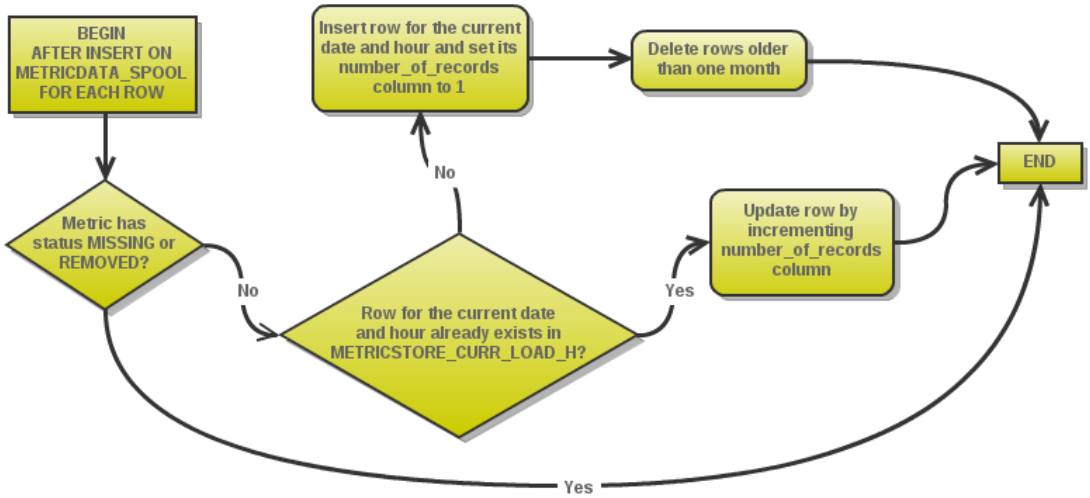

Figure 8: Trigger METRICDATA_SPOOL_INSERT_H

Table **METRICSTORE_CURR_LOAD_H_DET** is simply a more detailed version of table METRICSTORE_CURR_LOAD_H. This table does not consider all metrics that are being inserted the same, but splits them into groups according to their profile (e.g. ALICE_CRITICAL, ATLAS_CRITICAL, OSG), their service flavour (e.g. SRMv2, CE, CREAM-CE, gLite-CE) and the name of the corresponding National Grid Initiative (NGI) from where they originate (e.g. NGI_IT, NGI_NL, AsiaPacific). New rows are inserted into this table each hour for every (profile, service flavour, NGI) combination. Uniqueness is ensured by the UNIQUE key constraint (rec_date, hour, profile_id, flavour_id, ngi_name). Data is kept in this table only for one month (30 days), after which it is purged (via trigger METRICDATA_SPOOL_INSERT_H_DET). The table has the following columns (see Figure 9):

- *id* NUMBER : holds the row / record identification number; always use the associated sequence **METRICSTORE_CURR_LOAD_H_D_SEQ** when inserting to ensure that each new row has a unique ID
- *rec_date* VARCHAR2 : a string keeping the date when the record was inserted in easy-to-read format DD-MON-YYYY (DEFAULT is current UTC date)
- *hour* NUMBER : this column stores the hour when the record was inserted into the table (DEFAULT is current UTC hour)
- *timestamp* TIMESTAMP : this columns stores the exact time of insertion in TIMESTAMP format (DEFAULT is current UTC timestamp)
- *number_of_records* NUMBER : this is the column that keeps the measurements regarding metric load per hour
- *profile_id* NUMBER : this is the metric's profile; the same metric can belong to multiple profiles, or can have no associated profile, in which case this column has the value NULL
- *flavour_id* NUMBER : this is the metric's service flavour; cannot be NULL
- *ngi_name* VARCHAR2 : this is the name of the metric's originating NGI; can be NULL

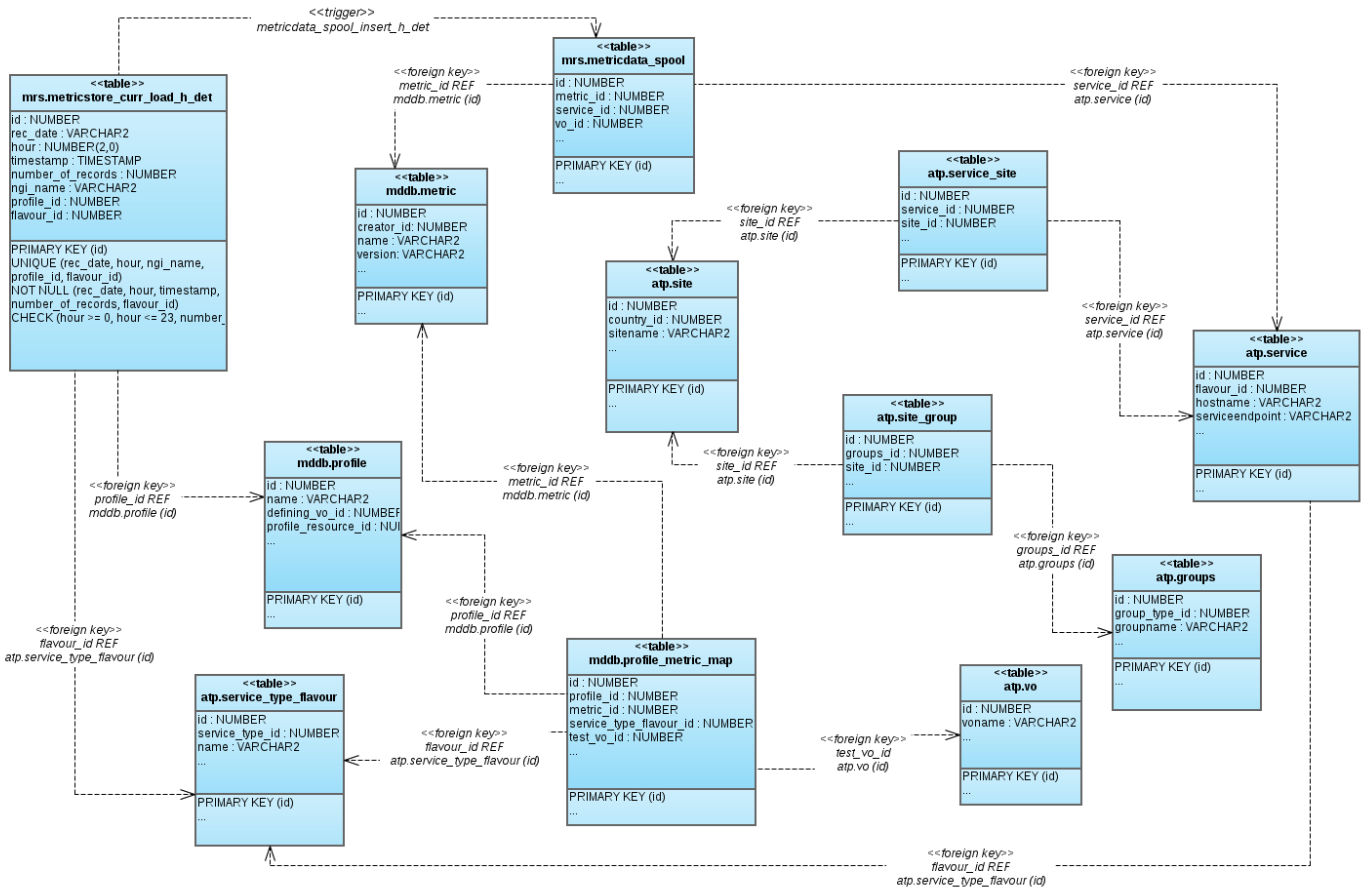

Figure 9: Table METRICSTORE_CURR_LOAD_H_DET

Trigger **METRICDATA_SPOOL_INSERT_H_DET** operates on METRICSTORE_CURR_LOAD_H_DET after each new metric result is inserted into METRICDATA_SPOOL. The trigger functions according to the flowchart in Figure 10.

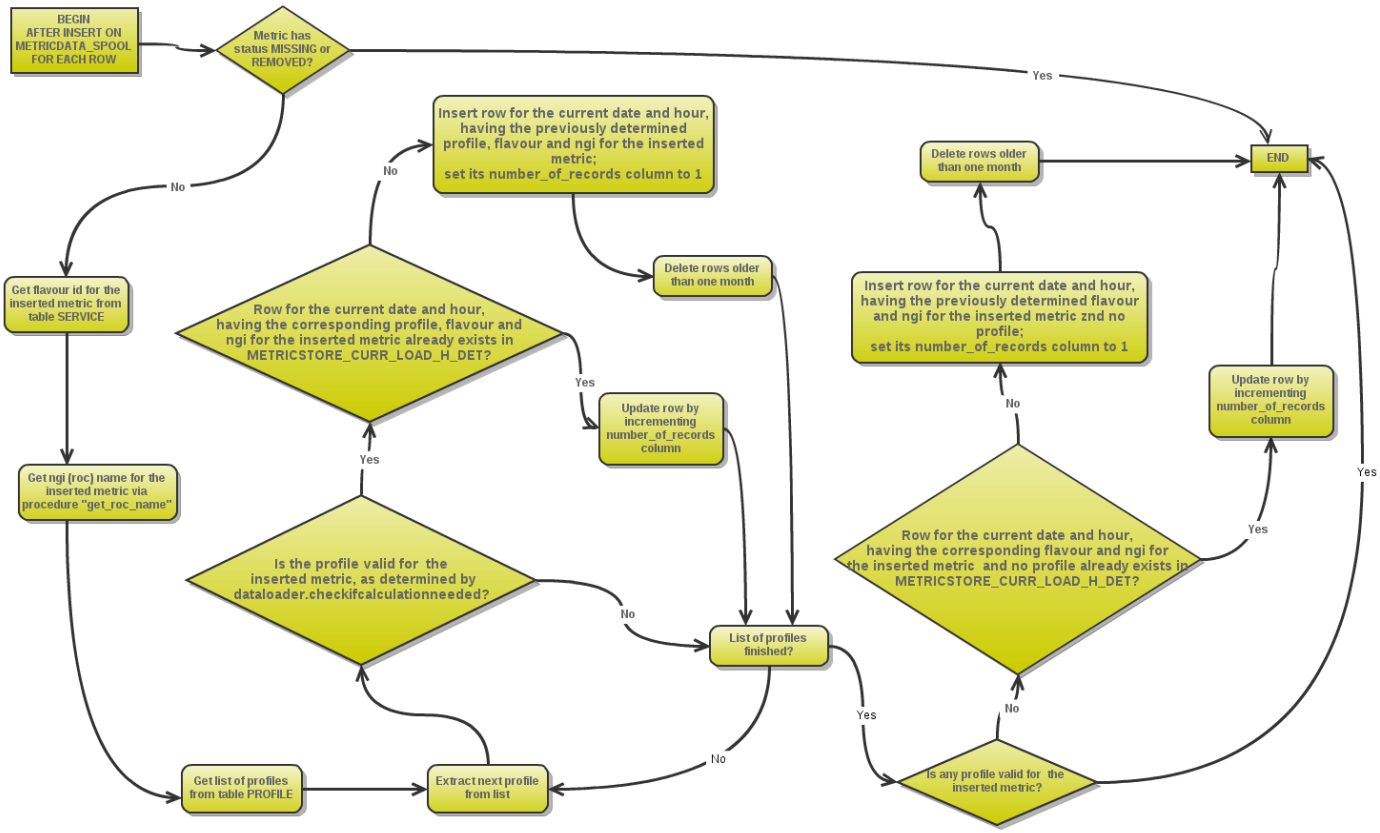

Figure 10: Trigger METRICDATA_SPOOL_INSERT_H_DET

Because of the three new added columns, *ngi_name, flavour_id and profile_id*, after which the inserted metrics are grouped, trigger METRICDATA_SPOOL_INSERT_H_DET is much more complex. In order to find out the values of these three columns for each metric, we must use the existing associative tables (see Figure 9). We obtain the *flavour_id* simply by looking in table SERVICE for the *flavour_id* corresponding to the inserted metric's *service_id*.

Obtaining the *ngi_name* is a little more complicated. Starting from the *service_id* in METRICDATA_SPOOL, we find the associated *site_id* by looking in the associative table SERVICE_SITE. Then, we look in the associative table SITE_GROUP to find the *group_id* associated with the respective *site_id*. Finally, using the *group_id* we get the *groupname* from table GROUPS, which is the *ngi_name* that we are looking for (see Figure 9). All these operations are performed by the stored function `get_roc_name(service_id IN NUMBER) RETURNS VARCHAR2`.

Getting the metric's corresponding *profile_id* is difficult because a metric can be associated with multiple profiles or with no profile whatsoever. Therefore, we must check for each existing profile if the association exists. To do this, we start by getting the list of profiles (profile IDs) from table PROFILE via a cursor. For each profile, we check for the association with the current inserted metric by looking in the associative table PROFILE_METRIC_MAP. This verification is performed by the function `checkIfCalculationNeeded(profileId INTEGER, serviceId INTEGER, metricId INTEGER, voId INTEGER) RETURNS BOOLEAN`. We simply pass the parameters *service_id*, *metric_id* and *vo_id* from the inserted metric row and every *profile_id* from the list in a loop. Whenever the function returns true, it means that the metric belongs to the respective profile. If no profile matches the inserted metric, we write NULL in the corresponding column.

Table **METRICSTORE_CURR_LOAD_M** contains the minutely number of metrics inserted in METRIC-DATA_SPOOL. A new row is inserted each minute, uniqueness being ensured by the UNIQUE key constraint (minute, hour). Data is kept in this table only for one day, after which it is purged (via trigger METRICDATA_SPOOL_INSERT_M). The table has the following columns (see Figure 11):

- *id* NUMBER : holds the row / record identification number; always use the associated sequence **METRICSTORE_CURR_LOAD_M_SEQ** when inserting to ensure that each new row has a unique ID
- *minute* NUMBER : this column stores the minute when the record was inserted into the table (DEFAULT is current UTC minute)
- *hour* NUMBER : this column stores the hour when the record was inserted into the table (DEFAULT is current UTC hour)
- *timestamp* TIMESTAMP : this columns stores the exact time of insertion in TIMESTAMP format (DEFAULT is current UTC timestamp)
- *number_of_records* NUMBER : this is the column that keeps the measurements regarding metric load per minute

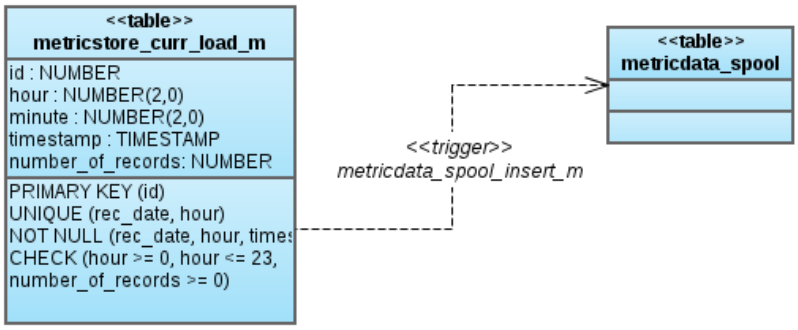

Figure 11: Table METRICSTORE_CURR_LOAD_M

Trigger **METRICDATA_SPOOL_INSERT_M** operates on METRICSTORE_CURR_LOAD_M after each new metric result is inserted into METRICDATA_SPOOL. The trigger functions according to the flowchart in Figure 12.

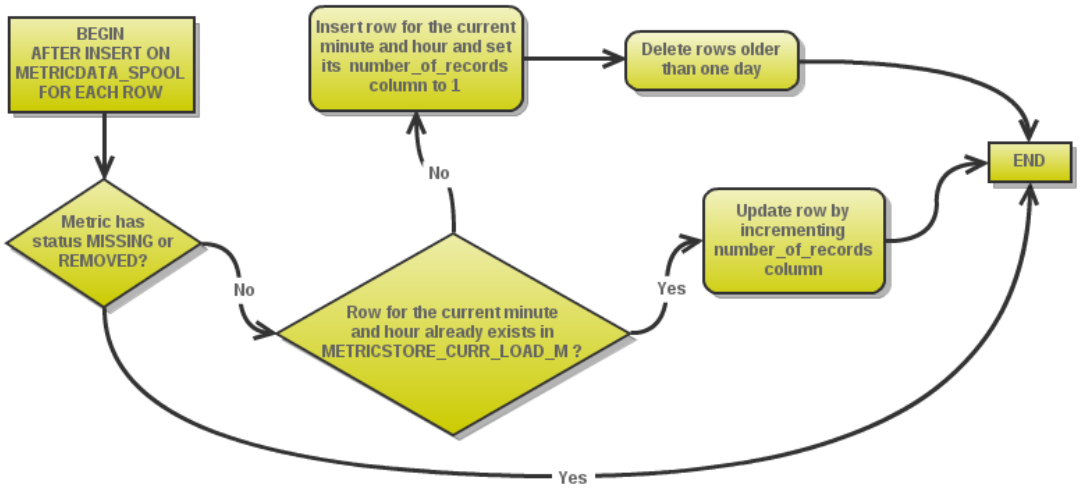

Figure 12: Trigger METRICDATA_SPOOL_INSERT_M

The above tables and triggers were written both in Oracle, for the CERN central database, and in MySQL, for the local NGI databases. During the migration from PL/SQL to MySQL code, some design-level modifications had to be performed. The functionalities of the three Oracle triggers were amassed in a single trigger for MySQL because the latter does not support multiple triggers with the same action time and event for one table. In addition, the sequences used for generating table unique IDs in Oracle were replaced by the AUTOINCREMENT feature in MySQL.

3.1.1 Use Statistics to Determine if Metric Data Should Be Loaded

Like we mentioned in 2.2, the Oracle DBMS_scheduler job SAM_MS_LOADDATA loads the metric data from the MRS buffer (METRICDATA_SPOOL) to the appropriate metric data storage tables every 3 minutes by calling *dataloader.loadmetricdata*. However, this process should not occur when the number of metrics inserted in METRICDATA_SPOOL is too high, signalling an abnormal situation. Instead, the process should be delayed until the parameter "number of metrics inserted" returns to normal values.

To solve this problem, we define a new *startThreshold* parameter for the procedure *dataloader.loadmetricdata*. Then, using the data stored in METRICDATA_CURR_LOAD_M, we can determine how many metrics have been inserted in the previous minute in table METRICDATA_SPOOL (we do not consider the number of metrics that is inserted in METRICDATA_SPOOL in the current minute, because that number keeps changing until the minute is over). We compare this value with the startThreshold and if it is above the threshold, the procedure returns immediately without affecting the database state in any way. By doing this, we simply delay the metric data loading because SAM_MS_LOADDATA will call dataloader.loadmetricdata after 3 minutes. The number of inserted metrics per minute will be checked again and if it returned to normal values (below *startThreshold*) the procedure can execute normally. In Figure 13 we can see the execution flow of *dataloader.loadmetricdata*. New elements are emphasised by means of the colour red.

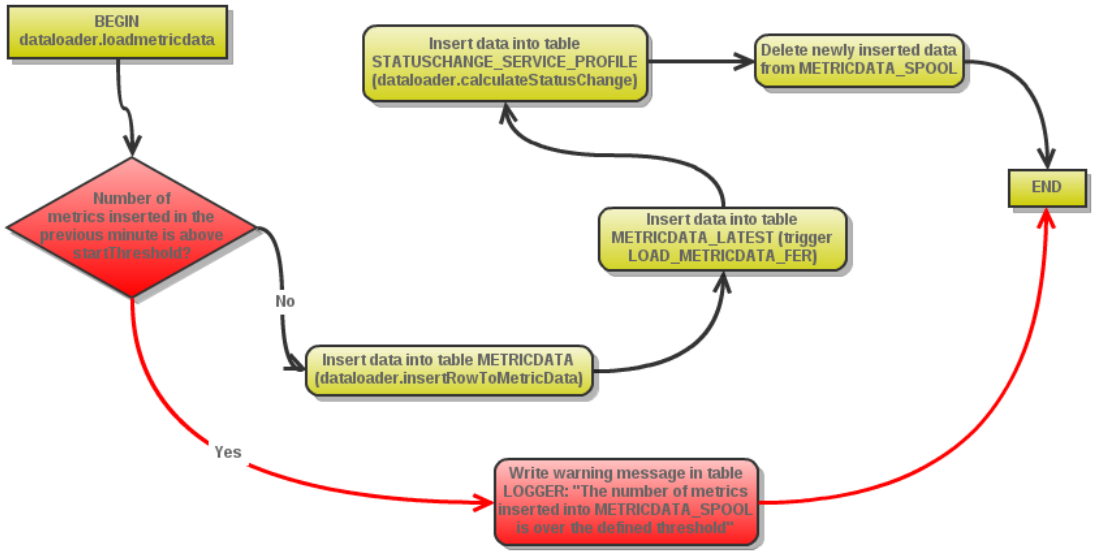

Figure 13: Procedure dataloader.loadmetricdata

3.2 Automatically Reject Old Metrics

Not all metrics coming to the Metric Results Store via the ActiveMQ Messaging Infrastructure are inserted into METRICDATA_SPOOL. Some are filtered out before entering the buffer / spooler for various reasons: service_id is NULL or metric_id is NULL or both. This filtering is done in procedure *dataloader.loadmetricdatatospool* for metrics coming from the EGI and in procedure *dataloader.loadmetricdatatospool_OSG* for metrics coming from the OSG.

From time to time, because of the inadvertent delays in the system, metrics are distributed to METRICDATA_SPOOL at a late time. These old metrics should be filtered out since the information they contain is not recent enough to be considered relevant. Our solution was to improve the filtering process which already existed in the two aforementioned procedures from the DATALOADER package. We added a new integer parameter called *metric_rejected_age_days* that clears up the definition of "old metric". If a metric is older than *metric_rejected_age_days* (for this we compare the metric's insert time with the metric's check time) it is inserted into METRICDATA_REJECTED or METRICDATA_REJECTED_OSG depending on its grid of origin.

The following flowchart shows the improvements made to *dataloader.loadmetricdatatospool* in red. As a side note, METRICDATA_REJECTED has a column that contains the reason for the rejection. Since "metric_id is

NULL and service_id is NULL" is considered a different reason when compared to "metric_id is NULL" or "service_id NULL", it follows that there are separate control flow branches for all these reasons. The flowchart for *dataloader.loadmetricdatatospool_OSG* is identical except for adding OSG to the names, so we will not show it as well.

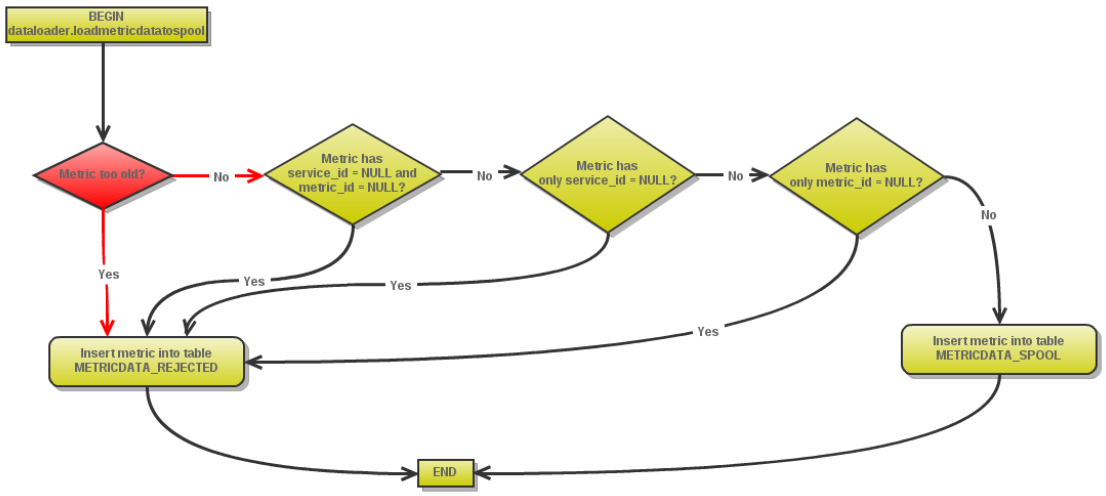

Figure 14: Procedure dataloader.loadmetricdatatospool

3.3 Add Oracle Data Purging Mechanism

Taking into consideration the huge amount of data that has to be kept in the SAM databases in general and in the Central Data Warehouse in particular, it is clear that old or stale information should be removed from the databases as soon as it is no longer required and in an automatic way if possible. We implemented the purgeMetricstore procedure in package DATALOADER in order to provide a partial solution to this issue. We also created the SAM_MS_PURGE_METRICSTORE job, which calls *dataloader.purgeMetricstore* daily.

The MRS should keep data for a full 6 months in table METRICDATA, one month in table METRICDATA_REJECTED and table METRICDATA_REJECTED_OSG, 12 months in table STATUSCHANGE_SERVICE_PROFILE and 7 days in LOGGER. For each table, there is a corresponding parameter in *dataloader.purgeMetricstore* that indicates how long the data should be kept in the respective tables. These parameters are suggestively named: *mdata_full_months_kept*, *mdata_rej_months_kept*, *logger_days_kept* and *statuschange_months_kept*. SAM_MS_PURGE_METRICSTORE calls *dataloader.purgeMetricstore* by assigning the values mentioned at the start of the paragraph to their respective parameters.

For computational ease, we consider a month to be a 31-day period. We must also explain what the meaning of "full months" is in this case. In table METRICDATA, data is guaranteed to be available for 6 full months. For instance, if we are in October, it is guaranteed that table METRICDATA still contains all the data from each day of April, May, June, July, August and September. This means that the data could be kept for more than 6 * 31 days, as long as it is needed to get at least 6 full months.

The new procedure *dataloader.purgeMetricstore* takes into consideration aspects such as partitions or foreign key dependencies when deleting data (Figure 15). Removing data from LOGGER, METRICDATA_REJECTED or METRICDATA_REJECTED_OSG requires a simple DELETE command. However, when deleting data from METRICDATA_REJECTED and METRICDATA_REJECTED_OSG, we noticed that it was too lengthy. We created the indexes MDATA_REJ_CHECKTIME_IDX and MDATA_REJ_OSG_CHECKTIME_IDX respectively so that this process would run in a much shorter time. As a side note, the MDATA_REJ_CHECKTIME_IDX index was also added to the MySQL databases for the same purposes.

When a metric is removed from table METRICDATA, the metric must also be deleted from METRICDATA_LATEST together with its metric details from table METRICDETAILS. These extra operations are done via newly created trigger DELETE_METRICDATA_DETAILS. Regarding table STATUSCHANGE_SERVICE_PROFILE, for each (profile, service) UNIQUE combination, the newest record older than 12 months is not deleted but updated to be exactly 12 months old (more exactly, its *timestamp* column value is updated to "current UTC time - 12 months"). This is done

because we must always keep the last status change that occurred before the defined interval (12 months) for every metric.

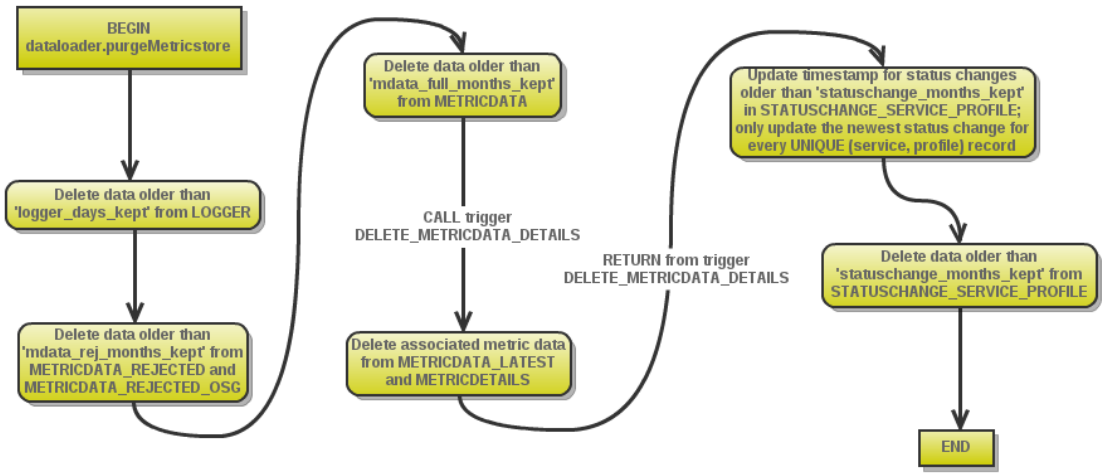

Figure 15: Procedure dataloader.purgeMetricstore

4 Central Data Warehouse Monitoring

As we have already mentioned in the introduction to Section 3, there are situations in which incoming metric output data is "blocked" on the messaging broker. We can detect this anomaly by examining manually the tables METRICSTORE_CURR_LOAD_H or METRICSTORE_CURR_LOAD_H_DET (Section 3.1), but we wish to be notified of such situations in an automated way. The underlying Nagios monitoring system provides an excellent means of achieving just that. Using the information stored in the aforementioned tables, we can develop probes that will monitor these particular anomalies.

4.1 MrsCheckDBInserts Probe

MrsCheckDBInserts is a probe which monitors if MRS is receiving metric results. It uses the data collected from METRICSTORE_CURR_LOAD_H: number of records inserted in MRS (METRICDATA_SPOOL) per hour. In its initial version, the probe would connect to the database and extract the data directly from the tables via SQL queries. In the current version however, the probe gets its data indirectly via a specialised web service, both for security reasons and to avoid unnecessary database connections in case multiple probes require the same data. The probe is written in Perl and employs the highly useful Nagios::Plugin package.

MrsCheckDBInserts has the following input parameters:

- *host_url* — default value: "http://localhost/" — This is the URL of the server that hosts the service. Concatenated with web_service_path, it forms the complete service URL.
- *web_service_path* — default value: "myegi/sam-pi/metricstore_current_load_per_hour?" — This is the name/path of the service from where the MRS load data is obtained. Concatenated with host_url, it forms the complete service URL.
- *warning* — default value: 3000 — If the number of metric results inserted per hour in METRICDATA_SPOOL is below this threshold, a simple warning is generated (return code 1).
- *critical* — default value: 1000 — If the number of metric results inserted per hour in METRICDATA_SPOOL is below this threshold, a critical warning is generated (return code 2).

The probe is divided into three phases: argument parsing, XML parsing and data interpretation. In Figure 16 we can see these phases illustrated. This flowchart is general enough so that it can be applied to any monitoring probe that gets its metric data via a web service that generates XML output. In our case, the flowchart applies to this probe as well as to the *MrsCheckDBInsertsDetailed* probe in the next section.

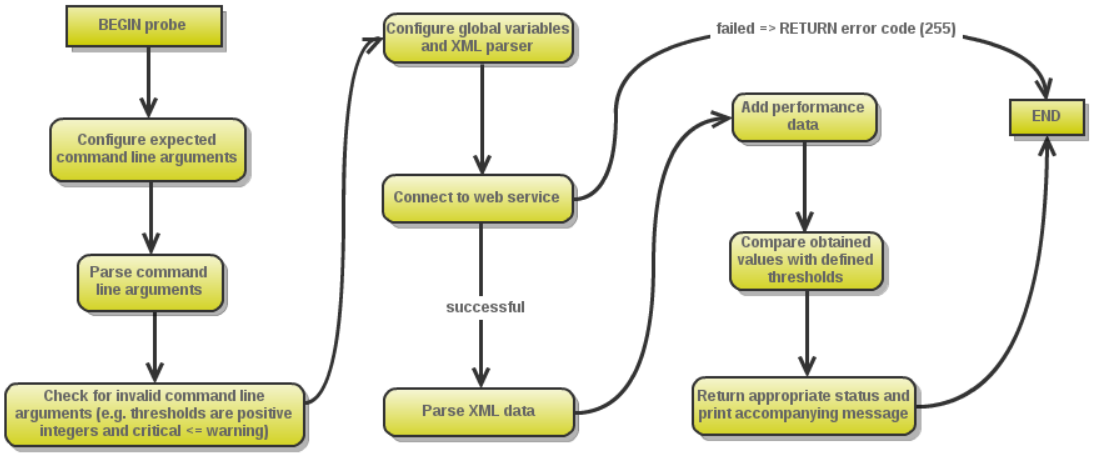

Figure 16: Probe flowchart

The initial phase is done using the Perl `Nagios::Plugin` module. Options are added with the *add_arg* method. Options are then parsed with the *getopts()* method. The `Nagios::Plugin` does some automatic command line argument check-in: the number of arguments passed is correct, the type passed is correct, each options that requires an argument gets an argument. The rest of the checking needs to be done manually in the Perl script. For all command line argument errors, the Nagios UNKNOWN code is returned, as specified in the Nagios Plug-in Development Guidelines. [6] After the configuration phase, the data needed is in raw XML format.

The MRS web service offers data in XML format, so the probe needs an XML parser. The `XML::Parser` module offers an efficient and flexible solution. After the XML data is downloaded (using the `LWP::Simple` module), it is passed to the manually configured parser. The parser has assigned four handler routines: Start (start of tag), End (end of tag), Char (normal text) and Default (all other content). Each can be configured to reference any Perl subroutine. After the XML parsing phase, the data is completely processed and stored in appropriate variables.

Adding performance data and returning the appropriate code to Nagios is simple when using the `Nagios::Plugin` module. The performance data is added with the *add_perfdata* method, while for code returning, the *nagios_exit* method is used. The final processed data is compared with the defined "warning" and "critical" thresholds in order to determine which code the probe should return. Informative output is attached to the return code. In the case of *MrsCheckDBInserts* the number of metrics inserted per hour is printed. The same number along with the number of metrics inserted per day is added as performance data.

4.2 MrsCheckDBInsertsDetailed Probe

MrsCheckDBInsertsDetailed is a probe which monitors if MRS is receiving certain metric results, selected by profile, service and NGI name. It uses the data collected from METRICSTORE_CURR_LOAD_H_DET: number of records inserted in MRS (METRICDATA_SPOOL) per hour, having a separate entry for each (profile_id, service_id, ngi_name) tuple. The probe, which is just a finer grained version of *MrsCheckDBInserts* gets its data indirectly via a specialised web service. The probe is written in Perl and employs the highly useful Nagios::Plugin package.

MrsCheckDBInsertsDetailed has the following input parameters:

- *host_url* — default value: "http://localhost/" — This is the URL of the server that hosts the service. Concatenated with web_service_path, it forms the complete service URL.
- *web_service_path* — default value: "myegi/sam-pi/metricstore_current_load_per_hour_detailed?" — This is the name/path of the service from where the MRS load data is obtained. Concatenated with host_url, it forms the complete service URL.
- *warning* — default value: 0 — If the number of metric results inserted per hour in METRICDATA_SPOOL is below this threshold, a simple warning is generated (return code 1).
- *critical* — default value: 0 — If the number of metric results inserted per hour in METRICDATA_SPOOL is below this threshold, a critical warning is generated (return code 2).

- *profile* — default value: WLCG_CREAM_LCGCE_CRITICAL — This is the profile (group of metrics) we wish to check.
- *ngi* — default value: all — This is the NGI name of the desired metrics. This option can receive singular NGI name or a list of names separated by commas (,). The option can also be specified multiple times and all the lists of arguments will be considered. If there is no NGI name mentioned, each service is considered by the probe for all NGIs.
- *service_flavour* — default value: all — This parameter holds the service flavour(s) that we are interested in. This option can receive a singular service flavour or a list of flavours separated by commas (,). The option can also be specified multiple times and all the lists of arguments will be considered. If there is no service flavour mentioned, the probe checks all service flavours.
- *hours* — default value: 2 — Through this option, we parametrize the time period we are interested in. We can evaluate how many metrics were inserted in the last "n" hours, were "n" is variable.

The probe is divided into three phases (argument parsing, XML parsing and data interpretation), just like *MrsCheckDBInserts* and follows the same flowchart (Figure 16). The only difference between this probe and the other is the extra number of arguments, which add extra flexibility to the data examined. There are some remarks to make about the new arguments.

The *ngi* and *service_flavour* options can receive lists of arguments separated by commas. These options can also be called multiple times and their lists will be concatenated. For example:

- `...  --ngi NGI_NL,NGI_IT,AsiaPacific ...` is the same thing as
- `...  --ngi NGI_IT --ngi AsiaPacific,NGI_NL ...` or
- `...  --ngi NGI_IT --ngi NGI_NL --ngi AsiaPacific ...`

The same option can be called multiple times even if interleaved by different options, like in these examples:

- `...  --ngi NGI_IT --service_flavour CREAM-CE,CE,SRMv2 --ngi Russia,CERN ...`
- `...  --service_flavour SRMv2 --ngi AsiaPacific --service_flavour CE --ngi Russia,CERN,NGI_NL . ...`
- `...  --warning 0 --ngi NGI_CZ --service_flavour SITE_BDII --hours 4 --ngi NGI_UK --critical 0 ...`

This probe provides much more detailed output. For each chosen service flavour via command line arguments, the number of inserts in the last "n" hours is written on a separate line. Also, the complete list of NGI names included as valid points of origin for the desired service flavours is printed as output. Regarding return codes, it should be noted that the number of hourly inserts for each service flavour in the argument list is tested against the given thresholds and generates a return code. Thus, it is possible that we obtain multiple return codes (we are talking about OK, WARNING and CRITICAL, i.e. normal execution return values; UNKNOWN is a separate situation), but it the end we can return a singular value. To solve this problem, we combine the results obtained for each service flavour using the star (*) operation described in the following table.

*	OK	WARNING	CRITICAL
OK	OK	WARNING	CRITICAL
WARNING	WARNING	WARNING	CRITICAL.
CRITICAL	CRITICAL	CRITICAL	CRITICAL

Table 2: Return codes composition (*) operation

4.3 Probe Configuration

The probes are configured according to the NCG specifications mentioned in Section 2.1.2. Here are some configuration examples:

```
$SERVICE_TEMPL->{30}->{path} = $NCG::NCG_PROBES_PATH_GRIDMON;
$SERVICE_TEMPL->{30}->{interval} = 30;
$SERVICE_TEMPL->{30}->{timeout} = 30;
```

```
$SERVICE_TEMPL->{30}->{retryInterval} = 5;
$SERVICE_TEMPL->{30}->{maxCheckAttempts} = 3;

$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{native} = 'Nagios';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{probe} = 'ch.cern.sam/MrsCheckDBInserts';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{metricset} = 'nagios';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{config} = {%{$SERVICE_TEMPL->{30}}};
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{flags}->{NOHOSTNAME} = 1;
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{flags}->{PNP} = 1;
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{parameter}->{'--warning'} = '10000';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{parameter}->{'--critical'} = '5000';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{parameter}->{'--host_url'} =
    'http://grid-monitoring.cern.ch/';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInserts'}->{parameter}->{'--web_service_path'}
    = 'myegi/sam-pi/metricstore_current_load_per_hour?';

$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{native} = 'Nagios';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{probe}
    = 'ch.cern.sam/MrsCheckDBInsertsDetailed';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{metricset} = 'nagios';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{config} = {%{$SERVICE_TEMPL->{30}}};
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{flags}->{NOHOSTNAME} = 1;
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{flags}->{PNP} = 1;
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{parameter}->{'--warning'} = '\0';
    # 0 threshold must be escaped
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{parameter}->{'--critical'} = '\0';
    # 0 threshold must be escaped
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{parameter}->{'--host_url'}
    = 'http://grid-monitoring.cern.ch/';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{parameter}->{'--web_service_path'}
    = 'myegi/sam-pi/metricstore_current_load_per_hour_detailed?profile_name=';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{parameter}->{'--hours'} = '2';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{attribute}->{NGI_NAME} = "--ngi";
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{attribute}->{PROFILE_NAME} = '--profile';
$WLCG_SERVICE->{'ch.cern.sam.MrsCheckDBInsertsDetailed'}->{attribute}->{SERVICE_FLAVOUR_NAME}
    = '--service_flavour';

$WLCG_NODETYPE->{opsmonitor}->{'GridMon'} = ['ch.cern.sam.MrsCheckDBInserts',
'ch.cern.sam.MrsCheckDBInsertsDetailed',];
```

Some observations are in order. $SERVICE_TEMPL is used as a template to provide configuration for multiple Nagios probes. It provides certain common configuration parameters. Passing "0" as an integer argument poses a problem since in Perl it is considered a "null" or "false" value. Therefore, "0" must be escaped as can be seen in the previous code. Some arguments can be passed via a static file rule (Section 2.1.2) by using the {*attribute*} key instead of the {*parameter*}. In this case, the given argument can be attributed only to a certain host or a certain service (group of probes) in *ncg.localdb*. Of course, these parameter passing methods can be combined.

```
HOST_ATTRIBUTE!grid-monitoring.cern.ch!NGI_NAME!"CERN,NGI_IT,NGI_NL"
HOST_ATTRIBUTE!grid-monitoring-preprod.cern.ch!PROFILE_NAME!"WLCG_CREAM_LCGCE_CRITICAL"
SERVICE_ATTRIBUTE!GridMon!SERVICE_FLAVOUR_NAME!"SRMv2,SITE-BDII"
```

5 Conclusion

The new tables created in the Central Data Warehouse (METRICSTORE_CURR_LOAD_H, METRICSTORE_CURR_LOAD_M and METRICSTORE_CURR_LOAD_H_DET) will provide a better overview of the hourly and minutely metric data output. This overview will enable better control and monitoring over the loading of metric results into the Metric Results Store. Already the two new probes *MrsCheckDBInserts* and *MrsCheckDBInsertsDetailed*, which make use of the data stored in these tables, constitute an initial monitoring subsystem that alerts the human supervisors in case of abnormal data flow. Moreover, the data gathered permits better control over the data flow in case of unusual situations by delaying SAM_MS_LOADDATA.

The purging mechanism is another welcome addition to the MRS database. It ensures automatic cleaning of stale data and it has determined the creation of new indexes that permit faster deletion. Least but not last, the modifications brought to the procedures that load data into METRICDATA_SPOOL will produce a better data filtering by not allowing the insertion of metrics that might be considered too old.

References

[1] CERN - The European Organisation for Nuclear Research
http://public.web.cern.ch/public/Welcome.html

[2] Science and Technology Facilities Council - The Large Hadron Collider
http://www.lhc.ac.uk/default.aspx

[3] Worldwide LHC Computing Grid Overview
https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03_Documents/4_Presentations/Slides/2011-list/Markus%20Schulz%20_OpenLabWLCG-2011.pdf

[4] Worlwide LHC Computing Grid
http://lcg.web.cern.ch/LCG/public/

[5] Nagios Core 3.x Documentation
http://nagios.sourceforge.net/docs/3_0/toc.html

[6] Nagios Plug-in Development Guidelines
http://nagiosplug.sourceforge.net/developer-guidelines.html

[7] Nagios::Plugin
http://search.cpan.org/dist/Nagios-Plugin/lib/Nagios/Plugin.pm

[8] Monitoring in GT (Grid Technologies)
http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=0&materialId=slides&confId=127638

[9] Physics Computing at CERN
https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03_Documents/4_Presentations/Slides/2011-list/H.Meinhard-PhysicsComputing.pdf

[10] CERN - IT Department - Grid Technology
https://it-dep.web.cern.ch/it-dep/gt/

[11] SAM Public - Confluence
https://tomtools.cern.ch/confluence/display/SAMWEB/Home

[12] Probes Development Policy - SAM Documentation
https://tomtools.cern.ch/confluence/display/SAMDOC/Probes+Development+Policy

[13] Regional Grid Monitoring: Introduction and Database Components
http://indico.cern.ch/contributionDisplay.py?contribId=264&confId=55893

[14] White Areas - Monitoring of Grid Services in WLCG
http://indico.cern.ch/conferenceDisplay.py?confId=88718

[15] gLite 3.1 User Guide
https://espace.cern.ch/WLCG-document-repository/Technical%20Documents/gLite-3-UserGuide.pdf