CDF/???
October 7, 2008
Version 0.1

CAF Monitoring

Federica Moscato, Doug Benjamin, Hans Wenzel

Fermilab

Abstract

We describe the current CAF Monitoring system and discuss its strength and limitations. We propose a new monitoring system which uses a central relational database to collect the monitoring information. A Web based framework connects to the data base and displays both static views as well as allows for interactive queries and creation of graphs and charts on the fly.

Contents

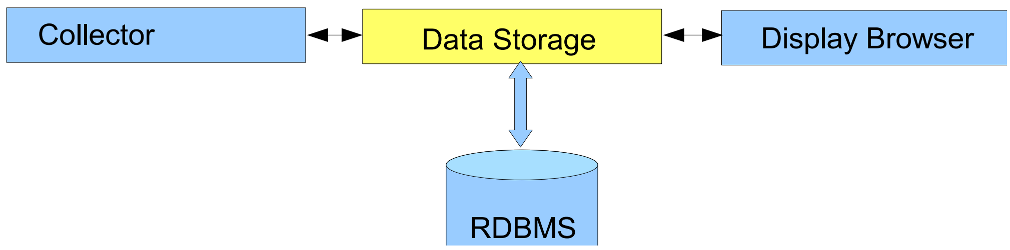

Figure 1: *Basic Components of a Monitoring framework.*

1 Introduction

Figure 1 shows the basic components of a Monitoring system.

- **Collector:** are processes/agents that collect the monitoring data (.e.g. in case of condor the quill demons.). The information is either pushed to the Data storage or pulled. The collectors must be robust and lightweight (if they don't run we don't get the information) and should have very little impact on the processes they are monitoring.

- **Data storage:** way to store the monitoring data in the most convenient way. In this article (see section 5) we argue that a relational database is the most natural choice for the data storage.)

- **Display Browser:** this is the process making the data available on the web. There is a large number of Modern web application frameworks [2] (e.g. ruby on rails, django ..) that use data bases as their back ends. The display and Data storage components should be clearly separated. All data we want to monitor should be available in the Data storage (db). The display should have no access to the Collectors.

2 The current system

The current system is shown in Figure 2 it is composed of a central sever, running on the head node of the condor pool, and a monitoring client. The server consists of 3 demons: monitor, state_monitor and xml_monitor. The monitor provides the communication between the user and his jobs; the xml_monitor collects the data and provides the communications with the client system, while the state_monitor produces the cache system. The client calls the monitor server and takes the data to store it in python pk and RRD (round robin database) files. Creation of this data file is obtained via cron jobs in the background (comment: not very robust since cron jobs might fail

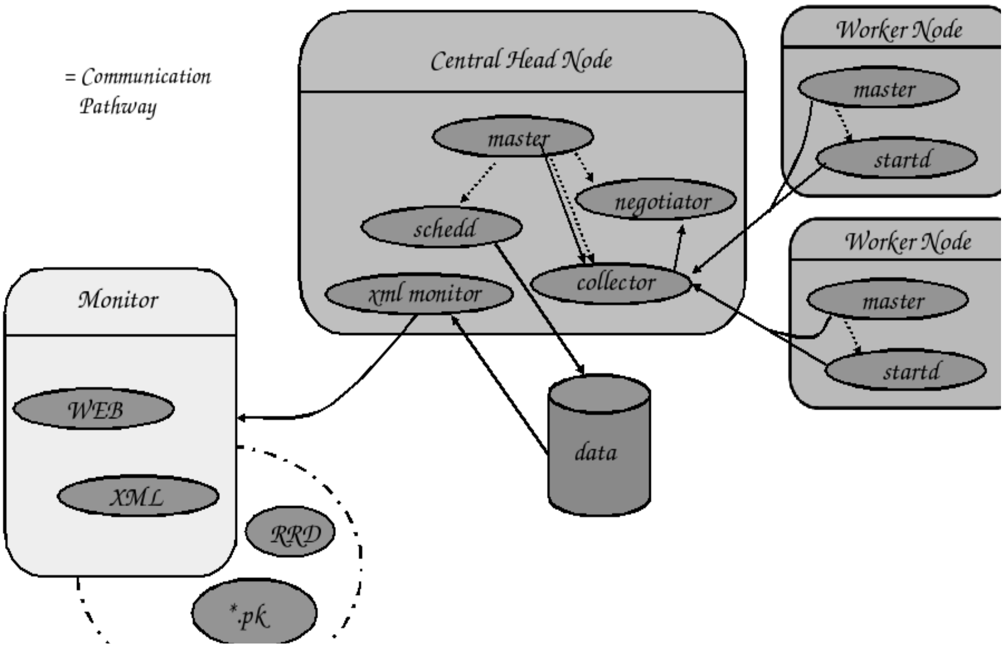

Figure 2: *Components of the current Monitoring framework.*

to start when the system is very busy) To get appropriate key attributes into the .pk file, the current model needs to parse many Condor and CAF log files. (not very robust as anyone knows who relied on parsing log files or the output of processes). Currently this causes many failures and it takes a lot of manpower fixing it. The user information can be accessed via the following web page [16], which provides various static views of the system and graphs prepared with rrdtools. The various views are

There is a document on progress describing the current system [4].

3 What triggers the need for updates now

The main trigger is the need to be able to operate with less man power and therefore need to automate and simplify operation where possible. It is planned that the main data tasks will migrate from dedicated resources to the general purpose centrally managed fermigrid. CDF's main data processing tasks are:

- **Production:** reconstruction of the raw data as recorded by the detector to obtain high level physics objects like jets, tracks etc.
- **Calibration:** Calculates various calibration constants and populates the calibration database. This calibration constants are then later used in production.

- **ntuple making:** uses the output of the production jobs and creates ntuples for the various physics groups to be used in physics analysis.

Compared to MonteCarlo/Simulation production all the tasks above are both CPU and IO intensive and the processing of this tasks will remain here at Fermilab. So monitoring the CDF jobs here at Fermilab must have the highest priority. Also CDF plans to use glideinWMS [3] for OSG (like FermiGrid) sites. So the monitor needs to keep track of glidein and VO classads etc..

There are dedicated and opportunistic computing resources available at various Grid sites using either LCG or OSG middle-ware. The monitoring framework should provide an overview and monitor all computing resources world wide available to the CDF VO. This can be done by collecting the information locally on the remote site and then synchronize the remote data bases with central CDF database.

The views presented on the web by the current monitoring system are views one would expect for a dedicated cluster. But since the model is changing and dedicated resources will be replaced by quotas (granted to the CDF VO) on general purpose grid sites (mainly FERMIGRID) not operated by the experiment. So the current views need to be replaced by VO based views that reflect the resources taken up by the CDF VO on this site. For example we need to know: how many of our allocated slots we are using, how many glideins are running and how many of them are actually running jobs, how efficiently our jobs make use of the resources etc. . Also if we want to use opportunistic resources on a site for e.g. MonteCarlo/Simulation jobs we need to know what metric is used to calculate the efficiency to make sure we don't get penalized for low efficiency and therefore loose the resources. We might also want to monitor other resources like space used in the storage element of the grid site, transfer rates of data out and into the site etc.

4 Requirements for a monitoring system

This list of requirements was presented at the CDF off-line operations meeting.

- conserve the current functionality After review of current functionality decide what features to maintain and may be what features can be dropped. E.g. monitoring the status of all Virtual machines (VM's) on a cluster might only be of interest for the operator of the site.
- adhere to standards (GRAM ...) if possible.
- whatever we monitor has to return sensible well defined exit status error codes (see e.g. [1])
- system should allow scoring and evaluation of efficiencies.
- Robustness

- reduce load on the head nodes
- web-based
- simplify configuration
- Allow specific queries e.g. use of a database will allow for specific queries allowing to improve the system e.g. want to know
  - job failures/successes by user
  - failures/success by Error return code
  - failures/success by submission site
  - failures/success by worker node
  - glidein failures
  - user: what are my jobs doing
  - MC data what is produced and where – help in accounting, justify resources
  - ... anything you can think of.
- allow for specific views (admin,user, summary) – certificate based authentication.
- allow to monitor multiple sites (LCG and OSG) CNAF, NAMCAF ,...... This could be done by collecting data into a local site specific data base which then gets synchronized with a central database at Fermilab.
- Additional functionality that isn't currently available

5 Planned Data Storage Solution

In this section we argue that the best replacement for the data storage solution is a relational database. In section 5.1 we collect the arguments for such a solution. In section 9.2.1 we describe the information that's kept in the current pk files how a possible translation into database tables could look like.

5.1 A relational database to store monitoring information

Using a database is much more appropriate than storing the information in various text log files, python pk files and RRD round robin databases as it is done now. Collecting the data by scanning log files or the output of e.g. condor commands is not very robust and prone to errors. Modern monitoring tools like Zabbix [17] (hardware and process monitoring), ARDA dashboard (VO based monitoring) use data bases as backbone. Modern web application frameworks (e.g. ruby on rails, ..) use data bases as their back ends. The condor scheduling system now provides Quill [6][7] which allows to store the status and history of the condor system in a relational data base.

Using a central relational database, one provides fast and flexible access to operational and history data where the data collectors can come from a variety of sources and therefore allow to consolidate and homogenize the source of status and monitoring data. SQL provides a common interface to the monitoring system. All access is reduced to SQL queries which can be local or remotely. Many frameworks in different programming languages exist to access the data in a database and to make this information available via the web.

- more robust
- the data source for the monitoring is consolidated and browse-able (SQL queries).
- allows for flexible queries besides static views.
- Many web based tools and software frameworks are available to access and visualize (making graphs on the fly) the data. (e.g.: the data source in netbeans)
- Data from different monitoring sources can be cross referenced and compared. (Zabbix, Condor (Quill), CAF...)
- After the data is collected and stored in the data base the monitoring will cause no load on the system: all communication is done with the (separate) data base server. In general services can and should be distributed over many servers (db, webserver, submitter...).
- Tools exist to backup the data, condense (vacuum) older data, synchronize databases etc.
- Database engines provide the infrastructure to control who and from where the database can be accessed. So making the monitoring framework secure only requires to configure the database access correctly.
- Database maintains referential integrity of the data.
- allows to automate specific tasks like generating reports, re-indexing etc.

The database we chose is PostgreSQL a free advanced open source database supporting all the features (transactions, vacuuming, foreign keys ..) we were looking for. There are no licence fees and the software is available for Fermi Linux. Detailed instruction exist how to set up Quill with PostgreSQL. The current installations of Quill here at Fermilab use PostgreSQL.

5.2 The current Data Storage and how to migrate to a DB

The current data that is stored in python dictionaries. An overview of the current monitoring system and all the data collected for caf monitoring purposes can be found in [4]. In this section we restrict ourselves to the data collected for completed jobs. We list the data fields in the python dictionaries, the log files that were parsed to obtain the data and then we describe how this data can be stored in a database. To demonstrate that it is easy to transfer the collected data into a data base we wrote python scripts that take the current python pk files and store the data in a postgres data base

5.2.1 JobID

Each caf job can have multiple sections as controlled by the meta-scheduler for Condor DAGMan (Directed Acyclic Graph Manager). The collection of sections is identified by a JobId dictionary. Below we list the fields of the JobID dictionary and the meaning of this fields. Table 1 lists the fields and the log files that have to be parsed to get this information. Finally the listing 5.2.1 shows how a database schema could look like.

JobId: dictionary containing JobId information

- schedd_name - None if the default is used
- schedd_node - Node on which the schedd is running
- group - CAF group (like "short")
- user - CAF user (like "sfiligoi")
- user_domain - domain used by the schedd (usually == schedd_node)
- caf_submit - when the caf submitted the job
- maxtime - Real time limit for the job (like 7200)
- cmdline - Submitter command line (like "./myjob.sh $")
- outurl - Where the output goes
- email - Email (line sfiligoi@fnal.gov)
- dhmode - SAM, DFC, rootd, GenMC, None, Unknown
- iomapfile - IOMap file used by mailer
- iomonfile - IOMon file
- logfile - User output file name

- errfile - User error file name
- prio - priority (between -20 and +20, the higher the better)
- optional:
  - acctuser - Accounting user (like "group_italy.sfiligoi"), if not present, use user as acctuser
  - dagman_id - if present, the dagman for jid is still running
  - tar_size - the size of the tar ball (should be present when running)
  - end_time - when dagman ended, present if dagman_id is not
  - sam - dictionary of dataset,project,station,group (only if dhmode==SAM)

Table 1: *JID dictionary fields and the log files parsed to obtain the information*

Field	Log parsed
dagman_id	dagman.log
schedd_name schedd_node user_domain cmdline outurl email dhmode iomapfile iomonfile logfile errfile	job.Descriptor
Group User	job.Descriptor (parsing the directory name)
caf_submit tar_size end_time Sam	job.ClusterId /stage/_job_in_.tgz job.dagman.dagman.out sam.ClassAd

Program 1 Schema replacing the JID python dictionaries.

```
CREATE TABLE JID
(
id              integer  NOT NULL,
dhmode          varchar(100),
group_name      varchar(10),
schedd_node     varchar(20),
prio            integer,
iomonfile       varchar(100),
errfile         varchar(100),
user_domain     varchar(100),
cmdline         varchar(300),
end_time        numeric(38),
schedd_name     varchar(20),
iomapfile       varchar(100),
caf_submit      numeric(38),
acctuser        varchar(100),
maxtime         numeric(38),
logfile         varchar(100),
email           varchar(100),
outurl          varchar(100),
user_name       varchar(100),
Primary Key (id)
);
```

5.2.2 Section_Summary and Sections

As mentioned before each caf job can have multiple sections the collection of sections is identified by a JobId dictionary. Below we list the fields of the JobID dictionary and the meaning of this fields. Table 2 lists the fields and the log files that have to be parsed to get the information for each section. Finally the listing 5.2.2 shows how a database schema could look like.
Key_section_summary: dictionary containing some global summary of a caf job.

- total_main_utime
- total_utime
- max_mem
- total_rtime
- datasets_str
- nr_sections
- main_max_mem
- main_exe

Sections: dictionary of Condor IDs where each id contains a dictionary of sections, each section contains:

- status - status of the job (any status_code (from DAGManParser))
- prio - priority (between -20 and +20, the higher the better)
- nr_starts - number of times the job was started
- nr_suspends - number of times the job was suspended
- nr_holds - number of times the job was held
- lost_secs - number of seconds lost due to preemptions and/or a removal
- optional: condor_id - Condor Cluster
- waited - number of seconds waited to be run (running, suspended or completed)
- elapsed - number of seconds used by the job (running, suspended or completed)
- cpu - number of CPU resources used by the job (running, suspended or completed), dictionary user,sys

- nodename - name of the node on which the job run
- nodeip - IP of the node on which the job run
- node - can be either value of nodename or nodeip
- vm - VM of the node on which the job run
- port - port of the node on which the job run
- group - CAF group
- submit_time - when the section was submitted to the condor queue
- start_time - when the section started
- end_time - when the section was completed
- abort_time - when the section was removed
- exit_data - data about a completed section dictionary of:
  - status - string, one of
  - Unknown - file not found or file empty
  - Error - Generic error
  - UserError - User error
  - Completed - Section completed
  - comment - error comment optional: (if status=="Completed")
  - retcode - return code
- cpu_used - dictionary of [Completed]
  - real - real time used (sec)
  - cpu - CPU time used (sec)
  - wait - how long it waited (sec)
- ioinfo - see CAFLogParser.CafExeLogParser
- process - tree of processes (completed only (optional), if want_process)
- pid - Process ID
- start_time - When the process stared (unix time)
- real_time - Number of real time seconds

- user_time - Number of seconds spent in user code
- system_time - Number of seconds spent in system code
- major_faults - Number of accesses to disk
- max_mem - Max known amount of memory used
- cmdline - Job command line

Table 2: *Sections dictionary fields and the log files parsed to obtain the information*

Field	Log parsed
status	job.log
nr_starts	
nr_suspends	
nr_holds	
lost_secs	
condor_id	
waited	
elapsed	
cpu	
nodeip	
node	
port	
submit_time	
start_time	
end_time	
Prio	section#.ClassAd
nodename	vms.log
vm	
exit_data{status, retcode,cpu_used,ioinfo}	CafExe_#.log
process {pid,start_time,real_time,user_time, system_time,major_faults,max_mem,cmdline}	section_#.procs

Program 2 Schemas replacing the Section_summary and Sections python dictionaries.

```
CREATE TABLE SECTION_SUMMARY (
id               integer  NOT NULL references JID(id),
total_main_utime numeric(38),
total_utime      numeric(38),
max_mem          integer,
total_rtime      numeric(38),
datasets_str     varchar(100),
nr_sections      integer,
main_max_mem     integer,
main_exe         varchar(100),
Primary Key (id)
);
CREATE TABLE SECTIONS (
id               integer  NOT NULL references JID(id),
secnr            integer  NOT NULL,
status           integer,
node             varchar(20),
nodename         varchar(20),
prio             integer,
start_time       numeric(38),
vm               integer,
elapsed          numeric(38),
condor_id        integer,
assign_time      numeric(38),
nr_holds         integer,
nr_starts        integer,
lost_secs        integer,
submit_time      numeric(38),
nr_suspends      integer,
port             integer,
waited           numeric(38),
Primary Key (id,secnr)
);
```

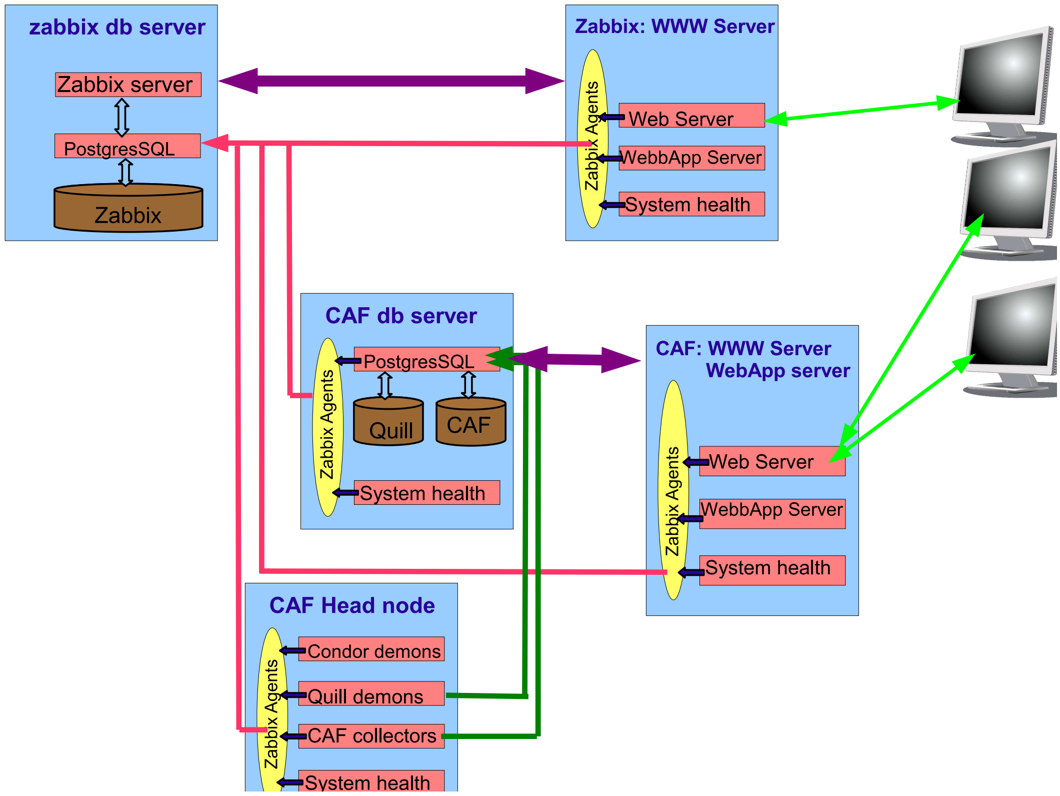

Figure 3: *One possible Monitoring scheme.*

6 The new monitoring system

Figure 3 shows the intended monitoring scheme. The Monitoring information that's collected is stored in a postgresql data base. The collection of condor specific data is done by quill. Everything else is in the CAF specific part. The data is displayed by a web application that uses the data base as back-end.

In a first development step the presentation on the web could be exactly the same pages that the users are accustomed to now, with the only difference that the information source is now a data base. Later on one wants to make use of the fact that the data is stored in a more convenient way by e.g. allowing specific queries, allowing to make graphs on the fly etc.

In this scheme zabbix [17] would be used to "monitor the monitor". The job of zabbix is to monitor the system health of the server nodes and to monitor that all necessary processes (web servers, data base servers, condor demons) are up and in a healthy state. Zabbix can be set up to raise automatic alarms when things go wrong. The information stored in the zabbix database can be used to correlate problems on the CAF with e.g. problems on the servers (out of memory, out of disk space etc.).

7 Future enhancements beyond Monitoring

Besides Monitoring we want features that improve operation. e.g. allow for data mining, reports, summaries and discovery of e.g. Bottlenecks of the system.

Grid sites score experiments/VO by efficiency and might reduce slots for VO's that don't perform very well. This can really hurt the experiment when trying to get access to opportunistic computing resources. E.g. when jobs are just waiting for IO and unnecessarily occupying slots this should trigger a warning/recommendations how to avoid the bottleneck and improve performance. We might think of blacklisting sites with known problems.

Grid sites allow for dedicated and opportunistic use of computing resources to VO's. Here it is important to consider that Grid sites might score experiments or Virtual organizations for how efficiently they make use of the provided computing resources. Experiments or virtual organizations that perform poorly might get penalized by getting less resources assigned to them. Therefore it is very important for the experiments to know exactly how their applications perform to ensure that the available computing resources are used efficiently to avoid loosing them.

The system should allow for automatic detection of error states, be able to issue alerts and be able to trigger 'automatic' action. Issue warnings that make sense and might include suggestions how to remedy the problem. (e.g. if jobs are held because a proxy expired, message should say that and point to instructions how to renew the proxy and release the jobs) (ErrorLogger, ErrorReceiver, MessageSystem) System should include a StateManager that checks that all demons and processes are running and reports the status to the web. (see CDF online Monitoring)

8 Status as of October 2008, Plan and deliverables for the next weeks

8.1 Accomplished so far

Here we summarize what has been accomplished until now:

- described and documented the current monitoring system.
- deployed quill on fcdftest010, documented the installation procedure and wrote the start scripts to start condor configured for CAF and Quill. Checked that the database information represents the status of the condor system correctly.
- deployed quillweb a php based web application which accesses the quill database to make the status of condor available from the web. Quillweb doesn't seem to be developed very actively, there are only a few very rudimentary graphs but the SQL queries are a good example how to access the quill database.

- quill contains all the information concerning the batch system but doesn't provide all the information that the current monitoring system provides and stores in python pk files (e.g. the dagman information described in is not kept in the quill database. We developed python scripts that read the current pk files and store the data in a relational database (PostgreSQL). Still need to replace the collectors to that we get the information into the database in real time.

- evaluated web applications, IDE's and frameworks to be used for the development of the CAF monitoring.

In summary we have documented the current system and provided pilot programs for the various components of a proposed CAF monitoring system. The next step is to develop a prototype that puts it all together.

8.2 Towards a prototype

We propose to deliver a prototype application that demonstrates the future caf monitoring framework. The development and deployment should take about 1 FTE for 2-3 weeks (end of October). This prototype should allow CDF to decide how to proceed with regards to a better monitoring solution.

Concerning the current monitoring that is in place we have two options: one possibility is to leave the current monitoring system as is and develop and deploy the new monitoring system in parallel, another possibility is to use the current web pages that the users are accustomed to but replace the back-end (data collectors and data storage). This would be completely transparent to the user. The second option would mean extra development effort. We assume that we proceed the development and deployment in parallel and leave the current monitor as is.

The prototype application would consist of:

- **Collectors:** quill to store the condor information in a relational database. In addition we need CAF specific collectors which write information into the database not available in quill. Quill is already deployed but we need to add the collectors for the caf specific part which needs probably about a day.

- **Data Storage:** Here the choice is a PostgreSQL data base with quill schema for the condor specific part in addition the schema described in section 9.2.1 is used to describe the CAF specific database tables for the completed jobs. We don't have a database schema yet for the current jobs and have no procedure yet how the current jobs migrate to the completed database table.

- **Display Browser** Use a web-application to connect to the database and make it available on the web. For the prototype we propose to use a web application based on Java server Pages: JSP [21], the open source Apache Tomcat web application server [22] and jfreechart [25] to display the results. As IDE and development environment we propose netbeans [23] which nicely integrates all the components for the display application. Netbeans is an open source product it provides the means to connect to the database, the tomcat webserver is build in and web application are nicely integrated. There are several examples and tutorials available describing how to connect to a database using a web-application. An alternative IDE is eclipse [24] but we don't have experience with it. For the prototype we would take example queries from quillweb implement them in java and make graphs using jfreechart. This should just serve as a prove of principal displaying a few graphs and status webpages.

8.3 Things we need to develop the prototype

- Test CAF system and server for the database: We have fcdftest010 with the caf software, postgresql and quill properly configured.
- CVS repository to develop the Webbapplication. (no yet)
- CVS repository for set up scripts, condor configuration files, sql schema definitions, python script to migrate the current historical data to a database etc. (not yet)
- PC to host the web application. Currently we use nara (hans's laptop) but we need to migrate to a different machine (fcdftest10?)

9 Tasks

9.1 Document and review the current system.

This is done see section 2 and [4].

9.2 Replace collectors

The initial deployment should be completely transparent to the CAF users while improving the system. This is achieved by introducing quill to collect the condor specific information. There are several installations of quill here at fermilab. Quill doesn't collect the complete information therefore additional CAF specific collectors and database schemas will have to be provided see task 9.2.3

9.2.1 Evaluate Quill (condor specific).

Quill [6] is an optional component of Condor that maintains a mirror of Condor operational data in a relational database. Quill is maintained and developed by the condor team. The database schema is well described in [7]. This tasks consists of deploying quill on a condor system and document the configuration and installation procedure. (done see [5]). The CAF condor setup is not standard, it involves multiple condor schedd demons (a master, scheeds for the glideins, and schedds for the regular jobs). Each schedd is started separately by a script and to be monitored by quill we need to start a separate quill demon for each of the schedds. Since quill is condor specific it can be only part of general monitoring solution but it provides us with the means to collect all the condor information (with the exception of the information contained in dagman for this see section) and puts it into the data base. So the use of condor commands and the scanning of condor log files is replaced by sql queries.

9.2.2 Deploy Quillweb

Quillweb is a php based web application [8] which allows access to the information stored in the database via a web application.

Quillweb will provide an example to query the quill database and obtain the desired information. (done see [9])

9.2.3 Create database schema

The task is to analyze the data structure of the current python pk files and map this structure to database tables. Some of the information is collected by executing condor commands. Since condor quill already collects the data of the condor processes in a database one should determine what part of the data collection can be left to quill and could be replaced to references within the database. This is done for the completed jobs but might need tweaks and requires advice from people with database experience. (Dennis Box?) In addition we need to know how to deal with jobs currently running and develop a scheme how jobs are transfered from running to completed.

9.2.4 Replace pk files by db

The task is to write a converter that reads the pk files for completed jobs and populates the database with historical data. This data base can then be used to develop the web applications. (this has been done as described in section 5).

9.2.5 Review and optimize database layout

The current implementation of the database as described in 5 might not be ideal and might not be complete. Help from database experts would be needed to optimize the layout.

9.2.6 Develop a complete CAF specific database schema?

The solution to use quill and some additional CAF specific database tables is condor specific and doesn't apply to e.g. LCG sites.

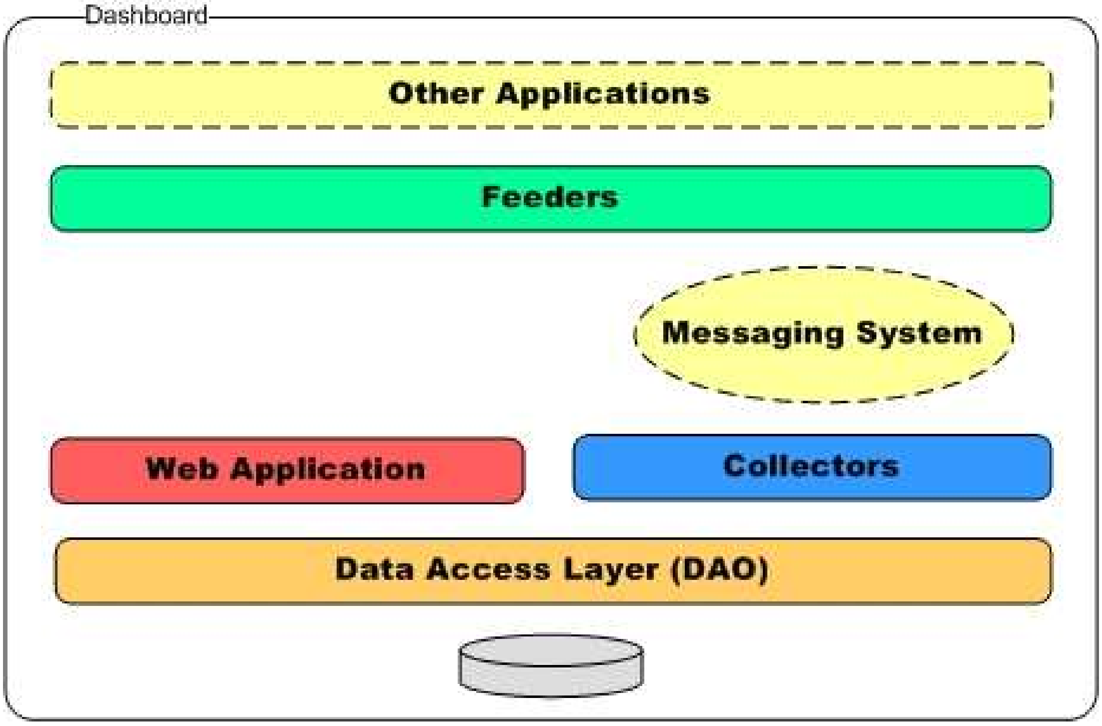

Figure 4: *ARDA dashboard design.*

9.3 Evaluate available solutions

There are ZABBIX[17], Dashboard [18], Quill[6] (Quillweb)

9.3.1 How to score a system.

- availability: (license ...)
- modularity
- extendibility (plug ins...)
- how easy to install (packaging) and maintain, Is the project alive. and who is developing it..
- how easy is it to describe the system you want to monitor
- support
- meets requirements or is easily adaptable to meet them

9.3.2 ARDA Dashboard

The ARDA Dashboard is a VO based monitoring framework that is used by all LHC experiments. It is supported by grid collaborations and the project is well documented. To see the dashboard at work have a look at the following web page[18]. The developers guide and the savannah project page can be found at [19] and [20] respectively. Dashboard stores the monitoring information in a ORACLE data base, the software is written in python using Apache 2 and PyDev for the web application.

To evaluate the ARDA dashboard, we installed their development environment. (eclipse, CVS, PyDev eclipse module....). The installation is quite involved and consists of:

- installation of extra tools: xslt, xpath, xsltpfoc, pychart,jw, pythonpgsql ....
- Configuration of Apache 2 and mod_python
- configuration of postgres/oracle database adding the arda schema, configuration of the web interface to the postgres data base. unfortunately implementations use oracle (commercial software)

Overall evaluate tools that might come in handy for this project. It did not take long to get the development version installed. It uses a nice development environment eclipse/netbeans. Below we list some of th components of the ARDA dashboard. The mayor components are also shown in Figure 4

- Dashboard Clients Scripts: pycurl, Command line tools (optparser + pycurl) Shell based: curl,
- Web Application Apache + mod_python Model View Controller (MVC) pattern multiple output formats: plain text, CSV, XML, XHTML GSI support using gridsite
- Agents collectors: RGMA, ICXML, BDII, stats generation, alert managers, Service Configurator pattern common configuration (XML file) and management: stop, start, status, list common monitoring mechanism
- Data Access Layer (DAO) interfaces available to different back-ends (Oracle, easy to add additional ones?) connection pooling
- Build and development environment based on python distutils (with several extensions) covers code validation, binaries and documentation generation, unit testing and reports automatic build for each of the release branches packaging uses RPMs APT repository available
- Release procedure three main branches: nightly, unstable, stable releases per component enforced versioning scheme (no manual versioning or tagging, all done via distutils command extensions)

9.3.3 ZABBIX

ZABBIX [17] is software that monitors numerous parameters of a network and the health and integrity of servers. ZABBIX uses a flexible notification mechanism that allows users to configure e-mail based alerts for an event. This allows a fast reaction to server problems. ZABBIX offers reporting and data visualisation features based on the stored data. This makes ZABBIX ideal for capacity planning. centralized monitoring system. All information (configuration, performance data) is stored in relational database Zabbix provides a web-based interface to the data. We found the installation fairly straightforward (see [10]). Overall ZABBIX seems to be a nice tool to replace other tools like ganglia [15] to monitor the health of servers and NGOP [14] for process monitoring. Therefore we use it in the proposed new monitoring framework described in section 6.

9.4 Decide on Web-Application framework and Development environment

Nowadays there are many powerful frameworks and development environments available (see e.g. [2]). One should decide early which one to use. See the proposed prototype solution described in section 8.2 for the proposed solution.

9.5 Configuration management

Identify and document the configuration files and changes that have to be made to this files. This files should be included in backups and should be added to Cluster management tools like Rocks or cfengine so that they can easily been restored or deployed to a new system.

9.6 Installation Procedure

Installation should be fairly automatic and basically should involve only running a script that installs and configures the software and all necessary packages that it depends on. (e.g. Apache, PostGres ....) A good example is e.g. the zabbix installation on ubuntu linux:

- installs software packages (....)
- configures the database (upload schemas, create necessary database accounts, allow for remote access by configuring data base and possible firewall policies)
- adds scripts so that all necessary services are started at boot time.

9.7 Monitoring the Monitor

Identify and document all processes/demons that need to be running for the system to work properly. This processes should be watched by an appropriate process monitoring tool such as zabbix or ngop.

References

[1] https://twiki.cern.ch/twiki/bin/view/CMS/JobExitCodes

[2] http://en.wikipedia.org/wiki/Model-view-controller#Pattern_description

[3] http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/

[4] Federica Moscato, Doug Benjamin *et al.*, CDF note in preparation, *The Central Analysis Farm Monitor Module Specifications*

[5] http://groups.google.com/group/cdf-offline-initiative/web/configuration-of-postgres-and-quill

[6] http://www.cs.wisc.edu/condor/quill_overview_07-18-2007.pdf

[7] http://www.cs.wisc.edu/condor/manual/v7.0/3_11Quill.html

[8] http://code.google.com/p/quillweb/

[9] http://nara.fnal.gov/quill/login.php

[10] http://groups.google.com/group/cdf-offline-initiative/web/configuring-zabbix-on-sl5-1?msg=ns

[11] http://www.nagios.org/

[12] http://cmssrv03.fnal.gov/condor/

[13] http://cdcvs.fnal.gov/cgi-bin/public-cvs/cvsweb-publnagiosic.cgi/condormon/?cvsroot=lpc

[14] http://www-isd.fnal.gov/ngop/

[15] http://ganglia.info/

[16] http://cdfcaf.fnal.gov/groupcaf/user.html

[17] http://www.zabbix.com/

[18] http://arda-dashboard.cern.ch/

[19] ARDA Dashboard Developers guide: http://dashb-build.cern.ch/build/nightly/doc/guides/common/html/dev/index.html

[20] ARDA Dashboard Savannah Project: http://savannah.cern.ch/groups/dashboard

[21] http://java.sun.com/products/jsp/

[22] http://tomcat.apache.org/

[23] http://www.netbeans.org/

[24] http://www.eclipse.org/

[25] http://www.jfree.org/jfreechart/