University of Florence
Degree in Computer Science
December 2005

Monitoring of a distributed computing system: the Grid AliEn@CERN

Meoni Marco - id. 02461991

Master Degree

Internal commitee:

Prof. Gregorio Landi, Doct. Carminati Federico

To my wife Valentina for her encouragement throughout the 8 years of my university studies, to my parents Giampiero and Vanna, to my relatives, to Latchezar and Federico for their review of the english version and analysis of the monitoring results.

Contents

Part I

Introduction

An unprecedented challenge

CERN, the European Organization for Nuclear Research [8], is the world's largest particle physics research centre. The aim of CERN is to explore what matter is made of and what forces hold it together. The laboratory, founded in 1954 and including now 20 member states, provides the equipments to chase this goal: particles accelerators, which accelerate particles to almost the speed of light, and detectors to make them visible.

The Large Hadron Collider (LHC) is the particle accelerator which is currently being built and will be commissioned for physics in 2007, probing deeper into matter than ever before. Its purpose is to collide, at high energy, beams of protons or beams of heavy nuclei.

ALICE (A Large Ion Collider Experiment) [4] is one of the four experiments at CERN LHC. The ALICE detector is dedicated to heavy-ion collisions to study the physics of strongly interacting matter at extreme energy densities, where is expected the formation of a quark-gluon plasma (QGP). QGP is supposed to have been present in the early universe approximately 10 microseconds after the BigBang.

High energy physics (HEP) experiments such as ALICE will produce massive amount of data (2-3PB/year). The processing and analysis of these data will necessitate unprecedented amount of computing and storage resources. Grid computing provides the answer to these needs: it has the design goal of solving problems too big for any single computer centre, providing the flexibility to use a distributed heterogeneous computing resources in a transparent way.

Document organization

These pages represent the result of a one-year collaboration with CERN of the author. The first four chapters of this document provide a general overview of the activities in ALICE Off-line and the fundamental concepts of Grid Monitoring, with focus on the MonALISA framework [19]. The remaining chapters, more experimental, are a detailed account of the pioneering work of implementing and extending a system offering a complete monitoring for tens of computing farms wordwide on one of the largest Grid in operation.

The first chapter introduces the concepts that will be used in the whole document: the concepts of distributed computing and Grid, the ALICE ex-

periment at CERN LHC, the ALICE Off-line group and, finally, the high level architecture of AliEn, the Grid system used in ALICE.

The second chapter describes the Grid Monitoring from a theoretical point of view, and in particular the components of the GMA architecture: Directory Service, Producer and Consumer. Follows and overview of R-GMA, a relational implementation of GMA, and of Jini, the Sun technology for distributed computing.

Chapter three runs over the distributed monitoring features of the MonALISA framework. It has been chosen by ALICE for its flexibility and easy interfacing to external tools.

Chapter four is the most important from the point of view of the work done. It analyzes on large scale the modification and extension of the MonALISA framework to suit the specific monitoring needs in the ALICE data production during 2004, called Physics Data Challenge 2004 (PDC'04). The tuning process has been realized step by step, in order to set up a functional system from the very beginning of PDC'04, with reduced functionality though. In particular, the most important services for the Grid system have been selected, and on top of them have been deployed the first monitoring agents, by using a data retrieval mechanism based on a Repository Web in place of GUI clients.

The definition of a set of strategic monitoring parameters has needed technical choices according to the type of measurement acquisition, according to whether it was done in a centralized or distributed manner. The MonALISA modularity has allowed to develop large portions of source code to add specific services to the basic framework, mostly about the presentation formats of the Web Repository and the storage of data on replicated servers.

Chapter five completes the previous one as far as the analysis of result is concerned. A so huge monitoring experience has produced thousands of analyzed parameters and millions of measured values. We have analyzed the overall data, depending on the strategic importance and provenance of the information, trying to localize eventual problems, bottlenecks or failures and proving how the monitoring activity provides an important support for the improving of AliEn and the success of PDC'04.

The last chapter, the sixth, summarizes the important results discussed in chapter five, describing lessons from a whole year of data production and looking at the possible next future scenarios.

The final appendix shows the relevant sections of code of custom modules

developed to extend the basic MonALISA framework. Additional source code to perform service tasks is included in the cd-rom provided with this document and is briefly described in the second part of the appendix.

Acknowledgments

I need to acknowledge my colleagues of the ALICE experiment to have used part of the experiment budget to fund my collaboration and permanence at CERN, the INFN section in Florence for the support provided. A particular acknowledgment to doctors F.Carminati, L.Betev and P.Buncic for the enthusiasm they transmitted during this work.

List of Figures

Part II

Grid general aspects

Chapter 1

ALICE's Grid: AliEn

1.1 The Grid concept overview

Grid computing consists of a coordinated use of large sets of different, geographically distributed resources in order to allow high-performance computation.

The first Grid efforts began in the 90s as projects whose aim was to link supercomputing sites and provide computational resources to a set of high performance applications. One of those projects has been particularly important because was the precursor of Globus [12], the de facto standard for developing Grid applications.

The second generation of Grids define a middleware (such as Globus) able to integrate software applications running in distributed heterogeneous environments. A layered architecture is in charge to manage allocation, monitoring and control of computational resources, handle file transfer protocols and authentication/security services, allow distribute access to hierarchical information (LDAP [18]) and provide application caching.

The more Grid solutions were developed, the more was felt the necessity to reuse existing components to build new Grid applications: the new needs pointed out the adoption of a service-oriented model and an increasing attention to metadata.

The standardization of the Web Services technology seemed to be well suited at the goal and so the Open Grid Services Architecture (OGSA) [22] has been proposed to support the creation, maintenance and application of services for any Virtual Organization (VO). OGSA is based on Web Services and integrates some Grid-specific requirements in order to implement *Grid Services*.

Basically a Grid Service defines standard mechanisms for naming and discovering service instances, provides locations transparently and supports integration with underling native platform facilities. Sharing resources such as computers, softwares and data must be highly controlled to define clearly what is shared, who is allowed to share, and the conditions under which sharing occurs.

1.2 The ALICE experiment at CERN's LHC

ALICE is one of the four LHC experiments at CERN. When the experiment start running, it will collect data at a rate up to 2PB per year and probably run for twenty years generating more than 10^9 data files per year in more than 50 locations worldwide.

The aim of ALICE experiment is to explore the primordial state of matter that existed in the first instants of our Universe, immediately after the initial hot Big Bang. At that time the matter passed through a state called Quark-Gluon Plasma (QGP) a very dense and hot mixture of quarks and gluons free to roam. Then the Universe expanded and cooled down in few microseconds producing protons and neutrons, with quarks bound together by gluons and both locked inside them.

QGP may exist at the heart of neutron stars where the density of matter is incredibly high. ALICE experiment will use LHC, the new CERN's particle accelerator that will increase the collision energy to 330 times the old SPS energy, to create QGP in the laboratory by head-on collisions of heavy nuclei. The collisions will heat and squeeze protons and neutrons trying to melt them back into QGP that will freeze out into (possibly new) particles.

Figure 1.1 shows the main phases of the ALICE's experiment functionig: the particles produced by the collisions will be recognized by the detector and analyzed with specialized software by the physicist involved in the experiment:

ALICE has chosen lead-lead collisions being one of the largest nuclei available with 208 protons and neutrons, but the experiment studies also proton-nucleus and proton-proton collisions, giving the physicist the possibility to study the different phases and transitions in which the matter has passed through since the creation of the Universe.

ALICE's detector will be the most advanced detector to collect and study data coming from heavy nuclei collisions. Picture 1.2 shows its main

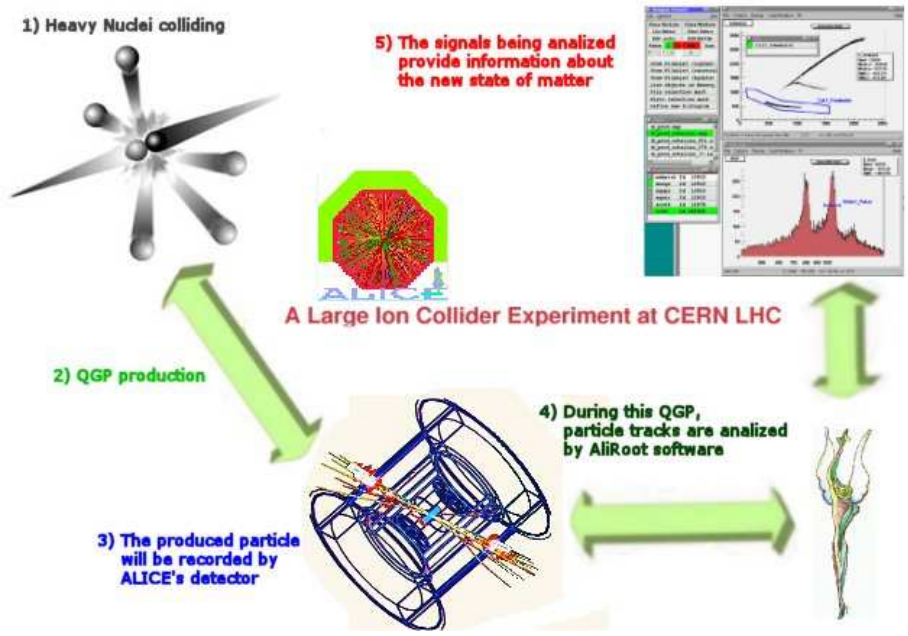


Figure 1.1: *ALICE experiment at CERN's LHC*

components that are briefly described here below:

- *Inner Tracking System*
- *High Momentum Particle Identification*
- *Dipole Magnet*
- *Time Projection Chamber*
- *Muon Spectrometer*
- *Time of Flight*
- *Photon Spectrometer*
- *Transition Radiator Detector*

1.3 ALICE Off-line

In 1998, the ALICE Off-line Project has started developing a simulation tool for the Technical Design Reports of the ALICE detector using the OO programming technique and C++ language. ROOT [24] has been chosen as

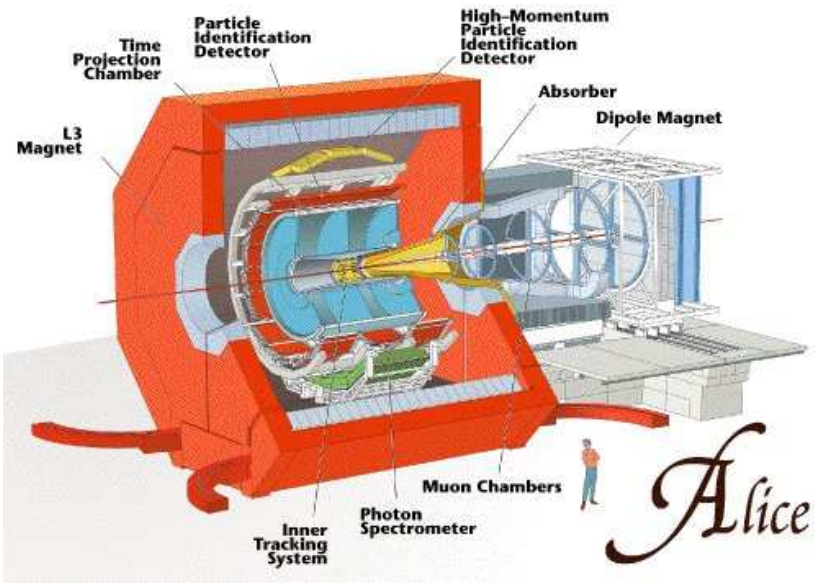


Figure 1.2: *ALICE's detector*

official framework offering a common set of features and tools for all domains shown in figure 1.3, and GEANT3 has been chosen as a simulation software (later GEANT4 and shortly FLUKA).

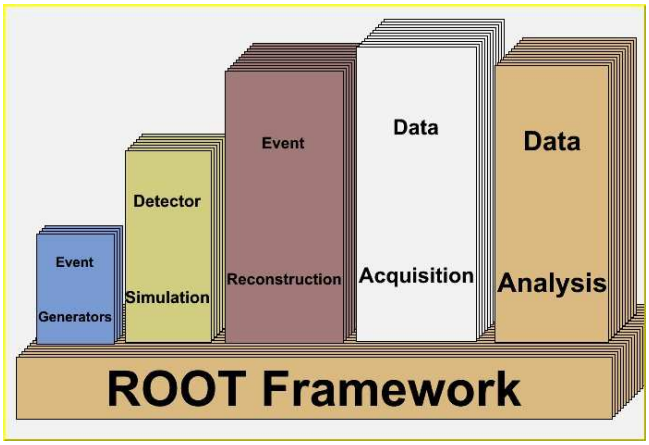


Figure 1.3: *ROOT architecture*

AliRoot is the name of the ALICE Off-line framework for simulation, reconstruction (with lots of optimized algorthims) and analysis. At the base of the choice of ROOT there are its complete set of data containers, C++ as a scripting language, a large amount of utility functions (math, random

numbers, multi-parametric fit etc...), a GUI toolkit, a documentation tool and a complete analysis framework. Moreover, a new package has been developed for building, browsing, tracking and visualizing a detector geometry, with the characteristic that the core functionality is independent from the AliRoot framework and the aim to use the same geometry for several purposes such as simulation, reconstruction or event display.

Since 1998 the ALICE experiment and the CERN/IT division have experienced several large-scale high troughput distributed computing exercises, called ALICE Data Challenges, to define the data acquisition, prototype the computing systems and test hardware and software components.

Monte-Carlo (MC) raw data production, shipment to CERN, reconstruction of the row data at all tier-1 sites and analysis using the AliRoot framework have been the tasks carried out during the three main phases of the Physics Data Challenge 2004 (PDC'04), with the aim to accomplish the all jobs on the Grid.

The distributed computing framework developed by ALICE is called AliEn [1] (Alice Environment): it provides a complete implementation of a Grid where handling a large number of files is required; distributed analysis is additionally achieved interfacing AliEn with PROOF, a part of the ROOT framework allowing parallel facility.

1.4 AliEn Architecture

AliEn is a distributed computing environment developed by the ALICE Offline Project offering to the ALICE user community a transparent access to wordwide distributed computing and storage resources. In 2004 has been used for a massive production of Monte-carlo data for detector and physics studies and for user job analysis.

AliEn interfaces to common Grid solutions and provides a native Grid environment based on a Web Services model. It has been built on top of a large number of Open Source components (Globus/GSI, OpenSSL, OpenLDAP, SOAPLite, MySQL, perl5) re-using their functionality, and the latest Internet standards for information authentication (PKI, SOAP). Once the AliEn components are installed and a Virtual Organization is configured, a number of Web Services at central and remote sites must be started to collaborate each others, in order to execute user jobs and constitute the AliEn Grid. Picture 1.4 shows the main web services.

The AliEn building blocks can be grouped in three different levels: low-

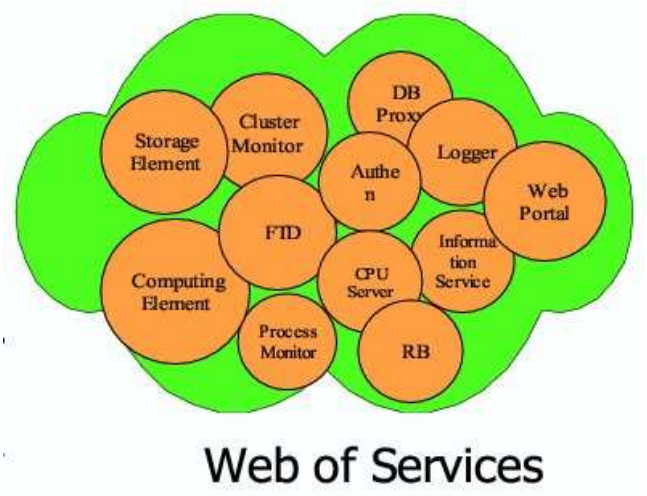


Figure 1.4: *AliEn services*

level external software components, AliEn core services and high-level user interfaces. Picture 1.5 shows the architecture.

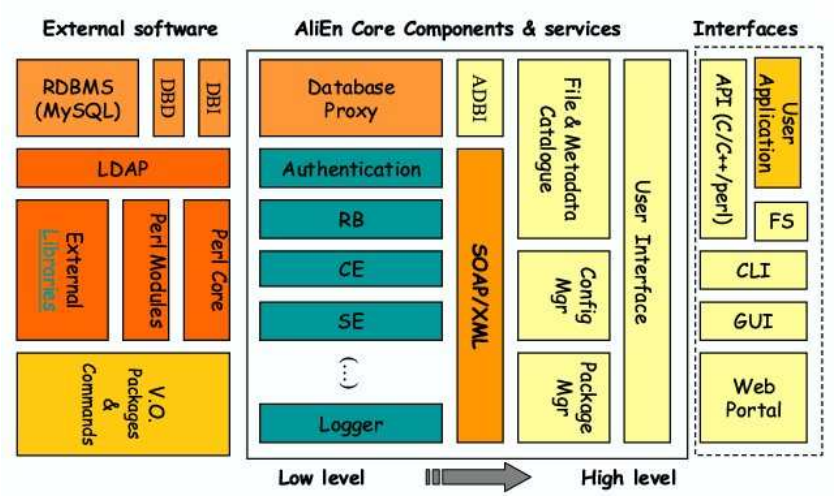


Figure 1.5: *AliEn architecture*

One of the first component developed in AliEn has been the *file catalogue*: it maps one or more physical file name (PFN) to a logical file name (LFN). It is up to the system to show the users only the LFNs and translate them into the closest and most appropriate PFN depending on the client location and capabilities. The catalogue provides an interface similar to a

UNIX filesystem and allows the user to store metadata information to additionally describe the files content. The implementation of the catalogue has done using a relational database accessed by several interface layers.

The catalogue capability to handle large distributed files is a basic need considering that it takes about 20 hours to simulate the complete detector response and the resulting output file for an event is up to 2GB.

The core service of the AliEn execution model is the *Broker*, whose aim is to receive a resource JDL (Job Description Language) describing resource's availability and check it against all the task JDLs scheduled in the system Master Queue. If it finds a match, the description of the selected task will be sent to the resource, otherwise the resource sleeps for a while and tries again later: this is basically the functioning of a Grid system based on a *pull-architecture* (as opposed to the push mechanism traditionally implemented in other Grid systems) that, as described briefly in the next section, is very flexible to interface even other Grids.

In fact, according to this implementation, the scheduling service does not need to know the status of all other resources in the system. The job description in the form JDL is simply stored in the Master Queue database waiting for Computing Elements to connect and to advertise their status and capabilities. This result is a robust and fault tolerant system where resources can come and go at any point in time. Given this weak coupling between resources and the system, it is possible to abstract and interface an entire foreign Grid as a large Computing and Storage Element.

AliEn's Master Queue, Broker and remote sites interactions will be discussed in details in chapter 4 to analize the monitoring system built on top of the system.

1.4.1 An example of foreign Grid: LCG

The LCG (LHC Computing Grid) project has deployed a worldwide computational grid service, integrating the capacity of scientific computing centres spread across Europe, America and Asia into a big virtual computing organisation.

As a proof of concept, AliEn has been interfaced to LCG as a real example of how a foreign Grid can interact with it: the LCG Resource Broker has seen by AliEn as a standard Computing Element (CE) thanks to a suited interface.

Chapter 2

Grid Monitoring

2.1 Introduction to Grid Monitoring: GMA

The ability to monitor distributed computing components is critical for enabling high-performance distributed computing. This monitoring feature is needed at a variety of levels, from the ability to determine server status, to being able to determine software availability and application progress, to logging performance data of CPUs, networks and storage devices in order to make predictions of behavior. As more application scientists make use of shared grids resources, the need for better monitoring will continue to expand.

Usually the reason to collect a substantial amount of monitoring data in large distributed systems such as computational and data Grids is to yield various services like:

- fault detection as entry-point of recovery mechanisms to figure out if a server is down and decide if restart the server itself or redirect service requests elsewhere;
- analysis to detect the source of performance problems;
- tuning to adjust the system and applications in case of detected problems;
- prediction to implement services that take monitoring data as input to a prediction model;
- scheduling to define which resources to assign to a job depending on the prediction model above.

At the moment have been developed several monitoring systems to gather performance data, in these papers we will talk about R-GMA [23] and will

go deeply inside MonALISA. Both of them are implementations of the Grid Monitoring Architecture (GMA) design developed by the Global Grid Forum Performance Working Group [14]. It addresses the characteristics of a Grid platform providing:

- a standard terminology;
- a description of the major components;
- a minimal specification of the required functionality;
- a list of critical characteristics to a proper functioning monitoring system.

2.1.1 GMA architecture

The main difference between a Grid monitoring system and a general monitoring system is that the former must easily scale across wide-area networks and include lots of heterogeneous resources: in fact, potentially, a Grid system could be built over thousands of resources at geographically distant sites and could be used by tens-of-thousands users at the same time. Naming and security features must be included in a Grid design as well; last but not least, is a set of common protocols for messaging, data management and filters.

Unlike program-produced data, monitoring information are characterized by a usually fixed and short lifetime utility because performance information usually age quickly and by the fact that they are updated more frequently than they are read. Moreover, it is often impossible to define the performance of a resource or an application component just using a single value: performance information should carry additional metrics quantifying the accuracy of the value, the lifetime and other parameters computed from the raw data.

Evidently the Grid monitoring system must satisfy certain requirements in order to not overload the underlying monitored system and to provide secure and scalable data-collecting mechanisms. In chapter four we will talk about MonALISA, the Grid monitoring system currently in use in ALICE; we will try to understand if and in which matter the system respects some GMA issues that any implementation should satisfy. Infact, they have proved by development experience as being very important to the success of monitoring systems. Moreover, we will analyze if it reliably delivers timely and accurately performance information without perturbing the underlying monitored system.

Here below the list of issues a GMA implementation should provide:

- **Fault tolerance**: the components of a monitoring system must be able to tolerate and recover from failures, reconnect and synchronize themselves automatically because in a large-scale distributed system failures will occur (monitoring servers, directory servers, network...);
- **Low latency**: as we said before, performance information have usually a short lifetime utility therefore the transmission time between where the data is measured and where it is needed must have a low latency;
- **High data rate**: the monitoring system should be able to handle performance data even if they are generated at high rates; moreover, average and burst data should be specified in advance avoiding to overload the consumer;
- **Low overhead**: the measurement must not be intrusive and has to use an acceptable portion of the available resources otherwise what is measured would be mostly the introduced load itself; in other words, the system monitoring components must control their intrusiveness on the resources they monitor;
- **Security**: the monitoring system must provide access control policies for each owner of the performance information in order to preserve integrity: this services involve publishing of new performance data, subscriptions for event data and lookup features;
- **Scalability**: as previously stated, a Grid monitoring system could be tuned over thousands of resources, services and application and used by tens-of-thousands users at the same time. Therefore it is important that such system scales up in terms of measurements, transmission of information, security and reduced overload; public key-based X.509 identity certificates [27] presented through a secure protocol such as SSL (Secure Socket Layer) are the common solution to identify users.
- **Adaptation to performance changing**: performance data can be evidently used to determine if the shared Grid resources are performing fine or whether the current load will allow a specific application. The data management system cannot itself be rendered inaccessible by the very performance fluctuation it is trying to capture, but must use the data it gathers to control its own execution and resources depending on the dynamically changing conditions;
- **Distribute components**: a single centralized repository for information or control has two problems: it is a bottleneck because experience has shown it cannot handle the load generated by actively monitored resources at Grid scales, and represents a single point of failure for the entire system in case of host or network failure;

- **Event format efficiency**: the data format choice is a compromise between ease-of-use and compactness. ASCII text is the easiest and most portable format to describe and send data, but it is also the least compact; compressed formats fall at the opposite side.

In order to achieve this requirements, any monitoring system must gather data from the local system producing the lowest overhead possible and deliver the performance information with low latency.

To provide a precise control of the two features, data discovery and transfer are usually separated and metadata must be abstracted and kept in a universally accessible location called *directory service* or *lookup service* together with enough information to start up the communication between the data producer and the consumer. The architecture above can easily scale organizing the metadata in a way that the directory service itself may be distributed.

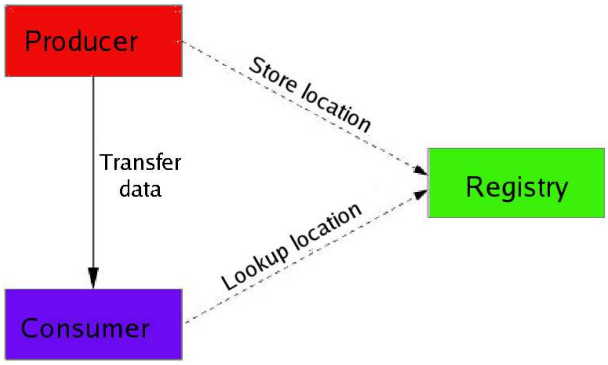


Figure 2.1: *GMA architecture*

Figure 2.1 shows the three main components of a Grid Monitoring Architecture: the Directory Service provides the functionality to publish and discover the performance information, the Producer makes data available and the Consumer receives data. Performance information are sent as timestamped events, typed collection of data with a specific structure defined by an event schema optionally stored in the Directory Service.

2.1.2 Directory Service

The Directory Service is in charge of providing the functionality to describe and discover performance data on the Grid. It stores information about producers and consumers: when they publish their existence within the Directory Service, they basically also describe the types of events they produce

or consume, together with information about the accepted protocols and security mechanisms.

The information supplied by a Directory Service allows other producers and consumers to find the types of event data that are currently available or accepted, the structures and the ways to gain access to the data themselves. It can optionally handles the event schema to provide directly the performance information structure.

The term *Directory Service* is not meant to imply a hierarchical service such as LDAP [18] because any lookup service could be used. Consumers use the Directory Service to discover producers of interest, and producers use the Directory Service to discover consumers of interest. At that point, control messages and data transfer occur directly between each consumer/producer pair without any further involvement of the Directory Service.

A Directory Service supports four basic functions:

1. *Add*: add an entry to the directory;
2. *Update*: modify an entry in the directory;
3. *Remove*: delete an entry from the directory;
4. *Search*: search for producer or consumer of interest, depending on some selection criteria. The client can specify if should be returned a single match: in case of multiple matches it can iterates through the results using a "get next" query-statement.

2.1.3 Producer

A Producer is by definition "any component that uses the producer interface to send events to a consumer": it can provide access control to the event data allowing different access to different classes of users. A Producer may have several interfaces, each acting independently and sending events. Since Grids generally have multiple organizations controlling the monitored resources, there may be different access policies, different measurements and different details depending on the kind of consumers.

Producers can provide additional services to the core GMA functionality such as event filtering, caching, alarm/message triggering according to specific thresholds or data ranges, intermediate processing of raw data requested by the consumer.

2.1.4 Consumer

A Consumer is by definition "any component that uses the consumer interface to receive events from a producer": it can have several interfaces, each acting independently and receiving events.

Basically there are 3 kind of interactions supported by the GMA architecture between producers and consumers:

- *publish/subscribe*: this communication is based on 3 phases. First of all the initiator (it can be either a producer or a consumer) contacts the server (if the initiator is a consumer, the server is a producer, and viceversa) to specify the events of interest. This usually happens via an exchange of control messages to negotiate the destination of the performance events, encryption mechanisms, buffer sizes, frequency of transmission and so forth. During the second stage, called *subscription*, the producer sends some test events and finally they terminate the process with few additional control messages. The pairs of functions supported by the producer and consumer respectively to handle this type of interaction are *accept/initiate subscribe*, *accept/initiate unsubscribe*;

- *query/response*: in this case the initiator must be a consumer. The first phase of the process sets up the transfer in the same manner of the previous interaction. Then the producer transfers all the performance events in a single *response*: this system works fine with a request/response protocol such as HTTP. The pair of functions supported by the producer and consumer respectively to handle this type of interaction is *accept/initiate query*;

- *notification*: it is a one-phase interaction started by a producer: it transfers all the performance events to a consumer in a single *notification*. The pair of functions supported by the producer and consumer respectively to handle this type of interaction is *notify/accept notification*.

It is possible to define different types of consumer, depending for example on the lifetime of the performance data and the kind of analysis they are used for.

An *Archiver* is a consumer that aggregates and stores data in long-term storage for later retrieval or analysis and can act as a GMA producer if the data is retrieved from the storage back-end. If the consumer has to collect monitoring data in real time for online analysis purposes (i.e. to plot a real time chart about cpu_load information) is called *RealTime Monitor*. Finally an *Overview Monitor* is a consumer that retrieves performance events

from several sources and computes derived information otherwise difficult to supply on the basis of data from only one producer (i.e. to trigger messages/alarms in case the number of assigned jobs by the central resource broker is greater than the number of jobs a remote farm can queue).

One of the most interesting benefits coming from the separation between data discovery and data transfer is that each GMA component can implement both the interfaces, consumer and producer, at the same time in order to become an intermediary that forwards, filters, caches, summarizes the performance events.

A consumer interface might also collect event data from different producers and compute derived event data type making them available to other consumer through a producer interface. This is the common case of archiver consumers: they usually store event data for later analysis performed by further consumers; moreover, the network traffic can be reduced if the compound component filters the event or is placed close to the data consumer.

2.2 R-GMA

The Relational Grid Monitoring Architecture (R-GMA) [23] is an implementation of GMA. In R-GMA, *"the information and monitoring system appears like one large relational database and can be queried as such"*, therefore the letter 'R' means a relational implementation of GMA.

Providing the capability to use the relational model in a Grid architecture doesn't mean that R-GMA is a distributed RDBMS. The key point is that a Producer can announce and publish performance events via SQL CREATE TABLE and INSERT statements and the Consumer can retrieve the needed information via an SQL SELECT statement.

R-GMA provides API in Java, C, C++ and Python allowing Grid Services and applications to interface with it; besides, a web browser to browse the status of a Grid is available. Figure 2.2 shows the main high-level components.

2.2.1 Functioning

R-GMA organizes the Grid resources information for a specific Virtual Organization (VO) in a so called *virtual database* that contains a set of *virtual tables*. R-GMA keeps a *schema* to keep name and structure of each virtual table, and a *registry* to keep the list of producers that publish *tuples*

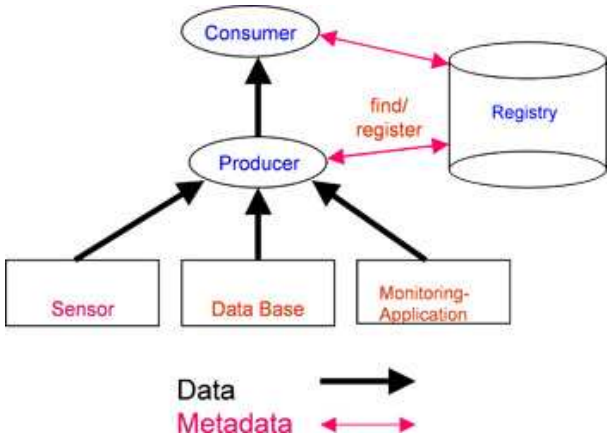


Figure 2.2: *R-GMA architecture*

(data rows) per each virtual table using SQL insert statements. Whenever a consumer runs an SQL query to retrieve performance information from the database, the registry start a *mediation* process to select the best producer(s) answering the query. At this point the consumer can connect directly to each producer. The database is called virtual because there is not any central repository holding the content of the tables.

R-GMA is based on the Web Service Architecture [25] and provides API for several languages (Java, Python, C, C++) to hide to SOAP interface to the user. The interaction between the user code and each Web Service operation is feasible thank to the methods provided by the API: each of these methods pack up the parameters into a SOAP message and sends it to the Web Service getting eventually back errors or exceptions and transparently handling any authentication required by the server. Direct interaction between user applications and R-GMA Web Services are also supported.

The R-GMA architecture defines 3 classes of producers, depending on where the tuples of data come from:

- *Primary*: the tuples generated by the user application are stored internally by the producer that answers consumer queries retrieving data from this storage. Each tuple published by a primary producer also carries a *timestamp* for history pourposes.
- *Secondary*: this service answers queries from its storage as well, but populates the storage itself querying the virtual tables and getting tuples from other producers;

- *On-demand*: these producers don't have internal storage and provide data by the user code.

A consumer executes a single SQL SELECT query within the virtual database. The query is initiated by the user code and handled by the consumer service via the *mediation* process explained above.

R-GMA has a Resource Framework based on WSRF [26] to protect itself from accumulation of redundant resources keeping track of the life-cycle of resources running on each server. This mechanism is hidden from the user.

2.3 Jini

This section explains the functioning of Jini Network technology [17], an on the cutting edge Sun solution grown from early work in Java to make distributed computing easier. This technology provides the classes and the necessary level of abstraction to implement a Grid monitoring framework based on lookup servers and dynamic discovery of distributed services according to the GMA architecture described above. MonALISA, the monitoring framework for AliEn, is fully based on Jini/Java and is described in the next chapter.

The problem that Jini tries to solve consists of how different applications, services or devices recognize each other existence without any knowledges about their names or locations. The central service in Jini is called *lookup*: it is the place where all the other services register and by which they are mediated.

Each Jini service can be any hardware device or a program: in the former case it must have its own processor with enough memory to run a Java Virtual Machine (JVM). Jini carries out the communication between producers and consumers via the Java Remote Method Invocation (RMI), a system that allows an object running in one JVM to invoke methods on an object running in another one.

Jini new services don't need any special knowledge to be added to the Jini network; they must register to a *lookup server*, that can be even unknown, via a *Discovery* protocol; once a lookup server has been found, the new service registers with it providing the interface it implements and eventual further attributes describing the service itself: this second step is called *Join*.

Discovery and Join are defined in the Sun specification as protocol and can be implemented using the sockets: anyhow Sun provides a class to easily handle the communication. Specifically, the Discovery can be addressed in three different ways:

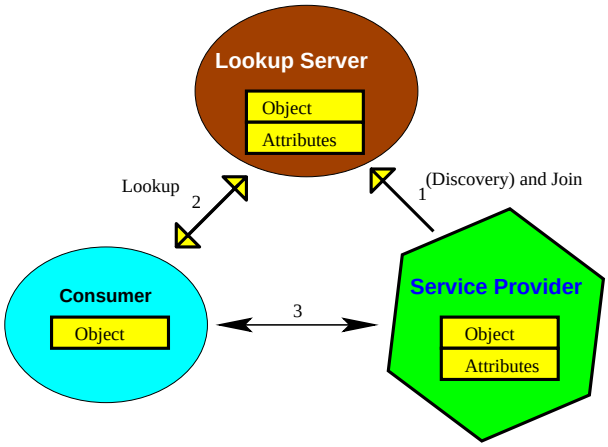


Figure 2.3: *Jini Components*

- the new service sends a UDP datagram packet into the local network to the IP address and port reserved for multicast;
- the new service addresses a lookup server directly via unicast, overpassing local network limits;
- the lookup server sends a multicast after being out of service for a while.

A service can belong to - and a lookup server responsible of - one or several groups. It must register with every lookup server handling a group to which the service shall belong.

A consumer looks for a service via the *Lookup* protocol: it specifies the Java interface or class for which it searches an implementation. The connection between the consumer and the lookup server is handled by a proxy loaded onto the local computer that can be obtained by applying the *Discovery* protocol.

Jini incorporates also a mechanism called *Leasing* to handle the aging of the services. In a distributed system, relations between services and their consumers are not as tight and reliable as in traditional local systems. In Jini, services are not registered permanently but only for a specific time during which their utilization is guaranteed. If the Leasing is not extended, the service expires and is removed from the lookup server via a distributed event mechanism of JavaBeans.

Chapter 3

MonALISA

3.1 Overview

MonALISA (Monitoring Agents in A Large Integrated Services Architecture) is a distributed computing oriented monitoring system. It provides extended functionality to monitor system and performance information on remote sites. It has incorporated dynamic protocols to ship this information to any kind of client, either web repositories handling data histories and providing advanced graphical interface, or GUI application that can be executed via Java Web Start [15] technology to monitor farms status. In addition, MonALISA provides API to send directly performance information to the agents from user custom applications, remote control facility and many other dynamic and self describing features.

The flexibility of the MonALISA framework has represented the main reason why it has been chosen. This characteristic has achieved easy adapting and tuning of the system to the needs of the ongoing ALICE's Data Production.

During the ALICE Physics Data Challenge 2004 and beginning of 2005 most of the MonALISA features have been successfully tested, tuned and improved. Lots of interesting feedbacks have risen up thank to the unprecedented opportunity to deeply monitor the behavior of an heavily stressed Grid system such as AliEn.

To fully understand the whole system, it is necessary to spend some pages describing MonALISA design, data collection engine, registration and discovery mechanisms and, finally, clients type.

3.2 Design

The MonaLISA system provides a distributed service for monitoring of complex systems. MonALISA is based on Java/JINI and Web Services technologies: picture 3.1 shows how each MonALISA server acts as a dynamic service system and provides the functionality to be discovered and used by any other services or clients that require such information.

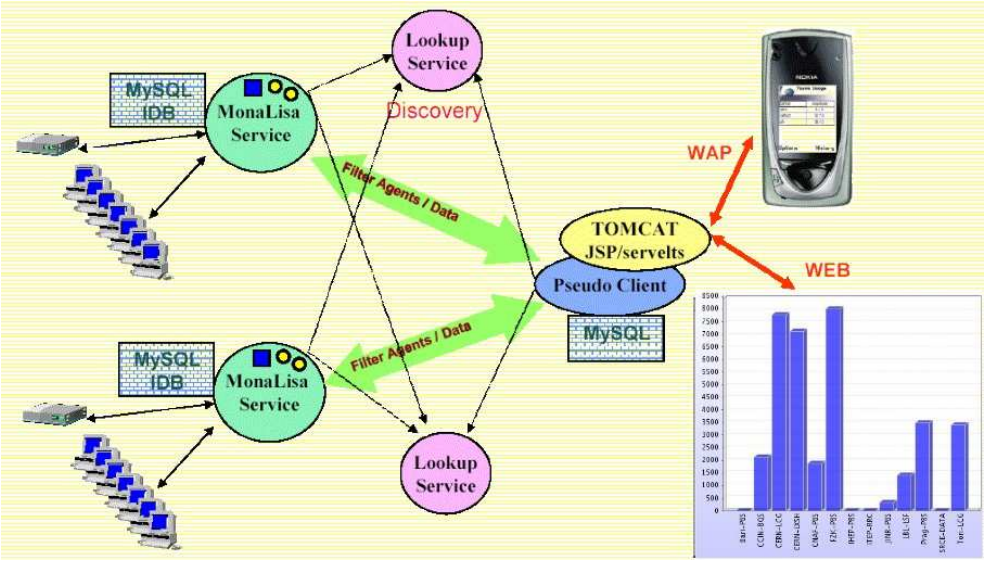


Figure 3.1: *MonALISA framework*

3.2.1 The Monitoring Service

As stated in the previous chapter, an essential part of managing a complex system, like the Grids, is a monitoring system that is able to track in real time the many site facilities, networks, and task in progress.

MonALISA is an ensemble of autonomous multi-threaded, agent-based subsystems which are registered as dynamic services and are able to collaborate and cooperate in performing a wide range of monitoring tasks in large scale distributed applications.

It is designed to easily integrate existing monitoring tools and procedures and to provide performance information in a dynamic, self describing way to any other services or clients. MonALISA services are organized in groups and this attribute is used for registration and discovery.

3.2.2 Data collection engine

Each MonALISA monitoring service can directly interact with sites, network links, routers, or interface with user custom scripts and existing monitoring tools (i.e. Ganglia [10]). The core of the service is based on a multi-threaded system used to execute the data collection modules (*Monitoring Modules*) in parallel, independently. If a monitoring module fails or hangs due to I/O errors, the other tasks are not delayed or interrupted, since they are executing in other, independent threads.

This schema allows to easily monitor a large number of heterogeneous nodes with different response times, and to handle monitored modules which are down or not responding, without affecting the other measurements.

Basically a Monitoring Module is a dynamic loadable unit, in general a simple class, which executes a procedure, script, program or SNMP request to collect a set of parameters (monitored values) by properly parsing the received output.

3.2.3 Registration and Discovery

The registration and discovery processes for the MonALISA services and clients are based on Jini technology described in paragraph 2.3.

In compliance with GMA Directory Service principles, each MonALISA service *registers* itself with a set of Jini Lookup Discovery Services (LUS) as part of a group, specifying a set of attributes. The LUSs are also Jini services and each one may be registered with the other LUSs, making possible to build a distributed and dynamic network for registration of services.

The registration process is based on a *lease* mechanism that is responsible to verify periodically that each service is alive. In case a service fails to renew its lease, it is removed from the LUSs and a notification is sent to all the services or clients that subscribed for such events for a specific group, called "community".

Any monitoring client uses the LUSs to *discover* the active MonALISA services running as part of one or several communities. The discovery mechanism is based on a set of matching attributes shipped via a remote event notification mechanism which also supports subscription.

At this point the client application (that can be also a service) connects directly with each service it is interested in for receiving monitoring infor-

mation: it first downloads the proxies for the selected service from a list of possible URLs specified as an attribute of each service, and than it instantiate the necessary classes to communicate with the service itself.

3.3 Repository

A generic framework for building "pseudo-clients" for the MonALISA services allows to create dedicated Web service repositories with selected information from specific groups of monitoring services. The repositories use the same LUSs approach described in the Registration and Discovery section to find all the active MonALISA services from a specified set of groups and subscribes to these services with a list of predicates and filters.

These predicates or filters specify the information the repository wants to collect from all the services: it stores all the values received from the running services in a local database (MySQL by default), and uses procedures written as Java threads to compress old data.

A Tomcat based servlet engine is used to provide a flexible way to present global data and to construct on the fly graphical charts for current or customized historical values, on demand. Dedicated servlets are used to generate Wireless Access Protocol (WAP) pages containing the same information for mobile phone users. Multiple Web Repositories can easily be created to globally describe the services running in a distributed environment.

3.4 GUI

The MonALISA GUI client allows to discover all the active remote MonALISA monitoring services. It displays real-time global views for connectivity and traffic as well as the usage and load of the monitored entities. By accessing the configuration of any site, it displays real-time values or short histories for any monitored parameter in the system. MonALISA GUI clients use the Java WebStart technology to be started from anywhere, with just a single click from the browser window.

The monitored parameters can be displayed on a 3D Globe or on a flat world map that can be manipulated (zoom in/out, rotate...) with the mouse. Additionally, graphical or tree layouts are available.

Clicking on a specific entity the corresponding property window is shown: it shows node's local time, IP address, MonALISA version and various site

information. The monitored information are organized in a tree view containing clusters and nodes. For each node and cluster on the right side is available a parameters view that displays history or realtime plots whose properties (3D/2D, stacked/simple, line width, graph background, axis labels etc.) can be customized.

Finally, the GUI client interface can be also used to manage the MonALISA service's Application Control Interface (AppControl) capabilities. It allows the farm administrator to remotely start, stop, restart and configure different applications. For each controlled application it must exist a corresponding module. The security part is implemented by communication between clients and server over SSL. The server has a keystore with the clients' public keys, so only the administrators can access this application.

Part III

Grid Experimentation

Chapter 4

MonALISA adaptations and extensions

This chapter describes the MonALISA components that have been adapted to satisfy the particular requirements of the ALICE Physics Data Challenge 2004 (PDC'04) monitoring. A detailed description of PDC'04 results can be found in the next chapter. The adaptions occurred to the framework have been incremental in the sense that the initial aim was to set up a working environment able to provide a basic monitoring of the ongoing data production. At that point, specific software developments and better presentation formats have been realized accordingly.

4.1 Farms Monitoring

In order to start monitoring a specific farm, a MonALISA service must be installed. From now on, the terms MonALISA service and MonALISA agent will be used to mean the same entity.

Once the MonALISA service is installed and started on a site, gathers performance data using dedicated modules to collect values provided by snmp daemons or by the kernel. In the first case, it is necessary that the snmpd daemons are installed and configured on the remote nodes or network elements (routers or switches) the user wants to monitor. In the second case MonALISA modules query the */proc* filesystem. The latter approach is mainly designed to be used on the local node MonALISA service is running but it may also be used on remote systems via rsh or ssh.

As described in the previous chapters, the core of the MonALISA service is based on a multi-threaded system that executes the data collection modules in parallel. The mechanisms to run these modules under independent threads (to control a snmp session, to perform the interaction with the op-

erating system or with user modules) are inherited from a basic monitoring class. The user has to provide the mechanism to collect the performance values, parse the output and generate a result object, as well as the names of each parameter collected by the monitoring module. On the other hand, any kind of specific site's information has to be provided by a custom (dynamically loadable) module.

Our approach to set up the communication between any installed MonALISA agent and the procedure responsable to produce user site's information has consisted of writing a generic Java class that runs custom monitoring shell scripts, providing outputs in a standard format.

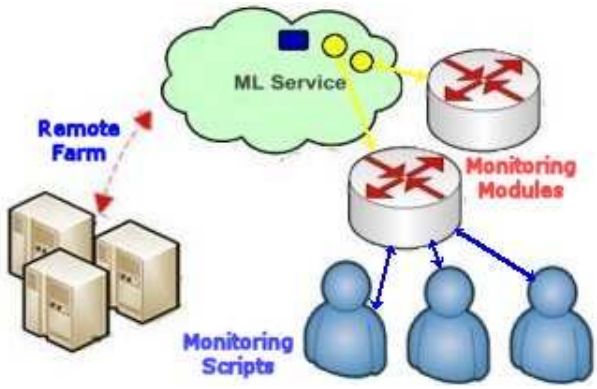


Figure 4.1: *Interaction between MonALISA service and any Interface module and Monitoring script*

In order to create a new MonALISA dynamically loadable module it is necessary to write a java class that extends the lia.Monitor.monitor.cmdExec class and implements the lia.Monitor.monitor.MonitoringModule interface. This interface has the following structure:

```
1   public interface lia .Monitor  .monitor.  MonitoringModule
2            extends lia .Monitor  .monitor.  cmdExec
3   {
4      public MonModuleInfo    init ( MNode  node , String  args  ) ;
5      public Object  doProcess  () throws Exception   ;
6      public String[]   ResTypes()   ;
7
8      public String  getOsName  ();
9      public MNode  getNode  ();
10
11     public String  getClusterName   ();
12     public String  getFarmName  ();
13     public String  getTaskName  ();
14     public MonModuleInfo    getInfo  ();
15
16     public boolean isRepetitive  () ;
17  }
```

It is up to the user to implement the following methods:

- *init*: initializes the useful information for the module, for example the cluster that contains the monitoring nodes, the farm and the command line parameters for this module. This function is actually the first called when the module is loaded.
- *doProcess*: collects and returns the results. Usually the return type is a Vector of lia.Monitor.monitor.Result objects. It can also be a simple Result object.
- *isRepetitive*: says if the module has to collect results only once or repeatedly. If true, then the module is called from time to time and the repetitive time has to be specified in the farm.conf file described below. If not, the default repetitive call time is set to 30 seconds.

The other functions return different module information.

4.1.1 A user class to interface MonALISA services

The Java class used to implement the MonALISA MonitoringModule interface is shown in Appendix A.1.1. This module, called *AlienCMD*, is in charge of gathering information that will be processed by the MonALISA service.

The doProcess method gets the output of the monitoring script presented in the next section and parses it via the Parse function. *AlienCMD* is dynamically uploaded by the MonALISA agent being defined in the farm.conf file by an entry in the following general format:

```
-------------------------------------------------------------------------------
*Cluster\{ loadable_module, farm_name, <parameters_list>; <shell_script> }%nsecs
-------------------------------------------------------------------------------
```

- **Cluster* is the functional unit used to aggregate parameters of the same type; for example monitoring parameters such as available_memory, processes number, load5, free_memory usually belong to the unit "Master";
- *loadable_module* is the name of the loadable monitoring module, AlienCMD in our case;
- *farm_name* is the node name;
- *parameters_list* is the list of performance parameters names: they will be used to name the published monitored values;
- *shell_script* is the name of the shell script producing the monitored values and, obviously, it has to return as many values as the parameters_list length;

- *%nsecs* is the number of seconds used by the isRepetitive method of the monitoring module.

Examples of custom clusters using the AlienCMD monitoring module are given below:

```
*AlienTcp{ AlienCMD, localhost, ntcp; ALIEN_ROOT/Java/MonaLisa/AliEn/Monitoring.sh Tcp ALIEN_ROOT/ }%60

*AlienLocalDisk{ AlienCMD, localhost, 1kblocks-root, usage-root, 1kblocks-tmp, usage-tmp;
                 ALIEN_ROOT/java/MonaLisa/AliEn/Monitoring.sh RootTmpDisk ALIEN_ROOT/  }%300

*AlienMQ{ AlienCMD, localhost, proc_inserting, proc_waiting, proc_assigned, proc_queued, proc_started,
                               proc_running, proc_saving, proc_done, proc_zombie, proc_failed,
                               proc_error_a, proc_error_i, proc_error_e, proc_error_r, proc_error_s,
                               proc_error_sv, proc_error_v, proc_error_vn, proc_expired, proc_idle,
                               proc_interactiv, proc_killed, proc_splitting, proc_split;
          ALIEN_ROOT/java/MonaLisa/AliEn/Monitoring.sh MasterQueue ALIEN_ROOT }%180

*AlienMQload{ AlienCMD, localhost, queueload, runload, queued, maxqueued, running, maxrunning;
              ALIEN_ROOT/java/MonaLisa/AliEn/Monitoring.sh MasterQueueLoad ALIEN_ROOT }%300
```

They are used to retrieve performance information from the MQ site about TCP/IP packets, disk usage, job status and job-queues occupancy respectively. We will discuss later about these parameters and their utility in a Grid monitoring system.

Technically speaking the AlienCMD class executes the script specified as shell_script and maps the first parameter name with the first output value produced by the script, the second name with the second value and so forth... The following section contains a description of the Monitoring.sh script used to produce strategic ALICE's performance information at the remote site.

4.1.2 A monitoring script run by MonALISA services

The Monitoring.sh script given in Appendix A.1.2 is in charge of retrieving the monitored values at the remote site and it now belongs to the standard AliEn monitoring distribution since it is quite flexible and easily to maintain. It contains a function per each class of information to be locally monitored (such as Tcp, Disk, Procs) or gathered from central servers and used for performance comparison (such as CE or SE). In the latter case it uses AliEn native commands or the AliEn API.

4.2 AliEn Jobs Monitoring

The AliEn monitoring keeps track of every job submitted through its entire lifetime. AliEn keeps track of the various states a job can assume and eventual error conditions that may occur.

Submitted jobs are stored as JDL (Job Description Language) scripts where the user can specify:

- Name of the executable that has to be run on the selected remote worker node;
- Arguments to pass to the executable;
- Eventual physical requirements that the worker node has to fulfill;
- Input and output data, code, libraries;
- Software packages;

An example of JDL can be found in section 4.7.

The user is only obliged to specify the executable application, while is up to the AliEn job manager to complete the other necessary fields and job requirements. At this point the job execution can be split over several sites.

Each site has at least one service called *Cluster Monitor* used to handle all the connections with the central services (Resource Broker and Job Manager) and to control one or more Computing Element (CE).

A CE asks the Broker for jobs to execute (see section 1.4), sending its JDL describing name, membership Grid partitions, near SEs and packages installed in the WNs. If the Broker finds a match between the JDL of the CE and the jobs, it will send the job's JDL to the CE, otherwise sleeps for a while and asks again.

Once the CE gets a job's JDL, the job is queued in the local batch queue and afterwards it will start running creating a web service called *Process Monitor*. The Process Monitor is the interaction point between the WN where the job is running and the rest of the AliEn services through the CE. Picture 4.2 shows services and actions involved during the job execution.

4.2.1 Job Status Flow

A submitted job goes through different status and, as stated before, the ability to keep track of this behavior is critical to understand the success or failure of a complex Grid system. Here below there is a description of the job status flow in AliEn:

1. when a job is submitted for processing, its status is "*WAITING*": the job's JDL is optimized by the Job Optimizer service to be queued by the Broker;
2. once the Resource Broker has found CE with adequate resources, the CE will pick the job's JDL up and the job will be "*ASSIGNED*";

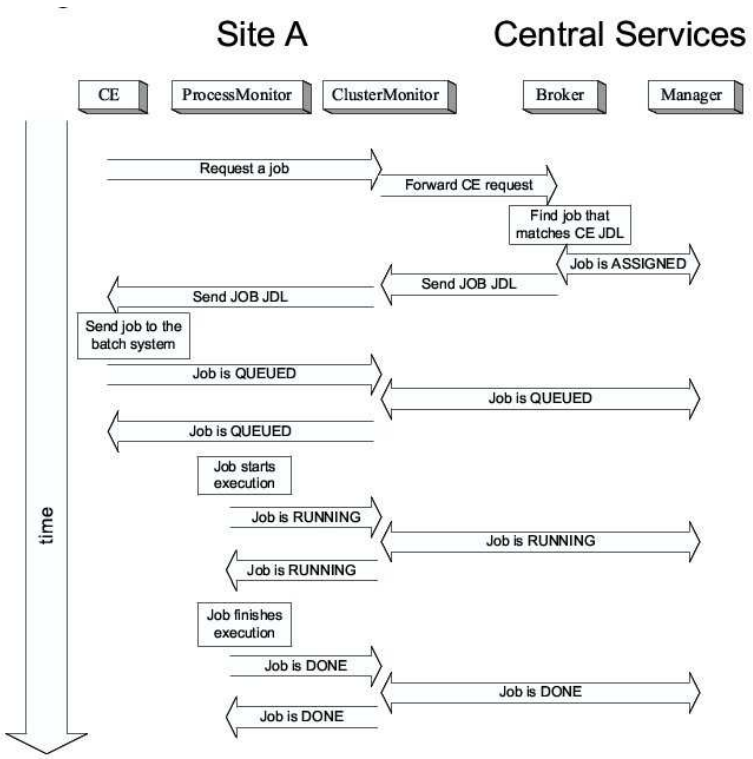


Figure 4.2: *Job execution - Resources versus Time*

3. the job is "*QUEUED*" when it is submitted to the local CE batch system;

4. the Process Monitor service starts on the WN and downloads input files, configures software packages and forks the user application: the job is "*STARTED*";

5. the user application is "*RUNNING*" on the WN and the Process Monitor sends an heartbeat to the AliEn server via SOAP calls;

6. if the job is writing log and output files into SEs, its status is "*SAVING*";

7. finally the job will be "*DONE*" when it finishes successfully;

8. jobs can also ask to be validated: the validation is a process that depends on the commands that are executed and usually parses the produced job output checking if there are any errors during the execution. The status of the job will be "*VALIDATED*" if it passes the validation procedure, or "*FAILED*" if it does not.

Figure 4.3 shows the AliEn job status flow. In case of failure in any of the different steps above, AliEn has several job status accordingly. Moreover, a job can stop interact with the central system for different reasons: in this case, if any heartbeat from the job is not received for more than 1 hour the job status will be "*ZOMBIE*" (but can recover from this status); if any heartbeat is not received for more than 3 hours the job status is "*FAILED*".

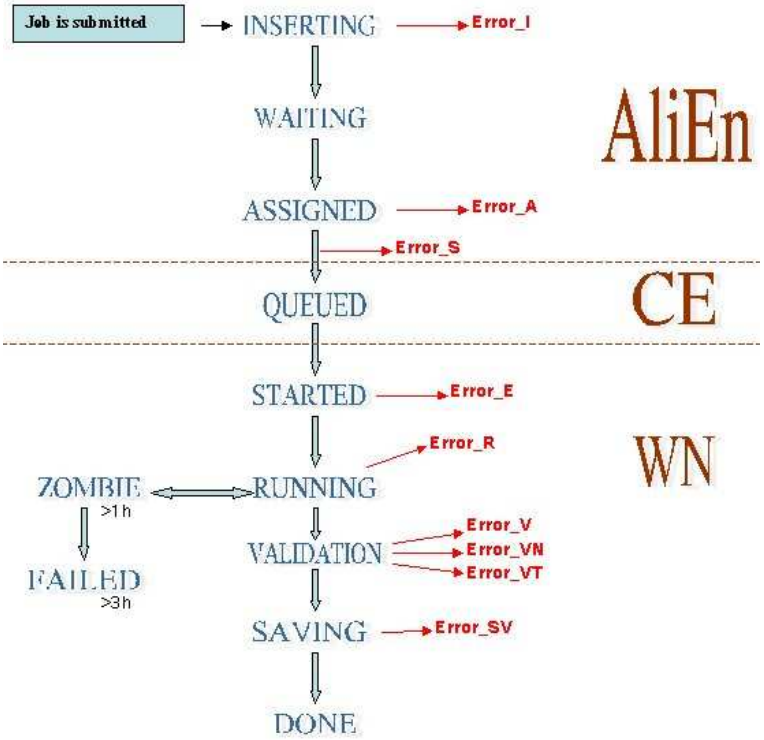


Figure 4.3: *Job status flow*

Further status can be "*INTERACTIVE*" if an interactive job is running, "*IDLE*" if the job is interactive but is not running or "*KILLED*" if it is killed by an operator call.

4.2.2 Centralized or distributed job monitoring?

Implementing a centralized or distributed system for monitoring jobs information has been the biggest dilemma during the first phase of PDC'04. The reliability of querying directly AliEn central servers to get job status information has played a critical role in choosing centralized solutions in place of pushing the MonALISA distributed monitoring agents philosophy, at least during the first two phases.

In fact, the first urgent goal of AliEn monitoring has been keeping history information of the job flow. The AliEn core services, via their mechanism of constant handshaking with the remote Cluster Monitors, provide the answer. By asking for jobs information at specific time intervals (this parameter depends on the jobs type, so it is fixed per each phase) it is possible to get a snapshot of the current situation at all the remote sites in one shot.

On the other hand, getting the same results for the whole AliEn Grid in a distributed manner would have been redundant because the remote monitoring scripts should query anyway the AliEn core services and then filter the answer just for the site where it is running.

For instance, an AliEn job status snapshot can be easily got by executing the AliEn queueinfo command (the output table is much larger because it reports all the possible job status and error conditions):

```
# alien login -exec queueinfo

Site                    Blocked         Status            Statustime    I    W    A    Q    ST   R    SV   D
-----------------------------------------------------------------------------------------------------------------------------
Alice::Bari::PBS        open            closed-maxqueued  1105957879    0    0    2    0    0    0    0    13
Alice::Bergen::PBS      open            down              1105957889    0    0    3    0    0    0    0    38
Alice::Calcutta::PBS    open            open-no-match     1105957883    0    0    1    0    0    0    0    5
Alice::Catania::PBS     open            down              1105957889    0    0    3    1    1    0    0    52
Alice::CCIN2P3::BQS     locked          closed-blocked    1105957798    0    0    0    1    0    0    0    11
Alice::CERN::FAKE       locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::CERN::LCG        locked          closed-blocked    1105957895    0    0    0    0    0    0    0    0
Alice::CERN::LCG2       locked          closed-blocked    1105957890    0    0    0    0    0    0    0    0
Alice::CERN::LCGtest    locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::CERN::LXSHARE    locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::CERN::Oplapro    locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::CERN::PCEPALICE45locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::CNAF::PBS        open            open-no-match     1105957873    0    0    1    1    1    1    1    4526
Alice::Cyfronet::PBS    locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::FZK::PBS         open            down              1105957889    0    0    0    0    1    2    0    12
Alice::GSI::LSF         locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::Houston::PBS     open            down              1105957889    0    0    0    0    0    0    0    0
Alice::IFIC::PBS        locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::IHEP::PBS        locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::ISS::PBS         locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::ITEP::RRC        open            down              1105957889    0    0    3    1    0    0    0    156
Alice::JINR::PBS        open            down              1105957889    0    0    5    1    0    0    0    193
Alice::KI::PBS          locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::LBL::LSF         open            down              1105957889    0    0    1    0    0    0    0    215
Alice::LCG::Torino      locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::Merida::gluon                    down              1105957889    0    0    0    0    0    0    0    0
Alice::ncp::PBS         locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::OSC::PBS         open            down              1105957889    0    0    5    0    1    0    0    131
Alice::Padova::LSF      locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::Prague::PBS      locked-err      closed-blocked    1105957878    0    0    0    0    0    0    0    0
Alice::PULV::FORK       locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::PULV::SGE        open            down              1105957889    0    0    0    0    0    0    0    0
Alice::PULV::Test       locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::SPBSU::PBS       locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::SRCE::DATAGRID   locked          down              1105957889    0    0    0    0    0    0    0    0
Alice::SUBATECH::PBS    open            closed-maxqueued  1105957870    0    0    4    1    1    0    0    82
Alice::Torino::LCG      locked          closed-blocked    1105957903    0    0    0    0    0    0    0    0
Alice::Torino::PBS      locked          closed-blocked    1105957865    0    0    4    2    1    0    0    138
Alice::Unam::PBS        open            down              1105957889    0    0    0    0    0    0    0    0
Alice::WISC::CONDOR     locked          down              1105957889    0    0    0    0    0    0    0    0
UNASSIGNED::SITE        locked          down              1105957889    0    0    0    0    0    0    0    0
-----------------------------------------------------------------------------------------------------------------------------
Sum of all Sites        ----            ----              ----          0    0    32   8    6    3    1    5572
-----------------------------------------------------------------------------------------------------------------------------
```

The first column shows the AliEn site names for the ALICE virtual organization (in a Grid fashion) and the used tasks queue system; the second

and third column show the site status, then follows a timestamp and the number of jobs per each different status. Evidently few status are supposed to be cumulative, for istance the "D" column stands for number of *Done Jobs*.

The next section will show how these snapshots are queried and parsed in order to feed the *DirectInsert* Repository custom module in charge to store the monitored values within the database.

4.3 ALICE's MonALISA Repository

A Repository has been the main analysis tool used in ALICE Off-line to monitor AliEn during the PDC'04. Basically the main advantage of the Repository (in respect of the GUI client) is the capability to keep data history allowing a full analysis of the gathered information.

The aim of the Repository is monitoring and storage of the running and cumulative parameters, tasks completion and resources status. It has been realized through a modification of the MonALISA framework to suit the ALICE production needs, retrieving data via one MonALISA agent running on one AliEn central server (Master Queue), AliEn monitoring commands, SOAP asynchronous communications, LCG monitoring scripts and several MonALISA services.

Monitored information have been gathered since March 13th 2004 across three different phases of the ALICE PDC'04. For a detailed description of PDC'04's phases, see chapter 5. At the end of phase 2 (September 2004) there were 17 millions records of data with one minute granularity stored within a relational database, for almost 2K different monitored parameters such as computing element load factors, storage element occupancy, job information and CERN network traffic in ALICE. In addition, 150 derived parameters, such as sites or system's efficiency, are computed starting from the basic ones by using a web front-end. Monitored data can be displayed in several formats: running history, bars and stacked bars, pies, tables, dials and active real-time geographic maps.

The Repository is independent from AliEn process tables and provides complete histories, using cumulative algorithms when necessary and allows for combination of basic parameters into derivative quantities for some basic analysis. It also provides API to get data either from distributed or central source. Every site can be monitored independently and in relation with others; the job status are separately monitored and the user can specify time interval for custom analysis.

4.3.1 Interfacing AliEn's API: DirectInsert

The current paragraph describes in which way the original MonALISA framework has been modified to allow a direct performance information retrieval from the AliEn central servers. As a Grid system, AliEn keeps track of the user job status through the Process Monitor service at the worker nodes level.

The MonALISA framework modification was concerning primarily the job status, with all the other site monitoring information generally provided by MonALISA services. The technical solution to gather data directly from AliEn API was provided by an additional Repository client thread called *DirectInsert*: it interfaces the Repository data collection module with a shell script querying AliEn via native commands, called *queueinfo.sh*.

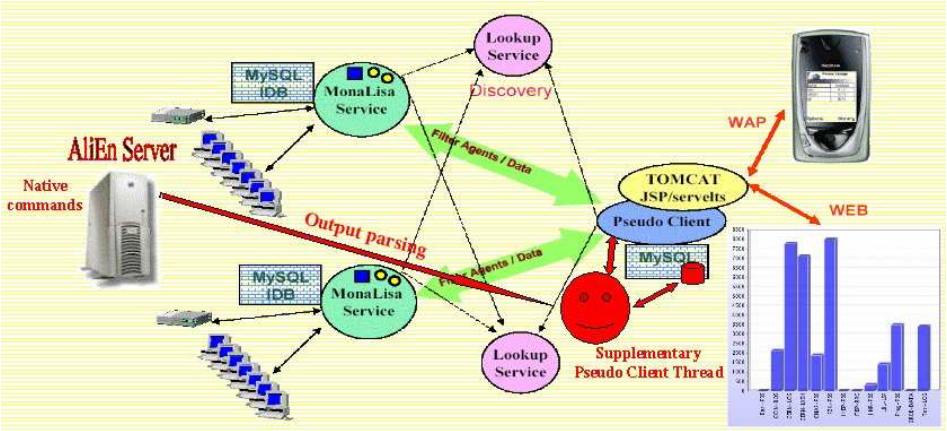


Figure 4.4: *Initial MonALISA framework modification*

Through this mechanism, performace data not coming from MonALISA services feed the database Repository preserving the automatic averaging functionality. The full source code of *queueinfo.sh* and *DirectInsert* modules is shown in Appendix A.1.3 and A.1.4 respectively.

The general format of the performance data the script must provide is the following:

```
-----------------------------------------------------------------
siteName clusterName hostName parameterName parameterValue timestamp
-----------------------------------------------------------------
```

The first step of the script is in charge to retrieve job status information, the second one provides local queue load factors.

In addition to the AliEn queries, modules for gathering information from the LCG Grid were added. Step 3 of the *queueinfo.sh* script parses the output report generated by the lcg-CEInfoSites script developed by the LCG's IT collaboration.

The following AliEn native Perl API calls have been used in place of the two initial command-line AliEn calls, and they have been provided a more reliable method for information retrieval.

```
1  use AliEn::UI ::Catalogue ::LCM:: Computer;
2  my $cat = new AliEn::UI ::Catalogue ::LCM:: Computer ();
3  while (1) {
4      sleep(120);
5      print `date `, ":=>producing queue list \n";
6      open SAVE_STDOUT,">&STDOUT" ;
7      $tmpstdout = "/tmp /queuelist .new";
8      open STDOUT ,"> $tmpstdout ";
9      $cat ->execute ("queue ","list ");
10     close STDOUT ;
11     open STDOUT , ">&SAVE_STDOUT";
12     system("mv /tmp/queuelist .new /tmp/queuelist ");
13
14     print `date `, ":=>producing queue info \n";
15     open SAVE_STDOUT,">&STDOUT" ;
16     $tmpstdout = "/tmp /queueinfo .new";
17     open STDOUT ,"> $tmpstdout ";
18     $cat ->execute ("queue ","info ");
19     close STDOUT ;
20     open STDOUT , ">&SAVE_STDOUT";
21     system("mv /tmp/queueinfo .new /tmp/queueinfo ");
22 }
```

In conclusion, at the end of 2004, there are four kind of different monitoring information sources feeding the Repository: the AliEn Master Queue providing job status and queues load, LCG interface providing LCG disks and jobs information, the MonALISA agents deployed at the remote sites in a number that slowly grows up month by month, and, finally, prototype SOAP based interface to monitor ALICE network traffic at CERN (that will be replaced by Java WSs as described in paragraph 4.8).

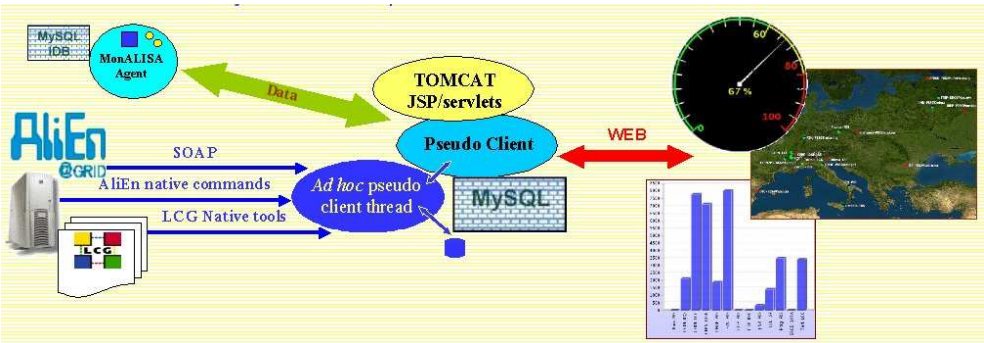


Figure 4.5: *Repository information source types at the end of 2004*

4.4 Repository Database

A MonALISA Repository can be interfaced to any kind of relational database. ALICE's Repository uses MySQL according to true experience with this backend.

4.4.1 Evolution

At the time the Repository has been initially set up, the DB schema basically consisted of a number of historical tables hosting the Grid monitoring information at different level of time granularity. Based on the MonALISA design and the GMA specifications, the Repository, in its role of consumer, asks the Lookup Service for MonALISA services (the producers) belonging to a virtual organization (ALICE) in order to establish connections with the services (via proxies) and retrieve performance information.

The gathered data are stored in a table called *1hour-table*, containing all the values acquired during the last hour. The internal caching mechanism and averaging system is forwarding averages of the monitoring values at different time intervals to different tables, as shown in picture 4.6. For instance, 10 bins of a monitored parameter from the 1hour-table are grouped together and averaged to produce one bin of the *10hours-table* (the averaging process is not trivial because it produces also an initial and final value and not only the average).

Data older than the hosting table period are removed: the aim is in fact to always hold *light* tables for fast plotting functionality and for simplify the creation of the most frequently used 1hour, 10hours, 1day and 1week plots.

After few months of use this schema showed several weaknesses:

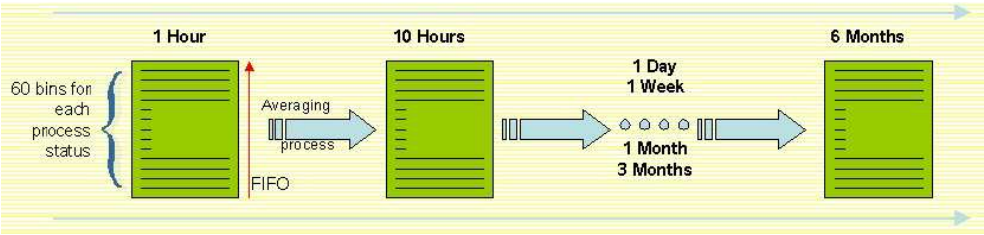


Figure 4.6: *Data binning in the MonALISA DB Repository*

- the plots did not have enough details in case of custom time interval selection since the old values are erased from the hosting tables and just averaged before the deletion, it was not possible to display detailed history between two custom dates laying in the past. The initial design did not foresee that historical data with high granularity are needed for performance analysis of the Grid;

- since became clear that the monitored data volume was too high, the need of a different and flexible storage format appeared accordingly. The history tables are hosting the identity of the monitoring parameters: initially as long strings in a format cluster/hostname/parameter_name/ and later by a numeric *id* based on a dictionary.

By creating a dictionary of the monitoring parameters (up to 2000 in the initial months) and changing the history storage system, the Repository data binning has changed as shown in Figure 4.7.

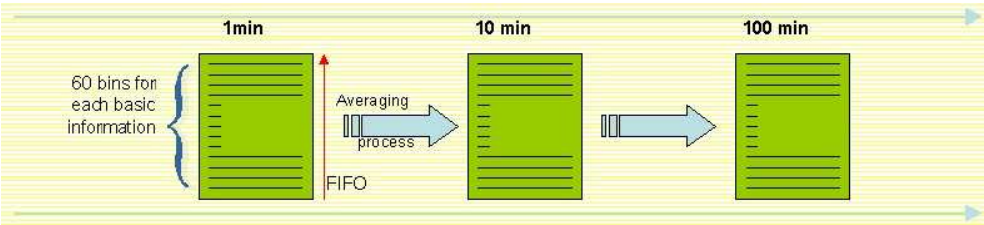


Figure 4.7: *Enhanced Data binning in the MonALISA DB Repository*

In this way it has been possible to fulfill the dual necessity of fully detailed and averaged histories, allowing to display rapidly charts without loss of information. An indexes system has given good performances for user time intervals selection and the structure has been proved to fit the ALICE's needs for the whole PDC'04 duration (9 months). New functionality

are coming out during the beginning of PDC'05 and a new DB schema based on split tables is under investigation.

4.4.2 MySQL replication

In parallel with the monitoring of the Grid, the Repository became an object of information for Grid performance and simulation studies. It has become clear that stressing the Repository database with heavy computations had the consequence to slow down the web Repository performance. In order to simulate the behavior of the Grid it is necessary to produce complex distribution histograms requiring the processing of substantial parts of the information stored in the Repository: for this purpose, a DB replication was set up.

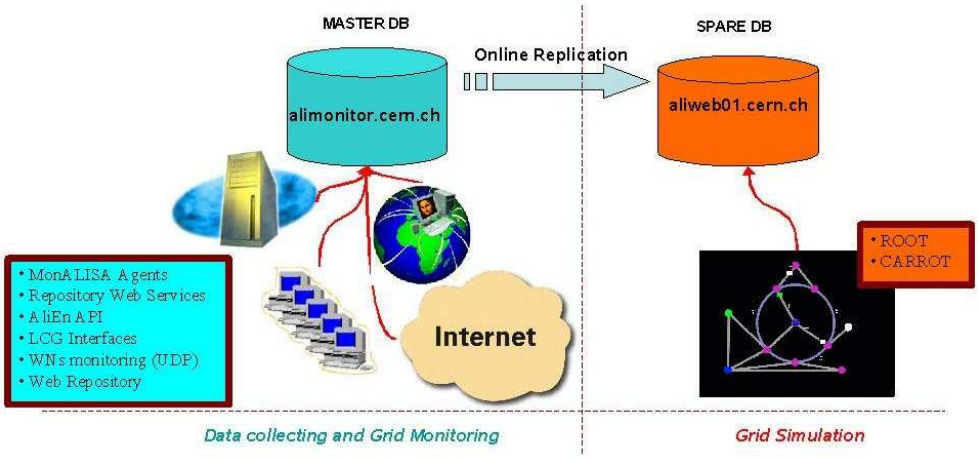


Figure 4.8: *DB replication to separate monitoring from simulation*

Generally, the reasons of a database replication are as follows:

- hot (on-line) spare;
- load (scalable) balancing;
- non-disturbing backups;
- separate environments (i.e. AliEn monitoring and Grid simulation).

Picture 4.9 shows basic concepts of a database replication mechanism and the possible configurations.

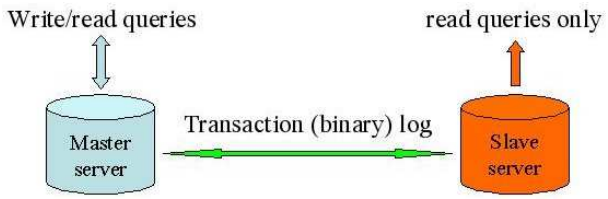


Figure 4.9: *Replication Schema*

MySQL provides a native reliable replication capability [21] and the setup process is quite straightforward. As first step, the master server has to be tuned specifying few configuration parameters; the master is not aware of whom are the slaves and where they are. Afterwards, the slave server has to be set up by specifying the master hostname and transactions log name used to retrieve the last updating statements executed at server side.

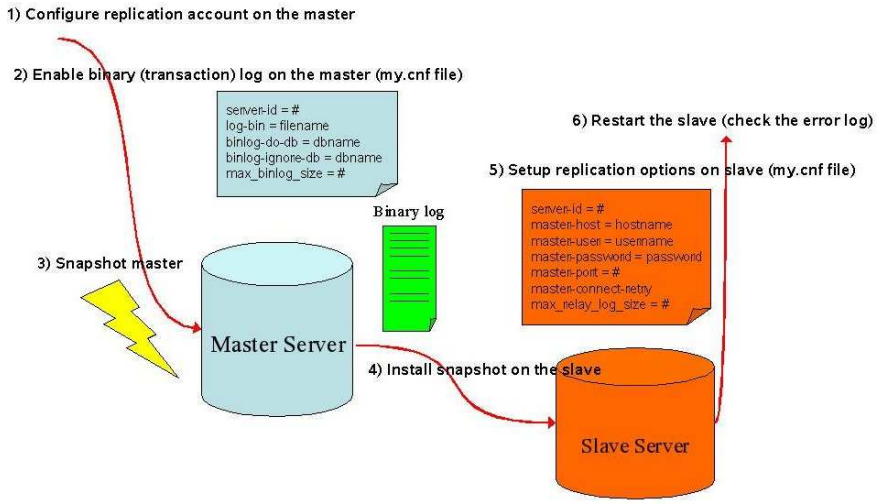


Figure 4.10: *Replication Setup*

The replication process is asynchronous: one slave thread asks the server for updates and a second thread is started to execute them once it got.

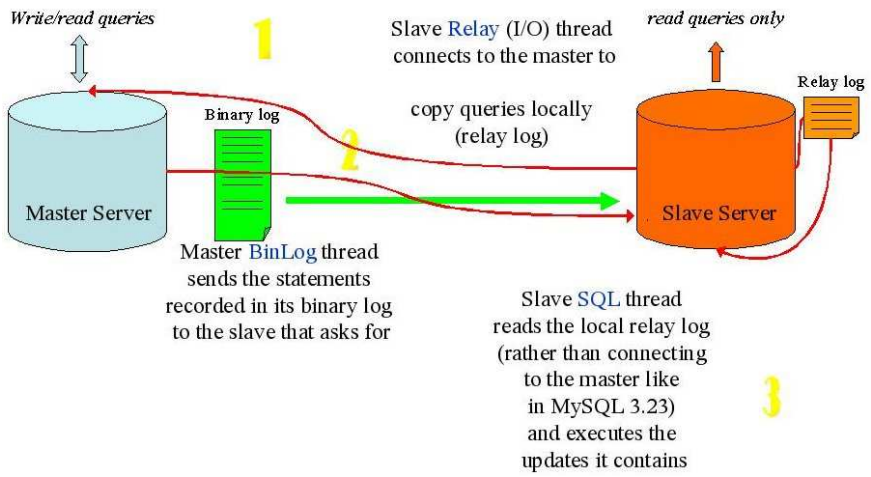


Figure 4.11: *Replication Functioning*

4.5 Web Repository

The Web Repository represents the Repository front-end. It is based on Java technology: a servlet running within Tomcat, connected to a MySQL database pool and providing an interface for custom development.

The Web Repository public url is http://alimonitor.cern.ch:8080. The title page, shown in picture 4.12, has the aim of geographically representing the ALICE's farms activity in Europe. By a mouse click on the "World map" button the visualization switches to farms world wide. Normally there is a unique correspondence between cities and farms, apart few cases such as Geneva itself (at CERN, ALICE uses several queues) and Torino, Italy.

Each site can be displayed in different solid colors or their combination. As the legend shows, sites can assume four main status:

- "*jobs running*". The green color indicates that the local queue at the remote farm is up, and there are running jobs in a number between one and the maximum allowed by the configuration.
- "*opened*". The cyan color represents the situation when a remote site is ready to process jobs but it is not running any of them. Reasonably, sites in this status might switch shortly to the green one. Technically speaking, the Resource Broker hasn't found any match between the site JDL and all the task JDLs scheduled within the Master Tasks Queue.
- "*error-locked*". The blue color shows an error situation. Although

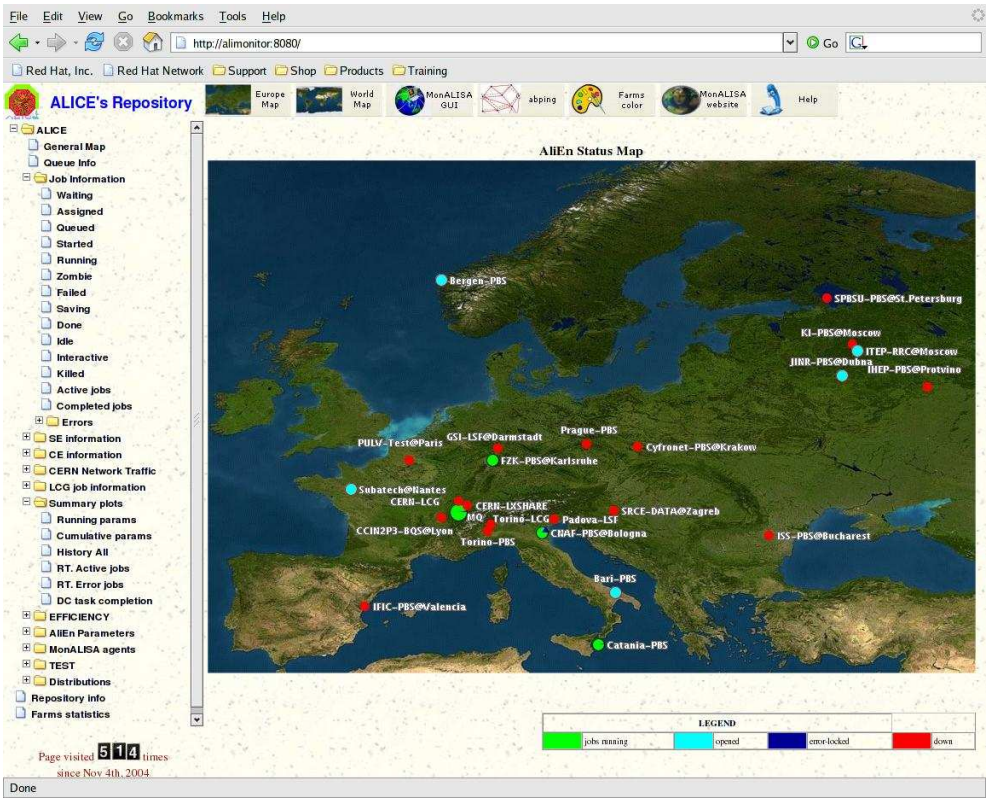


Figure 4.12: *Web Repository*

the site has been started, at a certain point its Cluster Monitor has communicated a local failure.

- "*down*". The red color indicates that the AliEn services are not running at the site.

Moreover, if a site is up and running jobs, the color is usually not just flat but it looks like a pie showing the ratio between jobs running and maximum allowed to run at that site. This is a useful visualization to give an overall idea of the sites occupancy at a glance.

Detailed situation and performance per each site can be explored by the appropriate histories, pies and dials from the web Repository menu. Most of them will be discussed in the next chapter.

4.5.1 Monitored Parameters

At the beginning of PDC'04 phase 2, the number of monitored parameters was already up to 1868 coming from four different main sources as summa-

rized in the table below:

SOURCE	CATEGORY	NUMBER	EXAMPLES
AliEn API	CE load factors	63	run load, queue load
	SE occupancy	62	used space, free space, files number
	Job information	557	running, saving, done, failed
SOAP calls	Network traffic	29	MBs, files
LCG	CPU - Jobs	48	Free CPU, job running and waiting
ML service on MQ	Jobs summary	34	running, saving, done, failed
	AliEn parameters	15	MySql load, Perl processes
ML services	Sites info	1060	Paging, threads, I/O, processes

Derived classes have been computed from the basic parameters listed above: basic efficiency values about AliEn, LCG and AliRoot [5], the ALICE Off-line framework for simulation, reconstruction and analysis.

EFFICIENCY MEASURE	FORMULA
Job Execution	$\varepsilon_j = \frac{proc\_done}{Total}$ %
System	$\varepsilon_s = 1 - \frac{\sum_{i=1}^{n} proc\_error_i + proc\_failed}{Total}$ %
AliRoot	$\varepsilon_A = 1 - \frac{proc\_error\_v}{Total}$ %
Resources	$\varepsilon_{r1} = \frac{proc\_running}{max\_proc\_running}$ % $\varepsilon_{r2} = \frac{proc\_queued}{max\_proc\_queued}$ % $Total = (proc\_done + \sum_{i=1}^{n} proc\_error_i + proc\_failed)$

4.5.2 Presentation formats

Graphical improvements to the web Repository presentation layouts offer the possibility to plot, sort and group any kind of performance data either basic or derived in multitude of presentation formats. The web plotting engine is based on JFreeChart [16], a free Java class library to generate charts. The MonALISA framework provides running histories, bars and stacked bars charts for real-time purposes and pies.

Additional charts, such as the *dials* shown in picture 4.13, have been developed to extend the graphical formats and represent the status of completion of specific activities (for example each of the three Data Challange Phases) as well as the usage of each site (as ratio between number of jobs running over the maximum, or queued over maximum queueing).

Figure 4.13: *Dials displaying CEs occupancy*

Source code and description about dials creation can be found in Appendix A.2.2.

4.6 Distributed Monitoring

The MonALISA design heavily relies on distributed agent technology. This has clear advantages over a central approach; some of them are listed below:

- *Monitored parameters distributed by nature*: monitoring network traffic or sites performance produce information that must be retrieved only by distributed agents running at the remote sites. The only information that reliably comes from central sources is the job status since AliEn tracks these to keep jobs evolution under control: deploying remote scripts to gather jobs information would have been redundant;
- *Real Time GUI client*: as described in the previous chapter, MonALISA provides a powerful JWS based GUI client that allows to have a full monitoring of recent site's parameters when the Services are started;
- *Remote control and Software delivery*: MonALISA Services are not just monitoring tools. The newest versions contain an Application Control Interface (AppControl) that allows the farm administrator to remotely and safely start, stop, restart and configure different applications;
- *High-granularity data*: by nature, the Repository is storing historical data gathered with certain frequency. On the other hand, remote Services can acquire data at higher frequency, showing them via the GUI and subsequentially averaging and storing only at less granularity.

- *Filters*: apply filters is useful to reduce the ratio by which performance data are sent to clients, such as a Repository, in charge of storing historical data; besides custom filters are very important in case of Grid simulation toolkits working on top of the Monitoring framework (i.e. Monarc [20]);
- *Alarm and Messaging system*: the capability to trigger alarms from the Services is fundamental to implement messaging system in case of failures or values being lower or higher than specific thresholds.

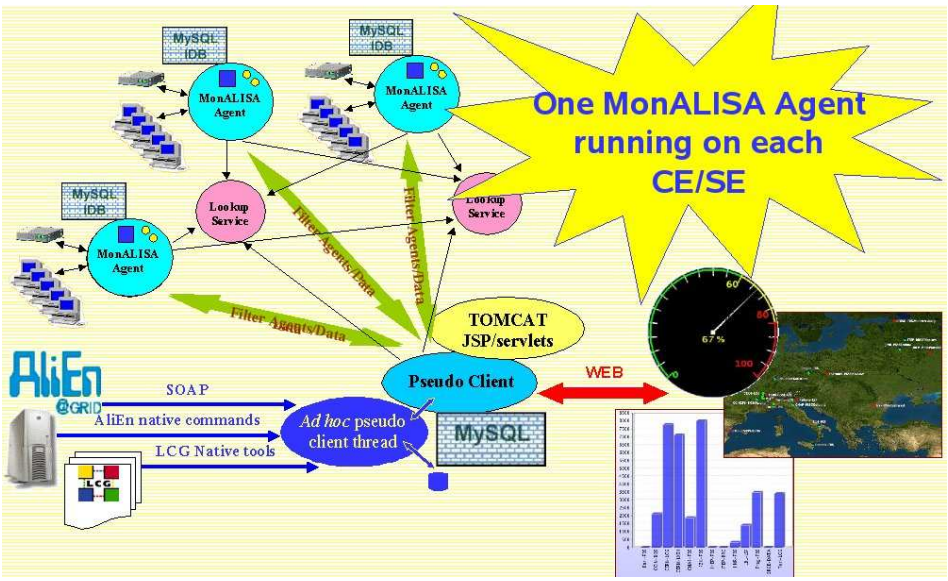


Figure 4.14: *MonALISA framework extensions allowing for a distributed monitoring through agents*

A further improvement to distributed monitoring Services at the remote farms is to monitor the Worker Nodes, on which the user application is running, one level deeper than the Computing Elements in the Grid hierarchy. The MonALISA framework provides a set of API called *AppMon* (Application Monitoring) to send monitoring data from any user application. Detailed descriptions can be found in paragraph 4.9.

4.7 Job Monitoring by User

By scanning the job's JDL and using the method to send monitored values directly to the Repository via the *DirectInsert* additional thread, job monitoring has been improved to keep track of the user identities and resources requirement. In other words, in a Grid system jobs can be submitted to

a central queue by different (potentially thousands) users: a deeper level to track down the evolution other than the status they go through, is to maintain information about the user each job is submitted by and moreover, resources and constraints specified within the JDL.

By using AliEn API it has been possible to write a Perl script running on the Repository server to retrieve the information above. The script gets the list of job IDs submitted to the Master Queue, scans the sub-jobs for each one, finally parses the JDL. The source code is shown below.

```
use AliEn::UI ::Catalogue ::LCM:: Computer;
my $cat = new AliEn::UI ::Catalogue ::LCM:: Computer ();

    open SAVE_STDOUT,">&STDOUT" ;
    open STDOUT ,"> /dev /null";
    my $loopcnt =0;
L1: while (1) {
        $loopcnt ++;
        open OUT," >/tmp /.AliEnJobsByUsers .out";
        select OUT;

        print SAVE_STDOUT "==> START exec : ",`date `;
LPSA :  foreach $job ($cat ->execute ("ps" ,"-a")) {
                my @words =split(/ +/,$job );
                my $jobID =$words [1];
                my $userID =$words [0];
                my @jobinfos =$cat ->execute ("jobinfo ","$jobID ");
                my $continue =0;
                my $psxstateargs ="";
LJI1 :          foreach $jobinfo (@jobinfos ) {
                        @words= split(/ +/, $jobinfo );
                        my $state =$words [1];
                        if ($state eq "SPLIT" || $state eq "SPLITTING ") {
                                next LJI1 ;
                        }
                        $psxstateargs .= " -st ". $state;
                        if ($state eq "WAITING " || $state eq "QUEUED " || $state eq "STARTED " ||
                            $state eq "RUNNING " || $state eq "SAVING ")
                        { $continue =1; }
                }
                if($continue ==0) { next LPSA; }
                my $sum =0;
                foreach $subjobinfo ($cat ->execute ("ps"," -X",$psxstateargs ,"-id $jobID ")) {
                        my $site =substr($subjobinfo ,19,27);
                        $site=" s/ //g";
                        my $shortstate =substr($subjobinfo ,46,8);
                        $shortstate =" s/ //g";
                        $shortstate =" s/1 x//g";
                        $shortstate =" s/0 x//g";
                        my $key = $userID ."::".getSiteName ($site)." ::".$shortstate ;
                        $sites {$key }++;
                        $sum++;
                }
        }
        open (TS,"date +\"%s\"|");
    }
```

The output is used by the *queueinfo.sh* script described in section A.1.4. The information is integrated into the web interface and provides the user with additional information within the same environment, as shown in picture 4.15.

JDLs content gives a possibility to extend the monitoring capability. A typical AliEn JDL is as follows. It shows the application(s) the user needs at the remote site to run the job, input and output files, resources require-

Figure 4.15: *Job monitoring by User*

ments and constraints.

```
[
    MaxInputFileSizeOld      = "2000000000  ";
    InputFileOld    =
        {
            "LF:/alice  /cern.ch  /user/p/  pshukla/CheckESDex    .C"
        };
    Packages   =
        {
            "GEANT3::  v0-6",
            "AliRoot  ::4.01.Rev  .04"
        };
    SplitOld   = "se ";
    OutputDir   =
        {
            "analysis  "
        };
    ExecutableOld    = "/bin/aliroot   ";
    MaxInputFileNumberOld      = "10";
    Executable   = "/bin/aliroot   ";
    RequirementsOld    = other .SPLIT  == 1 && ( member (other .Packages  ,"GEANT3  ::v0 -6") ) &&
                                               ( member (other .Packages  ,"AliRoot  ::4.01.  Rev.04")   );
    InputDataSetOld    =
        {
            "AliESDs  .root",
            "Kinematics   .root" ,
            "galice.  root"
        };
    InputDataOld    =
        {
            "LF:/alice  /production  /aliprod  /2004-03/V4.01.  Rev.00/cent1   /00156/00001/   galice  .root "
        };
    WorkdirectorysizeOld      =
        {
            "500MB"
        };
    InputDataAccessOld      = "local ";
    Type  = "Job";
    TypeOld   = "Job ";
    ArgumentsOld    = "Alice::  Commands::  AliRootS   -x CheckESDex   .C";
    PackagesOld    =
        {
            "AliRoot  ::4.01.Rev  .04",
            "GEANT3::  v0-6"
        };
    SpecialRequirementsOld      = ( other .GlueCEPolicyMaxWallClockTime       > 240  );
    OutputDirOld    =
        {
            "analysis  "
        };
    OutputFileOld    =
        {
            "CheckESDex   .root"
        }
]
```

4.8 Network Traffic Monitoring via Web Services

With the increasing volume of monitored information, a necessity to develop an interface to publish and store monitoring information from any context directly into the Repository (via the *DirectInsert* module) has emerged. To address this goal, WSDL/SOAP technology has been used to realize a communication between the Repository and any monitored application.

Deploying Web Services on the Repository server is a choice easy to explain: they provide interoperability between various software applications running on different platforms and, by using HTTP, can work through many common firewall security. Repository WSs are handled by Apache Axis, an implementation of SOAP fully integrated into the Tomcat servlet engine already running on the Repository server. In this way the same container is currently used either for the MonALISA client implementing the Repository or for the DirectInsert module and Web Services. Using the same technology for different kind of services allow easier debugging and further integrated developments.

Figure 4.16 shows the WSs currently running at the Repository server, focusing on the one in charge to directly store data within the Repository database.

Figure 4.16: *Repository Web Services*

The source code below shows the server implementation of the deployed Web Service, focusing the *directInsert* method: it produces an output file according the format read by the *DirectInsert* Repository custom thread.

```
1  package MLWebService1_pkg;
2
3  import java.io .*;
4
5  public class MLWebService1SoapBindingImpl        implements MLWebService1_pkg.MLWebService1     {
6      // ....
7      // ....
8      public int directInsert   (java .lang.String    site , java.lang  .String   cluster,
9                                 java .lang.String    node , java.lang  .String   function  ,
10                                java .lang.String    mval , java.lang  .String   rectime)
11                 throws java.rmi.  RemoteException     {
12      String   baseDir   = "/home/monalisa    /MLrepositoryNEW    /ALICE /";
13
14      try {
15         FileOutputStream     fout  = new FileOutputStream    (baseDir   + "MLfromSoap   .out ", true);
16         PrintStream    cout        = new PrintStream   (fout );
17
18         cout. println(site   +"\t"+ cluster+"   \t"+node+  "\t"+function   +"\t"+mval+"  \t"+rectime   );
19         cout. flush();
20         cout. close();
21         return 0;
22      }
23      catch (IOException    ioe)  { return -1; }
24    }
25    // ....
26    // ....
27 }
```

The client code is more complicated since it has to instantiate the stub, set the endpoint and finally invoke the method to pass the monitored values.

```
1  import org.apache   .axis .client.Call   ;
2  import org.apache   .axis .client.Service    ;
3  import javax. xml.namespace   .QName ;
4
5  public class MLWebService1Client
6  {
7     public static void main(String    [] args ) {
8         try { if (args. length  < 1) {
9                  System .out.println   ("Usage :: java  MLWebService1Client     <method > " +
10                                        "[parameters    list]");
11                 return;
12             }
13
14             String   methodName    = args[0];
15             String   retval        = "";
16
17             if (methodName   .compareTo  ("directInsert    ")==0  && args.length   < 7) {
18                 System .out.println   ("Usage :: java  MLWebService1Client     <method > " +
19                                        "[parameters    list]");
20                 return;
21             }
22
23           String   endpoint   = "http ://alimonitor   .cern .ch:8080/  axis2/ services/  MLWebService1   ";
24
25           Service    service   = new Service ();
26           Call       call      = (Call)  service .createCall   ();
27
28           call .setOperationName    (new QName (endpoint  , methodName   ));
29           call .setTargetEndpointAddress      ( new java .net.URL  (endpoint  ) );
30
31           if       (methodName   .compareTo  ("directInsert    ")==0)  {
32               retval  = String. valueOf(   (Integer)   call.invoke   (
33                         new Object  [] { new String (args[1]),   new String (args[2]),
34                                          new String (args[3]),   new String (args[4]),
35                                          new String (args[5]),   new String (args [6]) } ) );
36           }
37           else if (methodName   .compareTo  ("addInt"  )       ==0)  {
38               retval  = String. valueOf(   (Integer)   call.invoke   (
39                         new Object  [] { new Integer (Integer  .parseInt  (args[1])),
40                         new Integer (Integer  .parseInt  (args [2])) } ) );
41           }
42           else if (methodName   .compareTo  ("getVersion   ")   ==0)  {
43               retval  = (String  ) call.invoke   (new Object[]   {} );
44           }
45           System .out. println(methodName      + "()  returnValue    = " + retval );
46
```

```
47          } catch (Exception   e) { System.err   .println  ("Execution   failed . Exception  : " + e); }
48      }
49  }
```

Network traffic through ALICE's resources at CERN is an example where Repository Web Services have been used to send and store performance data. Picture 4.17 shows the data volume produced by the four main FTP servers in ALICE during the central part of PDC'04 Phase 2.

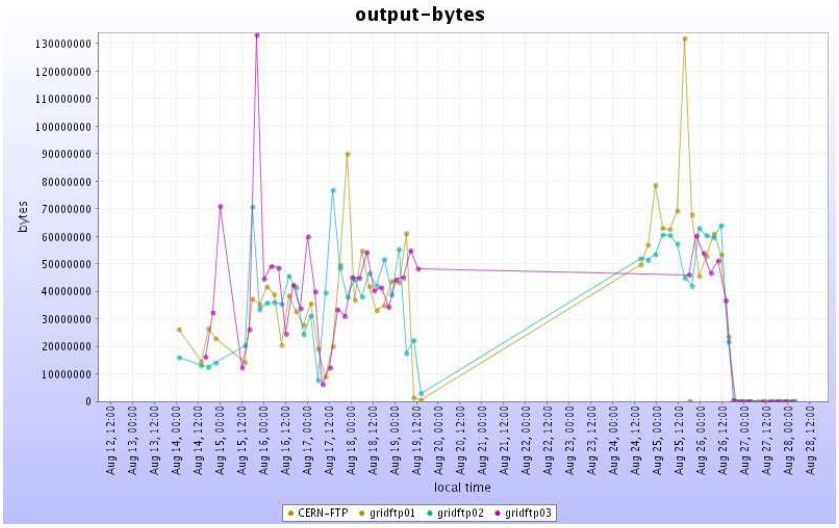


Figure 4.17: *Network traffic of ALICE's servers at CERN, monitored using web services*

4.9 Worker Nodes Monitoring

During Phase 3 of PDC'04 we have begun testing the monitoring feasibility at Worker Nodes (WNs) level, where user application effectively runs. The goal is to include a light monitoring module within the AliEn Process Monitor payload and start this module in order to get WN's performance data (such as storage, memory and cpu information, network traffic) sending them to the MonALISA services running at the CE upper level (not necessarily to an Agent running on the same CE the WNs belong to).

One of the last developments consist of a set of flexible APIs called ApMon (Application Monitoring) that can be used by any application to send monitoring information to MonALISA services. Picture 4.18 shows how the monitored data are sent as UDP datagrams (encoded in the XDR, eXternal Data Representation) to one or more hosts running the Services.

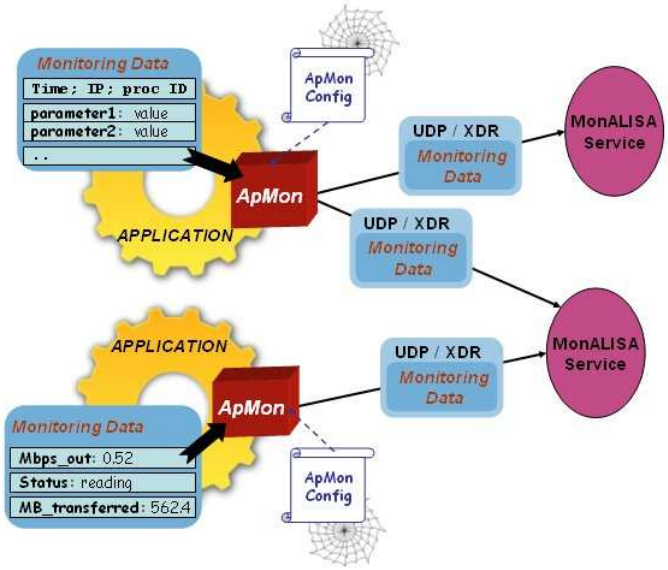


Figure 4.18: *WN monitoring by ApMon*

User applications can periodically report any type of information the user wants to collect, monitor or use in the MonALISA framework. An example of simple test application developed using ApMon C++ API is the following:

```
#include <stdlib .h>
#include <time .h>
#include "ApMon .h"

int main( int argc, char **argv) {
  char *filename = "destinations .conf";
  int nDatagrams = 20;
  double myval ; int i;
  srand (time( NULL));

  try
  { ApMon apm (filename );
    for (i = 0; i < nDatagrams ; i++)
    { myval = 2 * (double)rand() / RAND_MAX;
      try
      { apm.sendParameter ("ApMonCluster ", NULL, "myMonitoredParameter ", myval );
      }
      catch(r untime_error &e)
      { fprintf (stderr , "Send operation failed: %s\n", e.what ());
      }
      sleep(1);
    }
  } catch( runtime_error &e) {}
}
```

The tested example shows the instantiation of an ApMon object in charge of handling the encoding of the monitoring data in the XDR representation, building and sending the UDP datagrams. XDR, as well as XML, is a cross-platform format that works transparently either in big-endian or little-endian systems, but has been preferred because an XML package is

many times larger than the same XDR and it is important at this level of detail to keep the smallest footprint possible.

The destination.conf configuration file specified in the source code is used to indicate the MonALISA service(s) data must be sent to and the port it listens on (8884 by default).

4.10 Grid Analysis using Distributions

An extension of the MonALISA framework used for Grid performance analysis has been the possibility to display histograms and distributions. This is realized by the integration of additional components on a separate server. The server hosts the online replication of the Repository database and runs Apache Web Server. In addition, installation of the following CERN's packages have been required:

- *ROOT* [24]: an environment providing a set of OO frameworks with all the functionality needed to handle and analyse large amount of data in a very efficient way. In ROOT, for what concerning our goals, are available specialized graphic methods to access data and plot charts by using advanced histograming methods in one or multiple dimensions, curve fitting etc...
- *Carrot* [6]: an open source C++ scripting module for the Apache webserver based on the ROOT framework. It has the ability to embed C++ code into HTML pages or HTML into C++ source, access and manipulate databases and generate graphics on-the-fly (among many others that haven't been used for our purpose).

By using the TSQLServer class to query the MySQL database, it is quite simple to create and populate the histogram and draw the resulting graphics within the browser, which are generated by Carrot as JPEG pictures. The following example shows a method to plot an histogram based on data stored into a database:

```
1  void queryDB()
2  {
3      TSQLResult   *res;
4      char q2[200];
5      sprintf( q2,"select   mval,rectime    from  monitor_6m_1min,monitor_ids \
6                 where  id= mi_id and  mval >0 and  mi_key=\"%s"\",func_c);
7      if(res=db  ->Query( q2)) {
8          Int_t n=res ->GetRowCount  ();
9          x=( Double_t) malloc (n*sizeof (Double_t));
10         y=( Double_t) malloc (n*sizeof (Double_t));
11         Int_t i=0;
12         TSQLRow   *row;
13         while (row=res  ->Next ()) {
14                h->Fill( atof(row  ->GetField  (0)));
15                x[i]=atof  (row ->GetField  (1));
16                y[i]=atof  (row ->GetField  (0));
17                i++;
```

```
18          }
19          g=new  TGraph(n  ,x,y);
20      }
21  }
```

4.10.1 An example of Distributions Cache

In order to optimize the throughtput of the user queries producing distributions, a caching system has been implemented. By using the ROOT capabilities to manage objects serialization, the data flow between the database and Carrot is cached using the last timestamp of a distribution. Thus a new user query will only retrieve from the DB the not-cached entries.

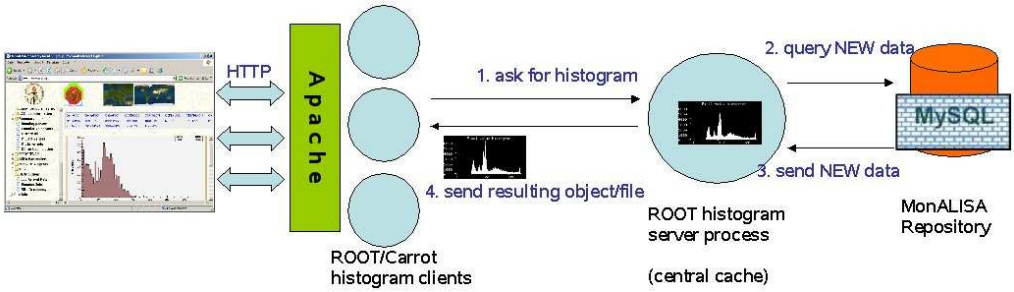


Figure 4.19: *A cache system for distribution analysis*

Picture 4.20 shows the UML of the java classes that implement the caching system.

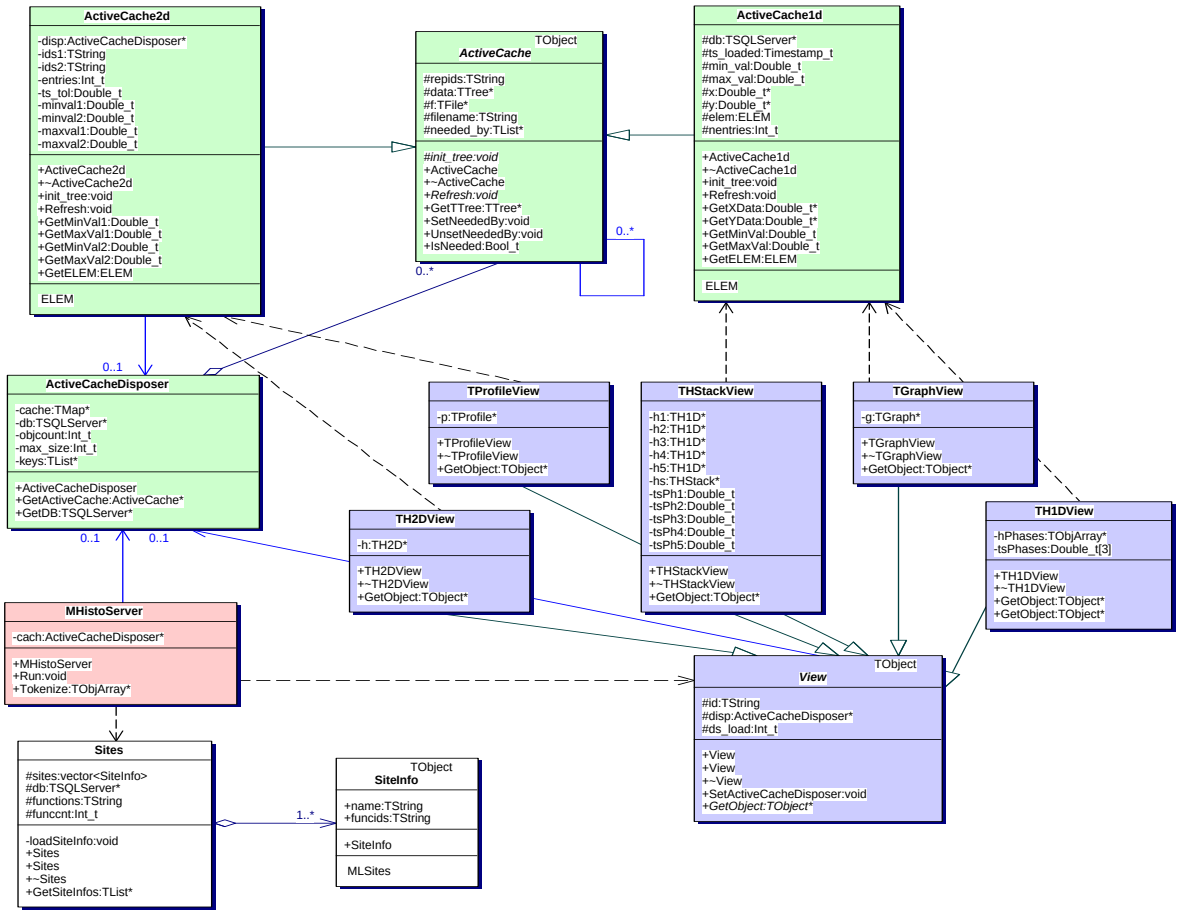


Figure 4.20: *UML of java classes implementing the caching system*

Chapter 5

PDC'04 Monitoring and Results

The role of the monitored data is to give an immediate and extensive representation of the current status and history of the parameters relevant to the Grid operation. In addition, they provide a basis for a realistic analysis of the Grid behavior in its complexity, where many components at different sites with interlocked hierarchical levels interact with each other.

5.1 PDC'04 purpose, structure and principles

The ALICE Physics Data Challenge 2004 (PDC'04) was a complete exercise of the distributed computing model chosen by the collaboration [9]. Its structure, task and duration were tuned to test extensively the functionality and reliability of the experiment's Grid. An integral part of the Grid is its monitoring which includes many "local" and "global" parameters. The terms are explained later in the chapter. To that end, the MonALISA framework provides a rich set of monitoring tools and methods for their application in a distributed framework.

The PDC'04 consisted of Monte-Carlo (MC) particle production, a subsequent particle trajectory reconstruction in the detectors, reconstruction and physics analysis. Its purpose, structure and principles are outlined below:

- *Purpose*: test and validate the ALICE Off-line computing model by producing and analyzing approximately 10% of the data sample of a standard data-taking year. Use the entire software framework described in Chapter 1 (AliEn, AliRoot, LCG, PROOF) with the dual purpose of making a complete test of the software and a production of a valid set of data to be used for detector studies and estimation of

the physics capabilities of the experiment;

- *Structure*: PDC'04 has been divided in three parts called *phases*. During Phase 1, a large number of underlying Pb+Pb events with different centrality were produced, as well as 10^6 p+p events. In Phase 2, signal events of various physics content (high-energy jets, open charm, di-muons) were merged with the underlying events and the resulting mix was reconstructed. To achieve a statistically significant sample of signal events, the underlying events were reused several times. In Phase 3, the reconstructed signal+underlying events are analyzed with various methods by the physicists in the collaboration. The analysis can be done in batch and interactive mode.

- *Principles*: the Monte-Carlo data production, subsequent reconstruction and analysis ware performed entirely on the ALICE Grid, using only AliEn for job submission and for access and control of the distributed computing resources and, through an interface, the resources provided by the LCG Grid.

The three phases of PDC'04 and their duration are shown in Figure 5.1. In practical terms, the control of the Monte-Carlo data production and reconstruction during phases 1 and 2 has been achieved centrally by a small number of operators. On the other hand, during Phase 3 many physicists submit analysis jobs.

The AliEn command line interface provides methods for job control and monitoring capabilities with several layers of access rights and details. In addition, the AliEn Web Portal [3] provides a web based interface. The portal has been designed to provide an entry point to AliEn, grouping together either a command and control interface or a full monitoring system. It is based on Open Source components and allows for an intuitive access to information coming from different sources. Users can check and manage job provenance and access process monitoring information from the MonALISA Repository.

5.2 Phase 1 - Monte-Carlo simulation

The aim of Phase 1 (April - May 2004) was to simulate the flow of data generated by the experiment during data taking in reverse. During normal data taking, the raw data produced by the spectrometer will be stored at CERN and subsequentially processed and analyzed at CERN and elsewhere. In the absence of the spectrometer, the data generation was done at remote sites, sent back to CERN over WAN and stored in the CERN Mass Storage Systems (MSS) CASTOR [7].

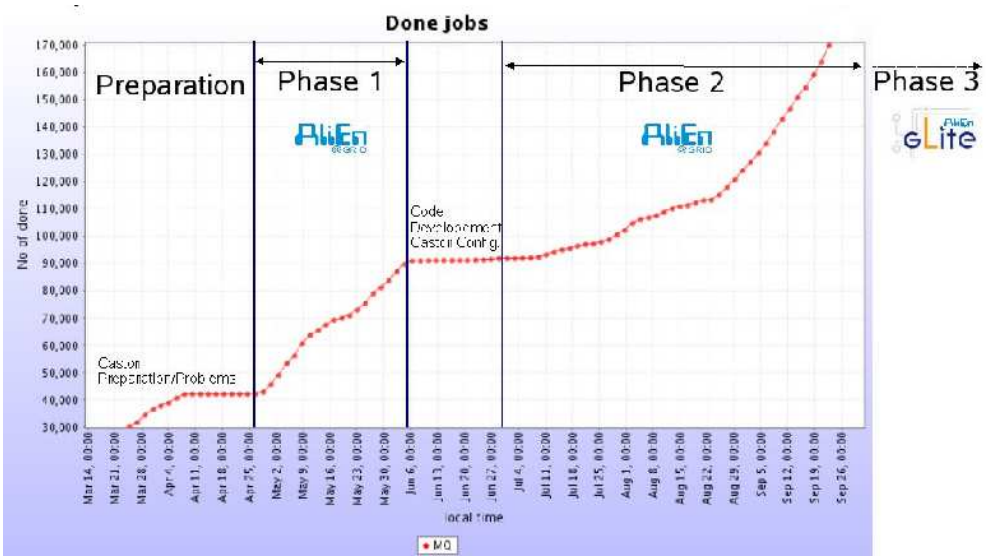


Figure 5.1: *Accumulation of number of completed jobs versus time during PDC'04. The vertical lines delimit the first two phases of the data challenge.*

Picture 5.2 shows the job path: jobs are submitted to the AliEn Task Queue, optimized and assigned to Computing Elements (CEs) by the Resource Broker (RB). An entire Grid, the LHC Computing Grid (LCG), was interfaced to and seen by AliEn as one CE. The produced data is shipped back to CERN at the end of the job.

The physics content of the generated data includes a complete description of the ALICE detector geometry and signal response of the various sub-detectors to the charged and neutral particles, which are produced from the colliding Pb ions. A summary of the physics signals produced during Phase 1 is shown in table 5.1.

Bin name	Impact parameter value [fm]	Produced events
Central1	0 -5	20K
Peripheral1	5 - 8.6	20K
Peripheral2	8.6 - 11.2	20K
Peripheral3	11.2 - 13.2	20K
Peripheral4	13.2 - 15	20K
Peripheral5	> 15	20K

Table 5.1: *Parameters of the Pb+Pb events produced during Phase 1*

The average track multiplicity has a maximum of around 87.5K tracks in

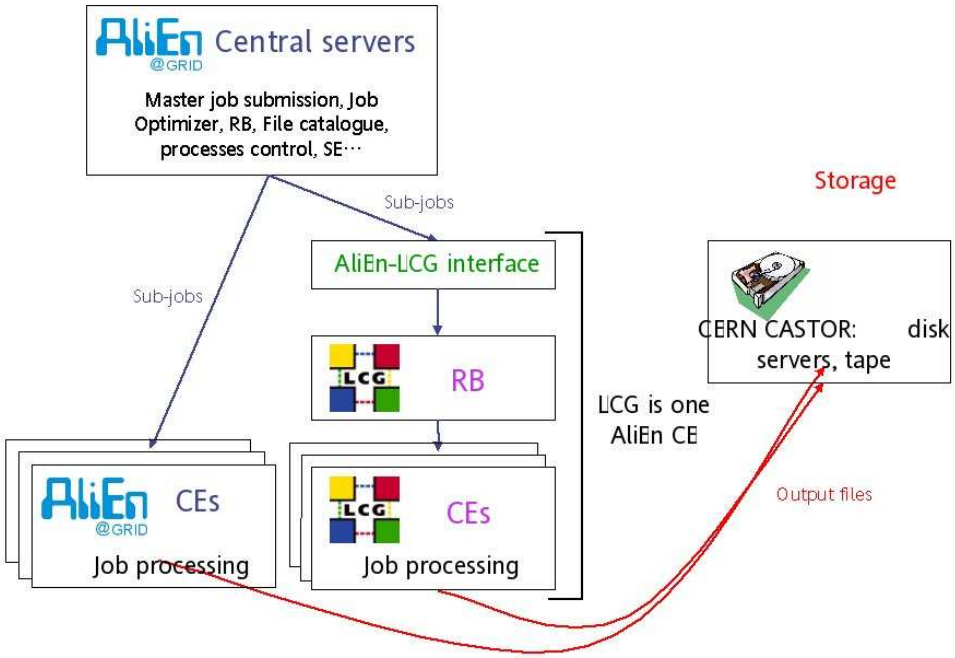


Figure 5.2: *Schematic view of jobs submission and data flow during Phase 1 of PDC'04*

the acceptance of the ALICE detector system for *central1* collisions, whilst this value falls down to 20K in case of *peripheral5* collisions.

Each event produces the 36 files shown in table 5.2, 24 of which contain various results from the simulation and reconstruction and are stored into CASTOR. The remaining 12 files (log and configuration) are saved in a special scratch Storage Element. A single event generation job lasts from 8 hours for the most *central* job to a couple of hours for the most *peripheral* on a typical PC (Pentium IV 3Ghz). Most of this time is spent in the reconstruction of the trajectories of the particles produced by the interaction through the detectors and the generation of the detector response.

Having lower number of particles generated, *peripheral* events produce smaller output than the *central* ones. The production scripts have been tuned in such a way that the size of the output files is the same for *peripheral* and *central* events. This is achieved by grouping more than one *peripheral* event into one single file.

The purpose of having similar sized events is for easier tracking and to

drwxr-xr-x	admin	admin	0	Apr	4	04:17	.
drwxr-xr-x	admin	admin	0	Apr	4	04:17	..
-rwxr-xr-x	admin	admin	6733728	Apr	4	04:17	AliESDs.root
-rwxr-xr-x	admin	admin	1903	Apr	4	04:17	check.log
-rwxr-xr-x	admin	admin	19173	Apr	4	04:17	check.root
-rwxr-xr-x	admin	admin	21920	Apr	4	04:17	CheckESD.C
-rwxr-xr-x	admin	admin	40191	Apr	4	04:17	Config.C
-rwxr-xr-x	admin	admin	4529139	Apr	4	04:17	ENCAL.Hits.root
-rwxr-xr-x	admin	admin	415818	Apr	4	04:17	ENCAL.SDigits.root
-rwxr-xr-x	admin	admin	2361545	Apr	4	04:17	FMD.Hits.root
-rwxr-xr-x	admin	admin	2245138	Apr	4	04:17	galice.root
-rwxr-xr-x	admin	admin	122	Apr	4	04:17	getESD.sh
-rwxr-xr-x	admin	admin	39298387	Apr	4	04:17	ITS.SDigitis.root
-rwxr-xr-x	admin	admin	39190211	Apr	4	04:17	Kinematics.root
-rwxr-xr-x	admin	admin	611497	Apr	4	04:17	MUON.Hits.root
-rwxr-xr-x	admin	admin	1378569	Apr	4	04:17	PHOS.Hits.root
-rwxr-xr-x	admin	admin	331878	Apr	4	04:17	PHOS.SDigits.root
-rwxr-xr-x	admin	admin	2217050	Apr	4	04:17	PND.Hits.root
-rwxr-xr-x	admin	admin	326285	Apr	4	04:17	PND.SDigits.root
-rwxr-xr-x	admin	admin	163	Apr	4	04:17	rec.C
-rwxr-xr-x	admin	admin	156683	Apr	4	04:17	rec.log
-rwxr-xr-x	admin	admin	1124	Apr	4	04:17	resources
-rwxr-xr-x	admin	admin	1304436	Apr	4	04:17	RICH.Hits.root
-rwxr-xr-x	admin	admin	686985	Apr	4	04:17	RICH.SDigits.root
-rwxr-xr-x	admin	admin	183	Apr	4	04:17	sim.C
-rwxr-xr-x	admin	admin	763456	Apr	4	04:17	sim.log
-rwxr-xr-x	admin	admin	956	Apr	4	04:17	simrun.C
-rwxr-xr-x	admin	admin	1048280	Apr	4	04:17	START.Hits.root
-rwxr-xr-x	admin	admin	8614	Apr	4	04:17	stderr
-rwxr-xr-x	admin	admin	35342	Apr	4	04:17	stdout
-rwxr-xr-x	admin	admin	172930	Apr	4	04:17	TOF.SDigits.root
-rwxr-xr-x	admin	admin	310028055	Apr	4	04:17	TPC.SDigits.root
-rwxr-xr-x	admin	admin	169565456	Apr	4	04:17	TrackRefs.root
-rwxr-xr-x	admin	admin	26123338	Apr	4	04:17	TRD.SDigits.root
-rwxr-xr-x	admin	admin	623	Apr	4	04:17	validation.sh
-rwxr-xr-x	admin	admin	4193865	Apr	4	04:17	VZERO.Hits.root
-rwxr-xr-x	admin	admin	830058	Apr	4	04:17	ZDC.Hits.root
-rwxr-xr-x	admin	admin	74173	Apr	4	04:17	ZDC.SDigits.root

Table 5.2: *Output files from one event. AliRoot files are associated with the simulation and the reconstruction*

avoid an overload of the MSS by producing a large number of small files. The fact that all *central1* and *peripheral1* events have been stored on disk has provided fast access for Phase 2 processing.

The duration of Phase 1 was 58 days: from March 13th to May 29th. The peak number of jobs running in parallel was achieved on March 27 - 1450 jobs. The average number of jobs for the period was 430. The job running history is shown in Fig.5.3. The uneven number of jobs throughout the Phase 1 was due mostly to number of free CPU constraints at the remote computing centres.

Due to the short time available for the initial MonALISA implementation at the beginning of the PDC'04, it hasn't been possible to set up and run monitoring agents at each remote site. In addition, the first versions of the agents exhibited instabilities and frequently crashed. This was one of the reasons a centralized information gathering approach was adopted for Phase 1 of the data challenge, as explained in section 4.2.2.

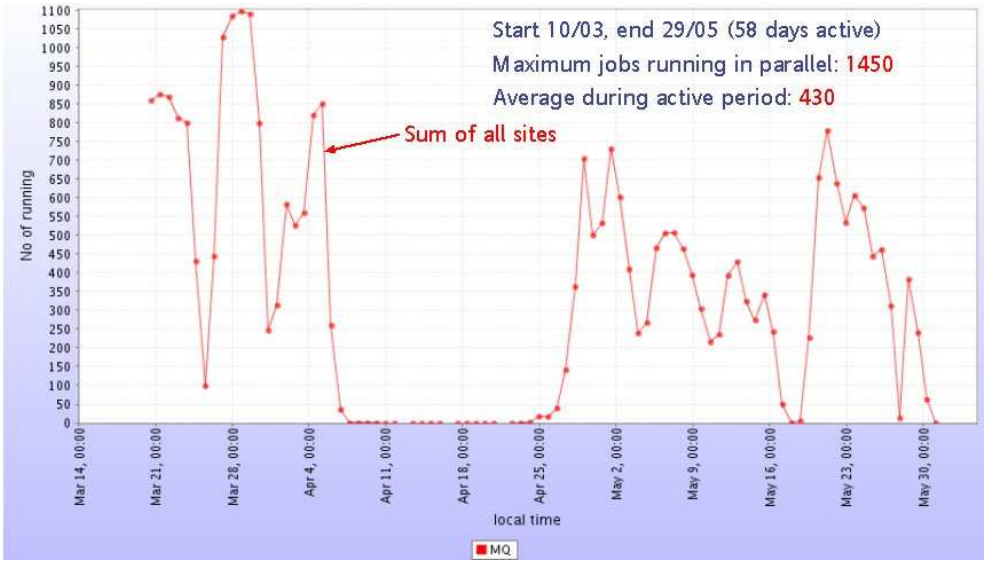


Figure 5.3: *Phase 1 - History of running jobs*

The average Phase 1 job distribution (see Fig.5.4) shows a balanced picture of jobs produced by 12 remote computing centers with no one center dominating the production. The slice representing LCG covers all centres running the LCG Grid middleware; the share of jobs successfully done is divided in 76% from AliEn native resources and 24% from LCG resources.

The plot in picture 5.5 shows the running job at the remote farms during PDC'04 Phase 1. We noticed that the sites CERN-LCG and CNAF-PBS reached the highest peaks; the table lists minimum, average and maximum values and confirms that the two sites satisfied the highest data volume throughout the period, mostly thanks to a major resources availability.

Table 5.3 shows the summary statistics for the jobs, storage, and CPU work during Phase 1:

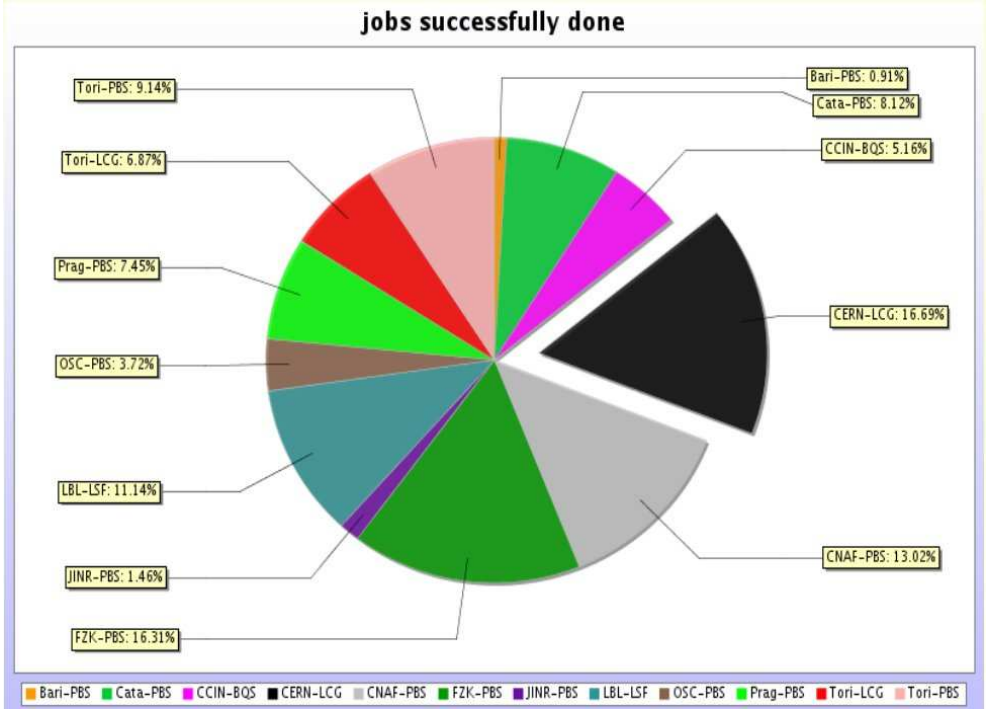
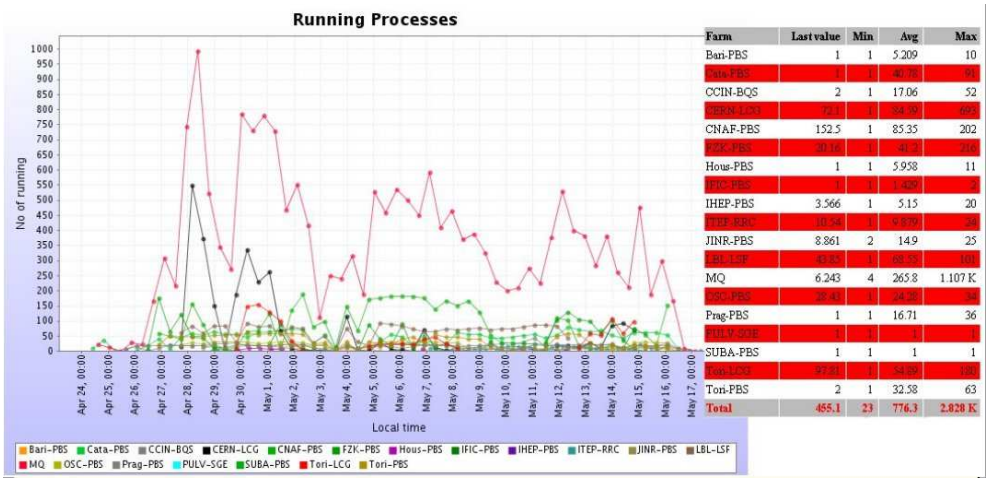


Figure 5.4: *Relative distribution of done jobs during PDC'04 Phase 1 among all participating computing centres*



Farm	Last value	Min	Avg	Max
Bari-PBS	1	1	5.209	10
Cata-PBS	1	1	40.78	91
CCIN-BQS	2	1	17.06	52
CERN-LCG	721	1	204.39	693
CNAF-PBS	152.5	1	85.35	202
FZK-PBS	30.16	1	46.2	216
Hous-PBS	1	1	5.958	11
IFIC-PBS	1	1	1.429	2
IHEP-PBS	3.566	1	5.15	20
ITEP-RRC	10.54	1	9.879	24
JINR-PBS	8.861	2	14.9	25
LBL-LSF	43.83	1	68.55	301
MQ	6.243	4	265.8	1.107 K
OSC-PBS	22.42	1	24.28	34
Prag-PBS	1	1	16.71	36
PULV-SGE	1	1	1	1
SUBA-PBS	1	1	1	1
Tori-LCG	97.81	1	54.29	192
Tori-PBS	2	1	32.58	63
Total	**455.1**	**23**	**776.3**	**2.820 K**

Figure 5.5: *Number of running jobs as a function of time during PDC'04 Phase 1 for all participating computing centres*

Number of jobs	56.000
Job duration	8h (cent1), 5h (periph1), 2.5h (periph2-5)
Files per job	36
Number of entries in AliEn file catalog	3.8M
Number of files in CERN CASTOR	1.3M
File size	26 TB
Total CPU work	285 MSI-2k hours
LCG CPU work	67 MSI-2k hours

Table 5.3: *Summary statistics during Phase 1*

5.2.1 Grid operation improvements

The Grid operation during Phase 1 has shown some problems that were resolved prior to Phase 2.

- The accumulation of processes in the AliEn Task Queue has led to a slow response, whereby new processes could not be managed effectively. To solve this problem, the Task Queue was split into "already completed tasks" (long) and "running tasks" (short) tables.
- The error handling and reporting capabilities have been improved by adding information of the remote sites status.
- The CASTOR MSS has shown limitations on the number of files which can be available simultaneously on a disk server. Due to the stager database design, this number cannot be larger than 300K files for optimal response. In Phase 2, to limit the number of output files kept on a server, all outputs of a job were saved as a single archived file, with AliEn functionality enhanced to be able to archive and pull out of archive a single file, as needed for the user analysis.

5.3 Phase 2 - Mixing of signal events

Phase 2 (July - September 2004) was aimed at a distributed production of signal events with different physics content and merging them with the underlying Pb+Pb events of Phase 1. This allows to simulate the standard schema of event reconstruction and remote event storage: network and file transfer tools and remote storage systems are being tested for performance and stability.

Picture 5.6 shows the Computing Elements (CEs) processing the underlying events stored in CERN CASTOR MSS and saving output files in local Storage Elements (SEs) and CASTOR itself. AliEn File Catalog (FC) maps LCG Logic File Names (LFN) in AliEn Physical File Names (PFN).

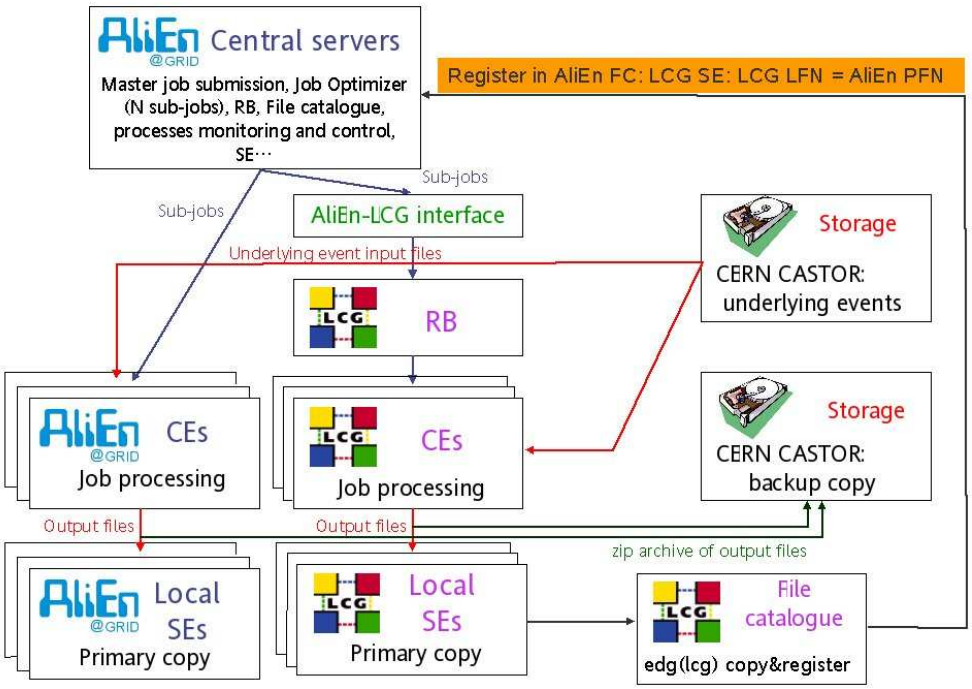


Figure 5.6: *Schematic view of jobs submission and data flow during Phase 2 of PDC'04*

For example, in the case of jet physics, to facilitate the events merging, the underlying events are grouped (through symbolic links) for specific signal event type in a directory. 1660 underlying events are used for each jet signal condition and another 1660 underlying events are used for the next and so on up to 20.000 in total. As such, we obtain the 12 conditions shown

in the first three columns of table 5.4.

Signal	No.of signal events per underlying	Number of jobs
Jets (un- and quenched) cent 1		
Jets PT 20-24 GeV/c	5	1666
Jets PT 24-29 GeV/c	5	1666
Jets PT 29-35 GeV/c	5	1666
Jets PT 35-42 GeV/c	5	1666
Jets PT 42-50 GeV/c	5	1666
Jets PT 50-60 GeV/c	5	1666
Jets PT 60-72 GeV/c	5	1666
Jets PT 72-86 GeV/c	5	1666
Jets PT 86-104 Gev/c	5	1666
Jets PT 104-125 GeV/c	5	1666
Jets PT 125-150 GeV/c	5	1666
Jets PT 150-180 GeV/c	5	1666
Total signal	399840	39984
Jets (un- and quenched) per 1		
Jets PT 20-24 GeV/c	5	1666
Jets PT 24-29 GeV/c	5	1666
Jets PT 29-35 GeV/c	5	1666
Jets PT 35-42 GeV/c	5	1666
Jets PT 42-50 GeV/c	5	1666
Jets PT 50-60 GeV/c	5	1666
Jets PT 60-72 GeV/c	5	1666
Jets PT 72-86 GeV/c	5	1666
Jets PT 86-104 Gev/c	5	1666
Jets PT 104-125 GeV/c	5	1666
Jets PT 125-150 GeV/c	5	1666
Jets PT 150-180 GeV/c	5	1666
Total signal	399840	39984

PHOS	**cent 1**	
Jet-Jet PHOS	1	20000
Gamma-jet PHOS	1	20000
Total signal	40000	40000
D0	**cent 1**	
D0	5	20000
Total signal	100000	20000
Charm & Beauty	**cent 1**	
Charm (semi-e) + J/psi	5	20000
Beauty (semi-e) + Y	5	20000
Total signal	200000	40000
MUON	**cent 1**	
Muon coctail cent1	100	20000
Muon coctail HighPT	100	20000
Muon coctail single	100	20000
Total signal	6000000	60000
MUON	**per 1**	
Muon coctail per1	100	20000
Muon coctail HighPT	100	20000
Muon coctail single	100	20000
Total signal	6000000	60000
MUON	**per 4**	
Muon coctail per4	5	20000
Muon coctail single	100	20000
Total signal	2100000	40000
Grand total	15239680	339968

Table 5.4: *Phase 2 - Repartition of physics signals embedded into the underlying events of Phase 1*

Table 5.5 shows the summary statistics for the jobs, storage, and CPU work during Phase 2.

Number of jobs Job duration	400.000 6 h/job
Conditions	62
Number of events	15.2M
Number of files in AliEn file catalog Number of files in storage	9M 4.5M distributed at 20 CEs world-wide
Storage at CERN CASTOR Storage at remote SEs	30TB 10TB
Network transfer	200TB from CERN to remote CEs
Total CPU work	750 MSI-2k hours

Table 5.5: *Summary statistics during Phase 2*

A requirement in Phase 2 is that a job must finish, unlike Phase 1 when they were Monte-Carlo: they are treated like real physics events and every failing jobs is analyzed for the cause of failure and then resubmitted for new processing.

Experience from the first phase of the PDC'04 has emphasized the need for a comprehensive set of monitoring tools, needed to run a large scale production and identification of problems and bottlenecks. The MonALISA Repository during this phase has been improved with new plotting functionality, dials and historical graphs, shown in section 4.5.

Sixteen AliEn sites and two big LCG CEs have been the workhorses of the data production. Individual job information, clearly the most important monitoring element of a distributed system, provided complete and detailed information and was based on the AliEn central tasks-queue.

At the same time few sites have been chosen to test and tune the MonALISA monitoring services concerning data volume being transfered (Bari, Prague and Master Queue itself). The stability of the remote monitoring agents will be in fact crucial from a Grid monitoring (and following Grid simulation) points of view. Through integrated filtering mechanism based on customizable predicates the individual job status can be aggregated for every individual site and transmitted to the MonALISA Repository thus preserving an average picture of the Grid at every moment in time.

The amount of gathered monitored information at the end of Phase 2 has been quite impressive: 7GB of data stored in 24 million records with one-minute granularity. This information is currently being analyzed with the goal of improving the Grid performance and discovering possible sources of AliEn job failures. Table 5.6 shows the error rates.

Failure type	Reason	Rate
Submission	CE scheduler not responding	1%
Loading input data	Remote SE not responding	3%
During execution	Job aborted (insufficient WN memory or AliRoot problems)	10%
	Job cannot start (missing application directory)	
	Job killed by CE scheduler	
	WN or global CE malfunction (all jobs on a given site die)	
Saving output data	Local SE not responding	2%

Table 5.6: *Main error types during job execution in Phase 2*

By allowing for real-time monitoring, MonALISA offers the possibility to point out the specific moment an error has occurred and the type of the problem accordingly. In cooperation with the administrators of the computing centres, the predominant causes of problems have been eliminated and the site stability improved. By the time of the end of Phase 2, the local site efficiency has increased by 20% and the failure rates have become very low.

5.4 Phase 3 - Distributed Analysis

The goal of Phase 3 (September 2004 - January 2005) is the distributed analysis of signal and underlying events produced during Phase 1 and 2. The users query the AliEn file catalog for the data relevant for their analysis. The results of the queries are stored as a single file collection, which is then used by the Job Optimizer to construct sub-jobs with groups of files, located at a given SE. The Job Broker submits the sub-jobs to the CE closest to the SE where the files to be processed are stored. All sub-jobs are run in parallel and deliver their results to the Merger job, which in turn returns the final result to the user. This is shown in schematic form in Fig.5.7. In this context, user interface and analysis application are ROOT and AliRoot, whilst the Grid middleware is used to steer the jobs and allocate the necessary computing resources.

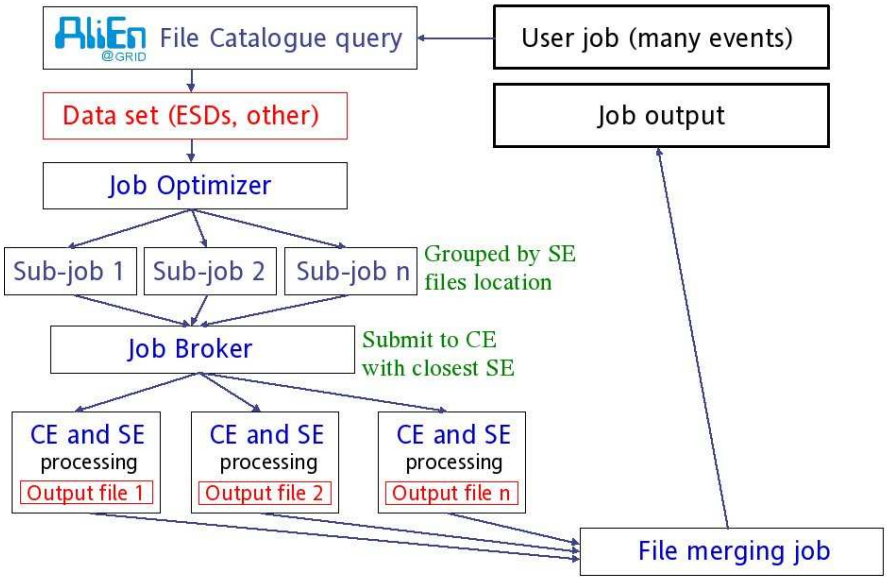


Figure 5.7: *Schematic view of Phase 3*

This phase has been characterized by efforts to tune monitoring parame-

ters via the MonALISA agents and testing light monitoring user applications sending performance data via the MonALISA AppMon API, embedded in the user job wrapper.

5.5 Analysis of the monitored data

In this section we present the analysis of the monitored parameters of jobs and AliEn Grid. The description follows the logical job processing steps and reflects the structure of the web Repository menu (Fig. 4.12).

5.5.1 Job Information

Job analysis is the most critical part of the AliEn monitoring. Jobs are tracked throughout their entire lifetime through several checkpoints inserted in the job processing path. Monitoring of the job flow has been used by the users and the computing site administrators to understand and solve network or resources allocation problems, or by the AliEn developers to optimize the job scheduling and synchronization mechanisms. In the current section we will run over the analysis of the monitoring results for each job status, whose meaning has been explained in section 4.2.1.

Waiting jobs

Throughout the entire Phase 1 the amount of waiting jobs was very low. The peak observed between June 29 and October 12 (see Fig.5.8) coincides with Phase 2 of PDC'04 and was due to the limited amount of CPUs at the computing centres. This amount was increased subsequentially and the value of waiting jobs returned to zero.

It is desirable, that there are low or zero waiting jobs in the AliEn Task Queue which signifies sufficient amount of computing resources available for processing.

Assigned jobs

The duration of the time interval during this status has revealed to be too short to be significantly monitoredi: picture 5.9 shows how the scale goes just up to 4 jobs.

Running jobs

Figure 5.10 shows that the number of jobs running in parallel was significantly oscillating through the entire period of PDC'04. In Phase 1 the the amount of free CPUs was significant, which allowed for a high number of

Figure 5.8: *Waiting jobs: full history*

Figure 5.9: *Assigned jobs: full history during PDC'04*

jobs to run in parallel. In Phase 2, the conditions changed due to the competition from other experiments running on the same resources. The zero jobs in the period April 20 and June 10 to 20 are due to technical pauses in the processing. Phase 3 has seen lower amount of running jobs due to development for distributed analysis and the end of massive computation.

Figure 5.11 is the result of a job analysis realized by the distributions charts built on top of the ROOT environment using Carrot as web interface, as described in the previous chapter. It shows the distribution of the number of running jobs for Phase 2 of PDC'04. This variable depends on several factors, with the most important of them being the number of waiting jobs

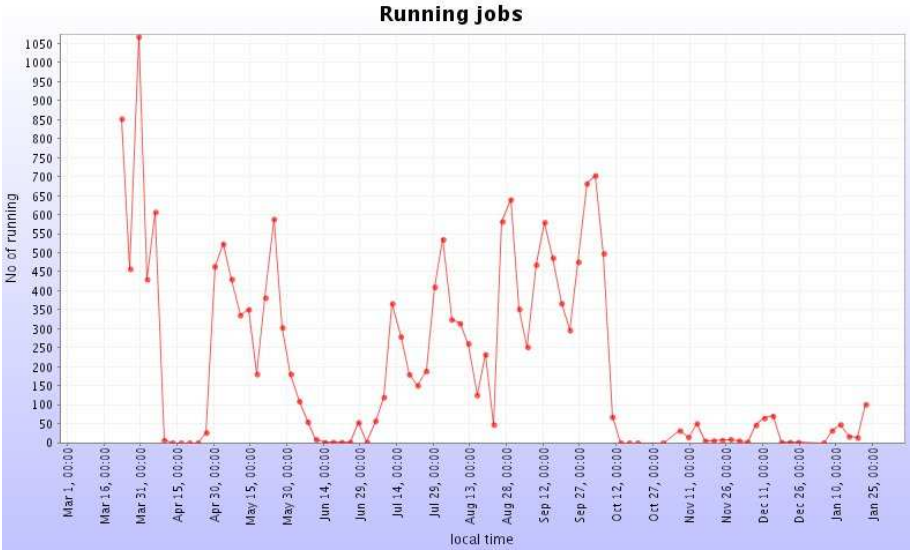


Figure 5.10: *Variation of running jobs during PDC'04*

in the Task Queue (TQ) and the availability of free CPUs of the remote computing centres. For this period, the TQ had always sufficient number of jobs waiting for execution. Under such circumstances the number of running jobs distribution is a direct measurement of the availability of CPUs at the remote centres.

Figure 5.12 shows the occupancy at a selected site, expressed in term of ratio between number of jobs running and maximum number allowed. It changes as a function of the number of queued job in the local batch system. Actually different batch schedulers (PBS, LSF, BQS) have different latency in scheduling jobs for execution and optimization of the number of jobs in the local queues is necessary to achieve a maximum occupancy with running jobs.

The example refers to the farm of CCIN2P3 in Lyon, France. One can see that there is an increase of the occupancy as more jobs are waiting in the queue and a saturation is reached around 60 jobs. The maximum number of running and queued jobs is specified as an LDAP parameter per each CE depending on the results from the analysis and the specification provided by the system administrators.

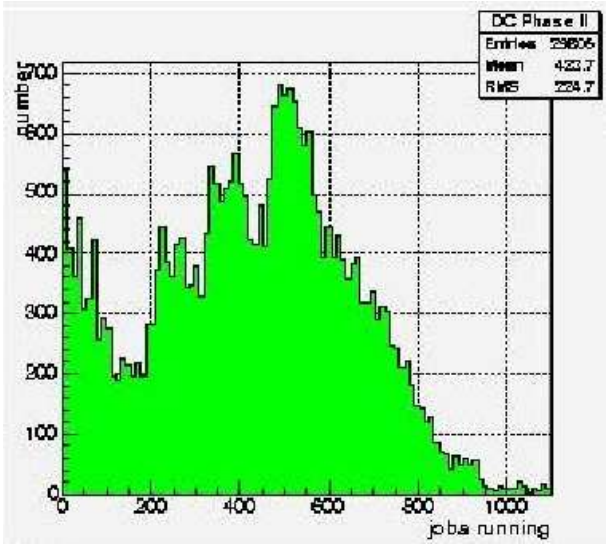


Figure 5.11: *Distribution of the number of running jobs for Phase 2 of PDC'04*

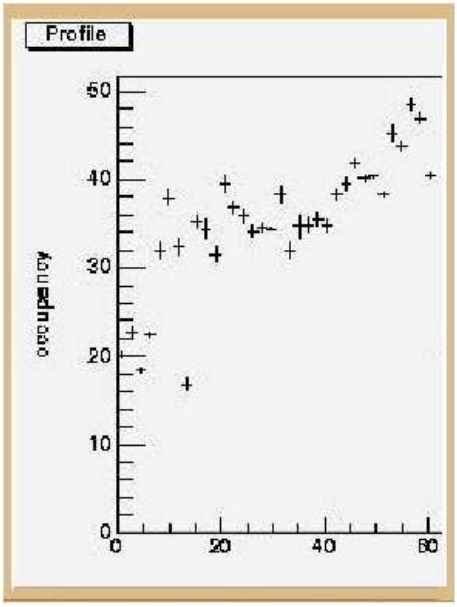


Figure 5.12: *Site occupancy as a function of queued jobs*

Done jobs

They indicate cumulative values and represent the level of completion of the tasks of PDC'04, since the number of jobs is known for all conditions

in PDC'04. As the values for the done jobs are taken from the AliEn process tables, and these tables can be reset to zero at any given moment of time, a dedicated code was written to enhance the MonALISA Repository functionality. Through this code, the resets of the AliEn process tasks was eliminated and the MonALISA Repository values kept consistent with the real number of done jobs. The source code is shown in Appendix A.2.4.

Failed jobs and killed jobs have the same incremental behavior of done jobs and they have been handled in the same way, as shown by the three plots in Fig. 5.13.

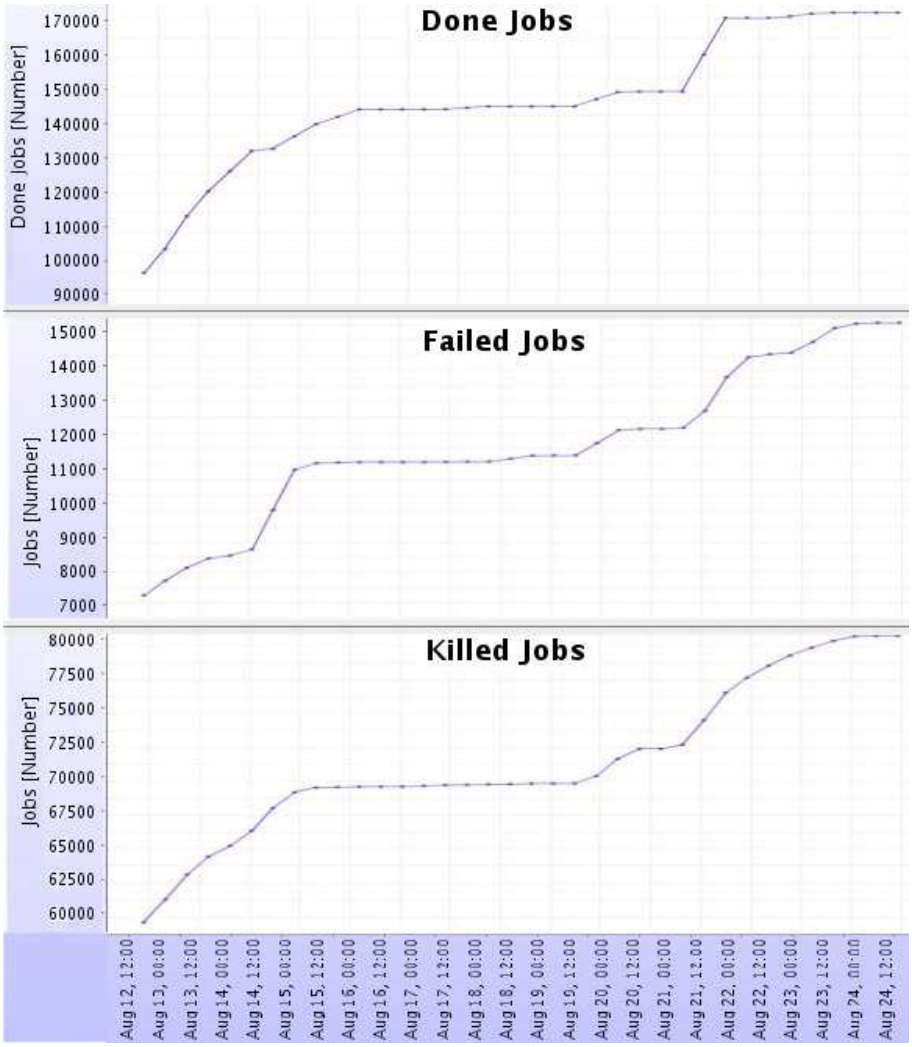


Figure 5.13: *Done, Failed and Killed jobs during the cetral part of PDC'04 Phase 2*

Error submitting

They represent 1% of Done jobs, mostly depending on CEs scheduler not responding.

Error executing

This error indicates that the job was aborted due to one of the following factors:

- killed by the local scheduler due to exceeding the allocated memory or time;
- could not start the application software (AliROOT, ROOT);
- problems with the shared software area. This particular type of failure can be extracted from individual job and fixed by the computing center experts.

Error saving

Failures during saving of the output files are tracked by the "error saving" procedure. Error of this type usually indicates a failure of the local site SE. It may necessitate intervention on the SE service, or the underlying storage system (NFS, MSS).

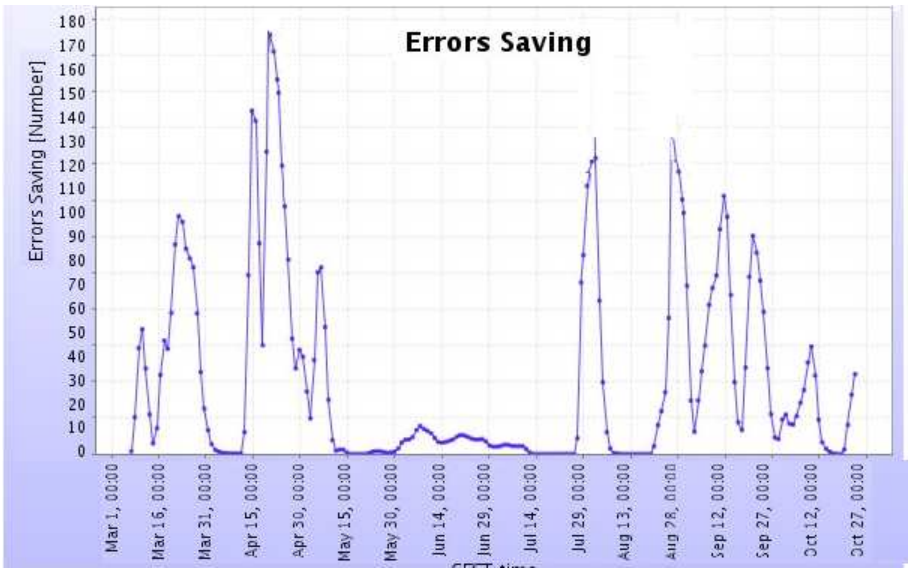


Figure 5.14: *Error Saving jobs: variation during PDC'04*

Error running

The percentage of these errors is about 10% of the running jobs. This error condition can occur if the Worker Node goes down or because the job runs out of memory.

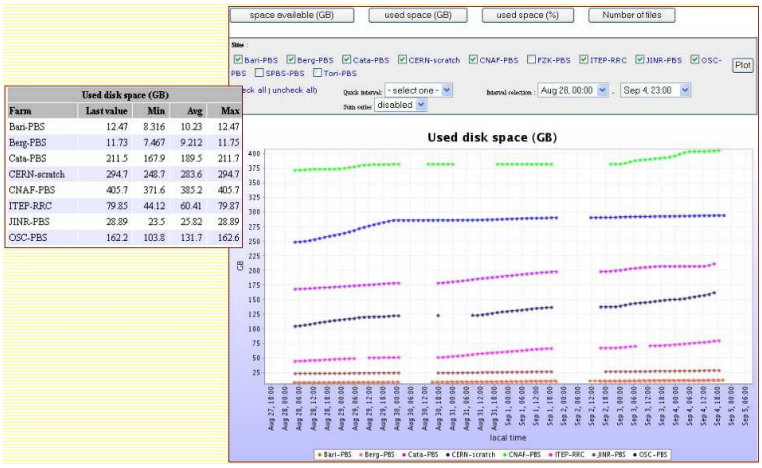
Error validation

By using a special script, at the end of the job all output files are checked for consistency (size, MD5 sum) and the log files are parsed for specific keywords like "segmentation violation" or "floating point exception". If these keywords are present, the job is not validated.

5.5.2 SE Information

We have noticed that, to fulfill the growth of activity, some sites (FZK, Torino-PBS, CERN-Scratch) have doubled the initial capacity of their SE: unfortunately the number of jobs in *saving* status has not always risen proportionally up. Starting from Phase 2 we have begun monitoring the occupancy of CASTOR (at CERN and CNAF), seeing that it has constantly increased because backup of reconstruction tasks during Phase 2.

The table in figure 5.15 shows minimum, maximum and average used disk space, in gigabytes, at the selected size and the plot gives an idea of the constant growth even in a short period of time. The visualization of the number of files has revealed to be useful during Phase 1 to determine the physical upper limit of the CASTOR stager.



Used disk space (GB)				
Farm	Last value	Min	Avg	Max
Bari-PBS	12.47	8.316	10.23	12.47
Berg-PBS	11.73	7.467	9.212	11.75
Cata-PBS	211.5	167.9	189.5	211.7
CERN-scratch	294.7	248.7	283.6	294.7
CNAF-PBS	405.7	371.6	385.2	405.7
ITEP-RRC	79.85	44.12	60.41	79.87
JINR-PBS	28.89	23.5	25.82	28.89
OSC-PBS	162.2	103.8	131.7	162.6

Figure 5.15: *Monitoring of available and used disk (tape) space at all local SEs at the remote sites*

5.5.3 CE Information

Special gauges have been used to observe in any moment the running load percentage per each site and verify the level of usage and stressing. The main two load factors have been computed by the ratios between running jobs and maximum allowed to run and, likewise, queued jobs and maximum allowed.

Figure 5.16 shows the real time monitoring of site occupancy. The dials can show either the percentage or the values according to the maximum number of jobs they are allowed to run. Moreover, each dial reports the last timestamp the monitored information refers to. They have represented a direct and ease tool to detect underloaded or overloaded sites. Bari, Catania and JINR have generally run always at the maximum capacity.

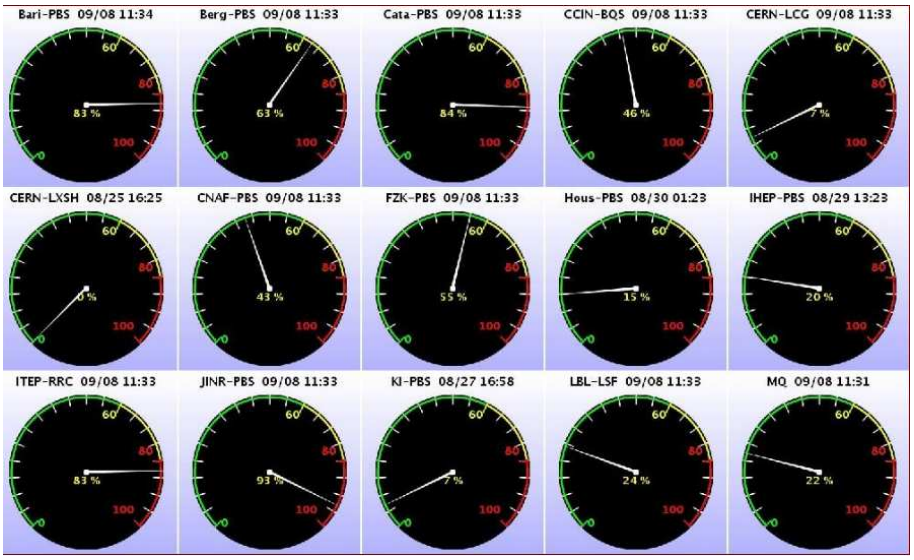


Figure 5.16: *Real time monitoring of sites occupancy*

5.5.4 CERN Network Traffic

In this section we have monitored the number of transferred files inward and outward CERN, other than the amount of traffic in MBs. Figure 5.17 shows the asynchronous data gathering of network traffic and data volumes at the three ALICE load-balanced servers at CERN.

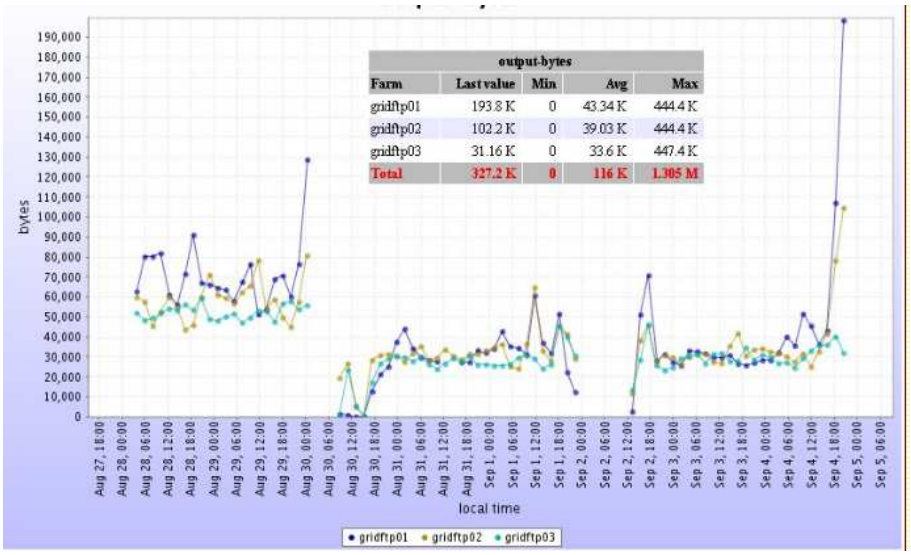


Figure 5.17: *Asynchronous (SOAP) data gathering of network traffic and data volumes*

5.5.5 LCG Job Information

The LHC Computing Grid (LCG) is an entire foreign Grid interfaced to AliEn. Monitoring tools have been provided by the LCG support and the MonALISA Repository has confermed its flexibility being easy to be interfaced to external tools. Figure 5.18 focuses on the number of jobs running in parallel at LCG remote sites: this number has revealed to be a significant percentage of the overall resources in ALICE's Grid. Additional LCG histories provide the number of allocated CPUs and queued jobs.

5.5.6 Summary Plots

This section has the aim to group and compare various monitoring parameters producing various plots depending on real time or user selection of time interval. A distinction between running and cumulative parameters has allowed to resolve the initial confusion due to different scales. Figure

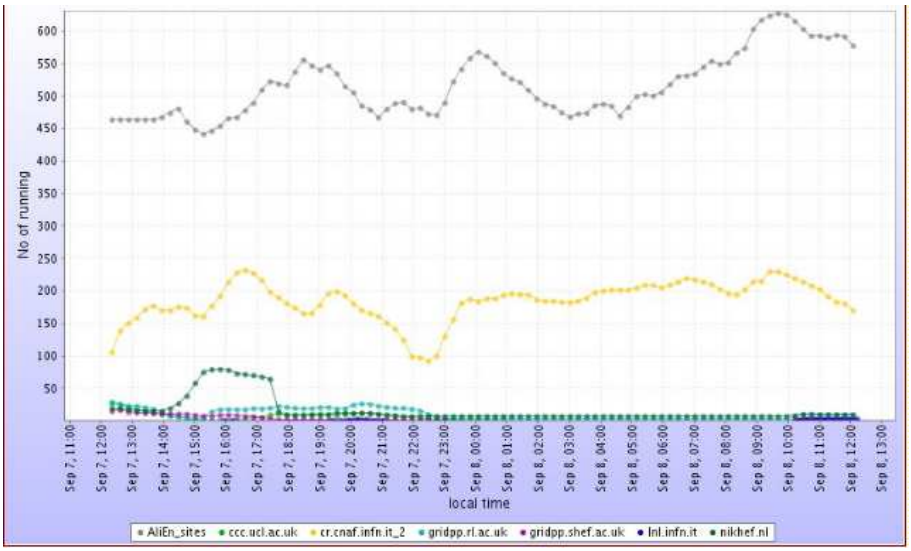


Figure 5.18: *Running jobs in LCG*

5.19 shows the Data Challenge completion of each phase.

Figure 5.19: *PDC'04 tasks completion*

5.5.7 Efficiency

Efficiency plots have been implemented to compare the 2 Grids (AliEn and LCG) and understand if their different philosophy of functioning (*pull-mode* against *push-architecture*) effects the overall performances. Figure 5.20 shows job execution, system and AliRoot measures of efficiency for AliEn, LCG and overall, but the same measures can be produced and compared per single sites, as shown in Figure 5.21.

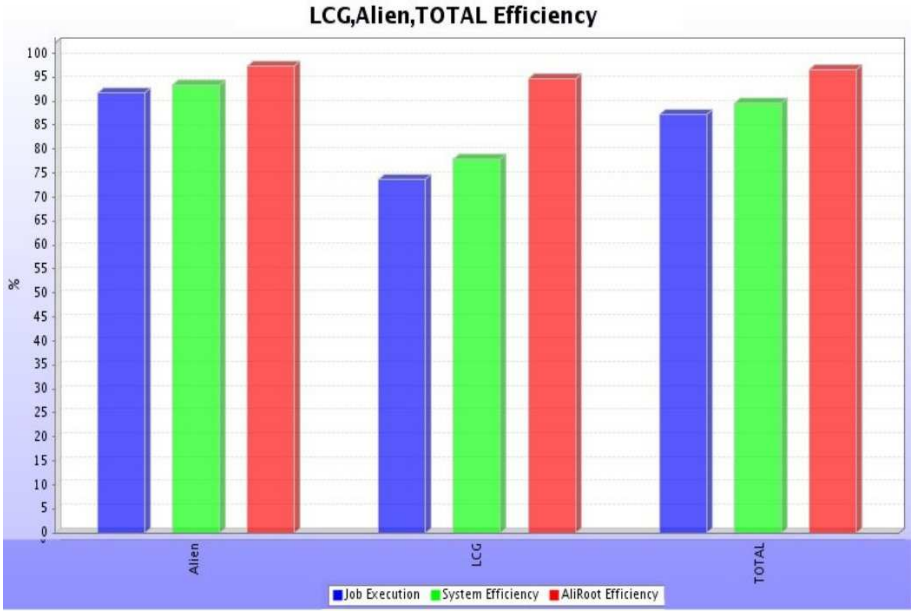


Figure 5.20: *Phase 1 - Groups Efficiency*

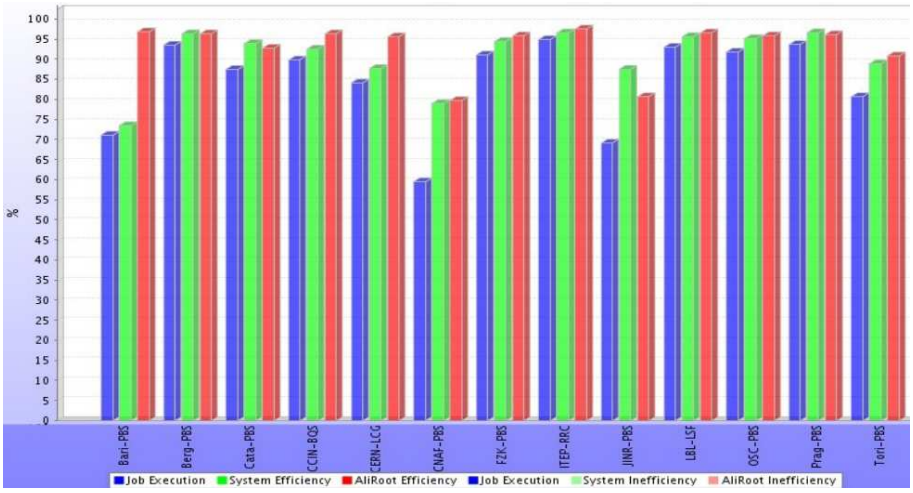


Figure 5.21: *Phase 1 - Sites Efficiency*

5.5.8 AliEn Parameters

Monitoring AliEn parameters has been useful for control and debugging of the central servers. Figure 5.22 tracks the memory usage by AliEn Perl modules. Measures of MySQL load, network traffic for internal communication, and disk consumption are also available for a complete monitoring.

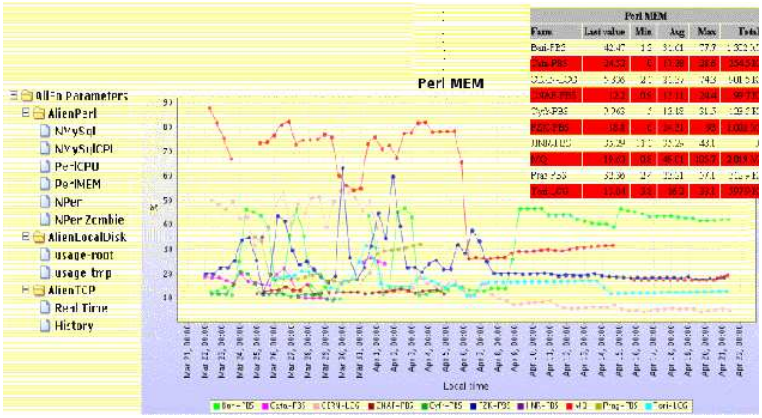


Figure 5.22: *Monitoring of AliEn parameters*

5.5.9 MonALISA Agents

As well as the MonALISA GUI, the Repository provides plenty of site-related useful monitoring parameters to understand what is happening at the remote Computing Element where the agents are running. MonALISA agents run into a Java Virtual Machine whose memory consumption in MB sometimes has seemed to be too intensive, as shown in Fig.5.23.

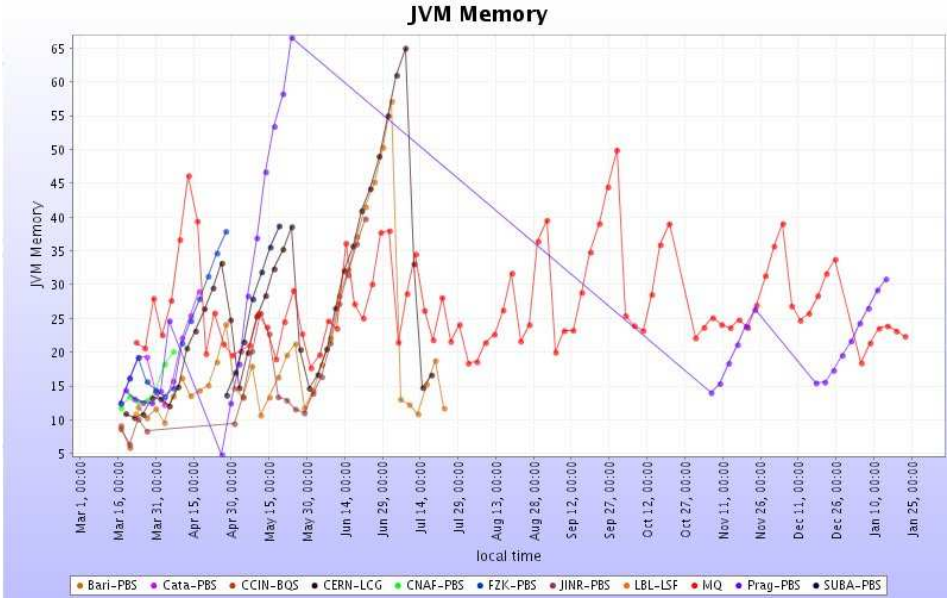


Figure 5.23: *JVM memory usage*

The Repository plots the memory that Java has allocated to the Virtual Machine and the remaining free memory. These are the values reported by the JVM itself using the java totalMemory() and freeMemory() functions. The memory value plus some fixed amount for the Java Runtime (JRE_memory) gives the real memory usage. Memory + JRE_memory corresponds to the value showed by the top command per each Java process, being the amount of memory that all of them share.

Chapter 6

Conclusions and Outlook

6.1 Lessons from PDC'04

During PDC'04, a large amount of events have been simulated to exercise the reconstruction of the particle identity and trajectory and the physics analysis. Practically, user jobs have been running for nine months using AliEn and since Phase 3 using the ARDA E2E analysis prototype and gLite [11].

AliEn constant improvements have been often implemented following the feedback coming from the monitoring of the data production and from increasingly complex user requirements: for example, the need of more functionality for the users involved into the data production (such as job handling and job resubmission) and the necessity of a stable and ad-hoc tuned monitoring tool.

MonALISA has been a flexible and complete monitoring framework successfully adapted to the needs of the Data Challenge and has given the expected results for performance tuning and load balancing in order to define the basic for a further Grid simulation project.

A step by step approach has clarified how, once jobs are digested by the system without long delays and at a high percentage of success, it is possible to switch from resources tuning to resources optimization, in order to find out and solve possible bottlenecks and defects of the system.

MonALISA framework has been able to gather, store, plot, sort and group any kind of data either basic or derived in a rich set of presentation formats. The Repository has proved to be a very useful tool as well as a helpful basis for the current Data Challenge. It's the only source of historical information and its flexible architecture has made possible custom modules

development as well.

The ALICE Offline computing model has been extensively tested and validated during the PDC'04, that has demonstrated the AliEn design scalability and has been a real example of successfull Grid interoperability by interfacing AliEn and LCG.

The framework and duration of the PDC has illustrated how many of the challenges encountered would not have shown in a short Data Challenge, such as operational problems of the Grid and Computing Element infrastructure for extended periods of time.

Last but not least, as expected the most challenging part has been the multi-user operation during phase 3: middleware protection is necessary in several areas and the monitoring capabilities should move down to a deeper level (from CEs to WNs) and provide user-related information being (hopefully) source of useful feedbacks to the gLite developers.

6.2 Outlook

Thanks to these results, the perspectives for the next future are very stimulating: the possibility to develop and improve a monitoring framework on top of a real functioning Grid, has provided and is providing an unprecedented experience to massively test the involved software technologies that are novel and in constant evolution.

Of all technologies deployed and tested, none has shown a clear dominant position, and they resulted complementary and, to some extent, interchangeable. The entire distributed monitoring system be built on top of Open Source components (OpenLDAP, perl5, MySQL, Jini to list some of them) and emerging standards makes it very flexible. In particular it is very easy to extend it, and to replace components with equivalent ones following the technical needs or the strategic choices that will emerge during its utilization.

Appendix A

Source Code

A.1 Fundamental source code

In this section we list the fundamental source code, either java classes or Linux shell scripts, developed in order to set the MonALISA framework up and fulfill the ALICE Data Challenge 2004 needs. See chapter four for a full explanation of each module.

A.1.1 AlienCMD.java

```
import lia.Monitor.monitor.*;
import java.io.*;
import java.util.*;
import java.net.InetAddress;

public class AlienCMD extends cmdExec implements MonitoringModule
{
   String[] tmetric; // a dynamic array of tag elements
   int NR;
   String cmd;
   String args;

   public AlienCMD()
   {  super("AlienCMD");
      info.ResTypes = tmetric;
      System.out.println( "Start the Interface to the Alien CMD module" );
      isRepetitive = true;
   }

   public MonModuleInfo init( MNode Node, String args )
   {  this.Node = Node;
      this.args = args;

      int ix = args.indexOf(";");
      if ( ix < 0 )
      {  System.out.println( " Input error in Alien CMD ... no tag / cmd delimiter !" );
         return info;
      }

      String tags = args.substring( 0, ix );
      cmd = args.substring(ix+1).trim();

      StringTokenizer tz = new StringTokenizer(tags,",");
      NR = tz.countTokens();
      tmetric = new String[NR];
      for ( int j=0; j < NR; j++ )
      {  String tag = tz.nextToken().trim();
         tmetric[j] = tag;
      }
      info.ResTypes = tmetric;
      return info;
```

```
42          }
43
44          public Object doProcess () throws Exception
45          { BufferedReader buff1 = procOutput ( cmd );
46
47             if ( buff1 == null )
48             { System .out .println ( " Failed to get the AlienCMD output " );
49                throw new Exception ( " AlienCMD output buffer is null for " + Node .name);
50             }
51             return Parse( buff1);
52          }
53
54          public Vector Parse ( BufferedReader buff ) throws Exception
55          { Result rr = null ;
56             Vector v = new Vector();
57             rr = new Result (Node.getFarmName (), Node .getClusterName (), Node.getName (), null, tmetric );
58
59             String lin;
60             rr.time = (new Date()). getTime();
61
62             try
63              { for ( int i=0; i <NR ; i++ )
64                { lin = buff .readLine ();
65                   if ( lin == null ) break;
66                   if ( lin .equals( "") ) break;
67                   String val = lin .trim();
68                   rr.param [i] = ( new Double (val)).doubleValue ();
69                }
70                buff .close ();
71                if ( pro != null ) pro.destroy ();
72             } catch ( Exception e )
73             { System .out. println ( "Exception in Parsing AlienCMD output Ex=" + e );
74               throw e;
75             }
76
77             // build a new Result
78             Result r = new Result( Node.getFarmName (), Node.getClusterName (),Node .getName(), null,null);
79             r.time = rr.time ;
80
81             for (int i=0; i<rr. param_name.length ; i++)
82             { if (rr.param [i]>0)
83                r.addSet(rr .param_name[i], rr .param[i ]);
84             }
85
86             v.add (r);
87             return v;
88          }
89
90          public MonModuleInfo getInfo () { return info ; }
91
92          public String[] ResTypes () { return tmetric ; }
93
94          public String getOsName () { return "linux"; }
95
96          static public void main ( String [] args )
97          { String host = "localhost " ;
98             AlienCMD aa = new AlienCMD ();
99             String ad = null ;
100            try { ad = InetAddress .getByName ( host ). getHostAddress (); }
101            catch ( Exception e )
102                { System. out.println ( " Can not get ip for node " + e );
103                  System. exit (-1);
104                }
105
106            MonModuleInfo info = aa .init( new MNode (host ,ad , null, null),
107                                          "tag1,tag2 ;echo -e \"10 \\n <file ://\\n> 20 \" ");
108
109            try { Object bb = aa. doProcess (); }
110            catch ( Exception e ) { System. out.println ( " failed to process " ); }
111         }
112      }
```

A.1.2 Monitoring.sh

```
#!/bin/sh
############################################################          #########  #########  #
# a script for retrieving MonALISA Monitoring Parameters
############################################################          #########  #########  #

Monitoring_Tcp()
{
    result=`netstat -t| wc -l | awk '{print $1}'`;
    if [ -n "$result" ] ; then echo "$result "; else echo "0"; fi
}

Monitoring_Disk()
{
    result=`df $1 |tail -1| awk '{printf ("%d \n %f \n",$4 ,$5); }'`;
    if [ -n "$result" ] ; then echo "$result "; else echo "0"; fi
}

Monitoring_SE()
{
    if [ $2 == "${ALIEN_HOSTNAME}.${ALIEN_DOMAIN}" ] ; then
        nfiles =`$1/bin /soapcall http://$2 :$3 /AliEn /Service /SE getLVMDF `;
    else
        nfiles ="0 0 0 0 0 0 0 0";
    fi;
    if [ -z "$nfiles" ] ; then nf=-1; nfree =-1;usage =-1; else typeset -i nf=` echo $nfiles |
        awk '{print $6}'`; typeset -i nfree =`echo $nfiles |
        awk '{printf("%d\n", $4); }'`; typeset -i usage =`echo $nfiles |
        awk '{printf("%d\n", $5); }'`; fi;
    echo -e "$nf \n $nfree \n $usage ";
}

Monitoring_CE()
{
        name=` hostname -f`;
        info=` $1/bin/soapcall http ://$name :8084 /AliEn/Service /ClusterMonitor getQueueInfo $2`;
        oldremember ="1 ";
        remember ="1";
        n_assigned=0; n_done=0; n_error_a=0; n_error_e=0; n_error_r=0; n_error_s=0; n_error_sv=0;
        n_error_v=0; n_error_vn=0;n _expired=0; n_idle=0; n_interactiv=0; n_killed=0; n_queued=0;
        n_running=0; n_saving=0; n_started=0; n_waiting=0; n_zombie=0;

        for name in $info ; do
        oldremember =$remember ;
        remember =$name ;
            if [ $oldremember = "RUNNING " ];    then n_running=$remember ;    fi
            if [ $oldremember = "WAITING " ];    then n_waiting=$remember ;    fi
            if [ $oldremember = "ZOMBIE" ];      then n_zombie=$remember ;     fi
            if [ $oldremember = "STARTED " ];    then n_started=$remember ;    fi
            if [ $oldremember = "QUEUED" ];      then n_queued=$remember ;     fi
            if [ $oldremember = "IDLE" ];        then n_idle=$remember ;       fi
            if [ $oldremember = "INTERACTIV " ]; then n_interactiv=$remember ; fi
            if [ $oldremember = "DONE" ];        then n_done=$remember ;       fi
            if [ $oldremember = "SAVING" ];      then n_saving=$remember ;     fi
            if [ $oldremember = "EXPIRED " ];    then n_expired=$remember ;    fi
            if [ $oldremember = "ERROR_A" ];     then n_error_a=$remember ;    fi
            if [ $oldremember = "ASSIGNED " ];   then n_assigned=$remember ;   fi
            if [ $oldremember = "KILLED" ];      then n_killed=$remember ;     fi
            if [ $oldremember = "blocked " ]; then
               if [ $remember = "open" ]; then n_blocked=1; else n_blocked=0; fi
            fi
            if [ $oldremember = "ERROR_E" ];     then n_error_e=$remember ;    fi
            if [ $oldremember = "ERROR_S" ];     then n_error_s=$remember ;    fi
            if [ $oldremember = "ERROR_R" ];     then n_error_r=$remember ;    fi
            if [ $oldremember = "ERROR_V" ];     then n_error_v=$remember ;    fi
            if [ $oldremember = "ERROR_VN" ];    then n_error_vn=$remember ;   fi
            if [ $oldremember = "ERROR_SV" ];    then n_error_sv=$remember ;   fi
        done;
        printf "$n_assigned \n$n_done \n$n_error_a \n$n_error_e \n$n_error_r \n
                $n_error_s \n$n_error_sv \n$n_error_v\n$n_error_vn \n$n_expired \n
                $n_idle \n$n_interactiv \n$n_killed \n$n_queued \n$n_running \n
                $n_saving \n$n_started \n$n_waiting \n$n_zombie \n$n_blocked\n";
}

Monitoring_MasterQueue()
{
        $1/bin /alien login -exec queueinfo | grep "Sum of all Sites " |
        awk '{printf("%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n
                      %d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d \n%d\n",
                     $8 ,$9,$10, $11,$12,$13 ,$14,$15 ,$16,$17 ,$18,$19, $20,$21,$22 ,$23,
                     $24 ,$25,$26 ,$27,$28 ,$29,$30, $31); }'
}
```

```

Monitoring_MasterQueueJobRunning ()
{
        $1/bin/alien login -exec queueinfo | grep "Sum of all Sites" |
        awk '{ if ($8 <0) $8=0;
         if ($13 <0) $13 =0;
         printf("%d \n", $13); }'
}

Monitoring_MasterQueueLoad ()
{
    $1/bin/alien login -exec queue list | grep "All" | awk '{print $2,$3,$4 ,$5 }' |
    cut -d "/" -f1 | awk '{printf("%s \n%s \n%s \n", $1,$2 ,$3); }';
    $1/bin/alien login -exec queue list | grep "All" | awk '{print $2,$3,$4 ,$5 }' |
    cut -d "/" -f2 | awk '{printf("%s \n%s \n", $1,$2); }';
    $1/bin/alien login -exec queue list | grep "All" | awk '{print $2,$3,$4 ,$5 }' |
    cut -d "/" -f3 | awk '{printf("%s \n", $1); }';
}

Monitoring_Zombie()
{
    result=` ps -C $1 o comm,state | grep $1 | grep -v Monitoring |grep -w Z | wc -l |
    awk '{print $1 }'`;
    if [ -n "$result" ] ; then echo "$result "; else echo "0"; fi
}

Monitoring_Procs()
{
    result=` ps -ax |grep $1 | grep -v Monitoring | grep -v grep | wc -l| awk '{print $1 }'`;
    if [ -n "$result" ] ; then echo "$result "; else echo "0 "; fi
}

Monitoring_ProcMemCpu()
{
    result=` ps -C $1 o comm,%mem ,%cpu | grep -v grep | grep -v Monitoring |grep $1 |
    awk '{nproc ++;mem+= $2;cpu+= $3; printf("%d \n %f \n %f \n", nproc,mem ,cpu); }' | tail -3`;
    if [ -n "$result" ] ; then echo "$result "; else echo -e "0 \n 0 \n 0 \n ";fi
}

Monitoring_PerlMysqlInfo ()
{
    Monitoring_Zombie perl ;
    Monitoring_ProcMemCpu perl;
    Monitoring_Zombie mysql;
    Monitoring_ProcMemCpu mysqld ;
}

Monitoring_RootTmpDisk()
{
    Monitoring_Disk "/";
    Monitoring_Disk "/tmp";
}

Monitoring_PingHosts()
{
    self="${ALIEN_HOSTNAME}.${ALIEN_DOMAIN}";
    rm -f .tmp .secret; touch .tmp .secret;
    alien -x $1/getServiceHosts .pl $2:$3 SE > .tmp .secret ;
    alien -x $1/getServiceHosts .pl $2:$3 CLC >> .tmp.secret ;
    alien -x $1/getServiceHosts .pl $2:$3 CE >> .tmp.secret ;
    alien -x $1/getServiceHosts .pl $2:$3 FTD >> .tmp.secret ;
    alien -x $1/getServiceHosts .pl $2:$3 Services >> .tmp .secret;
    cat .tmp. secret | sort | uniq |grep -v $self;
    rm .tmp.secret ;
}

#################################################### ########## ########
Monitoring ()
#################################################### ########## ########
{
  eval `alien --printenv `
  cmd =$1;

  case `type -t Monitoring_$cmd` in
     function)
           shift 1
           Monitoring_$cmd $*
            ;;
  esac

  exit
}

Monitoring $*;
```

A.1.3 Queueinfo.sh

```
#/bin /bash
cd ‘dirname  $0 ‘

proc_name ()
{ shift $1
  echo $1
}

$ALIENROOT  /bin /alien  login  −exec queueinfo   > /tmp /queueinfo
$ALIENROOT  /bin /alien  login  −exec queuelist   > /tmp /queuelist

# STEP  1) queueinfo   information  ..............................................................             .
echo ‘date‘  "; QueueInfo   lines: " ‘cat /tmp /queueinfo   | wc −l‘ >> log .out
for value  in ‘cat /tmp /queueinfo   | cut −c −24,40− |
    awk ’/Alice ::/ {printf("%s  %s  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d  %d
                    %d  %d  %d\n",
                    $1 , $3"000"  , $4,$5,$6  ,$7,$8,$9  ,$10,$11  ,$12,$13,  $14,$15,$16  ,$17,$18  ,$19,$20  ,
                    $21 ,$22,$23  ,$24,$25,  $26); }’ |
    cut −d ":" −f3,5‘;  do
   pos =‘expr index  "$value " ’:’‘;
   if [ $pos  −gt  0 ]; then
      i=0
      farmname  =${value :0:$pos  −1}; farmname  =${farmname  :0:4 }
      nodename  =${value :$pos }; nodename  =${nodename  :0:4 }
    else
      if [ $i −eq  0 ]; then
         rectime  =$value
         i=‘ expr $i  + 1‘
       else
         nvalue =‘proc_name $i proc_inserting proc_waiting proc_assigned proc_queued proc_started
                proc_running proc_saving proc_done proc_zombie proc_failed
                proc_error_a proc_error_i proc_error_e proc_error_r proc_error_s proc_error_sv
                proc_error_v proc_error_vn proc_expired proc_idle proc_interactiv proc_killed
                proc_error_ib‘

         if [ $value  −gt  0 ]; then
            echo −e $farmname  "−"$nodename  "\tAlienMQ  \tlocalhost  \t"$nvalue  "\t" $value"  \t"$rectime
            if [ $nvalue  == "proc_running" ]; then
               echo −e $farmname  "−"$nodename  "\tALICE \tlocalhost  \tjob_running\t"$value  "\t"$rectime
            fi
         fi

         i=‘ expr $i  + 1‘
      fi
   fi
done

# STEP  2) queue  list  information  ..............................................................             .
rectime  =‘date  +"%s"‘000
cat /tmp/queuelist    | cut −c 8− | awk −v rectime=$rectime    ’BEGIN  { myrectime  =rectime;   i=0 } /::/  {
     sitename    =substr ( substr( $1,1,index  ($1,  ":" )−1), 1, 4) "−" substr ($1,   index($1  , ":")+2,4)
     status      =$2
     running     =substr ($7,1,index  ($7,  "/ ")−1);
     maxrunning  =substr ($7,   index ($7,  "/ ")+1);
     if (maxrunning  >0) runload=(  running/  maxrunning  )∗100
     else runload  =0
     i++

     if (running  <0) running  =0

     if (i>1) {
       if (status  =="lock  −err")
          printf("%s \t%s \t%s \t% s \t%s \t%s \n",  sitename  , "AlienMQload  ", "localhost  ", "pie_run",
                 1 , myrectime  );
       if (status  =="open  ") {
          if (running==0)
             printf("% s \t%s \t%s \t%s \t%s \t%s \n",  sitename  , "AlienMQload  ", "localhost  ", "pie_maxrun",
                    1,  myrectime  );
          else {
             printf("% s \t%s \t%s \t%s \t%s \t%s \n",  sitename  , "AlienMQload  ", "localhost  ", "pie_run"   ,
                    running     , myrectime  );
             printf("% s \t%s \t%s \t%s \t%s \t%s \n",  sitename  , "AlienMQload  ", "localhost  ", "pie_maxrun",
                    maxrunning  , myrectime  );
          }
       }

       if (runload  >0) {
          printf("%s \t%s \t%s \t% s \t%s \t%s \n",  sitename  , "AlienMQload  ", "localhost  ", "running  "   ,
                 running     , myrectime  );
          printf("%s \t%s \t%s \t% s \t%s \t%s \n",  sitename  , "AlienMQload  ", "localhost  ", "maxrunning  ",
                 maxrunning  , myrectime  );
```

```
82            printf("%s \t%s \t%s \t%s \t%s \n", sitename , "AlienMQload ", "localhost ", "runload " ,
83                    runload , myrectime );
84        }
85
86      }
87  }'
88
89  # STEP 3) LCG information ..................................................... ......... .
90  rectime =`date +"%s"`000
91  /afs /cern.ch/ project/gd /www/eis /tools/lcg -CEInfoSites --vo alice ce --is lxb2006 .cern .ch |
92   awk -v rectime =$rectime 'BEGIN { myrectime =rectime } /-alice / {
93       pos_dot=index($6, "."); pos_colon=index ($6,": ");
94       sitename =substr($6 ,pos_dot+1, pos_colon-pos_dot-1);
95       if (sitename =="cr .cnaf.infn .it") sitename =sitename "_"substr($6 ,pos_dot-3,1);
96       if ($2 >0) printf("%s \t%s \t%s \t%s \t%s \t%s \n", sitename , "LCG_CE", "localhost ", "lcg_free" ,
97                         $2, myrectime );
98       if ($3 >0) printf("%s \t%s \t%s \t%s \t%s \t%s \n", sitename , "LCG_CE", "localhost ", "lcg_running",
99                         $3, myrectime );
100      if ($4 >0) printf("%s \t%s \t%s \t%s \t%s \t%s \n", sitename , "ALICE" , "localhost ", "job_running",
101                         $4, myrectime );
102      if ($5 >0) printf("%s \t%s \t%s \t%s \t%s \t%s \n", sitename , "LCG_CE", "localhost ", "lcg_queued" ,
103                         $5, myrectime );
104    }'
105
106 # STEP 4) AliEn = Sum of all Running except LCGs ..........................................
107 cat /tmp/queueinfo | grep -v LCG | grep -v '\-\-\-\-' | cut -c 103-106 |
108     awk -v rectime =$rectime 'BEGIN {myrectime =rectime ; sum =0} {sum= sum+$1 }
109          END {if (sum >0) printf("%s \t%s \t%s \t%s \t%s \t%s \n", "AliEn_sites", "LCG_CE", "localhost ",
110                              "lcg_running" , sum , myrectime )}'
```

A.1.4 DirectInsert.java

```
1  package lia.Monitor .JiniClient .Store;
2
3  import lia.Monitor .monitor .Result ;
4
5  import java.util .Vector ;
6  import java.util .Timer ;
7  import java.util .TimerTask ;
8  import java.util .StringTokenizer ;
9
10 import java.io .BufferedReader ;
11 import java.io .OutputStream ;
12 import java.io .InputStreamReader ;
13
14 public class DirectInsert extends TimerTask {
15
16     Vector vBuffer ;
17     String sProgram ;
18     Timer timer ;
19     boolean bOnlyPositives ;
20
21     public DirectInsert (String sProgram, long lInterval , boolean bOnlyPositives ) {
22         this.sProgram = sProgram ;
23         this.bOnlyPositives = bOnlyPositives ;
24
25         vBuffer = new Vector ();
26         timer = new Timer ();
27         timer. scheduleAtFixedRate (this, 0, lInterval );
28     }
29
30     public void run() {
31         int iLine = 0;
32         String sLine = null;
33         StringTokenizer st2;
34         Result r;
35
36         try {
37             StringTokenizer st = new StringTokenizer ( getProgramOutput (sProgram ), "\n" );
38             double d;
39
40             while (st.hasMoreTokens ()) {
41                 iLine ++;
42                 sLine = st .nextToken ();
43
44                 try {
45                     st2 = new StringTokenizer (sLine, "\t");
46                     r = new Result(
47                         st2.nextToken (), // farm
```

```
                    st2.nextToken (), // cluster
                    st2.nextToken (), // node
                    null,             // module
                    null);            // param_name

                  String sFunc = st2.nextToken ();
                  d = Double.parseDouble (st2.nextToken ());

                  if (bOnlyPositives && d<=0D) continue;

                  r.addSet (
                    sFunc, // function
                    d);
                  r.time = Long.parseLong (st2.nextToken ());
                  vBuffer.add(r);
                }
                catch (Exception e) {
                  System.err.println ("DirectInsert (" +sProgram +") ignoring exception at input " +
                                      "line : "+iLine );
                  System.err.println ("The line was : '"+sLine +"'");
                  e.printStackTrace ();
                }
            }

            System.err.println ("DirectInsert ("+sProgram +") : "+vBuffer.size());
        }
        catch (Exception e) {
            System.err.println("DirectInsert ("+sProgram +") caught exception at input line : "+
                               iLine);
            System.err.println("The line was : '"+ sLine+"'" );
            e.printStackTrace ();
        }
    }

    public Vector getResults () {
        Vector vTemp = new Vector ();

        synchronized (vBuffer ) {
            vTemp.addAll (vBuffer );
            vBuffer.clear ();
        }

        return vTemp;
    }

    protected String getProgramOutput ( String sProgram ) {
        try {
            Runtime rt = Runtime.getRuntime ();

            String comanda [] = new String [1];
            comanda [0] = sProgram ;

            Process child = null;
            child = rt.exec( comanda );

            OutputStream child_out = child.getOutputStream ();
            child_out.close ();

            BufferedReader br = new BufferedReader (new InputStreamReader (child.getInputStream ()));
            StringBuffer sb = new StringBuffer (20000);
            char cbuff[]= new char[10240];
            int iCount = 0;
            do {
                iCount = br.read( cbuff);
                if (iCount > 0)
                sb.append (cbuff, 0, iCount );
            }
            while (iCount >0);

            child.waitFor ();

            return sb.toString ();
        }
        catch (Exception e) {
            System.err.println(e.toString ());
            e.printStackTrace ();
            return "";
        }
    }
}
```

A.2 Additional code development

This paragraph runs briefly over the additional code that has been developed to add some features to the MonALISA framework. These improvements have been generally asked by the ALICE's analysts to provide a better interaction with the Web Repository and its monitoring capabilities.

A.2.1 ABping

ABping is the name of an internal module used by the MonALISA framework to perform simple network measurements using small UDP packages. The graphical effect, within the Web Repository or the GUI client, is the visualization of (bi)-directional lines indicating the connections quality using different colors and numeric scales. Being the links showing the ABPing measurement customizable from a configuration file (that can be stored either at an url address or local folder), it has been developed an interactive web form to select or de-select measurements at sites of interest.

A.2.2 Dials

By producing chart types not supported by the MonALISA Web Repository we have experienced how easy is to interface and improve its graphical capabilities, mostly based on JFreeChart java libraries. Dials have revealed to be useful to monitor the occupancy factor at the remote sites and to display the completion of the three major phases of PDC'04, as shown in sections 4.5.2 and 5.5.6 respectively.

A.2.3 Colors

Charts produced by the Web Repository have been intensively used during the whole 2004 as a source for many talks and documentations. As such, in order to be as clear as possible, it's important that plots showing information of many sites at the same time can clearly distinguish sites of interest by using different colors.

MonALISA provides an automatic mechanism to associate colors to sites. That mechanism is based on a color palette and, the more the sites number grow up, the more it's easy to confuse them because similar colors. That's why an additional web module has been developed to allow the Repository administrator to select the more appropriate colors and obtain the best view.

A.2.4 DBBrowser

DBBrowser is a java servlet developed for browsing the monitoring data stored within the Repository database. The main reason depend on the

necessity to have a look at the very rough historical data coming from the remote sites and have an exact idea of their values, otherwise averaged by the plot mechanisms.

Whether enabled, this feature allows to interact with the running history and, by just clicking by the mouse on the lines within the charts, shows the numeric detail of the monitored data.

A.2.5 WebQueueInfo

This module is in charge to display in the Web Repository the same snapshots about user job status at the remote site provided by AliEn native commands such as queueinfo or queue list.

Bibliography

[1] AP. Saiz, L. Aphecetche, P. Buncic, R. Piskac, J.-E. Revsbech and V. Sego, "AliEn-Alice environment on the GRID," *Nuclear Instruments and Methods in Physics Researh Section A: Accelerators, Spectrometers, Detectors and Associated Equipment,* Volume 502, Issues 2-3, 21 April 2003, Pages 437–440.

[2] P. Saiz, P. Buncic, A. Peters, "AliEn Resource Brokers", CHEP03, http://arxiv.org/abs/cs/0306068

[3] L. Betev, P. Buncic, M.Meoni, V. Pinto Morais, A. Peters, P. Saiz, P E. Tissot-Daguette "AliEn Web Portal", CHEP04, http://cern.ch/mmeoni/alice/chep04/AliEnWEBPortal.pdf

[4] "ALICE Technical Proposal for A Large Ion Collider Experiment at CERN LHC", CERN/LHCC/95-71, 15 December 1995.

[5] http://aliweb.cern.ch/offline

[6] http://carrot.cern.ch

[7] http://castor.web.cern.ch

[8] http://www.cern.ch

[9] http://aliceinfo.cern.ch/Collaboration/Documents/TDR/Computing.html

[10] http://ganglia.sourceforge.net/

[11] http://glite.web.cern.ch

[12] I. Foster, C. Kesselmann, "Globus: A metacomputing infrastructure toolkit," *The Grid - Blueprint for a new Computing Infrastructure,* pages 15-51. Morgan Kaufmann, 1999.

[13] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, R. Wolsky, "A Grid Monitoring Architecture", March 2000 - Revised 27 August 2002.

[14] Global Grid Forum Open Grid Service Infrastructure Working Group charter and document. http://www.gridforum.org/ogsi-wg/.

[15] http://java.sun.com/products/javawebstart

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

[17] The Jini Web Page. http://www.jini.org

[18] M. Wahl, T. Howes, and S. Kille, "Lightweight Directory Access Protocol (v3),". Available from ftp://ftp.isi.edu/in-notes/rfc2251.txt.

[19] http://monalisa.cacr.caltech.edu

[20] http://monalisa.cacr.caltech.edu/MONARC

[21] http://dev.mysql.com/doc/refman/5.0/en/replication.html

[22] I. Foster, C. Kesselmann, J. Nich, S. Tuecke, "The physicology of the Grid: An open grid services architecture for distributed systems integration," January 2002.

[23] http://www.r-gma.org

[24] http://root.cern.ch/

[25] http://www.w3.org/2002/ws/

[26] http://www.globus.org/wsrf/

[27] R. Housely, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure," IETF RFC 2459, Jan. 1999