ÉCOLE POLYTECHNIQUE
Promotion X2002
Levillain Olivier

Stage d'Option Scientifique

Comparison of several grid monitoring tools

Rapport Non Confidentiel

Option : Informatique
Champ de l'option : Architecture des Ordinateurs et Parallélisme
Directeur de l'option : Gilles Dowek
Directeur du stage : Jesús Labarta
Dates du stage : 2 Mai - 15 Juillet 2005
Adresse de l'organisme : Universitat Politècnica de Catalunya
Dep. d'Arquitectura de Computadores
Jordi Girona, 1-3
Mòdul D6 Campus Nord
08034 Barcelona (SPAIN)

Comparison of several grid monitoring tools

Olivier Levillain [*] Francesc Guim Bernat [†] Ivan Rodero [†]

olivier.levillain@polytechnique.org guim@ac.upc.edu irodero@ac.upc.edu

Julita Corbalán [†] Jesús Labarta [†]

juli@ac.upc.edu jesus@ac.upc.edu

Abstract *Grids allow large scale resource-sharing accross different administrative domains. Those diverse resources are likely to join or quit the grid at any moment, or possibly to break down. Thus grid monitoring is a complex task. The current usage of monitoring service is not very far from the usage of information service as they essentially track down breakdowns and give a more precise idea of the resource availability and load. However, job-oriented monitoring can be used for accounting purposes. Fine-grain resource-oriented monitoring can also allow more precise forecasting for resource allocation policies. The aim of this paper is to compare some existing monitoring tools, mainly on the following points: fine-grain resource monitoring, extensibility and integrability, possible solutions for job-oriented monitoring.*

Résumé *Les grilles de calcul permettent le partage de ressources très variées au travers de plusieurs domaines administratifs. Ces ressources sont susceptibles de rejoindre ou de quitter la grille à tout moment, et peuvent éventuellement tomber en panne. Ainsi, la surveillance de la grille est une tâche complexe. L'utilisation des outils de surveillance ressemble aujourd'hui à celle des systèmes d'information car ils servent à repérer les pannes et à donner une idée plus précise des ressources disponibles et de leur charge. Cependant, la surveillance au niveau des* jobs *peut être utilisée à des fins de décompte. La collecte d'informations détaillées sur les ressources permet également de meilleures prévisions pour les politiques d'allocation des ressources. Le but de ce rapport est de comparer quelques outils de* monitoring, *en particulier sur l'existence de données détaillées sur les ressources, les possiblités d'extension et d'intégration, et les solutions envisageables pour l'observation des* jobs.

[*] École Polytechnique, Palaiseau, France

[†] Departement d'Arquitectura de Computadores, Universitat Politècnica de Catalunya, Barcelona, España

Contents

Abstract **3**

1 Introduction **5**

1.1 The Grid 5

1.2 GGF and the Grid Monitoring Architecture 7

1.3 How to use a Monitoring Service 10

2 Tools presentation **12**

2.1 Monitoring and Discovery Service (Globus) 12

2.2 Mercury (GridLab) 14

2.3 Network Weather Service 19

2.4 G-PM/OCM-G (CrossGrid) 22

2.5 Ganglia 25

2.6 GRIA 28

3 Comparison **29**

4 Related work **31**

5 Conclusion **31**

A Installing Mercury **32**

B Adding a new sensor module in Mercury **34**

C Installing Ganglia **38**

D A simple job monitor with Ganglia **42**

References **44**

1 Introduction

In this report, we will introduce some existing grid monitoring tools and describe their features. We are interested in information such as the kind of resource monitored, at which level, the kind of API provided, and the way they can be modified to fit our requirements.

The remainder of the document is organized as follows: in this first section, we will first introduce the concept of grid computing and a monitoring architecture proposed by the GGF (Global Grid Forum, an organisation trying to define grid standards). Then we will define the goal of this study. The second section is dedicated to the presentation of several existing monitoring tools which will be compared in a table under different sorts of criteria, in section 3. The fourth section is about other papers comparing different grid monitoring tools. Finally, section 5 concludes this paper. Some appendices give details about specific work done with two monitoring tools, Mercury and Ganglia.

1.1 The Grid

Grid computing is a very general term describing a lot of different computing architectures, from internal grids (large commercial enterprises internal networks organized to fully eploit their computing power) to "a service for sharing computer power and data storage capacity over the Internet" (definition given by the CERN).

The way grids work has been summarized by the following points in [6]:

Resource sharing where resource can be computation power, storage facilities or specific software.

Security including access policies, authentification, authorization and data encryption.

Efficient use of resources.

Death of Distance thanks to high speed networks.

Open standards by the Global Grid Forum (GGF) which is a sort of standards body for grid-specific standards.

Actually, the Grid is a way to present to users a virtual cluster embedded in a distributed infrastructure, able to solve massive computational problems too big for any single supercomputer, or to provide a multi-user and flexible environment to work on multiple smaller problems.

Grid computing is often confused with cluster computing. The key differences are that clusters are homogeneous in the sense their scale of resources is small and these resources are all known; On the contrary, grids are very **heterogeneous** in their scales of resources, sub-architectures and systems, and the resource availability vary dynamically and so there is a need for information services to follow the existing resources; also, grids spread out and encompass user desktops while clusters are generally confined to data centers; finally, grids are meant to be extended **accross different administrative domains**.

To handle the variety of resources and domains, grids need *middleware* applications which represent the brain of the grid, whereas the different computation resources would be the muscles and the networks connecting those resources would be the nervous systems.

These middleware applications have to be distributed, flexible and scalable, to be fault tolerant and to be able to face the diversity of resources. They also need to use secure authorization techniques so as to allow remote users to control and monitor only their jobs and the resources these jobs are using.

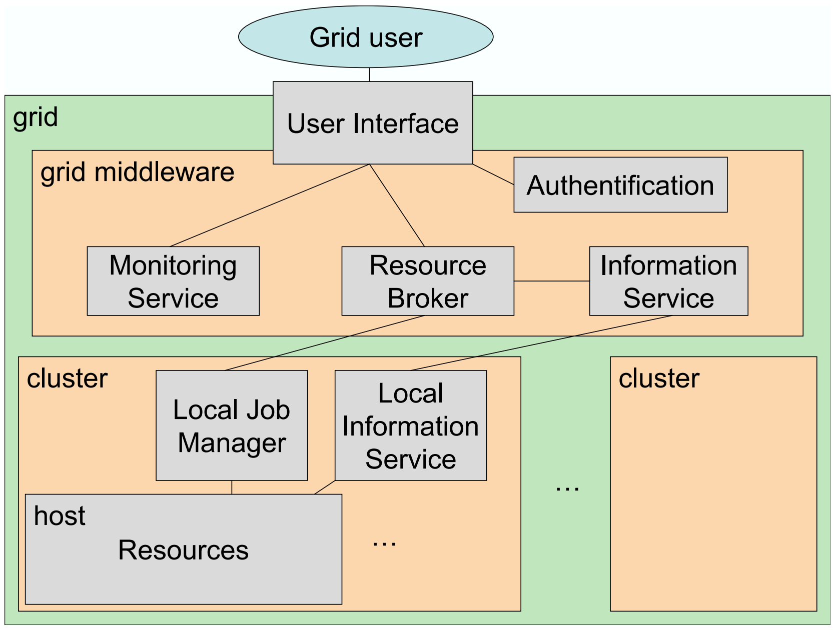


Figure 1: Grid general organization and middleware applications

Figure 1 shows the general organisation of grid computing. To understand the different roles played by the grid components, let's follow the life of a job on the grid.

First, the user connects to the User Interface, which requires an authentification. Once authorized, the user can submit a job, i.e. a program (e.g. a 3D rendering ray tracer), the data it uses (the scene and the textures) and a description of the job (the filenames, the owner) and its requirements; these requirements can concern hardware (architecture, minimum memory space needed, etc) and software (libraries or programs needed).

Then the Broker will try and find the best resources to meet all the requirements of the Job Description; it grabs information from the Information Service, a distributed component listing the available resources of the grid.

When resources are found, the job is submitted to a cluster[1] where the Local Job Manager executes the program, according to the Job Description. At the end of the job, the result is sent back to the Broker and to the user through the Interface.

During the execution of the job, the user can check which resources have been allocated to his job and known for how long it had been running; this information is collected and brought to the user by the Monitoring Service. When the application is instrumented, it is even possible with some Monitoring Tools to measure the performance and to forecast the remaining time, as described in [7, 8, 16].

1.2 GGF and the Grid Monitoring Architecture

The Global Grid Forum (GGF) is a community of users, developers and vendors aiming at defining specifications for grid computing. The forum was created in 2001 from the merging of different forums around the world. The Globus Alliance is a community of organizations and individuals implementing these standards through the Globus Toolkit, which has become the de facto standard for grid middleware.

The GGF Performance Working Group has developped a model for grid monitoring tools called Grid Monitoring Architecture (GMA) in [21]. The architecture they propose is designed to address the characteristics of Grid platforms.

[1]In this document, we call a cluster every administrative domain handled by the grid.

The paper first lists which requirements such a monitoring tool should meet. Then it presents the GMA model.

Requirements

Low latency for delivering data and high data rate Performance information has a fixed, often short, lifetime of utility. What is more, performance data are often more frequently updated than requested, whereas usual database programs are firstly designed for queries. This means permanent storage is not always necessary, and that the tool must be able to answer quickly before the data is obsolete.

Scalablity and Portablity A grid performance monitoring tool also needs to handle many different types of resources accross different administrative domains and should be able to adapt when communication links or other resources go down. The monitoring system should then be distributed.

Extensibility Depending on the use of the Grid, administrators and users may want to monitor non standard entities and would like to add them. A monitoring tool should provide a simple interface for adding new metrics[2].

Security As users can only have access to their files, it should be the same for the performance information given by the monitoring tool. So the tool has to include authentification and authorization services to define access policies.

Minimum intrusiveness What we finally expect is to keep up the performance we are monitoring. That is why we must pay attention to every resource used by the performance tool (processors, memory, storage and network).

There is often a compromise to find sometimes between these requirements. For example, using active sensors to measure network performance is more portable than passive OS-specific methods but also more intrusive.

Let's now look at the structure described in the GGF's paper.

Architecture proposed

The GMA is based on three types of components, producers, consumers and the directory service (fig. 2).

[2]*Metrics* is a term often used for *monitored data*, as described further; for example, the number of CPUs or the free memory are metrics supported by most tools

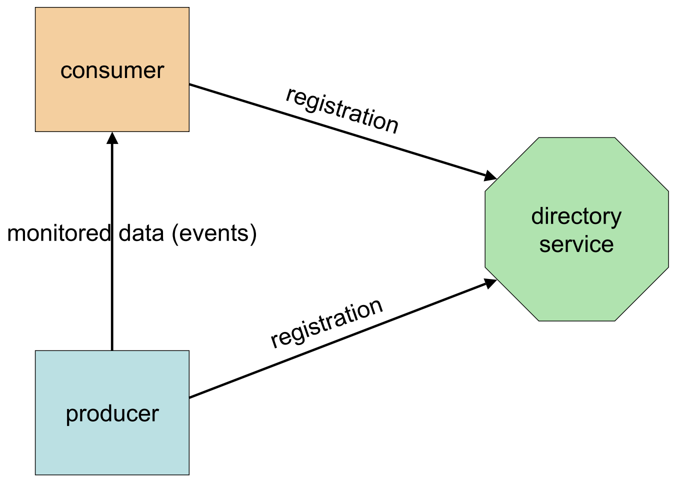


Figure 2: Grid Monitoring Architecture components

A producer is any component that can send events to a consumer, using the producer interface (accepting subscription, queries and ability to notify). In a monitoring tool, every sensor is encapsulated in a producer; however a producer can be associated to many different sources: sensors, monitoring systems, databases. A consumer is any component that can receive event data from a producer. The consumer interface contains subscription / unsubscription routines and query mechanisms.

To exchange data events, producers and consumers have a direct connexion, but to initiate the dialog, they need the directory service. When a consumer or a producer starts, it registers itself in the directory service. Then when a consumer is looking for a producer to get data, it has to perform a research in the directory service. Thus, it is possible to bootstrap correctly communication in a complex architecture and to incorporate security, because directory service takes into account understood wire protocols and security mechanisms. The directory service is in fact a distributed component, locally accessible from every other components.

This architecture provides a flexible way of distributing the system. It is possible to have a hierarchical distribution, with complex components having both consumers and producers interfaces (for example, a monitoring service can gather information from many producers/sensors, process them and present to the grid user a producer interface). It is then allowed to have data discovery, processing, storage and delivering separated, and to be able

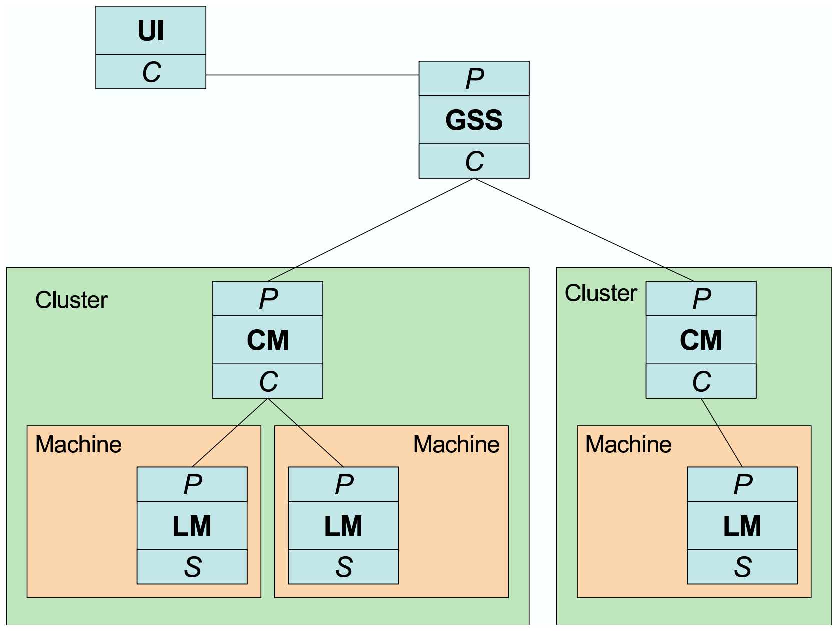


Figure 3: GMA example: S = Sensor(s); P = Producer Interface; C = Client Interface; LM = Local Monitor; CM = Cluster Monitor; GSS = Grid Storage Service; UI = User Interface

to furnish a consumer interface through a firewall. An example of what can be done following such recommendations is given fig. 3

We are now going to explain what we are aiming at doing with monitoring services.

1.3 How to use a Monitoring Service

Grid Monitoring Services is different from Grid Information Services, even if some toolkits as Globus furnish only one tool for both tasks. On the one hand, Information Service is supposed to register all the resources available in the grid. This corresponds to the Directory Service of the Grid Monitoring Architecture. Usually, Information Service works at cluster and grid level. On the other hand, Monitoring Service also works at host level, to get precise measurements thanks to host sensors and is looking for fresh and fine-grain data.

However, it is clear that both services need each other and that is why either they're grouped or there is a little Monitoring Service inside the Information one (and vice versa). For example, monitoring tools need to know all the existing resources to be able to display the state of the cluster to the user, and that's why it needs a Directory Service. Information tools are used to choose the resource to be allocated to a job, and sometimes it may communicate with the monitoring tool to get detailed data about resource load.

The current accounting mechanism is generally so simple as measuring the cputime consumed by each user using some standard UNIX log mechanism. However, this method doesn't take into account many aspects of the scheduling (e.g. we may want to pay attention to the time of day a job was scheduled or to the priorities system in a time-shared system) nor other resources than CPU (like memory, storage or TCP bandwith).

Similarly, many forecasting policies for launching new jobs are based on poor information (only CPU load and number of users) whereas it could be useful to know more fine-grain details about the use of all resources.

Thus our goal is to use some automatic mechanism to implement more accurate monitoring. Information obtained with this mechanism could be used in accounting, ensuring quality of service, or in forecasting, allowing the display of relevant information for the user. In both case, we can see the importance of having detailed resource monitoring information, and to know which job is using which resource: this is what we call job-oriented monitoring.

However, no monitoring tool can, alone, give job-oriented information. Indeed, job-oriented monitoring involve the cooperation between several grid middleware applications. A job is identified differently at grid, cluster and host levels. Yet, we would like to have host and resource information about jobs, and publish them at the grid level. We then have to let the monitoring service be able to bind jobs to their processes.

In [1], the Gridlab team propose a simple way to have monitoring at the job level. Once a job is submitted to the grid and the resources it will use are chosen, the processes of this job shall be locally executed on a new account, by a new Unix user. This allows the program to change its name and/or to fork, and still to be recognizable because the owner of the processes are uniquely associated to this job. What's more, security is enforced by the Unix users system.

Then, we only need to have a cooperation between the Monitoring Service and the Job Manager to get the correspondance between the GJID (Grid Job Id, identifying the job at the grid level) and the LJID (Local Job Id, identifying the job at the cluster level) from the one side, and between the Main Monitors and the Local Resource Mangement System to translate LJIDs in user names and so to find the corresponding PIDs and the CPUs used by every job.

Thus we see that job-oriented monitoring is possible if and only if the grid middleware is cooperating, hence there is a need of standards to plug different services together, and in this paper, we will try to give an idea of the possibilities of integration of each tool.

2 Tools presentation

2.1 Monitoring and Discovery Service (Globus)

Institution:	ANL, USC/ISI
Key persons:	Karl Czajkowski, Ian Foster, Carl Kesselman
Web page:	http://www.globus.org/toolkit/mds/
References:	[9, 12, 3]

MDS is the Grid Information Service provided in the Globus Toolkit. However it supports monitoring sensors to register to the Information Service. It supports a hierarchical architecture with Index Services aggregating the information collected from the Information Providers. Two version of MDS are currently available; MDS-2 uses the Open LDAP protocol wheras MDS-3 uses the Open Grid Services Infrastructure. MDS-4 is more or less the reunion of both older versions.

Architecture

MDS consists of two types of elements: information provider (producers) and data aggregation services. Fig. 4 represents this architecture; the GRIS are the producers (replaced by Service Data Elements in MDS-3) and are present at the cluster level. GIIS are in charge of the Directory Service (index of the resources available) and GRIS have to register regularly in GIIS which aggregate this information to publish it to the user or to other GIIS.

In MDS-2, the protocol used is LDAP and in MDS-3 it uses the Open Grid Services Infrastructure (OGSA). The data forma is LDIF (MDS-2) or XML (MDS-3).

As we can see, GIIS can regroup some clusters to form Virtual Organizations (VO) and can also be organized hierarchically. Authentification with GIIS is possible via LDAP or OGSA certificates.

Resource-level monitoring and host metrics

In fact, MDS does not really support fine-grain resource-level monitoring and host metrics. GRIS only collect information using scripts executed pe-

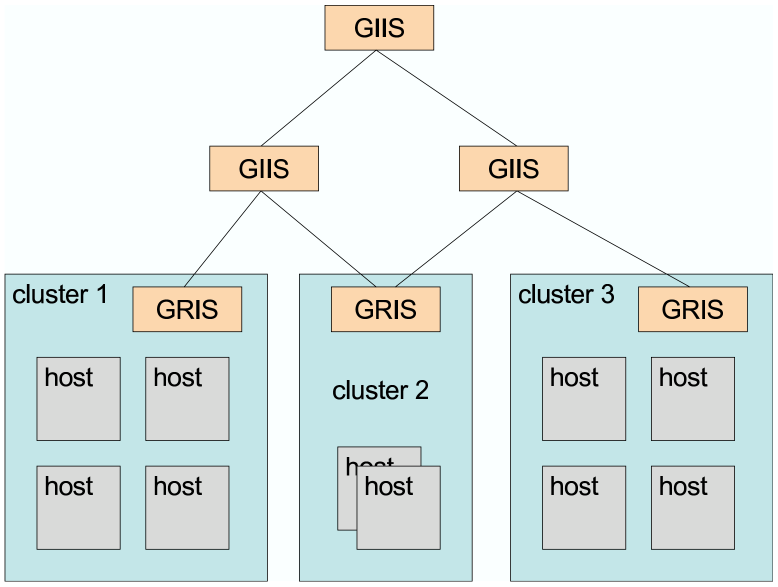


Figure 4: MDS-2 architecture; GRIS: Grid Resource Information Service, GIIS: Grid Index Information Service

riodically but it is also possible to add them because sensors can easily communicate with MDS thanks to known protocols.

However, MDS is more designed to be an Information Service and not a Monitoring tool, and the data given by the scripts is not updated frequently enough. Having fine-grain monitoring data would suppose to add sensors and to ask the GIIS hierarchy to refresh its data more often, which would certainly be very intrusive because it was not designed this way.

Developper Interface and Integration

To get a real monitoring system from MDS, we would need to program new sensors or try and integrate existing ones. Hopefully, adding sensors is possible, either by using the LDAP/OGSA interface, or by writing shell scripts. What is more, MDS-3 offers a standard data format through XML channels.

Security

Authentification with GIIS is possible via LDAP or OGSA certificates, but then users are authorized to see every resource in the VO. We will say that it is a VO-level authorization policy. Thus, beside this concept of VOs, the fine-grain access restriction in MDS could only be enforced outside the grid, by a portlet for example.

Data transmission can be encrypted through Secure Sockets Layer / Transport Layer Security (SSL/TLS) implementation.

Summary

MDS is an efficient Information Service and its architecture globally follows the Grid Monitoring Architecture, but it was not design to fit all the requirements needed for a Monitoring Tool. Indeed, even it may be extensible enough, the data are not supposed to be refreshed frequently enough for monitoring purpose. Different tools have then been developped to manage only the monitoring task and we are going to see some of them in the next sections.

2.2 Mercury (GridLab)

Institution:	MTA SZTAKI
Key persons:	Zoltán Balaton, Gábor Gombás, Péter Kacsuk
Web page:	www.gridlab.org/WorkPackages/wp-11, www.lpds.sztaki.hu
References:	[19, 1, 17]

GridLab is a European research project developping application tools and middleware for Grid environments. They produce a set of application-oriented Grid services and toolkits providing capabilities such as dynamic resource brokering, monitoring, data management, security, information, adaptive services and more.

Among them, the Mercury Monitor is designed to satisfy requirements of grid performance monitoring: it provides monitoring data represented as metrics via both pull and push access semantics and also supports steering by controls (also called actuators). It supports monitoring of grid entities such as resources and applications in a generic, extensible and scalable way. Its architecture is based on the GGF GMA, and implemented in a modular way with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system. [1] gives an overview of this architecture.

Appendix A is a brief manual to install and configure Mercury on your system.

Architecture

The design of the Mercury Monitoring System follows the recommendations of the Grid Monitoring Architecture (GMA) published by the Global Grid Forum [21].

The input of the monitoring system consists of measurements generated by sensors. Sensors are embedded in producers that manage the generated data and forward it to consumers. Sensors implement the measurement of one or more measurable quantities called metrics.

Mercury supports both event-like metrics when measurements are triggered by some external event (like log messages) and continuously measurable metrics when consumers must request the measurements explicitly (like CPU utilization). In the second case, as no measurement is made until explicitely requested, the intrusiveness is reduced. It is also possible to turn a continously measurable metric into a event-like one by requesting automatic periodic measurements.

Mercury extends the GMA by introducing actuators to enable the manipulation of monitored objects. As sensors regroup metrics, actuators implement one or more controls. But while metrics are used to gather data about an object without noticeably disturbing its behavior, controls are used to manipulate the monitored object.

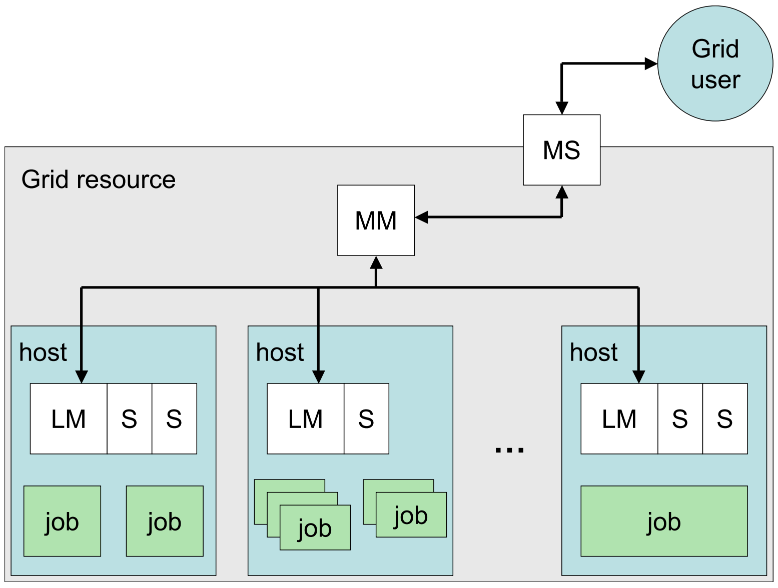


Figure 5: Mercury architecture; S: Sensor, LM: Local Monitor, MM: Main Monitor, MS: Monitoring Service

The Mercury Monitoring System has a distributed, hierarchical architecture (see fig. 5). A producer called the Local Monitor is running on every machine. It collects information about application processes and grid services running on that machine as well as information about the machine itself.

Local Monitors are supervised by Main Monitors. The Main Monitor distributes the requests it receives from remote consumers between the Local Monitors, and forwards any data generated by the Local Monitors back to the consumers. Local users can access those data directly by the Main Monitors. Grid users are provided the data via the Monitoring Service, which is a client of the Main Monitors.

Concerning reliability, Main Monitors - as Monitoring Services - can be duplicated, and share the same set of underlying Local and Main Monitors. Both duplicata won't share any data, but, as data is only asked on client demand, there won't be any significant overload due to this duplication.

This architecture with several layers allows a Main Monitor to work as a proxy, allowing data transmission even if parts of the grid are behind firewalls.

Resource-level monitoring and host metrics

First, Mercury can give general information on the system, like the architecture, the name and version of the operating system, the boot time or the list of the current users.

Concerning the CPU, Mercury can provide the following metrics:

- Number of CPUs (total installed, online and available),
- 1, 5 and 15 min. load averages,
- Frequency of the CPUs,
- Size of L1 and L2 cache,
- State of the CPUs ("Normal", "Offline", "Error", "PowerOff", "OS only", "User only" or "Restricted") and the date they enter this state, and
- CPU usage (per processor): time spent by the processor in
  - user context,
  - servicing system calls,
  - waiting for external I/O to complete,
  - idling,
  - servicing hardware and software interrupts, and

  - running niced processes.

Mercury also allows you to get information about disks and filesystems:

- Disks names, sizes and partitions
- Disks statistics
- Filesystems mounted
- Total, available and reserved space

Memory metrics are divided in 3 section:

- `mem` which gives the size and the usage of the memory (free memory, size used for IO buffers and for cached files).
- `swap` which describes the swap space
- `vm` for the virtual memory (memory moves, page faults)

Finally, Mercury's resource monitoring has a `net` section:

- List of interfaces with their status and addresses
- Statistics on incoming and outgoing traffic
- Counts of errors, drops and collisions

Job-oriented monitoring

As told in the introduction, the Gridlab suggests a solution to be able to monitor the grid at the job level. A way to do it is to add modules to help different services to communicate together.

We did not try to set up such a solution with Mercury, but appendix B describes a method to add new modules in Mercury. Based on such modules and on a communication between middleware applications, it could be possible to publish job-related information with the Monitoring System.

Application-oriented monitoring and application metrics

With Mercury it is also possible to have a more specific application monitoring. It is provided through the `app.*` metrics and is based on the GRM principles. GRM, GRid application Monitor [18, 2, 17], is an off-line monitoring tool providing application monitoring for one-process instrumented applications (i.e. applications with trace generation function calls) in the grid environment.

GRM is based on a Local Monitor / Main Monitor architecture, and is integrated into Mercury. [17] describes how this integration works. Application monitoring for instrumented applications allows Mercury / GRM to collect trace information. The main problem here is the need for an instrumentation of the application with GRM calls, which means that the job should be either recompiled with a specific compiler or instrumented by the submitter himself. What's more, GRM only provides an API for trace functions in C.

Developper Interface and Integration

The Mercury sources are open. The whole tool was developped in C. appendix A describes the installation of Mercury monitoring tool.

From the producer side, adding sensors and actuators is possible, as new modules (see appendix B). Even if programming a new module is a little hard at first (due to the large amount of C macros needed to define the structure of the module), the modular architecture of Mercury allows a simple way to add new components.

From the consumer point of view, we have to distinguish two kinds of consumers; if the client is inside the grid (for example the broker trying to forecast the use of the resources, so as to know whether to launch a new job or not), the API is in C. The protocols used are Mercury specific channels over TCP, using XDR (External Data Representation Standard, [20]) data format.

Users outside the grid should use the Monitoring Service which is integrated in a portlet which is part of the Gridlab project.

Security

By default, internally, anonymous IP authentification is used; though to make sure only authorized processes access monitored data, Mercury has a GSS-API authentification module, and it is also possible to write new modules for different authentification methods or to add an encryption layer (which is not included).

The GSS-API (decribed in [13]) is both transport and mechanism independent:

Transport independence means that GSS does not depend on a specific communication method or library. Rather, each GSS call produces a sequence of tokens which can be communicated via any communication library an application may choose. Transport layers to include raw TCP sockets, UDP, and Nexus.

Mechanism independence means that the GSS does not specify the use of specific security algorithms, such as Kerberos, SESAME, DES or RSA public key cryptography. Rather, the GSS-API is defined in terms of security operations. Each operation can be implemented via a range of different security mechanisms.

It is important to notice that Mercury authorization method does not apply at the cluster or Virtual Organization level, but at the resource level, through Access Control Lists, described in the configuration section of appendix B (in the Local and Main Monitor configuration files). It is thus possible to have very fine-grain access policies.

Grid security rules and local policies are also enforced into the Monitoring Service, allowing more dynamical access policies and possibly a cooperation with the Job Submitter.

Summary

The original version of Mercury allows us to monitor many interesting metrics, and authentification inside the grid is already possible. If you want to add metrics, controls, or authentification methods, you can write new modules in C, but the problem is a lack of documentation about the module format and the numerous C macros used in the existing modules.

So Mercury could be an interesting and evolutive solution for grid monitoring, in case you have time to invest in adding the modules you really need. The portlet GridLab provides for external Monitoring Service, cooperating with the job submitter may also be a very convenient solution to the job monitoring problem.

2.3 Network Weather Service

Institution:	University of California, Santa Barbara
Key persons:	Rich Wolski, Martin Swany
Web page:	nws.cs.ucsb.edu
References:	[22]

Network Weather Service (NWS) is a distributed system for producing short-term performance forecasts based on historical performance measurements. NWS provides a set of system sensors for periodically monitoring end-to-end TCP/IP performance (bandwidth and latency), available CPU percentage, and available non-paged memory. Based on collected data, NWS dynamically characterizes and forecasts the performance of network and computational resources.

Architecture

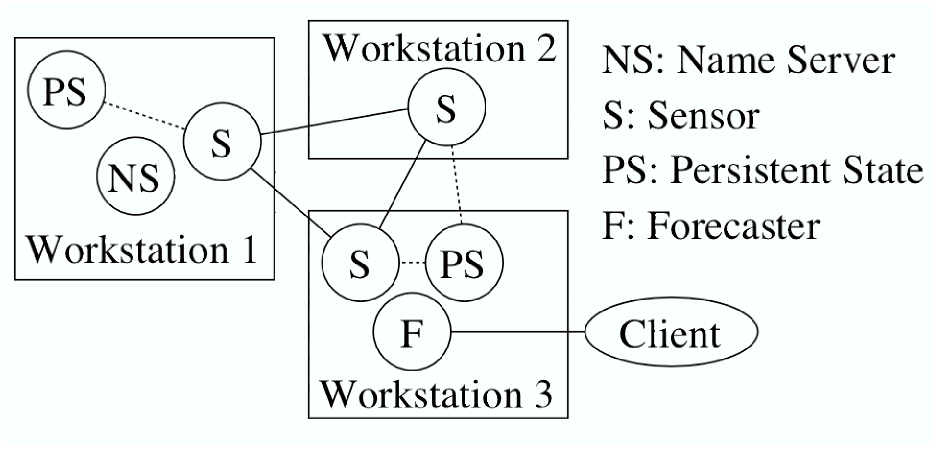


Figure 6: A NWS implantation sample

The NWS is composed mainly of four types of servers:

- **Directory service** (`nws_nameserver`) registering the active servers and the current monitoring activities. Every server must inform a name server of its presence, and every client must dialog with the name server to get monitoring data. The name server works with the LDAP (Lightweight Directory Access Protocol, [11]) protocol.
- **Persistent storage servers** (`nws_memory`) allowing sensors to keep previous measurements.
- **Resource monitors** (`nws_sensor`) providing monitoring skills and controls.
- **Forecasters** (`nws_forecast`) using previous measurements to estimate future resource availability.

An example of NWS implantation is described in fig. 6. We can see that NWS follows the GGF recommandation about the existence of the directory service, but it can not be distributed as in the Grid Monitoring Architecture. You can indeed have duplicated or different directory services, to create different Virtual Organizations (VO) as in Globus MDS, but all the information of one Name Server will be gathered on one host, which may be impossible in the grid context.

What is more, NWS does not provide a clear client / server interface, and so does not allow a hierarchical architecture across different administrative domains through firewalls. That is why we can say that NWS is more designed for clusters or internal grids than for the other types of grid.

Resource monitoring

Here are the resources NWS is able to monitor:

- **cpuMonitor** gives the fraction of CPU available for both newly-started and existing processes.
- **diskMonitor** indicates the space available on mounted partitions.
- **memoryMonitor** returns the amount of free memory available on the machine.
- **tcpConnectMonitor** monitors the time required to establish a TCP connection between a pair of machines.
- **tcpMessageMonitor** gives the bandwidth and the latency between a pair of machines.

Sensors combine active and passive monitoring methods. NWS provides periodic or single measurements, with forecasts when asked.

Developper Interface and Integration

NWS is a tool programmed in C. The code is not written in a modular way, so adding new sensors involves modifying many different files and hard-coded constants.

The tool is made of command line tools, and that is why it could easily be integrated as an accurate sensor tool in another monitoring tool.

Security

NWS, like Globus MDS, does not provide any authorization method to restrict the access to monitored data inside the Virtual Organizations. The authentification is made through LDAP certificates and then NWS apply a VO-level authorization. What is more, there is no portlet or other user interface furnished to enforce fine-grain access policies.

Summary

Finally, Network Weather Service does not seem to fit our needs, since its architecture is not as distributed as we need, and can not be organized in layers. Besides, NWS is not easy to modify. Yet it may be integrated as a local component in a more complex grid monitoring service.

2.4 G-PM/OCM-G (CrossGrid)

Institution:	Institute for Computer Science ICS-AGH and CYFRONET, Technische Universität München
Key persons:	Marian Bubak, Wlodzimierz Funika, Roland Wismüller
Web page:	grid.fzk.de/CrossGrid-WP2
References:	[4, 5]

The OMIS (On-line Monitoring Interface Specification [14]) is an API aiming at defining a standard interface for communication between various types of runtime tools for parallel and distributed systems (including monitoring services in the grid). It was developped in 1996-97 in the *Lehrstuhl für Rechnertechnik und Rechnerorganisation Institut für Informatik* in the Technische Univerität München. Today, OMIS is well documented, but doesn't seem to be much used in other grid projects than Crossgrid.

Later, the OCM-G (OMIS-Compliant Monitoring system for the Grid) project started in München, based on a Grid-enhanced version of OMIS. OCM-G is an application monitoring tool, which is part of the Crossgrid project. It provides configurable online monitoring via a central manager which forwards information requests to the local monitors. Crossgrid is also providing G-PM (Grid-oriented Performance Measurement tool) which is a graphical performance analysis tool that allows to request standard performance metrics as well as user-defined metrics at runtime. The measured data are periodically transferred from the monitor to the front end and visualized via various performance diagrams.

Architecture of OCM-G

As we can see in fig. 7, OCM-G has a distributed architecture, with Application Monitors (AM), linked into every Application Processes (AP) that is monitored; with Local Monitors (LM) to gather information at the host level; with Service Managers (SM) to collect data and to build the hierarchical structure of the monitoring system; and finally with a Main Service

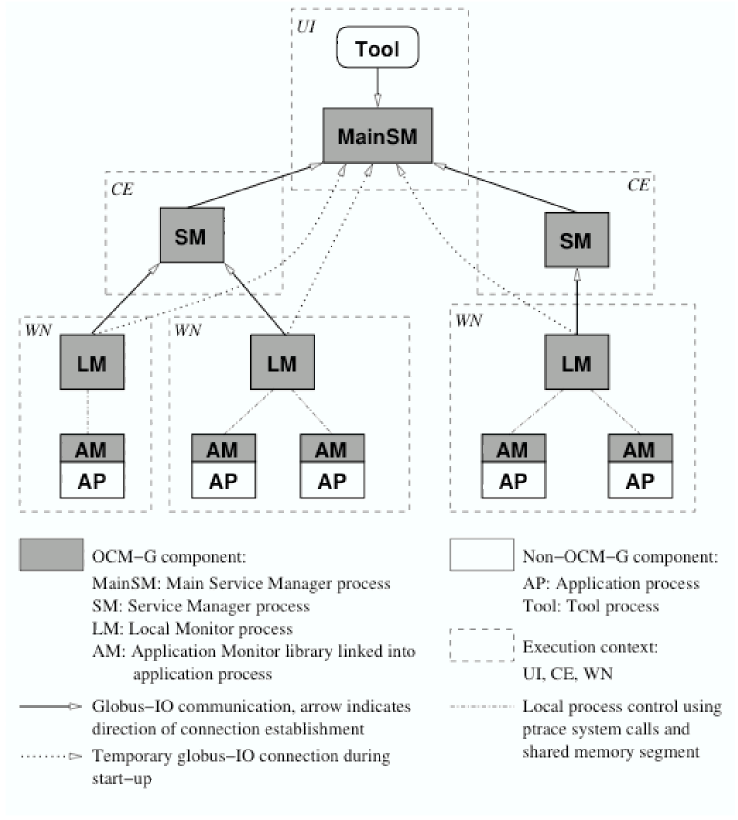

UI
Tool
MainSM
CE
SM
CE
SM
WN
LM
AM
AP
WN
LM
AM
AP
AM
AP
WN
LM
AM
AP
AM
AP
OCM−G component:
MainSM: Main Service Manager process
SM: Service Manager process
LM: Local Monitor process
AM: Application Monitor library linked into application process
Non−OCM−G component:
AP: Application process
Tool: Tool process
Execution context:
UI, CE, WN
Globus−IO communication, arrow indicates direction of connection establishment
Local process control using ptrace system calls and shared memory segment
Temporary globus−IO connection during start−up

Figure 7: OCM-G architecture

Manager (MainSM) to publish the data outside the grid.

When starting the monitoring, there has to be a Local Monitor running on each host and the SMs hierarchy in place. What's more, every application that can be monitored has to be instrumented and recompiled with the support of the Application Monitor library.

Application monitoring

OCM-G allows a fine-grain application monitoring, with trace events and CPU usage monitoring, as long as the applications have been instrumented with trace calls and linked to the Application Monitor library.

It is then possible to compare the efficiency of a function, eventually the speedup of a loop by comparing loop instances when a different number of processors are given to the process (see [7]).

The languages supported for instrumentation are C, C++ and FORTRAN.

Performance analysis tool

OCM-G is part of the Crossgrid project, and it is an OMIS-compliant application monitoring tool. Crossgrid also comes with a Performance analysis and visualization tool which can work with all kind of OMIS-compliant monitoring tools.

G-PM allows complex request, to determine higher-level performance properties and application specific metrics. To provide this kind of information, G-PM uses three sources of data: performance measurement data related to the running application, measured performance data on the execution environment, and results of micro-benchmarks (providing reference values).

At first glance, G-PM looks like visualization tools like Paraver, but there is a major difference between the Crossgrid tool and traditionnal tools: usual approaches (Pablo, Paraver, etc) only support a centralized and offline analysis whereas G-PM architecture allows a distributed on-line computation of raw data.

Developper interface and Integration

OCM-G is a very powerfool application monitoring tool, coded in C, and open-source. To add new apps measurements, we just need to put more trace calls and to recompile the programs. It can be integrated to other systems, since the OMIS interface is clearly described in [14].

Security

RSA-based encrypted connections (through GSS-API) and user authentication prevent unauthorized access to monitored applications and data. However, it seems that the authorization only works at VO-level.

Summary

OCM-G is designed to monitor applications, and can not provide resource- or job-oriented monitoring. Its structure also lacks of extensibility, and so it does not fit our requirements. What is more, OCM-G (as G-PM) only works on Linux systems. However, once available on different systems, it could be used as a component of a bigger monitoring tool.

2.5 Ganglia

Institution:	University of California, Berkeley, USA
Key persons:	Matt Massie
Web page:	http://ganglia.sourceforge.net/
References:	[15]

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters, relies on a multicast-based listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state. Data is represented in XML and compressed using XDR. The Ganglia Web Frontend can be used to inspect for example CPU utilization in the last hour or last month.

Architecture

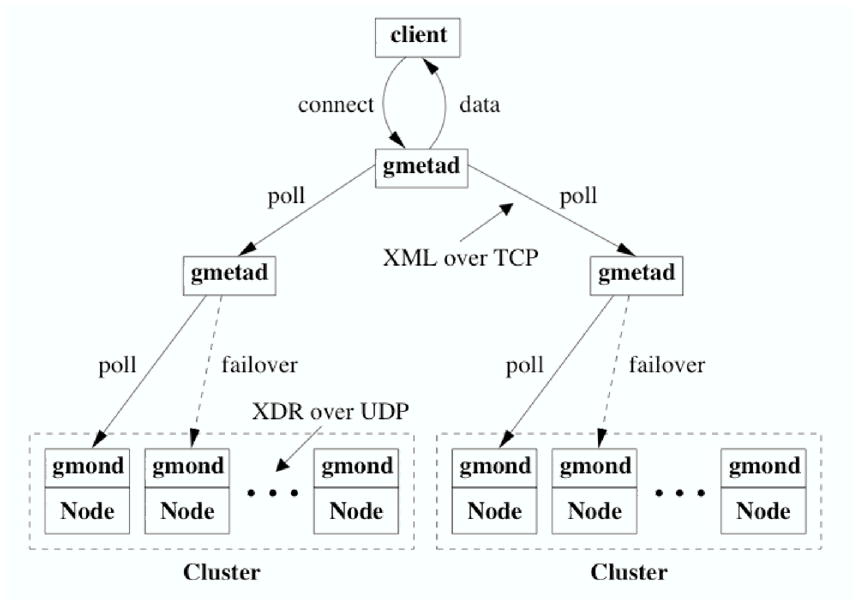


Figure 8: Ganglia architecture

Ganglia is based on a distributed and hierarchical structure represented on fig. 8. On every grid node that is monitored, a `gmond` daemon is running and collecting local data. It has four main responsibilities: monitor changes in host state, announce relevant changes, listen to the state of all other ganglia nodes via a unicast or multicast channel and answer requests for an XML description of the cluster state. `gmond`s are meant to communicate with every other gmond, so as any node may be able to know the status of the entire cluster. However, even if the data transmitted are reduced to the minimum (only when relevant changes happens) this may cause a significant network overload inside the cluster.

At cluster level, the data are collected by gmetad daemons by polling following a round-robin algorithm, thus distributing the load. There can be many different levels of gmetad applications, which can gather monitoring information from gmonds as well as from gmetads. They will play the role of proxies for every administrative domain; they can be replicated to make the system more reliable. Yet there may still be a problem of network intrusiveness because gmetads do not share data and are permanently gathering fresh updates.

Measurements are made periodically by gmond daemons, following the configuration given at startup. Communication between Ganglia agents use XDR over UDP or XML over TCP. At every level, users inside or outside the grid can connect to Ganglia through a PHP website.

A major drawback of Ganglia is that it has no real registration method. The gmetads have to know who is under their responsibility at startup. They will read the addresses of the daemons to query in a configuration file. That involoves the global structure of the grid must be known at startup; adding new hosts in a cluster would be possible, but new clusters could be difficult and restarting some daemons could be needed.

Ressource-level monitoring and host metrics

For each node, host metrics are nearly the same as in Mercury:

- OS name and release, machine type, boot time
- CPU numbers, speed, usage and average load
- Memory info (total, free, shared, buffers, cached)
- Swap info
- Disk info
- Network interfaces traffic

Job-level resource

Concerning job-level resource, we are confronted to the problem exposed in the introduction: the Monitoring System has to collaborate with the Job

Manager and the Local Resource Manager to get the correspondance between Grid Job IDs and PIDs.

Since Ganglia doesn't support arguments in metrics queries, it may be hard to create a metric giving the CPU usage for each process. However, depending on the number of interesting processes, we could create user-defined metrics for each process we decide to monitor.

In fact, in appendix D, that is a solution we have been working on. When a job is submitted with our very simple job submitter, the metrics associated (number of processes, CPU load) to the job are created and the processes are launched under a certain Unix user. Then these metrics are periodically updated.

Developper Interface and Integration

Ganglia is an open-source project written in C. The metrics are not coded as different modules to add at `gmond` startup, so it may be very hard to add metrics in the C code.

However, Ganglia provides another way to add metrics: the command line program `gmetrics`. For example, if we have a program `cputemp` outputing the CPU temperature, it is easy to add a new metric by running periodically (with `cron` for instance) the following command line:

```
% gmetric --name temperature --value `cputemp` --type int16 --units Celcius
```

The exit code tells whether the data was correctly sent or not.

Yet, in Ganglia the metrics queries cannot carry arguments (the only implicit argument is the host who the query is made to), and we may sometimes need to access the same metrics for different elements (processes for example) of the same hosts.

At the user end, the PHP web site publishing the data may be integrated and modified for more specific purposes.

Security

Concerning authentification methods, from inside or outside the grid, the only way available is IP authentification. Then there is no restriction in authorization, except that you have to see the machine you are asking information.

Summary

Ganglia is a project working on many different platforms, providing efficient monitoring in standard formats and also a simple way to register new metrics. As explained in appendix D, it can then be possible to establish an easy and simple job monitoring by associating the job submission to the metrics registration.

However, the communication between daemons, which ensures the freshness of data, may be too intrusive in a grid context, even if compressed formats can be used and if the load is distributed by a round-robin algorithm. What's more, the multicast used to simplify information sharing between a cluster's node may not be available, and so unicast, more expensive in network resources, should be used between every nodes. Finally, Ganglia won't suit very dynamic grids, because, there is no registration method and it would mean restarting daemons frequently.

2.6 GRIA

Institution:	IT Innovation Centre, Southampton, UK
	National Technical University of Athens
Key persons:	Mike Surridge, Hugo Kohmann
Web page:	http://www.gria.org/
References:	[16]

GRIA provides a package of four Web Services that together enable a service provider to provide access to shared remote computation and data storage, subject to a well-defined business process. The four GRIA Services are:

- an Account Service
- a Resource Allocation Service
- a Data Storage Service
- a Job Execution Service

In fact, as explaned in [16], the GRIA team doesn't integrate its own monitoring tool, but uses external benchmarks to compute the resource usage by the different jobs.

3 Comparison

	MDS (Globus)	**Mercury (GridLab)**	**NWS**	**OCM-G (Crossgrid)**	**Ganglia**
General					
Supported Systems	AIX, Darwin, Debian, Fedora Core, HP/UX, Red Hat, Solaris, SuSE	Darwin, Irix, Linux, Mach, OSF, Solaris, Tru64	Unix systems	Linux (RedHat)	AIX, Cygwin, Darwin, FreeBSD, HP/UX, Irix, Linux, OSF, Solaris
Source code	C, open source				
Tool architecture					
distributed	√	√	-[3]	√	√[4]
hierarchical	√	√	-	√	√
Code structure	modular		functionnal		
Resource oriented	partially	√	√	-	√
Job oriented	-	possible	-	-	possible
Application oriented	-	√	-	√	-
Resource monitoring					
Number of metrics					
AIX	15-20	14[5]	11	-	38
Linux	15-20	58	13	-	38
Architecture	√	√	-	-	√
CPU	√	√	√	√	√
Memory	√	√	√	-	√
Storage	√	√	-	-	√
Network	√	√	√	-	√
Application monitoring					
Instrumentation[6]	-	C	-	C, C++, FORTRAN	-

	MDS	**Mercury**	**NWS**	**OCM-G**	**Ganglia**
Integration and Development information					
Data format	XML (MDS-3)	Non standard	Non standard	Non standard (OMIS)	XML / XDR
Protocols	LDAP / OGSA	channels over TCP	channels over TCP	-	TCP / UDP
Metrics features					
Parameters	-	✓	✓	✓	-
Return types	-	scalar, complex	scalar	scalar	scalar
Notifications	✓	✓	-	✓	-
Method to add metrics	C or Java modules, Command line scripts	C module	-	-	Command line
Security					
Authentification	LDAP / OGSA	IP, GSS-API	LDAP	GSS-API	IP
Authorization	VO level[7]	Metric level[8]	VO level[7]	VO level[7]	domain level[9]
Encryption	✓	-	-	✓	-

[3]In NWS, the name server (also called directory service) is centralized. It can be replicated but not distributed, so we may have to choose between reliability and low load for a grid with many hosts

[4]Ganglia is fully distributed and even the data publication is distributed, thanks to an exchange of information between hosts. Thus the system is more failure tolerant

[5]On AIX systems, Mercury only provides UNIX general metrics, like average load, users online, mounted systems but no precise monitoring information; using Mercury on AIX would imply the development of a new module

[6]List of supported languages for application instrumentation: these are the lanhguages in which trace calls for application monitoring can be made.

[7]Inside a Virtual Organization (VO), every metric is visible without any restriction

[8]It is possible to define access rules for each metric

[9]The only restriction is to see the machine you are asking data and to have your IP allowed

4 Related work

In the litterature, there are several other works aimaing at describing the different grid solutions currently available. Some of them have been used to write this paper.

The APART community is a group regularly publishing a white paper called "Performance Tools for the Grid: State of the Art and Future" [10]. It gives a general overview of several monitoring and evaluation tools developped for the grid. The goal is to make a brief description of each and to categorize and compare them according to their functionnalities , features, instrumentation supports, architectures and provided interfaces. In the future, this paper will contain feedback about practical experiences of deploying and using the tools.

A similar work has been made in [23], describing monitoring tools and sorting them into categories, depending on the main kind of resources monitored, on the way it is organized and distibuted (level 0 tools are made of sensors directly answering with client, whereas level 3 monitoring systems allow a hierarchy of information republishers of different kinds, from the producer to the consumer).

5 Conclusion

Current monitoring systems do not fit our requirements in terms of job-oriented monitoring, because they need to cooperate with other grid middleware applications to achieve this goal, but some of them provide enough fine-grain metrics and are extensible enough to allow us to add job monitoring features.

We have particularly studied the Gridlab monitoring tool Mercury and Ganglia. Both of them seem to be extensible enough to provide job-oriented features. Mercury is more powerful but also more complex, whereas Gagnlia is more simple but could not be a good choice for dynamic grids.

The work we have done was essentially to try and find the extension possibilities of these tools. The next step should be to integrate them in a complete grid environnement and make them communicate with the Job Submitter, the Broker and the Job Manager.

A Installing Mercury

Where to find the sources

The main website for Mercury is

`http://www.gridlab.org/WorkPackages/wp-11/`

The source of the last version (2.4.1) can be downloaded at

`gridlab.org/WorkPackages/wp-11/mercury-monitor-2.4.1.tar.gz`

The Monitor Reference Manual is available in HTML at

`www.gridlab.org/WorkPackages/wp-11/monitor-manual/index.html`

Dependencies

Software	**Version**[10]	**Comment**
`pkg-config`	0.15.0	
`GLib 2`	2.2.0	
`gcc`	3.3	in case of building problems
`GNU make`	3.79.1	
`GNU autoconf`	2.59	if you intend to modify Mercury or if you build it from CVS
`GNU automake`	3.79.1	
`GNU libtool`	1.5	
`flex`	2.5.30	
`bison`	1.875	

Installation

Once all the requirements are met, you just have to do the usual following commands:

```
% cd [Mercury source directory]
% ./configure
% make
% make install
```

You can give many different options to the `configure` script. To know all of them, you can see the autoconf and libtool documentation for information about the standard options, or the Monitor Reference Manual, in the *Building and Installing the Mercury Grid Monitoring System* for Mercury specific options. In the following table, some options are described:

[10]The versions given in the table are the recommanded ones, given by Gridlab, and previous version may work; to get more information about the versions, have a look at the Monitor Reference Manual, in the *Software Requirements* section

`--prefix = `*PATH*	Installs Mercury in *PATH*`/bin`, *PATH*`/include`...
`--with-gridlab`	Use this option if building the package in a GridLab environment.
`--with-gssapi=`*FLAVOR*	Enable the use of GSSAPI for authentication. Supported flavors are heimdal, mit and gsi. Default is to autodetect

Configuration

`lm.conf / mm.conf / libmonp.conf`

These files contain the configuration information for the Local Monitor (`lm`), the Main Monitor (`mm`) and the consumer library (`libmonp`). The structure of the three files is the same. The detailed section of the manual dedicated to these files is called *Monitoring System Configuration*.

The `Common` section can contain two values: `ModulePath` which indicates the path where to look for loadable modules and `Autoload` which can enable the loading of every module in the `ModulePath` directory.

Next, each module can be configured in a `Module[‘‘modulename.so’’]` section. If the value `Load` is set to `Yes`, the module is loaded. The `Priority` of a module let the administrator decide which module will answer when several modules have registered the same metrics. The module can be loaded into a stand-alone process with the `External` option and in this case, the `User` one specifies the user account to run the external sensor as.

Finally, a `Daemon` section indicates where to store the pid of the daemon (`PidFile`), what is the log behaviour, the URL to listen on for request (`Listen`) and the Access Control Lists file (`AclFile`).

Access Control List files

The ACL files contain a list of lines respecting the pattern *`permission object`* `to` *`requirements`* and here is an example of such access control lists:

```
allow monitor * to everyone;
deny control * to everyone;

allow monitor app.events to user "admin";
```

The `cluster_hosts` file

In a cluster environment the main monitor needs a list of URLs of local monitors in `cluster_hosts` filled by the system administrator. The format of the lines is <*hostname*> <*URL of the local monitor on that host*> and here is an example of such a file.

```
frontend-node.cluster monp://localhost:3571
node1.cluster monp://node1.cluster
node2.cluster monp://node2.cluster
```

B Adding a new sensor module in Mercury

Here are the steps to follow to add a new sensor module. Adding an actuator module is quite similar, but the files to modify are slightly different.

Bootstrap

First thing to do is to compile the tool, to make sure everything is working, and to bootstrap the Makefiles structure (see fig. 9).

```
% cd [Mercury main directory]
% ./configure --prefix=/usr/local
% make
```

Figure 9: Bootstraping

Writing the module

The sensor modules are grouped in the `lib/modules` subdirectory. The application actuator module can be found in `lib/modules/appsensor`. To write your own module, you can start from another file, like `tru64.c` (or `actuator.c`).

We are now presenting a new module called `newmod.c` (the complete source is in the appendices), and describing the key sections in the conception of the module.

Declarations of the metrics

The metrics are defined by their class and their id; `host.cpu.frequency`, for example, belongs to the `cpu` class and is identified by `ID_cpu_frequency`.

To define which metrics your module is going to provide, you have to declare them twice, in the *Prototypes* section and in the *Global variables* section of the .c file. In the following figures (10 and 11), we are redefining an existing metric (`host.cpu.number`) and creating a new one (`host.newclass.newid`) in our new module .

```
// Prototypes

[...]

static DEF_SAMPLE(cpu, number);
static DEF_SAMPLE(newclass, newid);
```

Figure 10: lib/modules/newmod.c: part of the *Protoypes* declaration section

```
// Global variables


static const prod_metric_desc newmod_metric_table[] = {
    { "host.cpu.number",         SAMPLE(cpu, number) },
    { "host.newclass.newid",     SAMPLE(newclass, newid) },
    { NULL,                      NULL }};

[...]
```

Figure 11: lib/modules/newmod.c: part of the *Global variables* declaration section

Definitions of the metrics

Let's now define the metrics we have chosen to furnish. In the figures 12 we can see that `host.cpu.number` always returns 0 (in fact this won't necessary be the result we will have when asking the Local Monitor, because the result is given by the top-priority loaded module providing this metric) and that `host.newclass.newid` return a char string.

Module interface

To complete the module, we finally need to declare the public interface of the module. In the source code, it corresponds to the `prod_sensor_module` structure called `newmod_module`. The module is registered by this structure through a macro call `PROD_SENSOR_MODULE`.

```
static DEF_SAMPLE(newclass, newid) {
  char* ret = "Yeah ! New module added succesfully";
  return SEND(string, ret);
}

static DEF_SAMPLE(cpu, number) {
  unsigned long ret = 0;
  return SEND(uint32, ret);
}
```

Figure 12: lib/modules/newmod.c: Metrics definitions

The structure declares 4 functions to be defined in the source file:

- `module_init`
- `module_done`
- `check_metric` to check the name and parameters of the metrics asked for
- `start_instance` to prepare an instance of a metric

We have now written our new module, and with the `newmod.c` file, it is easy to understand the basic structure of a module. If you want to have a more complex module and you need to include net support in your module for example, you will have to look at other modules like `net-common.c` which is included in `unix.c`.

Registration of the new metrics

This step only concerns non-overloaded metrics. In the example developped here, `host.newclass.newid`. In the déclaration section, we have declared this new metric as the couple `{newclass, newid}`. To register the metric, we need to modify two files.

First we have to define the class and id enum constants `CLASS_newclass` and `ID_newclass_newid` which are used by the macros `DEF_SAMPLE` and `SAMPLE`. This is done in `lib/modules/hostsensor.h` (see fig. 13), but the only constraints for the constants are to exist and to have different values[11], wherever they are defined.

[11]In fact different classes must have different CLASS_xxx values and different metrics of the same class must have different ID_class_xxx values

```
/* Measurement classes */
enum {  CLASS_cpu = 1,
        CLASS_disk,
        [...]
        CLASS_newclass};

/* newmod measurement IDs */
enum { ID_newclass_newid};
```

Figure 13: lib/modules/hostsensor.h: Metric class and id registration

Next we have to define the type and parameters of this new metric in `etc/metric_defs` (for actuators, the file is `etc/ctrl_defs`. Figure 14 shows this step.

```
# newmod

name = host.newclass.newid
param = string host
type = string value;
```

Figure 14: etc/metric_defs: Metric class and id registration

Registration of the new module

Finally, we have to modify `lib/modules/Makefile.am` and `configure.ac`. The first modification (fig. 15) is to tell `make` how to compile `newmod.c`. The second one (fig. 16) is to tell `configure` to include the module if the condition (here `true`) is met.

```
EXTRA_LTLIBRARIES = \
                auth-anon.la \
                [...]
                unix.la \
                newmod.la

[...]

newmod_la_SOURCES = newmod.c
newmod_la_CPPFLAGS = $(MOD_NEWMOD_CPPFLAGS)
newmod_la_LDFLAGS = $(MODULE_LDFLAGS) $(MOD_NEWMOD_LDFLAGS)
```

Figure 15: lib/modules/Makefile.am: module compilation declarations

```
MON_CHECK_MODULE([newmod], [test module by OL], [true])
```

Figure 16: configure.ac: module compilation declarations

Installation and configuration

We are now ready to install the modified version of Mercury. The recompilation (`make`) will normally call `autoconf` and `automake` to regenerate the Makefiles. Type `make install` to finally put the compiled program where it is supposed to be.

Before launching the Local Monitor (or the Main Monitor), we still have to change the configuration file (`etc/lm.conf` or `etc/mm.conf`, see fig. 17) to tell the monitor to load our module.

Then, depending on the priorities, `host.cpu.number` will be taken in charge by a pre-existing module or by ours; and `host.newclass.newid` will return the string "Yeah ! New module added succesfully"

```
# Module added by OL
Module["newmod.so"] {
        Load yes;
        Priority 100;
}
```

Figure 17: etc/lm.conf: Configuration for module loading and priorities

C Installing Ganglia

Where to find the sources

The main website for Ganglia is

`http://ganglia.info/`

The source of Ganglia monitor core laste release (3.0.1) can be found at

`http://ganglia.info/downloads.php`

Ganglia documentation can be found at

`http://ganglia.info/docs/`

Dependencies

Software	**Comment**
`librrd`	if you want to compile `gmetad`[12]
`gcc`	in case of building problems
`GNU make`	
`GNU autoconf`	if you intend to modify Mercury or if you build it from CVS
`GNU automake`	
`GNU libtool`	

Installation

Once all the requirements are met, you just have to do the usual following commands:

```
% cd [Ganglia source directory]
% ./configure
% make
% make install
```

You can give many different options to the `configure` script. To know all of them, you can see the autoconf and libtool documentation for information about the standard options, or the Ganglia Readme. In the following table, some options are described:

`--prefix =` *PATH*	Installs Ganglia in *PATH*`/bin`, *PATH*`/include`...
`--with-gmetad`	Compile also the `gmetad` daemon. By default, only `gmond` is built
`--disable-shared` `--enable-static`	On AIX, Ganglia should not be compiled with shared libraries
`CFLAGS="-I/rrd/header/path"`	You might need to give the `librrd` path to the `configure` script when compiling `gmetad`
`CFLAGS="-I/rrd/header/path"`	
`LDFLAGS="-L/rrd/library/path"`	

Configuration

- `gmond.conf`

It is the configuration file describing the behaviour of `gmond`.

[12] As the `librrd` is statically linked, there is no need to have the library once `gmetad` is compiled

First, a (unique) **cluster** section will describe the cluster which the host belongs to with the following attributes:

`name`[13]	Name of the cluster
`owner`[13]	Administrators of the cluster
`latlong`	GPS coordinates of this cluster on earth
`url`	The url for more information on the cluster

Then, the **globals** section controls general characteristics of `gmond`. Here are the values this section contains:

`daemonize`	When true, `gmond` will daemonize
`setuid`	When true, `gmond` sets its effective uid to the uid of the `user` specified
`user`	
`debug_level`	Commands the verbosity of `gmond`
`mute`	When true, `gmond` will not send any data
`deaf`	When true, `gmond` will not receive any data
`host_dmax`	If `host_dmax` is a positive number, `gmond` will flush a host after it has not heard from it for `host_dmax` seconds
`cleanup_threshold`	Minimum about of time before `gmond` will cleanup and hosts or metrics where $tn > dmax$, i.e. expired data
`gexec`	Specify whether `gmond` will announce the host availability to run `gexec`[14]jobs.

You can define as many **udp_send_channel** sections as you like within the limitations of memory and file descriptors. If `gmond` is configured as mute this section will be ignored. The options are the following:

`mcast_join`	When specified `gmond` will send data out the interface `mcast_if` on the UDP multicast socket `mcast_join`
`mcast_if`	
`host`	If no `mcast` is specified, `gmond` will send unicast UDP messages to the host specified
`port`	

You can specify as many **udp_recv_channel** and **tcp_accept_channel** sections as you like within the limits of memory and file descriptors. If `gmond` is configured as deaf these sections will be ignored. The options for `udp_recv_channel` are `mcast_join`, `bind`, `port`, `mcast_if` and `family`; the options for `tcp_accept_channel` are `bind`, `port`, `interface`, `family` and `timeout`; these options specify the characteristics of the UDP or TCP channel to listen to; both can have an `acl` subsection.

[13]The pair `name / owner` should be unique in the world.
[14]`gexec` is part of an execution environment which is another part of the Ganglia project.

Here are some examples of **acl** subsections:

```
acl {
  default = "deny"
  access {
    ip = 192.168.0.4
    mask = 32
    action = "allow"
  }
}
```

```
acl {
  default = "allow"
  access {
    ip = 192.168.0.0
    mask = 24
    action = "deny"
  }
  access {
    ip = ::ff:1.2.3.0
    mask = 120
    action = "deny"
  }
}
```

Finally, `gmond.conf` contains as many **collection_group** sections as you like within the limitations of memory. A `collection_group` has the following attributes:

`collect_once`	Specify the frequency the metrics will be collected at
`collect_every`	
`time_threshold`	Max time before publiqhing again the metrics of the group
`metric`	Each collection group shall contain one or more metric subsections, which will be characterized by the `name` of the metric and by an optional `value_threshold` defining the percentage of the minimum variation triggering an update.

Here are some examples of `collection_group` subsections:

```
collection_group {
  collect_once   = yes
  time_threshold = 1800
  metric {
   name = "cpu_num"
  }
}
```

```
collection_group {
  collect_every = 60
  time_threshold = 300
  metric {
    name = "cpu_user"
    value_threshold = 5.0
  }
  metric {
    name = "cpu_idle"
    value_threshold = 10.0
  }
}
```

• `gmetad.conf`

For `gmetad` to do anything useful you much specify at least one `data_source` in the configuration. The format of the `data_source` line is as follows:

```
        data\_source "Cluster A" 127.0.0.1 1.2.3.4:8655

              data\_source "Cluster B" 1.2.4.4:8655
```

For each cluster `gmetad` has to collect data from, we have to add a line with the name of the cluster (corresponding to the `name` value in the `cluster` section of `gmond.conf`), and a list of the addresses of the nodes in the cluster able to publish the cluster state.

There are some other options available in `gmetad.conf`, which are described inside the configuration file itself.

D A simple job monitor with Ganglia

So as to give an example of the possibility in Ganglia to publish job-oriented metrics, we have developped little scripts and tools to submit locally a job and create associated metrics.

The job submitter

```
    Usage: submitjob <GJID> <local_user> <prog> <in> <out>
```

To submit a job, you have to give the Grid Job ID, because it will be the prefix of all the metrics associated to this job. You also have to give the local user which will be used to launch the program; in a real implementation, the user should be chosen by the job submitter in a pool of available local users.

Then you give the submitter the names of the program file and of the files used as standard inputs and outputs. The transfert of these files should be done by the Data Management Service in a complete grid environment.

The `submitjob` first launches the program with the standard input and output redirected, under the requested account `local_user`; then it periodically runs a script collecting and publishing data.

Data collection and publication

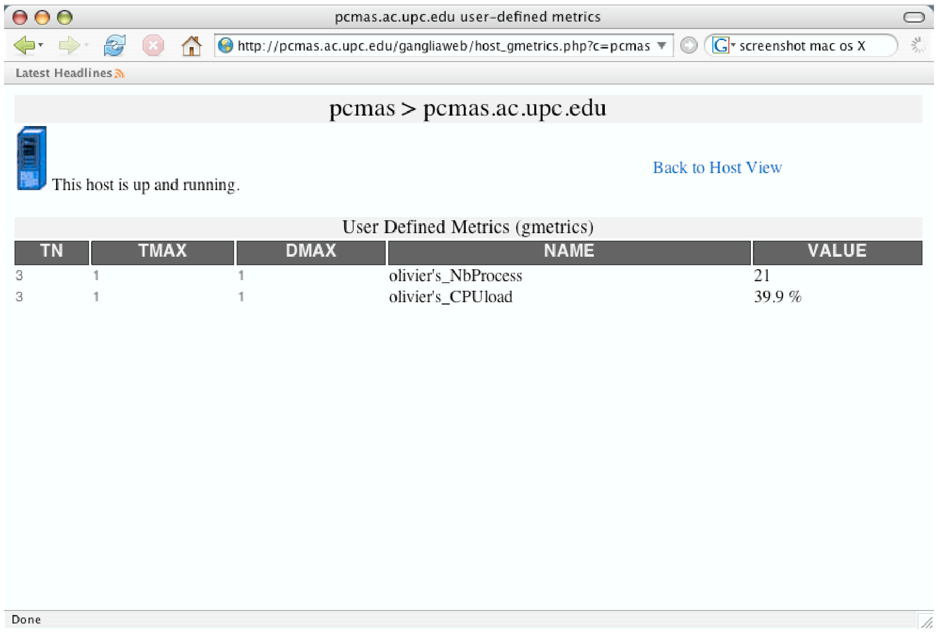
The metrics added are the number of processes run by the job and the CPU load used by the job. They are published through the following script:

```
#!/bin/sh
# metricsUser localuser metricprefix

[...] # computation of the CPUload using ps aux

gmetric -n $2_CPULoad -v $calcul -t double -u "%"
gmetric -n $2_NbProcess -v ‘ps aux|grep "^$1" | wc‘ -t int16
```

The script has two arguments, the local user running the job, and the metrics prefix that will be used. Then it computes the CPUload filtering the output of `ps aux`. Finally, it publishes both metrics using `gmetric`.

Data visualization

Finally, it is possible to see these metrics on Ganglia PHP website. Fig. 18 shows an example of such a job running on a machine called `pcmas`.


pcmas.ac.upc.edu user-defined metrics
http://pcmas.ac.upc.edu/gangliaweb/host_gmetrics.php?c=pcmas
screenshot mac os X
Latest Headlines
Back to Host View
Done

pcmas > pcmas.ac.upc.edu

This host is up and running.

User Defined Metrics (gmetrics)

TN	TMAX	DMAX	NAME	VALUE
3	1	1	olivier's_NbProcess	21
3	1	1	olivier's_CPUload	39.9 %

Figure 18: Job monitoring with Ganglia and the Simple Job

References

[1] Zoltán Balaton and Gabor Gombás. Resource and job monitoring in the grid. In *Euro-Par*, pages 404–411, 2003.

[2] Zoltán Balaton, Péter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. From cluster monitoring to grid monitoring based on grm. In *Euro-Par*, pages 874–881, 2001.

[3] Zoltán Balaton and Fernc Vajda. Grid information and monitoring systems, 2004.

[4] Bartosz Balis, Marian Bubak, Wlodzimierz Funika, Roland Wismüller, Marcin Radecki, Tomasz Szepieniec, Tomasz Arodz, and Marcin Kurdziel. Performance evaluation and monitoring of interactive grid applications. In *Lecture Notes in Computer Science, Volume 3241*, pages 345–352, Nov 2004.

[5] Marian Bubak, Wlodzimierz Funika, and Roland Wismüller. The crossgrid performance analysis tool for interactive grid applications. In *PVM/MPI*, pages 50–60, 2002.

[6] CERN. Gridcafé website.

[7] Julita Corbalán and Jesús Labarta. Dynamic speedup calculation through self-analysis, 1999.

[8] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-driven processor allocation, 2000.

[9] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing, 2001.

[10] M. et al. Gerndt. Performance tools for the grid: State of the art and future. apart white paper, 2004.

[11] J. Hodges and R. Morgan. Rfc 3377: Lightweight directory access protocol (v3): Technical specification, 2002.

[12] Helene N. Lim Choi Keung, Justin R. D. Dyson, Stephen A. Jarvis, and Graham R. Nudd. Predicting the performance of globus monitoring and discovery service (mds-2) queries. In *GRID*, pages 176–183, 2003.

[13] John Linn. Rfc 1508: Generic security service application program interface, 1993.

[14] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — Online Monitoring Interface Specification (Version 2.0). Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, July 1997.

[15] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[16] Athanasios Panagakis, Costis Christogiannis, and Theodora Varvarigou. End-to-end performance modelling in gria, 2003.

[17] Norbert Podhorszki, Zoltán Balaton, and Gabor Gombás. Monitoring message-passing parallel applications in the grid with grm and mercury monitor. In *European Across Grids Conference*, pages 179–181, 2004.

[18] Norbert Podhorszki and Peter Kacsuk. Design and implementation of a distributed monitor for semi-on-line monitoring of visualmp applications. *Distributed and parallel systems: from instruction parallelism to cluster computing*, pages 23–36, 2000.

[19] GridLab project. Mercury monitor reference manual, Version 2.4.0.

[20] Sun Microsystems R. Srinivansan. Rfc 1832: Xdr: External data representation standard, 1995.

[21] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture, 2002.

[22] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[23] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems., 2004.