Journal of Grid Computing (2004) 2: 247–260

Autonomic Management of Large Clusters and Their Integration into the Grid *

Thomas Röblitz[1], Florian Schintke[1], Alexander Reinefeld[1], Olof Bärring[2], Maite Barroso Lopez[2], German Cancio[2], Sylvain Chapeland[2], Karim Chouikh[2], Lionel Cons[2], Piotr Poznański[2], Philippe Defert[2], Jan Iven[2], Thorsten Kleinwort[2], Bernd Panzer-Steindel[2], Jaroslaw Polok[2], Catherine Rafflin[2], Alan Silverman[2], Tim Smith[2], Jan Van Eldik[2], David Front[3], Massimo Biasotto[4], Cristina Aiftimiei[4], Enrico Ferro[4], Gaetano Maron[4], Andrea Chierici[5], Luca Dell'agnello[5], Marco Serra[6], Michele Michelotto[7], Lord Hess[8], Volker Lindenstruth[8], Frank Pister[8], Timm Morten Steinbeck[8], David Groep[9], Martijn Steenbakkers[9], Oscar Koeroo[9], Wim Som de Cerff[9], Gerben Venekamp[9], Paul Anderson[10], Tim Colles[10], Alexander Holt[10], Alastair Scobie[10], Michael George[11], Andrew Washbrook[11] and Rafael A. García Leiva[12]

[1]*ZIB, Takustraße 7, D-14195 Berlin Dahlem, Germany*
[2]*CERN, CH-1211 Geneva-23, Switzerland*
[3]*Weizmann Institute of Science, PO Box 26, Rehovot 76100, Israel*
[4]*INFN, Viale dell'Università 2, I-35020 Legnaro (Padova), Italy*
[5]*INFN, Viale Berti Pichat 6/2, I-40127 Bologna, Italy*
[6]*INFN, P. le Aldo Moro 2, I-00185 Roma, Italy*
[7]*INFN, Via Marzolo 8, I-35131 Padova, Italy*
[8]*University of Heidelberg, Kirchhoff-Institut für Physik, Im Neuenheimer Feld 227, D-69120 Heidelberg, Germany*
[9]*NIKHEF, PO Box 41882, 1009 DB Amsterdam, The Netherlands*
[10]*University of Edinburgh, Old College, South Bridge, Edinburgh EH8 9YL, UK*
[11]*University of Liverpool, Oxford Street, Liverpool L69 7ZE, UK*
[12]*Department of Theoretical Physics, Universidad Autónoma de Madrid, Cantoblanco, 28049 Madrid, Spain*

Key words: autonomic computing, cluster computing, grid computing, system management

Abstract

We present a framework for the co-ordinated, autonomic management of multiple clusters in a compute center and their integration into a Grid environment. Site autonomy and the automation of administrative tasks are prime aspects in this framework. The system behavior is continuously monitored in a steering cycle and appropriate actions are taken to resolve any problems.

All presented components have been implemented in the course of the EU project DataGrid: The *Lemon* monitoring components, the *FT* fault-tolerance mechanism, the *quattor* system for software installation and configuration, the *RMS* job and resource management system, and the *Gridification* scheme that integrates clusters into the Grid.

* This work from the EU DataGrid project was funded by the European Commission grant IST-2000-25182.

1. Introduction

Large-scale projects like the forthcoming LHC experiments at CERN require computing resources in a scale that was never reached before [4]. In respect to their excessive demands on the computing infrastructure these projects are often equipped with a relatively low budget for computing, storage, and networking. Commodity clusters made of common-off-the-shelf technology therefore provide a welcome alternative to the more costly high-performance computers. The vastly grown requirements on the availability, however, clearly identify the neuralgic shortcomings in the current state-of-the-art of the cluster technology, namely their poor fault tolerance and high maintenance overhead [20]. Moreover, clusters are often incrementally installed or upgraded over time, typically resulting in heterogeneous hardware and software and thereby making it difficult to manage them in a consistent way.

At a conceptually higher level, clusters are often used as computing nodes in global Grid environments. Here, the mentioned problems become even more critical, because single clusters should (ideally) function autonomously without human intervention. The automated operation is not only necessary for reducing human administration effort, but also to limit possible sources of errors and to reduce service downtimes.

First bold visions for autonomic computing systems were presented, but their implementation are – of course – still in their infancy. Among the four self-management issues presented by Kephart and Chess [14], our work addresses self-configuration and self-healing. We aim at freeing virtual organizations from the burden of manually maintaining compute fabrics, thereby allowing them to concentrate on the higher-level organizational tasks in the Grid.

Our system facilitates *self-configuration* by describing the configuration of hardware and software components and deploying this information, e.g. for install and configure services. *Self-healing features* are achieved by monitoring the actual state of hardware and software components, correlating the sampled data with the goal state and automatically devising actions for repairing or updating the affected components. Additionally, our system manages the access to cluster resources for Grid jobs and it coordinates all tasks, that is, it provides a coherent inter- and intra-cluster job management.

The remainder of this paper is organized as follows. In the next section we present an overview of the architecture of the DataGrid fabric management system. Thereafter we present the building blocks in more detail (Sections 6, 7) and conclude the paper with a summary of our results.

2. Goals and Architectural Overview of Our Framework

In this section we discuss the objectives of our framework and present an overview of the components. The architecture reflects two key issues: (1) autonomous maintenance of the configuration of Grid computing centers and (2) job management.

2.1. *Autonomous Maintenance*

The main aim of our framework is to lower the administrative burden of managing multiple heterogeneous hardware and software components in a Grid computing site (also called *fabric*). Large computing centers may profit, because many repetitive tasks are handled automatically and more consistently. Although, our work was aiming at large installations, we found out that, research groups with smaller sized resources and usually less expertise in managing hardware and software are enabled to share their resources, because of the lowered administrative costs. The *self-management* of all hardware and software components includes (A) automatically adding new machines to a fabric – i.e. recognizing those machines, installing appropriate software packages on them and configuring the services hosted by a machine – and (B) ensuring the functional integrity of hardware and software components – including to switch off faulty services or even removing hosts from the active set of machines.

Hardware-based changes (A). As hardware is incrementally added to or upgraded in a fabric, the self-management must take notice of this and perform the necessary actions (OBJECTIVE AM.1). Assuming a new machine has been added or an existing one has been upgraded, the following actions are performed: (1) installing/running the core operating system and the self-management components, (2) deciding which services the machine should host, (3) updating the machine's configuration accordingly (installing software packages and configuring the services running on the machine). Both steps (1) and (2) can be initiated from the machine itself via some bootstrapping mechanism or externally from another machine that observes the network to recognize new machines. The architecture of our framework supports both methods equally.

Functional integrity (B). A Grid computing center is build of many services such as a file server, a DNS server, a Grid access node, a batch master, a batch worker, etc. The relationship of services to machines (hardware) is typically N to M:

– some services exist only once (e.g. DNS server),
– multiple machines host a single specific service only (e.g. file server and batch worker) and
– some machines host multiple services (e.g. Grid access and batch master).

The *self-management* must ensure that the services are running and function correctly (OBJECTIVE AM.2), e.g. by performing basic tests. Also, the hardware must be monitored, e.g. to simplify the identification of faults.

2.2. *Job Management*

Beside the self-management aspect our framework addresses issues related to the management of user jobs. User jobs may originate from different sources: (1) local user's community, (2) Grid community and (3) self-management part.

First, because certain self-management actions could corrupt the runtime environment of jobs, the job management must provide capabilities to coordinate user jobs with maintenance jobs (OBJECTIVE JM.1). Different Grid computing centers may deploy different job management systems each providing a specific set of scheduling features apart from common functionality. Hence, our framework must provide methods to enhance scheduling features. Otherwise, Grid jobs may not fully exploit the potential of a site (OBJECTIVE JM.2).

Second, jobs from the Grid community must pass an authorization check before they can be submitted to a local queueing system (OBJECTIVE JM.3). Authorization may depend on multiple factors, e.g. user's affiliation, resource requirements, current state of the resources, and may change dynamically. If a job request has passed the authorization step it must be provided with the necessary credentials for their execution (OBJECTIVE JM.4).

2.3. *Archtectural Overview*

In this section we present an overview of the components in our framework. The following sections will discuss these components in more detail and describe how each contributes to reach the objectives listed in the previous section.

Figure 1 depicts the interrelationships between the components, that we developed and implemented in the European *DataGrid* project [7], and shows how they work together to achieve an automated computing center management. A 'fabric' comprises one to several separate clusters with (possibly) different cluster

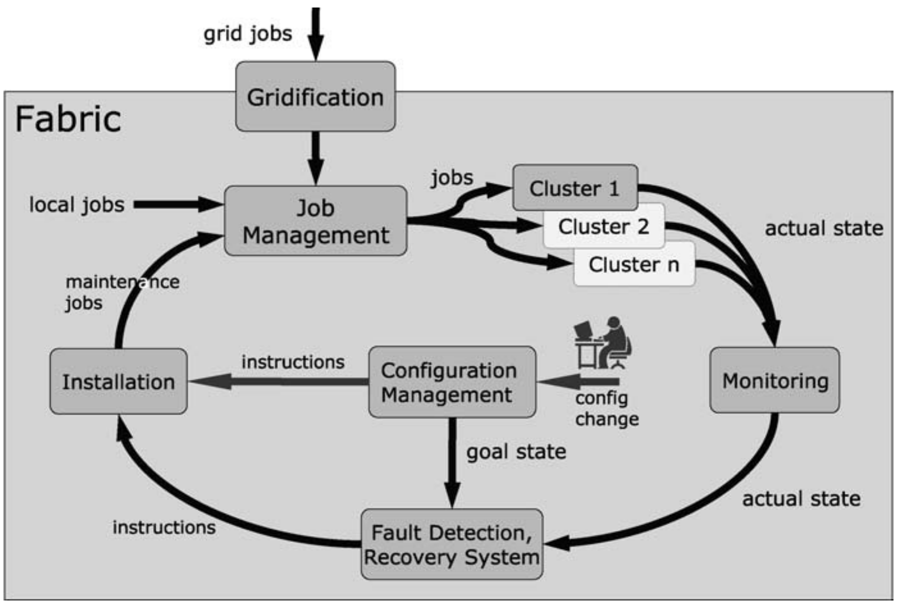

Figure 1. Management cycle for the automated maintenance of clusters in a compute center and their integration into a production Grid.

management systems (batch systems) and a couple of servers responsible for services, e.g. self-management components, file servers, DNS server, Grid access node, etc. (not shown in the figure).

Three kinds of jobs may enter the fabric:

– grid jobs (from the grid level above),
– local user jobs (injected from the left), and
– maintenance jobs (injected by the system itself).

Status information on the machines' target configuration, the *goal state*, is stored in a configuration database. The *actual state* is obtained by monitoring tools. Both states are compared in the *Fault Detection and Recovery System*, which checks for mismatches and initiates the necessary maintenance actions to fix them. Note that these actions are passed via the *Installation System* to the *Job Management* which schedules the maintenance jobs just like any other ordinary user job – but with specific requirements, of course. Even if the actions would not be executed on a worker node, their execution might require careful planning because (running) jobs may depend on affected services, e.g. file server hosted by a non-worker machine. In addition, local user jobs may be submitted for execution on specified clusters, or grid jobs may be injected from remote via the *Gridification* component.

The components *Configuration Management*, *Monitoring*, *Fault Detection and Recovery System* and *Installation* together provide a framework aiming at the objectives AM.1/2. The *Job Management* and the *Gridification* services provide a solution to the job-related objectives JM.1–4.

In the following sections, we discuss the components focussing on self-management, i.e. configuration and installation management (Section 3), monitoring (Section 4) and fault detection and recovery system (Section 5). Then, the job management system is described in detail (Section 6). Thereafter, we present the Grid access services (Section 7).

3. Configuration and Installation

Large fabrics consist of diverse hardware and software. Mainly, two reasons contribute to this observation. First, different machines serve different purposes, e.g. batch nodes, batch servers, file servers, network management machines, etc. Second, even those machines which are used for the same purpose may be installed or upgraded in an incremental procedure. Moreover, the list of software to be installed on all those machines differs significantly. Hence, the administration complexity is enormous. An automated configuration and installation management system is required to lower the administration overhead and facilitate a consistent configuration of all services.

Here, we describe a system called *quattor* that addresses these issues. Quattor manages configuration information, installs software packages and configures the services provided by a fabric. Thus, quattor addresses all objectives (by describing the configuration) and particularly the installation of software packages and the configuration of the whole system (OBJECTIVE AM.1).

Cfengine [6] is a set of tools building an expert system for the configuration and management of computer networks. Unlike quattor, Cfengine uses no central store for configuration information. Configuration elements are implicitly contained in policy rules organized by classes. Cfengine does not address software distribution. LCFGng [3] stores configuration information in a central database. Its configuration description language provides mechanisms for inheritance and mutation. The information is made available to specific machines by creating a machine profile and transmitting it to its target (i.e. the machine). Even though quattor and LCFGng share many architectural ideas, quattor uses a improved configuration language called "pan", which provides offline validation capabilities. Also, quattor interfaces different software package management tools.

3.1. *Architecture of the Configuration and Installation System*

The key design issues for the architecture are:

– *distributed approach*: operations are handled locally on the machines whenever possible,
– *efficiency*: machine profiles are stored locally to avoid a central bottleneck, and
– *adaptability*: interfaces existing tools.

Quattor consists of two building blocks, a configuration information management system and a software installation and service configuration system. The former is shown in Figure 2. It consists of a central database that stores configuration information. The information is inserted or updated using the *High Level Description* language *Pan*. Machine profiles are generated by compiling the *High Level Description* information into *Low Level Description* documents.

Validation of the configuration information is one of the most important features of the *Pan* language.

It is described by the validation code that is attached to a configuration tree either through paths (attached to a given path) or through types (added to a type definition). One configuration element can have more than one validation code attached. During validation phase, the code gets executed and evaluated. If it does not return the true value, a validation error is returned. The validation code can use the variable *self*, which is set to the value of the element that is being validated. Validation code cannot change any configuration information. Validation is the last phase of the processing of statements in *Pan*. It is performed just before creation of the *Low Level Description* documents.

Figure 3 shows an example for a configuration with validation described in *Pan*. First, the type `nfs_mount` is declared (lines 1–6). It contains attributes for the IP number of the server (line 2), the directory on the server (line 3), the mount point at the client (line 4) and a list of options (line 5). The values for the first three attributes must adhere to some format, which is given by the validation expressions (starting with `with`). For example, the IP number must contain four numbers separated by a dot. Both the directory on the server and the mount point are strings that start with a slash. Next, the specific mount point must be created in the configuration section for `nodeA` (line 10). Then, values for the attributes are set (lines 11–14).

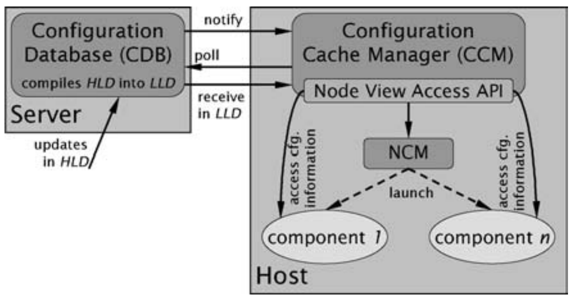

Figure 2. Components for managing configuration information.

The machine profiles are stored in the *Configuration Database* (CDB). If a machine's profile was changed the *Configuration Cache Manager* (CCM) on the respecting host is notified. The CCM of a machine polls for its profile and stores it in a local cache. The *Node Configuration Manager* (NCM) provides a framework for adapting the actual configuration of a node to its desired configuration, as it is described in the node's profile. Plug-in software modules called components are responsible for the configuration of local services, e.g. network, sendmail, NFS, scheduler, etc. These components or any other service on a machine may access configuration information via the *Node View Access API* (NVA).

A specific service using the configuration information management system is the software installation system. Figure 4 shows the main components of such a system. Software packages are stored in a repository on a server (multiple servers may coexist). The list of software to be installed on a machine is stored in a central configuration database and forwarded to this machine via its configuration cache manager. On each machine the *Software Package Manager Agent* compares the target software list with the currently installed packages and devises a list of packages to be

```
 1  define type nfs_mount = {
 2    "server_ip" : string with self =~ m/^\d+\.\d+\.\d+\.\d+$/
 3    "server_dir" : string with self =~ m/^\//
 4    "mountpoint" : string with self =~ m/^\//
 5    "options" : list
 6  }
 7
 8  template nodeA;
 9  include nfs_mount;
10  "/services/nfs/home" = create ("nfs_mount");
11  "/services/nfs/home/server_ip"  = "10.0.0.3"
12  "/services/nfs/home/server_dir" = "/home"
13  "/services/nfs/home/mountpoint" = "/global/home"
14  "/services/nfs/home/options"    = list("rw","noauto")
```

Figure 3. Declaration and configuration of a NFS mount point using *Pan*.

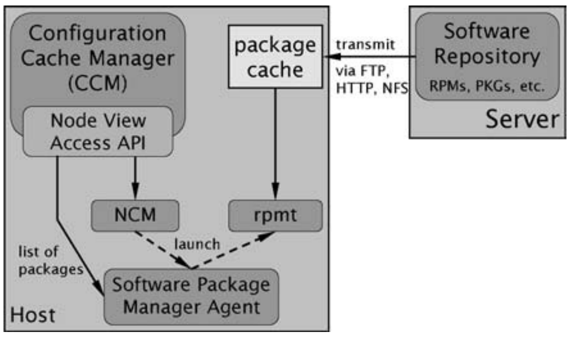

Figure 4. Components for managing software installation.

installed or removed, according to configuration policies e.g. for respecting existing local installation. The tool *rpmt* executes this list. For efficiency packages may be pre-staged to a machine in a cache.

4. Monitoring

Monitoring data about many different components must be gathered and stored to facilitate the self-healing of fabric services, i.e. aiming at the objectives AM.1 and AM.2. First, data must be collected. Then it must be stored and made available in an efficient manner for other components, such as the fault detection system (see Section 5). We developed *Lemon* a framework for monitoring components and storing the collected data both in a local and in a central repository.

SNMP (Simple Network Management Protocol) [24] is a widely used standard for facilitating device management over a network. Agents notify managers about events, while managers poll agents for data updates. A monitoring system building on SNMP may contain components from different vendors. However, SNMP was not used, because it is quite complex to implement, the communication between the agents and the manager is not efficient [2], and its data format is bounded to transfer commands. In contrast, Lemon uses a proprietary very simple data exchange format.

Ganglia [22] and Condor Hawkeye [10] use XML as data format. Lemon uses XML for querying and change notification.

In Ganglia, all nodes push all data to all cluster mates, and cluster representatives are polled by the manager. In contrast, Lemon stores data locally and forwards it directly to the manager, which is more efficient in large clusters.

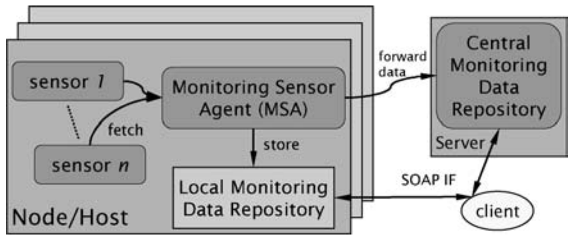
sensor 1
sensor n
fetch
Monitoring Sensor Agent (MSA)
store
Local Monitoring Data Repository
Node/Host
forward data
Central Monitoring Data Repository
Server
SOAP IF
client

Figure 5. The monitoring system consisting of sensors, a sensor controller, a local and a central monitoring data repository.

4.1. *Architecture of the Monitoring System*

The monitoring system consists of four components as shown in Figure 5.

On each machine different *sensors* periodically collect data. All sensors are controlled by the *Monitoring Sensor Agent* (MSA) which receives data samples and stores them in a *Local Monitoring Data Repository* (i.e. a cache). The MSAs also forward the data samples to a central repository.

4.1.1. *Data Sampling*

A configurable Monitoring Sensor Agent runs on all monitored hosts. The MSA is responsible for calling the plug-in sensors to sample the configured metrics. The system provides sensors for common performance and exception monitoring. Other sensors can be plugged in easily. The sampling frequency and the minimal change percentage required for a sample to be sent (smoothing) can be configured per metric. The interface is designed such that sensors are not required to answer to MSA sampling requests and may chose to trigger their own unsolicited samplings to MSA. The sensor communicates with the MSA over a normal UNIX socket using a proprietary simple text protocol.

The local monitoring data repository (cache) is available for local consumers of monitoring data. This is useful for allowing local fault tolerance correlation engines and may be used to resend data to the central repository in case of sending data has failed. The cache is implemented as a flat text file database, with one file per metric per day. Each line contains a single measurement in the format `timestamp value`.

4.1.2. *Data Access*

Data may be accessed through the repository API. The API does not distinguish different data types (must be handled by the client of the API). The API provides methods to insert samples into a repository, to query samples and to subscribe for change notification. The result of queries may contain one to many samples. Queries may be restricted to the latest measurement or may refer to measurements taken over a given period.

Bindings for various languages can be generated from the WSDL description of the API.

4.1.3. *Data Transport*

The transport of monitoring data from the monitored hosts to the central repository is also pluggable. Implementations for both UDP and TCP (prototypic) exist. The TCP based implementation uses permanently open sockets and includes a proxy like mechanism to fan-out the number of open connections on the central repository to a subset of the monitored hosts. On the proxy hosts the transport component of the MSA not only sends the monitoring data of the host itself, but it also receives and forwards data from other MSAs. The proxy environment must be configured statically.

4.1.4. *Data Storing*

The central measurement repository server uses the repository API to plug-in any database system as backend. So far, backends for flat files (same as for the local repository), Oracle (called OraMon), and ODBC (prototypic) have been developed.

5. Fault Detection and Recovery

The aim of the *Fault Detection and Recovery* system (FDR) is to provide automatic error detection and correction, i.e. self-healing of a fabric (objectives AM.1/2). In a large cluster or fabric one faulty node can cause serious problems for the whole grid. For example, a broken DNS server or a broken gateway may disconnect a whole fabric from the grid. Another problem may be that a normal computing node may cause a long delay in the analysis job or may even cause the total failure of a job.

For the described scenario, the Fault Detection and Recovery (FDR) system must cope with automatically running tasks that are not commonly found in todays cluster management systems:

- automatic error prevention, e.g. to prevent hardware damage caused by overheating,
- automatic error correction, e.g. by restarting crashed services,
- schedule repair tasks that would interfere with running jobs, e.g. freeing disk space on file servers.

The FDR system differs from existing tools like VACM [26], Patrol [18], and Performance Co-Pilot [19]. VACM is a centralized cluster administration system which is not able to react on alerts in more than one way. Patrol is very limited in its functionality, because it provides only a small set of services like CPU load monitoring, disk space controlling or watching the instances of running daemons. Performance Co-Pilot is useful for detecting performance problems in clusters, but does not support automatic recovery actions. Our fault tolerance software offers a non-centralized fault recovery system which is freely configurable and can combine results from more than one sensor to detect more complex faults.

5.1. *Architecture of the Fault-Tolerance System*

The FDR system is rule based. Each rule compares data retrieved from the monitoring system against configured limits. If the condition of a rule holds, the specified action is performed. By following a hierarchical approach, the FDR system supports handling faults both locally on each machine and remotely from specific machines dedicated to maintain the correct functioning of the diverse services in a Grid computing center. In Section 5.1.3 we show examples for both local and remote rules.

Figure 6 shows the components of the FDR system. These components are the *Fault Tolerance Correlation Engine* (see Section 5.1.1), the *Fault Tolerance Actuators* (see Section 5.1.2) and the *Fault Tolerance Rules* (see Section 5.1.3).

Because of the generic architecture of the fault tolerance components, any (soft) fault can be handled. In case of hardware faults, the system may not solve the problem but keep the fabric running by isolating faulty machines.

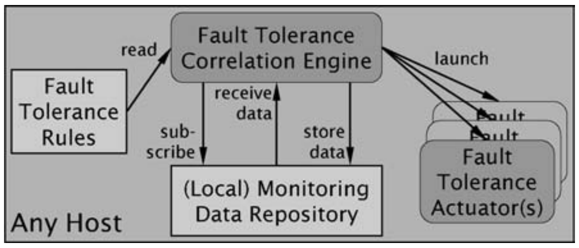
Fault Tolerance Correlation Engine
read
launch
Fault Tolerance Rules
receive data
sub-scribe
store data
(Local) Monitoring Data Repository
Fault Tolerance Actuator(s)
Any Host

Figure 6. Components of the fault-tolerance system.

5.1.1. *Fault Tolerance Correlation Engine*

The Fault Tolerance Correlation Engine (FTCE) is the active correlation engine. The FTCE runs as a daemon process on all hosts and is implemented to be robust to most system component failures. The FTCE processes observe one or several metrics stored in the MR to determine if something has gone wrong or is on its way to go wrong on the system. If so, it determines what recovery actions are needed, and launches the corresponding actions. Its output metric values contain a boolean flag that reflects if any fault tolerance actuators were launched, and if so, the identifiers of the actuators and their return status. The FTCE processing for a given metric is triggered either through a periodic sampling request from the MSA (see Section 4.1) or through the metric subscription/notification mechanism provided by the monitoring repository.

5.1.2. *Fault Tolerance Actuators*

A Fault Tolerance Actuator (FTA) is an implementation of the FaultToleranceActuator interface that executes automatic recovery actions. The FTA is typically driven by rules stored in the Configuration Management subsystem (see Section 3). A given FTA implementation can thus be used for several similar recovery actions, e.g. a single "daemon restart". Another example for an FTA, is a service restarter, i.e. by calling the restart method of the installed software packages.

5.1.3. *Fault Tolerance Rules*

A Fault Tolerance Rule (FTR) contains all necessary information about the controlled values on the nodes and the actuators that should be started if a value runs out of its defined limit. The interface for the actuator is as simple as possible: it may be a shell or an executable. The administrator is able to configure up to 64 levels of actuators, which may be started in sequence if the actuator that was started before was not able to fix the problem. FTRs are described in XML.

Generally, rules should be kept as simple as possible to prevent situations where different rules conflict each other. During the course of our project we found that (among others) rules concerning the observation of daemons, processor temperature, fan speed, disk usage, process zombies and partition table are useful on all machines. Also, the comparison of CPU load and/or memory usage for worker nodes can be used to identify unusual situations, for example caused by run-away processes from previous jobs.

```
<edg_ft_rule>
 <event>
  <level>
   <actuator shell="noshell">
    <actuatorname>sshd</actuatorname>
    <argument number="1">start</argument>
    <actuatorpath>/etc/init.d/</actuatorpath>
   </actuator>
  </level>
  <rule>
   <lookup>
    <node_id>node03.bar.foo</node_id>
    <metric_id>9501</metric_id>
   </lookup>
   <mo>!=</mo>
   <value>1</value>
  </rule>
 </event>
</edg_ft_rule>
```

Figure 7. Example of a fault tolerance rule being applied locally on a machine that serves as login (ssh) node.

```
<edg_ft_rule>
 <event>
  <level>
   <actuator shell="noshell">
    <actuatorname>switch_role</actuatorname>
    <argument number="0">node04.bar.foo</argument>
    <argument number="1">worker_node</argument>
    <argument number="2">login_node</argument>
    <actuatorpath>/opt/edg/sbin/</actuatorpath>
   </actuator>
  </level>
  <rule>
   <lookup>
    <node_id>node03.bar.foo</node_id>
    <metric_id>9506</metric_id>
   </lookup>
   <mo>></mo>
   <value>4</value>
  </rule>
 </event>
</edg_ft_rule>
```

Figure 8. Example of a fault tolerance rule being applied remotely to change the role of a machine from worker node to login node.

Figure 7 shows an example of a rule applied locally on a login machine. The example rule compares (op !=) a specific metric (id 9501) sampled at host node03.bar.foo against the value 1. If the condition holds, the command /etc/init.d/sshd start is executed, i.e. the daemon sshd is started.

Figure 8 shows an example of a rule applied remotely. The rule checks the load (metric 9506) on node node03.bar.foo against the value 4. If the condition holds (load higher than 4), the command switch_role is executed, i.e. the configuration database is updated such that node04.bar.foo will serve as login node in the future.

6. Resource Management

For the described scenario, the resource management system must cope with additional tasks that are not commonly found in today's cluster management systems:

– the support of jobs coming from various sources (OBJECTIVE JM.1),
– the interaction with different cluster management systems (OBJECTIVE JM.2),
– the provision of additional services for the grid layer (OBJECTIVE JM.2).

We developed a single component, the *Abstraction Layer* (AL) [21], that facilitates the deployment of advanced schedulers, like Maui [5] and provides a consistent interface to different cluster management systems.

Some cluster management systems already provide hooks to integrate external schedulers like Maui, but the integration is done differently for each system. Our system, in contrast, interfaces to various cluster management systems and schedulers via adaptors, without the need to implement separate adaptation interfaces for each combination.

A variety of cluster configuration suites, like OSCAR [16], SCMS [25], NPACI Rocks [17] exists, but none of them supports the scheduling of maintenance actions.

6.1. *Resource Management Architecture*

In most cluster batch systems, the scheduler interacts with a server/master to retrieve status information from the nodes to make its scheduling decisions. We illustrate different aspects of this interaction in the following sections. First, we discuss how a specific scheduler can interact with a specific server/master. Thereafter we show how to manage several clusters with one scheduler (within a computing center). With

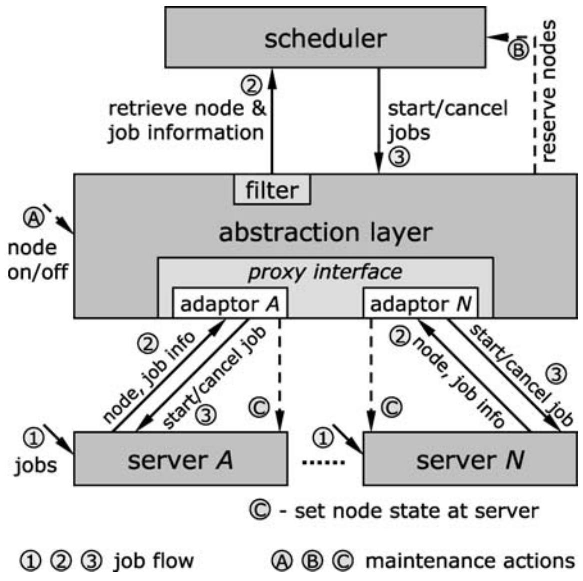

Figure 9. Support for job management and maintenance actions across several clusters.

our approach scheduling features can be added to cluster management systems in a non-intrusive way. We describe this at the hand of scheduling features that are missing in several cluster management systems.

6.1.1. *Consistent Management of Multiple Clusters*

We use an *abstraction layer* (AL) [21] between the scheduler and the cluster batch system's server. The AL filters information that is transmitted between the two components. This reduces the customization efforts in an environment with different schedulers and multiple cluster batch systems. When the source code of the server or the scheduler is not available, adding an abstraction layer may be the only possible solution. In addition, it helps hiding site-specific facilities or features like internal load balancing across clusters, resource brokerage, status information filtering, etc. The abstraction layer is kept generic and provides an proxy interface for plugging-in different adapters for different servers/schedulers.

The abstraction layer also allows to operate several clusters with a single scheduler as illustrated in Figure 9. Here, job management actions are depicted by solid arrows while maintenance tasks have dashed lines.

Incoming jobs are submitted to the server (step 1). The scheduler periodically asks for the current status of nodes and jobs, and the abstraction layer gathers this information by sending requests to the server. When finished, it sends the (filtered) information back to the scheduler (step 2). The scheduler determines a schedule and sends the decision to the server through the abstraction layer (step 3).

Only four operations are necessary to link scheduler and server by an abstraction layer: start job, cancel job, node info, and job info. Although the set of operations seems to be obvious, its composition was motivated by the *Wiki interface* of the Maui scheduler.

The execution of maintenance actions is illustrated by dashed arrows in Figure 9. First, the abstraction layer receives a request to switch a node on or off (step A). If it accepts the request, it reserves the specified nodes for the given time interval (step B) and sets the status of the nodes during that interval to *stopped* at the servers (step C).

6.1.2. *Extending the Functionality of Cluster Management Systems*

Most cluster management systems, like the Portable Batch System (PBS) [11], LoadLeveler (LL) [12], Sun Grid Engine (SGE) [23], Load Sharing Facility (LSF) [27], or the Computing Center Software (CCS) [13] provide a common set of scheduling features like fifo, backfill, etc. They mainly differ in their support for advanced scheduling capabilities like advance reservation. While users of stand-alone clusters may be able to cope with these feature lacks, Grid users are forced to confine themselves to the common set of basic scheduling features. Our approach, in contrast, allows to add those functions inside the abstraction layer if necessary (OBJECTIVE JM.2).

Note that up to now no cluster management system or scheduler supports the modification of node states like *on* and *off* in the future, which is necessary to support the planning and execution of maintenance actions.

6.1.2.1. Maintenance Actions As outlined in Section 1 automating the administration of large clusters is an important issue. Administrative tasks that may affect jobs running on the same node, must be taken into account by the scheduler by planning the maintenance task just like an ordinary job (OBJECTIVE JM.2).

A simple but effective method is to disable all affected nodes during the task. To schedule a maintenance action, the *admin* component contacts the resource management for a specific or flexible time slot on a set of nodes. The scheduler decides and schedules the node state change. If the request was successful, the administrative task can be performed during the agreed time slot. Our scheme of handling maintenance actions needs the capability to request time slots in the future for the coordinated planning of system maintenance. Not all cluster schedulers support advance reservations. Hence we replace them by a more powerful one, the Maui scheduler.

7. Gridification

Grid job submission and file access using GridFTP have traditionally been protected using a simple version of the Grid Security Infrastructure (GSI) [8]. Authorization in traditional GSI is based on a single user access list (the so-called `grid-mapfile`), explicitly naming individuals that are allowed to access a service. For those services that need a system-local principal, this list also provides a mapping between the user's distinguished name and that of the local principal.

The components LCAS (Local Center Authorization Service; OBJECTIVE JM.3) and LCMAPS (Local Credential MAPping Service; OBJECTIVE JM.4) represent two functions to access the local fabric: pure authorization and the assignment of local credentials, respectively. The main incentive for this split is to enable global authorization decisions to be made without the need to interact with system-local credential services.

Besides the traditional `grid-mapfile` solution for local site authorization described above, the PRIMA system [15] was developed, which is similar to LCAS. The PRIMA system is driven by an XACML policy and uses the globus authorization call-out mechanism in the Gatekeeper and GridFTP server. The development of this call-out mechanism was triggered among others by the development of LCAS, but it lacks the possibility to incorporate job characteristics in the authorization decision process, in contrast to LCAS and LCMAPS.

7.1. *Architecture of the Gridification Components*

Figure 10 shows the architecture of the Grid Access system and presents the interaction of the different components. A job request consisting of the usual description (e.g. executable name, in- and output files, wallclock limit, etc.) and a (proxy) certificate is sent to the Gatekeeper. Then the request must be authorized by the LCAS (indicated by (1) in Figure 10). For authorization the LCAS invokes a set of plug-ins until the request is denied or allowed to proceed (2). Next, the Gatekeeper asks the LCMAPS to acquire local credentials to the job and to enforce the use of these

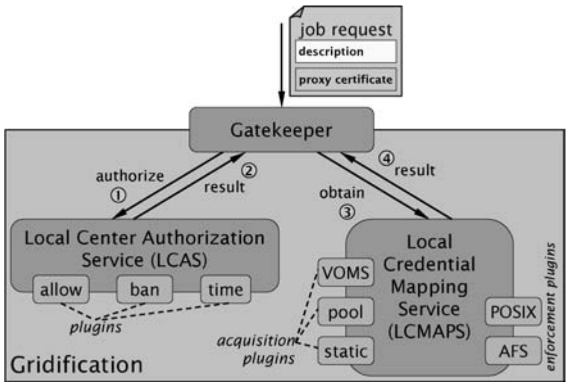

Figure 10. Components for managing access to fabric resources.

credentials, e.g. by setting the user ID (uid) and group ID (gid), etc. (3). If all actions have been passed successfully (4), the job is forwarded to the local cluster batch system.

7.1.1. *The Local Center Authorization Service*

The Local Center Authorization Service (LCAS) allows authorization decisions based on user credential information and service request characteristics. It provides a framework for pluggable authorization that is interfaced directly to the service daemon.

The LCAS framework is enabled by loading the LCAS service library in the service daemon (at this time both the Gatekeeper and GridFTP have been equipped with the appropriate hooks for communicating with LCAS). This LCAS library reads a text-based configuration file listing the authorization plug-ins to be invoked. Each plug-in is a stand-alone shared object, to be loaded on LCAS startup, with three pre-defined entry points (`initialize`, `confirm_authorization` and `terminate`). During authorization, the LCAS service will call each plug-in in turn, in the order specified in the setup file, until a module denies access to the request or no more modules are available. If any module denies the access, LCAS will return an authorization failure to the calling service (e.g. the Gatekeeper).

Since modules are stand-alone objects, they can be updated or replaced by the system administrator without recompilation of the service daemon itself. Also, new modules can be written and added to the authorization chain. Three standard modules are supplied with the system: an “allowed users” module (providing functionality similar to the traditional `grid-mapfile`), a “ban users” module (allowing instant denial of service to specific users) and a “time-slot” module (placing wallclock time constraints on acceptance of requests).

7.1.2. *The Local Credential Mapping Service*

The Local Credential Mapping Service (LCMAPS) is a pluggable framework like LCAS. However, in order to merge into pre-existing services that use the Grid Security Infrastructure, LCMAPS is equipped with a more advanced policy language (see Figure 11 for an example) and multiple entry interface. It can be used without recompilation or re-linking of either the Gatekeeper and GridFTP daemons. Virtually all existing computer systems require that a process or action is performed using one or more credentials. On traditional POSIX systems, this is a user ID (uid) and one or more group IDs (gids), with one specific uid and gid having elevated privileges. Other systems use AFS or Kerberos5 in lieu of, or next to the conventional POSIX authentication. Therefore, it is necessary to provide any user request that will create a process or access data directly via the filesystem layer with (a set of) local credentials.

Conventional GSI provides a direct one-to-one mapping between the client’s distinguished name (DN) and a pre-existing local credential. Moreover, the Gatekeeper service can also acquire a Kerberos5 ticket, if so instructed by the system administrator. There is no provision either for users that are not known to the system beforehand, or to acquire privileges based on VO membership, as e.g. provided by the VO Membership Service (VOMS) [1]. The former point (unknown users) has been addressed by the *pool accounts* extension to GSI [9], but this is still limited to conventional UNIX credentials (*uid* and *gid*), and does not support membership of multiple VOs.

LCMAPS provides a policy-driven framework for acquiring and enforcing local credentials, based on the complete security context, which includes the VOMS attributes contained in the user proxy certificate, and the job description. In addition a legacy interface is provided by which the credential mapping is based on only the user’s DN. For reasons of system integrity, LCMAPS comes only as a library and cannot operate as a stand-alone daemon. Also, since the operations that LCMAPS might perform can be expensive (like creating a new account on-the-fly) it is required that all relevant authorization decisions have been completed successfully by LCAS.

The LCMAPS system is initialized on service startup (Gatekeeper or GridFTP daemon). It consists of two principal components: a plug-in manager and an evaluation manager (not shown in the figure). The plug-in manager is in control of the plug-in modules and is the only component that has direct access to

```
 1  # module definitions
 2  vomsextract = "lcmaps_voms.mod                                 \
 3        -vomsdir /etc/grid-security/vomsdir/                      \
 4        -certdir /etc/grid-security/certificates/"
 5
 6  vomslocalgroup = "lcmaps_voms_localgroup.mod -mapmin 1         \
 7       -groupmapfile /opt/edg/etc/lcmaps/groupmapfile"
 8
 9  jobrep = "lcmaps_jobrep.mod                                     \
10        -vomsdir /etc/grid-security/vomsdir/                      \
11        -certdir /etc/grid-security/certificates/                 \
12        -jr_config /opt/edg/etc/lcmaps/jobrep_config"
13
14  posixenf = "lcmaps_posix_enf.mod -maxuid 1 -maxpgid 1          \
15          -maxsgid 32"
16
17  localaccount = "lcmaps_localaccount.mod                         \
18        -gridmapfile /etc/grid-security/grid-mapfile"
19
20  poolaccount = "lcmaps_poolaccount.mod                           \
21        -override_inconsistency                                   \
22        -gridmapfile /etc/grid-security/grid-mapfile              \
23        -gridmapdir /etc/grid-security/gridmapdir/"
24
25  # policies
26  voms:
27  vomsextract    -> vomslocalgroup
28  vomslocalgroup -> jobrep
29  jobrep         -> posixenf
30
31  standard:
32  localaccount -> jobrep | poolaccount
33  poolaccount  -> jobrep
34  jobrep       -> posixenf
```

Figure 11. Example configuration of LCMAPS.

them. The evaluation manager reads and compiles the policy description. Upon receipt of an LCMAPS request it asks the plug-in manager to run the plug-ins in the order prescribed by the policy.

There are two different logical module types: acquisition modules and enforcement modules. Acquisition modules look-up (or create) accounts in the system and assign group IDs, based on e.g. VOMS attributes. In the LCMAPS service a credential object is filled with the acquired credential identifiers. Enforcement modules take the content of the credential object and attempt to enforce the credentials listed. There is no difference in the interface between acquisition and enforcement modules. The result (success/failure) of the credential mapping is returned to the calling application.

Figure 11 shows an example policy definition for the LCMAPS state machine. There is a module definition (lines 2–23) followed by two policies called `voms` and `standard` (lines 26–34). The module definitions can be read like: `alias = "[path/]module_file [commandline_arguments_to_module]"`.

Next, are the policies which always start with their name followed by a colon (lines 26 and 31). These name declarations are followed by the actual rules, i.e. lines 27–29 for the `voms` policy and lines 32–34 for the `standard` policy. The policies are evaluated in the given order. For each policy, the rules are evaluated until the last rule or until a rule evaluates to false. If a rule of a policy evaluates to false, the next policy will be used or the whole call to the LCMAPS fails (i.e. if no further policy is available).

The arrow sign `->` combines two modules, e.g. `jobrep` and `posixenf`. The latter one is executed if the former is evaluated to the value true. If a rule evaluates to true, the next one of the same policy will be executed. The module following a pipe sign | is executed if the first module of that rule evaluated to false, e.g. line 32.

In the `voms` policy this means after a true state of `vomsextract` the next module to be executed is `vomslocalgroup`. If the state is `vomsextract` ended in a false state, there is nothing to execute. This means that this policy has failed. There still is another policy below this one (fail-over is only top to bottom). `standard` will be the active policy now starting with the `localaccount` module. On failure of this module `poolaccount` will be executed. If `poolaccount` successfully exits the `jobrep` module will be next. If this module also ends successfully `posixenf` will be executed. If `posixenf` turns out positive this policy has been completed and if a policy ends in a completed and successfull state, LCMAPS returns success to the caller. If `posixenf` failed then there is no other policy left to execute and LCMAPS will report failure.

8. Conclusion

In the course of the EU project DataGrid we designed, implemented and deployed a framework for the coordinated, autonomous management of multiple heterogeneous clusters in a fabric. The framework consists of the following building blocks:

– the *Resource Management System* for handling jobs from different sources,
– the *quattor* system for software installation and configuration,
– the *Lemon* component for monitoring the system's status,
– the *FT* mechanism for fault detection and recovery, and
– the *Gridification* scheme for integrating fabrics into a Grid.

These set of components allow to reduce the human maintenance in a cluster computing center drastically. With automation and rule based error handling the system is open for extensions and future requirements.

References

1. R. Alfieri, "VOMS: An Authorization System for Virtual Organizations", in *Proceedings of the 1st European Across Grids Conference, Santiago de Compostela*, Spain, 2003.
2. E. Anderson and D. Patterson, "Extensible, Scalable Monitoring for Clusters of Computers", in *Proceedings of the 11th Systems Administration Conference (LISA'97)*, San Diego, CA, USA, 1997.
3. P. Anderson and A. Scobie, "LCFG: The Next Generation", in *UKUUG Winter Conference*, 2002.
4. S. Bethke, M. Calvetti, H. Hoffmann, D. Jacobs, M. Kasemann and D. Linglin, "Report of the Steering Group of the LHC Computing Review", Technical Report, CERN European Organization for Nuclear Research, 2001.
5. B. Bode, D. Halstead, R. Kendall and Z. Lei, "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters", in *USENIX Conference*, Atlanta, GA, 2000.
6. M. Burgess, "Cfengine: A Site Configuration Engine", *USENIX Computing Systems*, Vol. 8, No. 3, 1995.
7. DataGrid, "EU DataGrid Project Homepage", 2004. `http://www.eu-datagrid.org/`
8. I. Foster, C. Kesselman, G. Tsudik and S. Tuecke, "A Security Architecture for Computational Grids", in *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, San Francisco, CA, USA, pp. 83–92, 1998.
9. A. Frohner, "DataGrid Security Design Report", Technical Report, EU DataGrid Project, 2003.
10. Hawkeye, "Condor Hawkeye Homepage", 2004. `http://www.cs.wisc.edu/condor/hawkeye/`
11. R. Henderson, "Job Scheduling under the Portable Batch System", in *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, Vol. 949, pp. 279–294, 1995.
12. S. Kannan, M. Roberts, P. Mayes, D. Brelsford and J. Skovira, *Workload Management with LoadLeveler*, IBM Redbooks, 2001.
13. A. Keller and A. Reinefeld, "Anatomy of a Resource Management System for HPC Clusters", in *Annual Review of Scalable Computing*, Vol. 3, 2001.
14. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing", *IEEE Computer*, Vol. 36, No. 1, 41–50, 2001.
15. M. Lorch, D.B. Adams, D. Kafura, M.S.R. Koneni, A. Rathi and S. Shah, "The PRIMA System for Privilege Management, Authorization and Enforcement in Grid Environments", in *Proceedings of the 4th International Workshop on Grid Computing – Grid 2003*, Phoenix, AR, USA, 2003.
16. OSCAR, "OSCAR Homepage", 2004. `http://oscar.sourceforge.net/`
17. P. Papadopoulos, M. Katz and G. Bruno, "NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters", *Concurrency and Computation: Practice and Experience*, Vol. 15, Nos. 7–8, 707–725, 2003.
18. Patrol, "Patrol Homepage", 2004. `http://www-d0en.fnal.gov/patrol/patrol_doc.html`
19. Performance Co-Pilot, "Performance Co-Pilot Homepage", 2004. `http://oss.sgi.com/projects/pcp/`
20. A. Reinefeld and V. Lindenstruth, "How to Build a High-Performance Compute Cluster for the Grid", in *2nd International Workshop on Metacomputing Systems and Applications (MSA2001)*, Valencia, Spain, 2001.

21. T. Roeblitz, F. Schintke and A. Reinefeld, “From Clusters to the Fabric: The Job Management Perspective”, in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster’03)*, Hong Kong, China, 2003.
22. F.D. Sacerdoti, M.J. Katz, M.L. Massie and D.E. Culler, “Wide Area Cluster Monitoring with Ganglia”, in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster’03)*, Hong Kong, China, 2003.
23. SGE, “Sun Grid Engine Homepage”, 2004. `http://www.sun.com/software/gridware/`
24. SNMP, “Simple Network Management Protocol”, 2004. `http://www.faqs.org/rfcs/rfc1157.html`
25. P. Uthayopas, J. Maneesilp and P. Ingongnam, “SCMS: An Integrated Cluster Management Tool for Beowulf Cluster System”, in *Proceedings of the International Conference on Parallel and Distributed Proceeding Techniques and Applications 2000 (PDPTA’2000)*, Las Vegas, NV, USA, 2000.
26. VACM, “VACM Homepage”, 2004. `http://vacm.sourceforge.net/`
27. S. Zhou, X. Zheng, J. Wang and P. Delisle, “Utopia: A Load Sharing Facility for Large, Heterogenous Distributed Computer Systems”, *Software – Practice & Experience*, Vol. 23, No. 12, 1305–1336, 1993.