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

Thomas Röblitz, Florian Schintke and Alexander Reinefeld
ZIB, Takustraße 7, D-14195 Berlin Dahlem, Germany

Olof Bärring, Maite Barroso Lopez, German Cancio, Sylvain Chapeland, Karim Chouikh, Lionel Cons, Piotr Poznañski, Philippe Defert, Jan Iven, Thorsten Kleinwort, Bernd Panzer-Steindel, Jaroslaw Polok, Catherine Rafflin, Alan Silverman, Tim Smith and Jan Van Eldik
CERN, CH-1211 Geneva-23, Switzerland

David Front
Weizmann Institute of Science, PO Box 26, Rehovot 76100, Israel

Massimo Biasotto, Cristina Aiftimiei, Enrico Ferro and Gaetano Maron
INFN, Viale dell'Università 2, I-35020 Legnaro (PADOVA), Italy

Andrea Chierici and Luca Dell'agnello
INFN, Viale Berti Pichat 6/2, I-40127 Bologna, Italy

Marco Serra
INFN, P.le Aldo Moro 2, I-00185 Roma, Italy

Michele Michelotto
INFN, Via Marzolo 8, I-35131 Padova, Italy

Lord Hess, Volker Lindenstruth, Frank Pister and Timm Morten Steinbeck
University of Heidelberg, Kirchhoff-Institut für Physik, Im Neuenheimer Feld 227, D-69120 Heidelberg, Germany

David Groep, Martijn Steenbakkers, Oscar Koeroo, Wim Som de Cerff and Gerben Venekamp
NIKHEF, PO Box 41882, 1009 DB Amsterdam, The Netherlands

Paul Anderson, Tim Colles, Alexander Holt and Alastair Scobie
University of Edinburgh, Old College, South Bridge, Edinburgh EH8 9YL, UK

Michael George and Andrew Washbrook
University of Liverpool, Oxford Street, Liverpool L69 7ZE, UK

Rafael A. García Leiva
Department of Theoretical Physics, Universidad Autónoma de Madrid, Cantoblanco, 28049 Madrid, Spain

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.

© 2004 *Kluwer Academic Publishers. Printed in the Netherlands.*

All presented components have been implemented in the course of the EU project DataGrid: The *Lemon* 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.

Keywords: autonomic computing, cluster computing, grid computing, system management

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 [24]. 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 improve the response time.

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 [17], 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

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

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 3 through 7) and conclude the paper with a summary of our results.

2. A Brief Outline of the Architecture

We illustrate the requirements on the architecture by the following scenarios. All scenarios assume a UNIX based fabric comprised of a single cluster and some management servers.

Job authorization: A user sends a job request to the fabric, including ordinary requirements (e.g. number of processors, input data, expected execution time, etc.) and a certificate. The entry point of the fabric validates both the ordinary requirements and the certificate. In particular, it checks whether the user is banned or permitted to access the fabric's resources. Before the request is handed over to the cluster batch system, a local UNIX account is provided for the request dynamically.

Fault tolerance: Each node provides a set of services. Most daemons may be both monitored and restarted remotely. However, some daemons cannot be restarted remotely, e.g. `sshd`, and require self-healing capabilities inside the node, because restarting daemons remotely is done via sshd. Hence, monitoring data about service daemons is periodically sampled. Then, the data is analyzed (i.e. comparing the actual state with the goal state) and repair actions are devised.

Scheduling of intrusive actions: If a new kernel has to be installed on a subset of the cluster nodes, this would affect running jobs at these nodes. Hence, the maintenance action is an intrusive one and must be scheduled at a time when no user job is executed at these nodes. Therefore the software installation system asks the resource management system to not schedule further jobs to these nodes. On each node the action may be scheduled at a different time.

The architecture reflects two key issues: (1) autonomous maintenance of the configuration of clusters and (2) job management.

Figure 1 depicts the interrelationships between the components, that we developed and implemented in the European *DataGrid* project [8], and shows how they work together to achieve an automated computing center management. A 'fabric' comprises several separate clusters with (possibly) different cluster management systems. Three kinds of jobs may enter the fabric:

– grid jobs (from the grid level above),

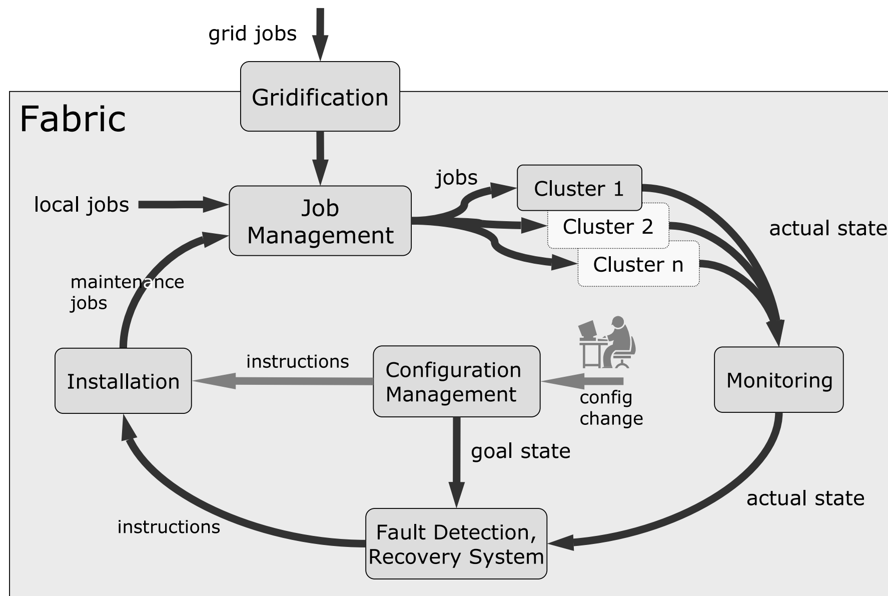

Figure 1. Management cycle for the automated maintenance of clusters in a compute center and their integration into a production Grid.

- local user jobs (injected from the left), and
- maintenance jobs (injected by the system itself).

Status information on the clusters' 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. 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.

3. 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 interaction with different cluster management systems,

- the support of jobs coming from various sources,
- the provision of additional services for the grid layer.

The Maui scheduler [5] provides scheduling algorithms and static configuration features required for our system. 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 [20], SCMS [29], NPACI Rocks [21] exists, but none of them supports the scheduling of maintenance actions.

3.1. 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. With 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.

3.1.1. *Consistent Management of Multiple Clusters*

We use an *abstraction layer* (AL) [25] 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 Fig. 2. 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).

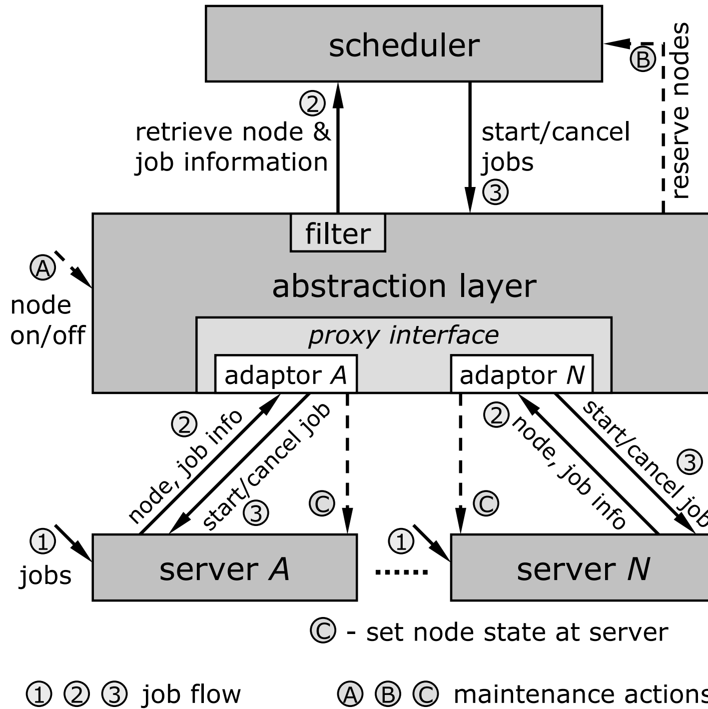

Figure 2. Support for job management and maintenance actions across several clusters.

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 Fig. 2. 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).

3.1.2. *Extending the Functionality of Cluster Management Systems*

Most cluster management systems, like the Portable Batch System (PBS) [14], LoadLeveler (LL) [15], Sun Grid Engine (SGE) [27], Load Sharing Facility (LSF) [32], or the Computing Center Software (CCS) [16] 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.

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

3.1.2.1. *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.

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.

3.1.2.2. *Advance Reservations* 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.

3.2. Current Status

We implemented the described architectural framework in a daemon that provides generic interfaces to both scheduler and server of a cluster batch system. Currently, our framework uses the Maui scheduler and supports OpenPBS and LSF as batch systems.

We performed experiments to determine the overhead of our implementation. The overhead of the main control methods (i.e. getNodes, getJobs and startJob) varies between 40 and 120 ms on a 16 node cluster. In [25] we discuss our implementation and the measurements in more details.

4. 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 briefly describe a system called *quattor* that addresses these issues. Quattor manages configuration information, installs software packages and configures the services provided by a fabric. The system is discussed in more detail in a separate paper in this journal [18].

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 comes with a new and feature-rich configuration language called "pan", improvements in the software management system, and more adherence to established standards, e.g. for initial system installation or service configuration management.

4.1. Architectural Overview

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.

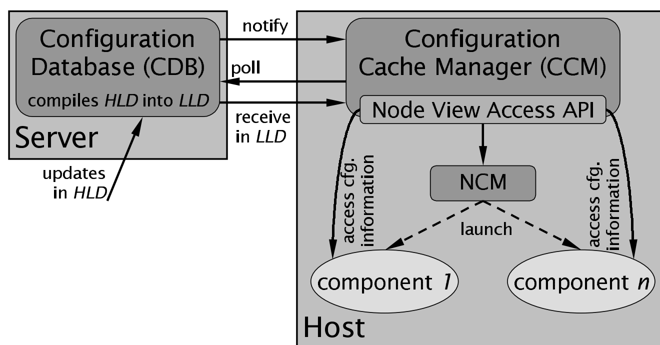

Figure 3. Components for managing configuration information

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 3. It consists of a central database that stores configuration information. The information is inserted or updated using a *High Level Description* language (HLD). Machine profiles are generated by compiling HLD information into *Low Level Description* documents using *Pan*. These machine profiles are stored in the 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).

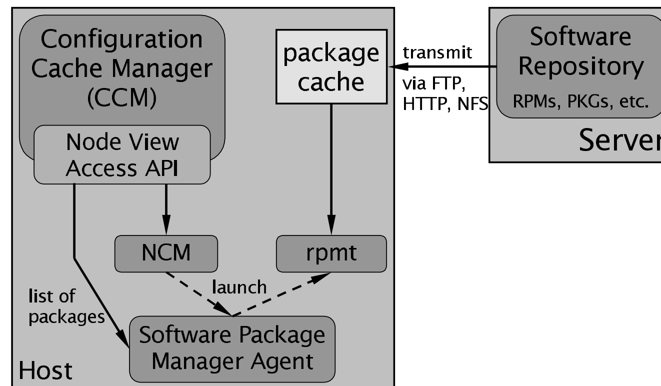

Figure 4. Components for managing software installation

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 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.

4.2. CURRENT STATUS

Quattor is successfully used at Cern's computing center. A detailed discussion of the results is included in the paper on quattor [18] in this journal.

5. Monitoring

Monitoring data about many different components must be gathered and stored to facilitate the self-healing of fabric services. 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 6). 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) [28] 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 [26] and Condor Hawkeye [13] 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.

5.1. ARCHITECTURE OF THE MONITORING SYSTEM

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

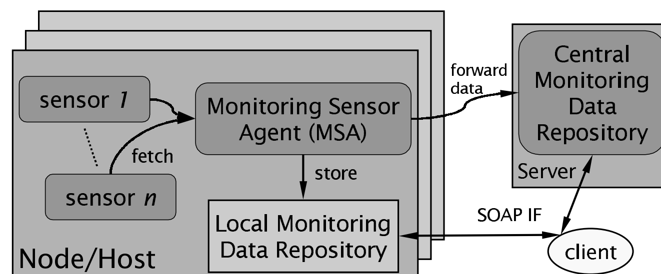

Figure 5. The monitoring system consisting of sensors, a sensor controller, a local and a central monitoring data repository.

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.

5.1.1. *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`.

5.1.2. *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.

5.1.3. *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.

5.1.4. *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.

5.2. Current Status

Lemon is has been used within the GridICE [12] project and at Cern's computing center for collecting data from over 1600 nodes for over a year. It has been observed that the MSA has a low footprint at monitored hosts, while approximately 150 sensors have been used. Also, a GUI was developed to let clients easily browse through data samples and handle alarms (prototype). OraMon provides an SQL interface to the repository.

6. 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. 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 system must cope with automatically running tasks that are not commonly found in todays cluster management systems:

- automatic error prevention,
- automatic error correction,
- schedule repair tasks that would interfere with running jobs.

The FDR system differs from existing tools like VACM [30], Patrol [22], and Performance Co-Pilot [23]. 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. The WP4 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.

6.1. 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. Fig. 6 shows the components of the FDR system.

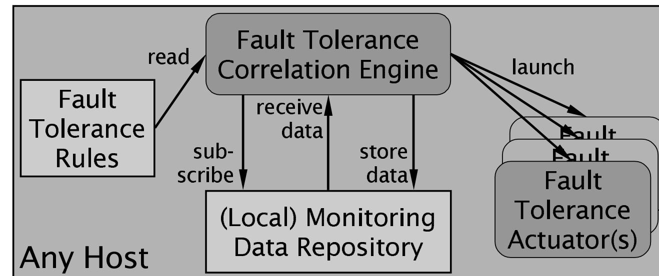

Figure 6. Components of the fault tolerance system

6.1.1. *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 5.1) or through the metric subscription/notification mechanism provided by the monitoring repository.

6.1.2. *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 4). 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.

6.1.3. *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. Fig. 7 shows an example of a rule.

```
<edg_ft_rule>
 <event>
  <level>
   <actuator shell="noshell">
    <actuatorname>pwd</actuatorname>
    <argument number="0"></argument>
    <actuatorpath>/bin/</actuatorpath>
   </actuator>
  </level>
  <rule>
   <lookup>
    <node_id>lxshare0314.cern.ch</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

The example rule compares (op !=) a specific metric (id 9501) sampled at host `faulty` against the value 1. If the condition holds, the command `/bin/pwd` is executed.

6.2. Current Status

We implemented the described architectural framework in a daemon (FTCE) that provides a generic interface for the rules. By design, the framework is highly scalable, because most of the activity may run locally on a single node. An administrator can easily setup correlations of many remote nodes and can start actuators remotely. Therefore a global fault tolerance server for a whole fabric may not be necessary, because the administrator can distribute this functionality on a large number of nodes to increase the reliability of the fabric. In this scenario a normal computing node may be responsible for its

own functionality and for one part of the fabric as well, e.g. for a switch, a UPS or a fabric wide air conditioning system which is connected to the monitoring.

7. Gridification

Grid job submission and file access using GridFTP have traditionally been protected using a simple version of the Grid Security Infrastructure (GSI) [10]. 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) and LCMAPS (Local Credential MAPping Service) 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. Such local services are likely to be more resource-intensive than the pure authorization decision itself.

Besides the traditional `grid-mapfile` solution for local site authorization described above, the PRIMA system [19] 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 8 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 Fig. 8). 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 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.

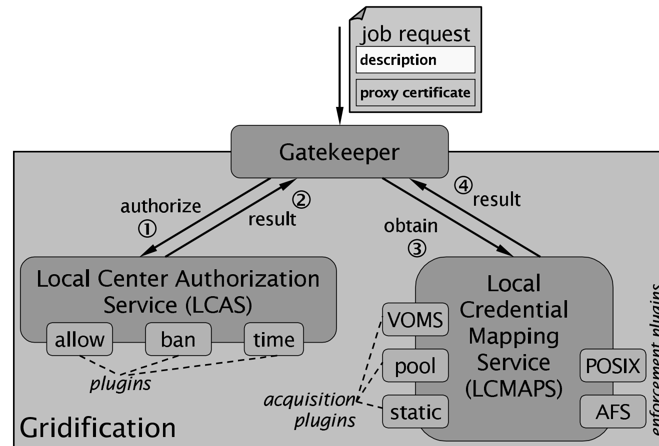

Figure 8. Components for managing access to fabric resources

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 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 [11], 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 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.

7.2. Current Status

All the Gridification components have been deployed in all EU DataGrid testbed sites. They have also been adopted by other grid projects such as Crossgrid [7]. Also the project VL-E (Virtual Laboratory for E-science) in the Netherlands [31] has chosen to use the Gridification components.

LCAS and LCMAPS will continue to evolve within a grid services architecture in the framework of the security joint research activity in the EGEE project [9]. LCAS will probably take on the use of XACML (a likely GGF standard for expressing access policies) for expressing VO access rights.

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. Alfieri, R.: 2003, 'VOMS: An Authorization System for Virtual Organizations'. In: *Proceedings of the 1st European Across Grids Conference, Santiago de Compostela, Spain*.
2. Anderson, E. and D. Patterson: 1997, 'Extensible, Scalable Monitoring for Clusters of Computers'. In: *Proceedings of the 11th Systems Administration Conference (LISA'97), San Diego, CA, USA*.

3. Anderson, P. and A. Scobie: 2002, 'LCFG: The Next Generation'. In: *UKUUG Winter Conference*.
4. Bethke, S., M. Calvetti, H. Hoffmann, D. Jacobs, M. Kasemann, and D. Linglin: 2001, 'Report of the Steering Group of the LHC Computing Review'. Technical report, CERN European Origanization for Nuclear Research.
5. Bode, B., D. Halstead, R. Kendall, and Z. Lei: 2000, 'The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters'. In: *Usenix Conference*. Atlanta, GA.
6. Burgess, M.: 1995, 'Cfengine: a site configuration engine'. *USENIX Computing systems* **8**(3).
7. CrossGrid: 2004, 'CrossGrid Project Homepage'. `http://www.crossgrid.org/`.
8. DataGrid: 2004, 'EU DataGrid Project Homepage'. `http://www.eu-datagrid.org/`.
9. EGEE: 2004, 'EU EGEE Project Homepage'. `http://www.eu-egee.org/`.
10. Foster, I., C. Kesselman, G. Tsudik, and S. Tuecke: 1998, 'A Security Architecture for Computational Grids'. In: *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference, San Francisco, California, USA*. pp. 83–92.
11. Frohner, A.: 2003, 'DataGrid Security Design Report'. Technical report, EU DataGrid project.
12. GridICE: 2004, 'GridICE Project Homepage'. `http://server11.infn.it/gridice/`.
13. Hawkeye: 2004, 'Condor Hawkeye Homepage'. `http://www.cs.wisc.edu/condor/hawkeye/`.
14. Henderson, R.: 1995, 'Job Scheduling under the Portable Batch System'. In: *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*. pp. 279–294.
15. Kannan, S., M. Roberts, P. Mayes, D. Brelsford, and J. Skovira: 2001, 'Workload Management with LoadLeveler'. IBM Redbooks.
16. Keller, A. and A. Reinefeld: 2001, 'Anatomy of a Resource Management System for HPC Clusters'. In: *Annual Review of Scalable Computing*, Vol. 3.
17. Kephart, J. O. and D. M. Chess, 'The Vision of Autonomic Computing'. *IEEE Computer* **36**(1), 41–50.
18. Leiva, R. G., and et al., 'Quattor: Tools and Techniques for the Configuration, Installation and Management of Large-Scale Grid Computing Fabrics'. *Kluwer Journal of Grid Computing*.
19. Lorch, M., D. B. Adams, D. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah: 2003, 'The PRIMA System for Privilege Management, Authorization and Enforcement in Grid Environments'. In: *Proceedings of the 4th Int. Workshop on Grid Computing - Grid 2003, Phoenix, AR, USA*.
20. OSCAR: 2004, 'OSCAR Homepage'. `http://oscar.sourceforge.net/`.
21. Papadopoulos, P., M. Katz, and G. Bruno: 2003, 'NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters'. *Concurrency and Computation: Practice and Experience* **15**(7-8), 707–725.
22. Patrol: 2004, 'Patrol Homepage'. `http://www-d0en.fnal.gov/patrol/patrol_doc.html`.
23. Performance Co-Pilot: 2004, 'Performance Co-Pilot Homepage'. `http://oss.sgi.com/projects/pcp/`.
24. Reinefeld, A. and V. Lindenstruth: 2001, 'How to Build a High-Performance Compute Cluster for the Grid'. In: *2nd International Workshop on Metacomputing Systems and Applications (MSA2001)*. Valencia, Spain.
25. Roeblitz, T., F. Schintke, and A. Reinefeld: 2003, 'From Clusters to the Fabric: The Job Management Perspective'. In: *Proceedings of the IEEE Intl. Conference on Cluster Computing (Cluster'03), Hong Kong, China*.

26. Sacerdoti, F. D., M. J. Katz, M. L. Massie, and D. E. Culler: 2003, 'Wide Area Cluster Monitoring with Ganglia'. In: *Proceedings of the IEEE Intl. Conference on Cluster Computing (Cluster'03), Hong Kong, China*.
27. SGE: 2004, 'Sun Grid Engine Homepage'. `http://www.sun.com/software/gridware/`.
28. SNMP: 2004, 'Simple Network Management Protocol'. `http://www.faqs.org/rfcs/rfc1157.html`.
29. Uthayopas, P., J. Maneesilp, and P. Ingongnam: 2000, '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, Nevada, USA.
30. VACM: 2004, 'VACM Homepage'. `http://vacm.sourceforge.net/`.
31. VL-E: 2004, 'Virtual Laboratory for E-science Homepage'. `http://www.vl-e.nl/`.
32. Zhou, S., X. Zheng, J. Wang, and P. Delisle: 1993, 'Utopia: A Load Sharing Facility for Large, Heterogenous Distributed Computer Systems'. *Software – Practice & Experience* **23**(12), 1305–1336.