Issues and Scenarios for Self-Managing Grid Middleware

Philippe Collet
Université de Nice Sophia Antipolis, I3S - CNRS — EPU, 930 route des Colles 06903 Sophia Antipolis, France
philippe.collet@unice.fr

Filip Křikava
Université de Nice Sophia Antipolis, I3S - CNRS — EPU, 930 route des Colles 06903 Sophia Antipolis, France
filip.krikava@i3s.unice.fr

Johan Montagnat
CNRS, I3S laboratory EPU, 930 route des Colles 06903 Sophia Antipolis, France
johan@i3s.unice.fr

Mireille Blay-Fornarino
Université de Nice Sophia Antipolis, I3S - CNRS — EPU, 930 route des Colles 06903 Sophia Antipolis, France
blay@polytech.unice.fr

David Manset
Maat Gknowledge
Méjico, 2.
45004 Toledo, Spain
dmanset@maat-g.com

ABSTRACT

Despite significant efforts to achieve reliable grid middlewares, grid infrastructures still encounter important difficulties to implement the promise of ubiquitous, seamless and transparent computing. Identified causes are numerous, such as the complexity of middleware stacks, dependence to many distributed resources, heterogeneity of hardware and software operated or incompatibilities between software components declared as interoperable. Based on failures that occurred during a large data challenge run on a grid dedicated to neuroscience, we identify scenarios that can be handled through autonomic management associated to the grid middleware. We also outline a flexible self-adaptive framework that aims at using model-driven development to facilitate the engineering, integration and reuse of MAPE-K loops in large scale distributed systems.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Experimentation, Reliability

Keywords

Grid Computing, Self-Adaptive Systems, Autonomic Computing, Model Driven Engineering, Medical Image Analysis, SALTY, neuGRID, SCA

1. INTRODUCTION

Grid infrastructures have become a critical substrate for supporting scientific computations in many different application areas. Over the last decade, world-wide scale grids (*e.g.* EGEE[1], OSG[2], PRAGMA[3]) leveraging the Internet capabilities have been progressively deployed and exploited in production by large international consortia. They are grounded on new middleware federating the grid resources and administration frameworks and enabling the proper operation of the global system 24/7. Despite all efforts invested both in software development to achieve reliable middleware and in system operations to deliver high quality of service, grids encounter difficulties to implement the promise of ubiquitous, seamless and transparent computing.

The causes are diverse and rather well identified. They notably include i) the complexity of middleware stacks, making it extremely difficult to validate code; ii) the dependence of the overall infrastructure to many distributed resources (servers, network) which are prone to hardware failures and exogenous interventions; iii) the heterogeneity of hardware and software operated, leading to almost infinite combinations of inter-dependencies; iv) the uncontrolled reliability of the application codes enacted that sometimes has side-effects on the infrastructure; v) the incompatibilities between software components although they were meant to be interoperable; vi) the difficulty to identify sources of errors in a distributed, multi-administrative domains environment; vii) the challenging scale of the computing problems tackled; The practice demonstrates that the human administration cost for grids is high, and end-users are not completely shielded from the system heterogeneity and faults. Heavyweight operation procedures are implemented by the grid administrators and users have to explicitly deal with unreliability issues [17].

Acknowledging the fact that middleware can hardly achieve complete reliability in such a challenging context, new operation modes have to be implemented to make grid systems resilient and capable of recovering from unexpected failures. Recently, there has been a lot of effort put into considering alternative paradigms and techniques that are based on principles used by biological system or in control engineering. These approaches, referred to as Autonomic Computing,

GMAC'10, June 7, 2010, Washington, DC, USA.
ACM 978-1-4503-0100-8/10/06.

[1]Enabling Grids for E-sciencE, `http://www.eu-egee.org`
[2]Open Science Grid, `http://www.opensciencegrid.org`
[3]Pacific Rim Applications and Grid Middleware Assembly, `http://www.pragma-grid.net`

aim at realizing computing systems and applications managing themselves with minimal or no human intervention [25]. Such systems then provide some self-management properties, mainly *self-configuration*, *self-healing*, *self-optimization*, *self-monitoring* and *self-protection*. Autonomic systems are thus characterized by their ability to detect, devise and apply adaptations when needed. There has been a considerable effort recently on combining Grid computing with techniques of autonomic computing [16, 20, 2, 21, 5, 18, 10, 9, 8]. In this paper we are focusing on an autonomic architecture for a grid middleware supporting computational science.

The objective of this work is to outline a flexible self-adaptive framework, SALTY[4] (Self-Adaptive very Large disTributed sYstems), which is designed to tackle the inevitable reliability problems encountered when operating grids and other large scale distributed systems. A generic solution to address the multiple potential sources of error is difficult to achieve. SALTY is therefore designed to be highly reconfigurable and can be instantiated at different levels of the controlled system. The SALTY framework follows a model-driven development approach to facilitate the engineering, integration and reuse of self-adaptive capabilities in large scale distributed systems. These capabilities are organized in a now classic MAPE-K feedback control loop [1], which architectures autonomic managers around four consecutive activities (*Monitor*, *Analyse*, *Plan*, *Execute*), sharing some abstract *Knowledge* on the controlled system.

To demonstrate the feasibility of our approach, we first analyze the requirements to integrate the SALTY framework in a specific Grid middleware. This sets the need to be as non-intrusive as possible in order to minimize the implementation effort and the possible side-effects (Section 2). A large part of the current design of SALTY is grounded on a production-level grid infrastructure deployed in the context of the neuGRID project[5] (an infrastructure for medical science) that addresses societal challenges related to Alzheimer's disease. Recent experience with the operation of the neuGRID platform was collected and analyzed (Section 3) and representative autonomic scenarios are deduced (Section 4). Some significant parts of their implementation within SALTY are considered while giving an overview of the framework, which is undergoing implementation, and discussing expected benefits (Section 5).

2. CHALLENGES AND REQUIREMENTS

SALTY defines a generic framework for building self-adaptations that tackle different use cases and adapt to different deployment scenarios. In the context of this paper, we are considering the deployment of SALTY over the neuGRID infrastructure. neuGRID is operating the pan-European gLite middleware [12] for core functionality. gLite adopts a Service-Oriented Architecture, although the heterogeneity of the software components integrated does not comply to a single interface definition. gLite was operated in production on the EGEE grid infrastructure over several years. Despite continuous improvements and fixes, some system limitations are regularly encountered that impact application performance, cause faults, and in some cases lead to core services crashes. Operational problems are usually identified manually through a grid-wide ticketing system. The problems reported are then treated according to their severity and either lead to service reconfiguration (including new hardware deployment if needed) or to software re-engineering.This entire procedure involves significant manual effort and makes operating the Grid quite expensive both time-wise and cost-wise.

SALTY is used to improve gLite operation on the target infrastructure by integrating autonomic capabilities through MAPE-K control loops – the reference standard from the IBM Autonomic Computing Initiative that codifies an external, feedback control loop approach in *Monitor-Analyze-Plan-Execute-Knowledge* model [1]. For each MAPE-K loop, the *Monitor* parts collect, filter and aggregate information from some sensors on the managed resources. The *Analyze* elements correlate and reify complex situations that might lead to some adaptations through the rest of the loop. When a change in the system context is identified in the analysis, the *Plan* parts, following high-level policies, build the necessary actions to achieve the loop adaptation objectives. Finally, the *Execute* functions implement the adaptation actions that directly interact with effectors on the managed resources. Applied to a gLite environment, the necessary control loops have to be driven by a network of sensors monitoring the grid activity. The sensors output should then be analyzed and autonomic system management is to be considered to improve system reliability and performance.

Management plans should address three major challenges related to infrastructure operation:

- Services self-protection. The most critical requirement is to detect system overloads and prevent services from crashes. It is important to ensure that service performances degrade gracefully rather than leading to a complete interruption.
- Services tuning. Multiple deployment and configuration parameters control the performance of the services operated. Service tuning is usually a costly manual process. Self-adaptation of service parameterization helps in achieving good performance by adapting to the operation conditions (variable workload and infrastructure resources volatility).
- Frequent job faults detection. The reason for some faults may be difficult to identify, yet some patterns causing frequent faults may be learned and avoided.

In the remainder of the paper, typical problems encountered when exploiting the neuGRID infrastructure for handling intensive data processing tasks are identified. The SALTY framework should tackle the distributed nature of the managed resources on a large scale to attempt solving these issues with autonomic capabilities.

3. NEUGRID DATA CHALLENGE

The neuGRID European infrastructure aims to support the neuroscience community in carrying out research on the neurodegenerative diseases. In neuGRID, a collection of large amounts of imaging data is paired with a grid-based computationally intensive data analyses. The infrastructure is developed to run neuroimaging and data-mining pipelines of algorithms, in particular specializing on Alzheimer's disease research with the analysis of cortical thickness from 3D Magnetic Resonance (MR) brain images. Capitalizing

[4] `http://salty.unice.fr`
[5] `http://www.neugrid.eu`

on the databases acquired in the US (ADNI Project[6]) and Europe (EU-ADNI Project[7]) respectively, up to 13,000 MR scans of the head should ultimately be archived in the infrastructure, thus constituting the largest ever standardized database in the field. Expected to be completed in early 2011, neuGRID will provide neuroscientists and potential pharmaceutical industries with a harmonized framework and a powerful distributed environment to seamlessly create, use, combine and validate algorithm pipelines to process acquired data and thus support clinical trials activity.

The neuGRID project is the first project within the neuroscientific community to use the Grid technology. Pipelines manipulated in neuGRID are computationally intensive as they enact a mixture of both short and long running I/O demanding algorithms that are applied over large data sets containing tens of thousands of images. It thus brings underlying Grid resources to their limits and highlights technological bottlenecks which must be addressed through appropriate scheduling optimization, data replication and fine tuning of the grid infrastructure. As an example, the formerly cited cortical thickness pipeline takes approximately 15 hours of CPU time when executed on a regular workstation and applied to only one brain. In the context of population pattern searching, applying the cortical thickness over 13,000 scans would simply be a waste of time with a single PC, bringing it to 22 CPU-years. In the target deployment of the neuGRID project with 4 European sites, each hosting about 20 quad-core CPUs paired with 5TB of effective storage, the execution time of the example case could shrink down to matter of weeks. To enable such a massive amount of data and to adequately service on demand computing power, neuGRID is utilizing a Grid infrastructure based on the gLite middleware [13]. The multiple institutions involved in the neuGrid testbed are another motivation for using a Grid middleware, since regular cluster-based solutions do not apply to an environment spanning over different administrative domains.

3.1 gLite Middleware Overview

The gLite middleware has been developed as a part of the European project EGEE which delivers a reliable and dependable European Grid infrastructure for e-Science. gLite *Workload Management System* (WMS), which is the subject of our scenarios, is architected as a two-level batch system that federates resources delivered by multiple computing *sites*. Each site is exposing its *Worker Nodes* computing units (WN) through a *Computing Element* (CE) gateway. A high-level meta-scheduler, called the WMS, is used as a front end to multiple CEs.

Grid applications are sliced in smaller computing *jobs*. Each job is described through a *Job Description Language* (JDL) document that describes the executable code to invoke and specifies the associated specific requirements. Jobs are submitted from a client *User Interface* (UI) to the WMS. The WMS is responsible for resources identification and job management across Grid resources, in such a way that jobs are conveniently and efficiently executed (fig. 1). Effectively, the job enters the WMS through a simple web service base interface (WMProxy) and is passed to the *Workload Manager* (WM) to be queued into a file system-based *Task Queue* (TQ). A matchmaking operation then takes place to identify available and suitable resources. The matchmaking is done by interrogating the *Information Supermarket* (ISM), an internal information cache, to determine the status and availability of computational and storage resources and query the *Logical File Catalogue* (LFC) to find locations of any required input files. Once an appropriate CE has been found, the WMS delegates the job processing to the CE batch manager where it is queued until a WN can process it. The job scheduling policy configured in neuGRID's WMS is eagerly scheduling, so that a job is matched against the resources and passed on for execution as soon as possible.

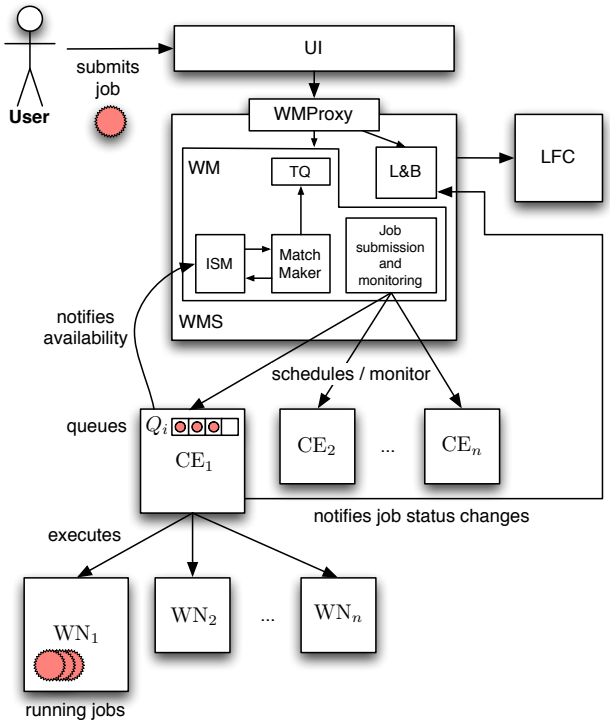

Figure 1: gLite job submission overview (a logical schema)

When submitted, a job goes through a sequence of states. The change from one state to another as well as other important events in the job life-cycle, such as finding a matching CE, are being tracked by the *Logging and Bookkeeping service* (L&B). These events are passed to a physically close component of the L&B infrastructure in order to avoid any sort of network problems. These components are responsible for persisting events and delivering them to one of the bookkeeping servers. This server processes them and provides a higher level view of the job states (*submitted, running, done, etc*) together with various attributes like the job's JDL, matched CE, exit code, etc.

3.2 Data Challenge

A part of the neuGRID project is a set of validation tests that are run within the infrastructure in order to verify its good performance while meeting user requirements specification. These performance tests are executed in the form of *data challenges* in which a very large data set of medical im-

[6] `http://www.adni-info.org`
[7] `http://www.centroalzheimer.it/E-ADNI_project.htm`

ages is analyzed, hence stressing the underlying infrastructure. The most recent data challenge (as of this publication) consisted in analyzing the entire dataset of the US-ADNI data using the CIVET pipeline [15], which contains 715 patients with 6,235 scans in MINC[8] (Medical Image NetCDF) format, representing roughly 108 GB of data. Each scan is about 10 to 20 MB and contains between 150 to 250 slices. The experiment ran for less than 2 weeks producing approximately 1TB of data and at peak performance utilizing 184 cores in parallel. The deployment schema of neuGRID is shown in Figure 2.

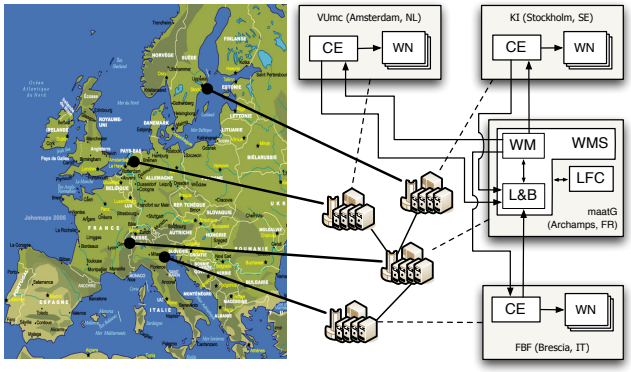

Figure 2: neuGRID deployment

The data challenge consists of a *parametric* job that is submitted into the gLite middleware in the very same way as presented in section 3.1. A parametric job is a job that allows the creation of a bulk of similar jobs that only differ in arguments, and submits them as a single job. The WMS then breaks the parametric job into many single jobs and submits them separately into CEs on the users behalf, thus significantly reducing the time needed for the jobs submission.

During the data challenge several observed problems required significant interventions from the operating personnel, not only resulting in prolonged execution time, but more importantly in higher cost. They vary in nature (hardware/middleware/application) and severity.

A representative hardware failures encountered was a power-failure resulting in the shut down of the entire site (CE), which had to be manually recovered. Submitted jobs were automatically pushed to the alternate CE and because of this sudden extra load, the alternate site got overloaded and crashed (see below).

Representative middleware failures encountered are

- WMS service overload - not able to handle all submitted jobs and had to be manually reconfigured and restarted.
- WMS service crashing - due to memory leak in the middleware and had to be manually restarted while pending jobs were rescheduled.
- CE service overload - not able to handle all submitted jobs and had to be manually reconfigured and restarted.
- LFC service overload - not able to handle too many requests from the many services within the Grid and required a workaround handling timeouts to be developed because of LFC not responding.

Representative application failures encountered:

- Library incompatibilities between CIVET pipeline and WN operating system. It is very difficult to trace what is the exact cause. The jobs had to be manually rescheduled.
- Bad data - ADNI images not fully quality assessed. In cases where workflows could not be recovered, they had to be manually rescheduled.
- Problems in the pipeline itself. Affected jobs had to be manually rescheduled.

Hardware failures are in general difficult to address, but here we focus more on their effect on the infrastructure than on their root cause.

During the data challenge run, the WMS had detected the problem of a computing site not being available and then correctly resubmitted all jobs to the other available site. However putting an additional load of approximately 3,000 jobs to the second CE caused its failure. So the effect of the hardware issue resulted in an additional failure in the middleware layer, finally leaving the entire Grid without any computational site.

All impacts of the described issues are quite significant to the normal operation and maintenance of the grid. Consequently, managing such problems through additional autonomic capabilities are likely to bring important benefits to other data challenge runs and on the normal day-to-day operation of the infrastructure.

4. REPRESENTATIVE AUTONOMIC SCENARIOS

Some recent work in the area of self-adaptive systems has been focused on how computational applications can benefit from autonomic computing concepts (for example [10, 9, 20]). In our case, the considered applications in neuGRID are based on the existing medical image analysis pipelines, which must not be modified. Our objective is instead to introduce self-adaptive capabilities to the Grid middleware itself, regardless of the applications that are executed on it.

The development of gLite is done in a fairly closed environment and not much information is available on how to change or extend its functionality. Furthermore, since gLite is rather a complex system, its deployment and configuration are quite difficult tasks [11]. Therefore the proposed solutions attempt to avoid any modification of the middleware code. The proposed adaptation is designed as an external subsystem that is deployed next to the middleware without deep intrusion. We consider gLite to be a black box with which we can only communicate using interfaces such as provided system commands, configuration files, log files, process signals, etc.

In order to be able to use directly these interfaces, we need to have an administration access to the infrastructure. Since neuGRID is a private Grid, this kind of access can be granted. This allows us to directly interact with the system, collect information about the runtime context from

[8] `http://www.nitrc.org/projects/minc/`

various sources (such as low operating system probes and logs), modify the configuration files, etc. Our aim is to first demonstrate the benefits of the self-adaptive behaviour in private Grid setups so that a potential adoption of the proposed techniques in other infrastructures like the EGEE grid can be envisaged.

The engineering of the following self-adaptive scenarios is based on a feedback control loop organized with the MAPE-K principles presented in section 2. The presented scenarios are motivated by the middleware related issues from the data challenge experiment, but also by the recurring issues on the EGEE Grid in which the gLite middleware is also deployed. We present the scenarios in a bottom-up way, first concentrating on a concrete failure and building up to a more generic solution that is applicable in other gLite based deployments as well as in other Grid systems.

4.1 WMS overload

The WMS overload is usually caused either i) by receiving more requests that it can handle or ii) because of a software problem in the component itself, *e.g.* a memory leak such as the one encountered during the data challenge.

To deal with this kind of failure, an additional self-healing control loop should be deployed into the infrastructure. This loop interacts with the WMS host's low level operating system probes and periodically monitors CPU and memory utilization of the WMS process. An overload is detected when the resource utilization exceeds a certain threshold value (fig. 3). We define two threshold values with associated adaptation mechanisms: 1. *blocking* threshold T_0 and 2. *restarting* threshold $T_1, (T_1 > T_0)$.

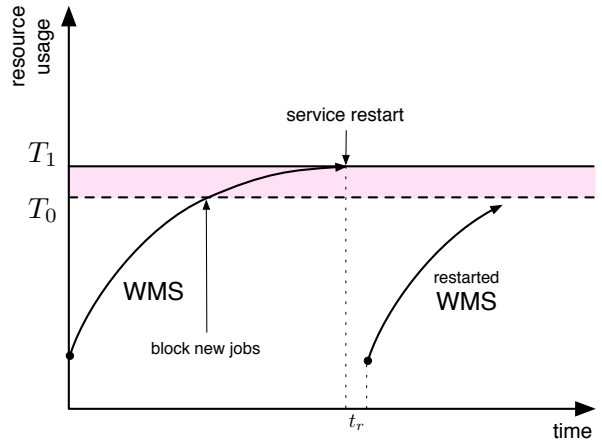

Figure 3: WMS overload

When the loop detects that T_0 is exceeded, the adaptation mechanism will block all incoming jobs from entering the system. It does that by putting the WMProxy into a drain mode that prevents it from accepting any new job submission requests. This should remove a part of the load and therefore enables the WMS to recover (unless the overload is caused by a software defect). This will also allow the service to process as many already queued jobs from its TQ as it can, before the resource usage reaches the second threshold T_1. At the point when T_1 has been exceeded, the adaptation mechanism will restart the WMS process itself. All the job management services, together with monitoring will cease for the duration of the service restart t_r. If the system has recovered and its resource usage has dropped below the blocking threshold T_0, the WMProxy will again be enabled to accept new job submission requests.

At first, both T_0 and T_1 are empirical, but the next step is to make them to evolve during the system life time so they adapt to the current system context [4].

The monitoring part of the control loop should also be self-adaptive. Instead of taking the resource usage samples at a constant rate, it should adapt the rate frequency based on the load in the system. The higher the load is, the shorter the sampling intervals should be in order to have very precise information about the system and execute the adaptation policy on time.

Figure 4 illustrates the adaptation of sampling rate according to the resource utilization. The concrete model of the monitoring adaptation is also to be improved and simple statistical models are intended to be experimented first [19].

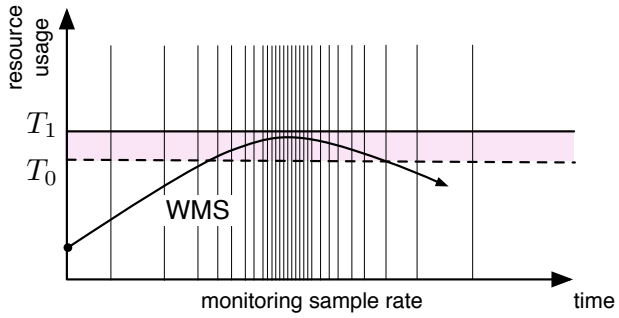

Figure 4: WMS resource usage adaptive monitoring

In a similar way, the same scenario can be applied to others Grid middleware components that tend to be overloaded. As identified during our data challenge both CE and LFC components were overloaded, due to the handling of large numbers of jobs. The above proposed scenario can be also applied to these cases and provides a self-healing autonomous capability to other components in a Grid. In the case of the CE, for instance, the scenario is merely the same, but instead of blocking jobs from users, it blocks jobs from WMS. Because of the extensive computational load that is being put on Grids, the components overload is not a rare event. Grid clients are usually fault resilient, resubmitting failed requests after some time). Therefore we expect this self-healing technique to make Grid middleware more robust against high system load and thus reducing manual intervention.

4.2 CE Starvation

During the data challenge run, when the CE had disappeared because of the power failure, the WMS correctly detected the situation and rescheduled all jobs to the other site that remained available. However, the sudden schedule of all these jobs resulted in a complete overload at the other site. This could have been fixed by setting a smaller queue size. Nevertheless, this introduces a different but more se-

vere issue. If the site receiving all rescheduled jobs was not overloaded and continued to work and the first site became available once again, the first site would have no job to execute. This would result into the situation when one site is very busy and the other completely idle, being able to only work on newly arrived jobs. Therefore, in this scenario, the objective is to keep all computing elements optimally utilized and prevent them from both extremes: an overload, due to large number of jobs getting scheduled, on one hand and a starvation, with no job to process, on the other hand.

The general rule should be to always keep some jobs in the WMS task queue rather than immediately submit them to corresponding CEs. The standard behaviour of WMS (when configured in eager scheduling mode, like are the one in neuGRID or EGEE) is that it schedules a job as soon as there is a matching CE resource available i.e. when it has a free slot in its batch queue. So in order avoid empty TQ, the size of the queue at CE level must be set to a reasonably small number according to the context. On the other hand, the number should not be too small, because when the execution time of jobs is short, the site will then be running out of work to do.

The proposed solution is to have a control loop for each CE that monitors the number of jobs in the site's batch queue, readjusting it when necessary. The initial model should maintain two thresholds that relate to the minimum and the maximum number of jobs in the queue. The minimum should be that amount of queued tasks necessary to avoid empty CE queue. The maximum should not be much more than that to keep TQ non-empty. Both values should be subject to adaptation and change as the system evolves. Every batch queue size has a directly proportional tolerance zone associated. When the number of jobs at the site drops below this zone an adaptation might be triggered and the queue size increased. The concrete model, which is to be experienced very soon, should be based on a discrete ratio between the number of jobs to be scheduled and the size of the batch queue.

In case of neuGRID, the CE is LCG-CE[9] which is based on torque[10]. Adjusting the queue size in torque has very little impact on the running system, hence we can often modify it. However, there might be different batch systems used in other gLite deployments, in which a queue size change has a more significant impact. In that case a different approach will be developed, for instance by setting an artificial threshold on the queue size and by adjusting the WMS job scheduling as well.

4.3 Job Failures

Job failures can be divided into two categories: one where the failure is caused by an application specific problem and the other where it is because of a problem in the Grid middleware. The first category includes invalid job descriptions, application software "bugs" or invalid input data. The cause related to the middleware may be for example some unresolved library dependencies that lead to systematic failures on some jobs. Indeed a job expresses its requirements in a specific JDL file, but there is no fine-grained manner to express precise library dependencies. Therefore a job might be scheduled to run on a WN that does not satisfy the actual job library requirements. The larger the grid considered, the more critical this issue is, as heterogeneity and possible incompatible configurations are more likely to appear in large systems.

Identifying the exact cause of a job failure requires extensive expertise and debugging skills. Furthermore, coordinated investigation over multiple administrative domains is often needed in Grids. To address this problem without resorting to costly human intervention, it is possible to collect statistics to identify recurring source of failures. Although it does not provide insight on the exact reason of the failure, it may be sufficient to avoid situations that are known to fail. A first practical approach consists in building a self-monitoring subsystem (cf. Figure 5) that gathers information relevant to job failures and indexes them in a database with their job type (i.e. the full value of the `Executable` directive in the JDL file). It can then be queried to decide some adaptations based on gradual information about failures as well as statistics such as job executable against failure rate.

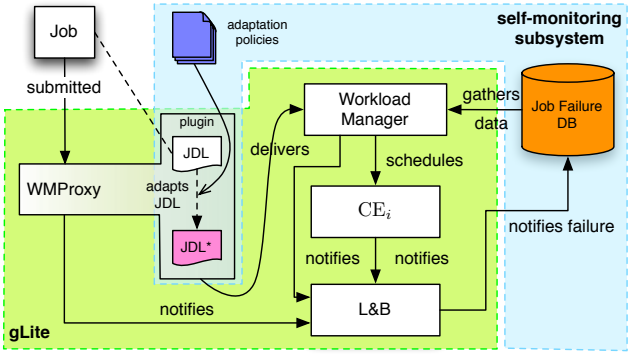

Figure 5: Job failure self-monitoring subsystem

Unlike the others monitoring facilities that are already present in the middleware, our proposed self-monitoring subsystem interacts with the job submission mechanism and adjusts the amount of information it obtains and the sensors it uses according to the current state and some specified high-level policies.

Typically, when a job fails, unless explicitly specified, there is very little information available when the cause is not properly identified by the middleware, *i.e.* usually only the exit code. This makes root cause analysis very difficult. However, by interacting with the L&B service, the self-monitoring subsystem can be notified upon such a job failure and records it into the database together with all other available information. Next time, when the same kind of job is submitted for execution, the subsystem can adapt the job's JDL according to some specified high-level monitoring policies — technical studies show that this can be achieved by developing a plug-in for the request delivery module in the WMProxy [3] —. For example, these policies might extend the job's output sandbox to include standard output, error and core dump file, or wrap the executable with some tracing utility such as `strace`. Another type of adaptation would be to verify whether a particular job type fails on all CEs or only on a certain subset of them, so the

[9] `https://twiki.cern.ch/twiki/bin/view/EGEE/LcgCE`
[10] `http://www.clusterresources.com/products/torque-resource-manager.php`

subsystem could modify the JDL to black list one or more CEs. In this way the system learns about the context of the job failures.

4.4 CE Black Hole

Under certain circumstances a CE might defect and start to fail all scheduled jobs for some unknown reason. Since it fails all jobs immediately, it will process its queues very quickly hence becoming a *black hole* in the Grid as it will attract all newly incoming jobs that are matched to its configuration. This scenario is not directly linked to failures observed during the data challenge, but it is a well-known issue in the gLite middleware [6].

The self-healing adaptation in this case involves a control loop that monitors execution time, IO activity using low level operating system probes and results of job execution using the L&B service or the CE log. When it observes the black hole pattern – a series of jobs with a very short execution time and a low disk activity – it will put the CE into a drain mode. This will be reported back to the ISM and after several minutes, the WMS will no longer submit jobs into this site. It will also be propagated to a system administrator who should take a closer look at the problem. It is the system administrator who is responsible for bringing the site back up and running.

In this scenario there are multiple options on the concrete loop deployment. For example there may be one control loop per CE or one *master* control loop that manages all CEs in the infrastructure. In the former option, another loop will be required to manage loops together with CE life-cycles, so when a new CE joins the infrastructure a new properly configured loop will be deployed into the system and vice-versa. The different pros and cons of these approaches are to be further experimented and one of the aims of the SALTY framework is to facilitate and capitalize such experimentations.

5. TOWARDS MODEL DRIVEN SELF-ADAPTIVE GRID MIDDLEWARE

In this section, we present the main principles underlying the SALTY framework, which is undergoing implementation, as well as its expected benefits.

5.1 Principles

In order to build the SALTY framework, the main approach consists in applying end-to-end model-driven engineering to all elements of the necessary control loops. To understand the process and the realized abstraction of the framework, we detail the different stages of usage from execution back to deployment and design times. Some illustrations are also given using the scenarios that have been previously described.

Execution time.

At runtime, control loops are executed to manage the self-configuration and self-optimization of the controlled system. In SALTY, the control loops are made of one or several SCA components. SCA (Service and Component Architecture) is a standard specification that defines a distributed component model aiming at complement the Service Oriented Architecture (SOA) paradigm. SOA promotes a way for exposing and composing coarse-grained services, *e.g.* implemented with web services, while maintaining a loose coupling between clients and remote suppliers. Composition of services is usually described as orchestrations, but the SOA approach does not really address the service implementation issue. SCA entities are thus software components that may provide and require interfaces and may expose properties. They are connected through *wires* and can also be contained in other *composite* components, making the SCA model hierarchical. An XML-based language helps in specifying and configuring component assemblies. SCA components can be implemented by different languages (Java, C++, PHP, BPEL, COBOL), interfaces can be specified as WSDL or Java interfaces and different protocols can be used in some cases, ranging from SOAP for Web Services, to Java RMI and REST.

Using the SCA infrastructure at runtime allows providing an architecture similar to Rainbow [7], with some explicit description of the main self-adaptive entities. For instance, sensors and effectors, which are connected to the control loops elements and implement respectively the basic probes and elementary modifiers on the controlled systems, are also wrapped and exposed as SCA components. SCA also permits all possible granularities for control loops, i.e.:

- as a single component making a complete autonomic managers with connections to sensors and effectors.
- as separate components (possibly in a surrounding component), with one component per loop activity: monitor, execute, etc. This may be useful to separate activities in some loops.
- as several component *layers* going through sensors to effectors as a flow, from aggregated probes pushing or pulling higher level events to some analysis and planning components, which are then communicating with one or several effector components. This architecture can notably be used to explicit and share the aggregation of information from the basic sensors to the aggregated resource usage probes, and more generally to deploy a very fine-grained decomposition of the loops. In this case, it is planned that the SALTY framework will be able to generate loops compatible with SPACES [23], a distributed context processing architecture based on SCA components supporting the REST protocol.
- a combination of all the previous architecture style, as SCA supports hierarchy of components. This can lead to some loops visible as a single component at the higher level, but decomposed inside in more elements, or a mixed architecture with one single monitoring component acting as a database with other components acting as processes that analyze and make changes [9].

As for the proposed scenarios on the Grid middleware, some loop elements are already designed, such as sensors on CPU and memory usage on (virtual) machines. They are to be aggregated as monitoring components, e.g. in the WMS overload scenario. Similarly, effector components wrap code and scripts, e.g. to block jobs on the WMS and to restart it if needed. These components will then be integrated in different possible loop architectures according to the scenarios.

This will enable us to compare the different loop architectures on their capabilities and performances.

Furthermore, instances of loop will also be created to control different aspects of other loops on a large scale:

- some loops are to be instantiated to coordinate other loops at the same level, forming a hierarchy of loops. In our scenarios, there will be a loop on the WMS managing other loops dedicated per scenario, i.e. the WMS overload and CE starvation loops. For instance, queue size could be adapted to manage threshold for CE starvation according to the load of the WMS.

- some other loops will aim at controlling loop elements, thus being loops at the meta-level. This will notably allows for self-adaptive monitoring, with the self-configuration of sampling intervals on probes [14] or triggering threshold [4]. More generally, any loop elements can be self-managed in the same way.

- similarly some loops will have to control the behavior of several or all loops, also from the meta-level. For example, this will be used to enforce time constraint on the overall self-adaptive parts or any constraints on the features of the loops.

Deployment time.

All running instances, loop elements and loops themselves, are created through factories that have access to the type definition of all elements. These element types are defined through models, which can be directly instantiable SCA definitions or other specifications like EMF (Eclipse Modeling Framework). These latter necessitate additional code but allow the direct usage of design time models [22].

As the SCA specification only defines the static description of components wiring, no reconfiguration of components is directly supported. In the SALTY framework, the Frascati implementation [24] of the SCA specifications is used, which enables dynamic reconfigurations of any component at any level, while providing consistency checking on the architecture. This allows for several reconfiguration scenarios on loop elements (updating a sensor, an effector, a monitoring component, etc.) on loop architectures (replacing an autonomic manager implemented by a single SCA component by a two-levels components with subcomponents, and vice versa).

The used type definitions are stored in repositories together with necessary integration code for sensors and effectors. These two elements of the autonomic framework cannot directly be generated through model transformations. Still, they can be provided by developers or integrators, and then wrapped into appropriate SCA components. These elements can be reusable for other deployments or be platform specific. For example, in the WMS overload scenario, scripts and codes are going to be reused and wrapped to provide resource usage sensors. As for effectors, some code will be integrated in an architectural addition to block jobs on the WMS, and some script will be wrapped to restart the WMS when triggered.

Design time.

Model-driven development is a style of software development where the primary software artifacts are models from which code and other artifacts are generated or controlled. A model is a description of a system from a particular perspective, omitting irrelevant detail so the characteristics of interest are described more clearly.

In SALTY, models of each activity of the loops, loops themselves, SCA components and infrastructures, are available at design time. All these models conform to respective metamodels so that they can be extended and tailored, while being as technology agnostic as possible. Model transformations are then used to produce the whole or part of types of the loop elements.

It should be noted that two concepts of models will be manipulated in the SALTY framework:

1. reification of autonomic elements, for model-driven engineering, as described above.

2. models for model-based reasoning, i.e. statistical and probabilistic models at monitoring level, as well as different kinds of Markov decision processes at the analysis and planning levels. These latter models will be encapsulated into some component-based elements with common facades so that they can be easily composed and reused. In some ways, model-driven engineering will enable the use and reuse of model-based techniques.

5.2 Expected Benefits

We now focus on the expected benefits of using a model-driven approach to develop and extend the SALTY framework and shows how they are essential to tame complexity of grid computing and focus on relevant information for self-adaptation.

Abstraction and efficiency. Using models to design software is a well-established practice to convey some aspects of a system. In our context, we use adaptation models to design autonomic scenarios by means of concepts such as "queue size" and "average time to perform a job". To implement interactions between models and middleware, we refine these models and design the details of these models expressing correspondences between these models and the artifacts (code generation or existing mechanisms) at the middleware level. Consequently models are at the same time a support to the design, the comprehension and the implementation of MAPE-K loop in the middleware.

Models are described according to meta-models. Meta-models themselves are described using a meta-meta-model. Thus, the designer can use a modeling tool and a well-known language to make the necessary changes to the meta-model, and modify transformations to propagate changes at grid middleware level. Meta-Model enables the definition of middleware configurations supporting self-adaptation management. Meta-elements describes the structure and semantics of entities in an infrastructure. They support description of static configurations of the middleware and dynamic adaptations. They can be used as a catalog of specialized configurations and a repository of models and codes referencing mechanisms to be deployed to observe and control some middleware entities.

According to the second scenario, component defined to dynamically modify the queue of a CE (stopping it, changing the configuration file, maybe restarting the CE) will be referenced in the catalog as an effector. To each effector

correspond different factories supporting build of the corresponding entities at the platform level. Each scenario corresponds to a specific policy of adaptations. Several policies can be defined simultaneously on a same middleware. But the SALTY framework should help to master this complexity by detecting possible interactions between policies.

Cost reducing and quality of code. To deal with auto-adaptation, we have to consider sensors and effectors at platform level, implement management and reasoning on observations, evaluate results of adaptations and eventually deploy new probes or configure middleware to deal with frequency of observations, etc. It is a hard and cumbersome work that usually requires expertise in middleware, loop management, analyze, etc. Model-driven development is supposed to automate implementation patterns with transformations, which eliminates repetitive low-level development work. Rather than repeatedly applying technical expertise manually when building solution artifacts, the expertise is encoded directly in transformations, offering the advantages of both consistency and maintainability.

Reuse. Depending on the middleware and on the adaptation mechanisms, suitable off-the-shelf sensors, effectors, transformations, adaptation components are available for use directly or as a basis for extension. Adaptation policies such as the ones described in the proposed scenarios can then be deployed by reusing existing components or by adapting them on different middlewares. Moreover experts may customize these policies according to their own applications, improving them and enriching the community with new algorithms. Consequently our approach should capture the expertise of technical, analyst, business people, making them available to other teams through SALTY tooling.

5.3 Ongoing and Future Work

Ongoing work is split into two complementary activities. A bottom-up work consists in implementing the described scenarios without any SALTY architectures, in order to validate all implementation details. First SCA component wrapping this code will be specified and implemented. In parallel the first drafts of all metamodels are going to be produced soon, focusing on some core features of some simple but complete MAPE-K loops. Necessary transformations will then be implemented to generate the equivalent SCA specifications from the bottom-up implementations. Additional features will next be incrementally added, while experimentations will be conducted in parallel to get feedback and improve the SALTY framework. These experimentations will also cover another large-scale distributed system, with a geo-tracking application dealing with several thousand trucks, many control loops and a huge amount of events.

As for the grid, on a longer term, catalogs of the developed models are going to be provided to the community to be reused and extended. In order to consolidate validation, we are planning to deploy the SALTY tooling on other gLite deployments on private grids and to develop new self-adaptive scenarios on the application side for deployments with the EGEE grid.

Acknowledgements

The work reported in this paper is partly funded by the ANR SALTY project (`http://salty.unice.fr`) under contract ANR-09-SEGI-012 and by the neuGRID EC/FP7 project (`http://www.neugrid.eu`) under grant agreement 211714.

The authors would like to thank Remi Mollon for providing valuable details and remarks on some technical parts of the scenarios.

6. REFERENCES

[1] An architectural blueprint for autonomic computing. `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`, June 2006.

[2] M. Z. 0002, J. Xu, and R. J. O. Figueiredo. Towards autonomic grid data management with virtualized distributed file systems. In *ICAC*, pages 209–218. IEEE, 2006.

[3] G. Avellino. Flexible job submission using web services: the glite wmproxy experience. (EGEE-PUB-2006-024), 2006.

[4] D. Breitgand, E. Henis, and O. Shehory. Automated and adaptive threshold setting: Enabling technology for autonomy and self-management. *Autonomic Computing, International Conference on*, 0:204–215, 2005.

[5] G. Dasgupta, O. Ezenwoye, L. Fong, S. Kalayci, S. M. Sadjadi, and B. Viswanathan. Runtime fault-handling for job-flow management in grid environments. In Strassner et al. [26], pages 201–202.

[6] A. Duarte, P. Nyczyk, A. Retico, and D. Vicinanza. Monitoring the egee/wlcg grid services. *J. Phys.: Conf. Ser.*, 119:052014, 2008.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[8] C. Germain-Renaud and O. F. Rana. The convergence of clouds, grids, and autonomics. *IEEE Internet Computing*, 13(6):9, 2009.

[9] S. Jha, M. Parashar, and O. Rana. Investigating autonomic behaviours in grid-basedcomputational science applications. In *GMAC '09: Proceedings of the 6th international conference industry session on Grids meets autonomic computing*, pages 29–38, New York, NY, USA, 2009. ACM.

[10] Y. E. Khamra and S. Jha. Developing autonomic distributed scientific applications: a case study from history matching using ensemblekalman-filters. In *GMAC '09: Proceedings of the 6th international conference industry session on Grids meets autonomic computing*, pages 19–28, New York, NY, USA, 2009. ACM.

[11] A. Kretsis, P. Kokkinos, and E. Varvarigos. Developing scheduling policies in glite middleware. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:20–27, 2009.

[12] E. Laure, S. Fisher, Á. Frohner, C. Grandi, and P. Kunszt. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12(1):33–45, 2006.

[13] E. Laure, F. Hemmer, F. Prelz, S. Beco, S. Fisher, M. Livny, L. Guy, M. Barroso, P. Buncic, P. Z. Kunszt, A. Di Meglio, A. Aimar, A. Edlund,

D. Groep, F. Pacini, M. Sgaravatto, and O. Mulmo. Middleware for the next generation grid infrastructure. (EGEE-PUB-2004-002), 2004.

[14] B. Le Duc, P. Châtel, N. Rivierre, J. Malenfant, P. Collet, and I. Truck. Non-Functional Data Collection for Adaptive Business Processes and Decision Making. In *4th Workshop on Middleware for Service Oriented Computing(MW4SOC 2009) AR=45%*, page 6. International Conference Proceedings, ACM Digital Library, Nov. 2009.

[15] J. P. Lerch and A. C. Evans. Cortical thickness analysis examined through power analysis and a population simulation. *NeuroImage*, 24(1):163–173, January 2005.

[16] Z. Li and M. Parashar. Rudder: A rule-based multi-agent infrastructure for supporting autonomic grid applications. In *ICAC*, pages 278–279. IEEE Computer Society, 2004.

[17] D. Lingrand, J. Montagnat, and T. Glatard. Modeling user submission strategies on production grids. In *International Symposium on High Performance Distributed Computing(HPDC'09)*, pages 121–130, June 2009.

[18] Y. Liu, S. M. Sadjadi, L. Fong, I. Rodero, D. Villegas, S. Kalayci, N. Bobroff, and J. C. Martinez. Enabling autonomic meta-scheduling in grid environments. In Strassner et al. [26], pages 199–200.

[19] M. A. Munawar and P. A. S. Ward. Leveraging many simple statistical models to adaptively monitor software systems. In *Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007, Niagara Falls, Canada, August 29-31, 2007, Proceedings*, volume 4742 of *Lecture Notes in Computer Science*, pages 457–470. Springer, 2007.

[20] M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt. Enabling autonomic grid applications: Requirements, models and infrastructure. In *Self-star Properties in Complex Information Systems*, pages 273–290. 2005.

[21] J. Perez, C. Germain-Renaud, B. Kégl, and C. Loomis. Utility-based reinforcement learning for reactive grids. In Strassner et al. [26], pages 205–206.

[22] L. L. Provensi, F. M. Costa, and V. Sacramento. Management of Runtime Models and Meta-Models in the Meta-ORB Reflective Middleware Architecture. In *Proceedings of the 4th Workshop on Models@run.time, held at the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09), Denver, USA, October 5th, 2009.*, pages 81–88. CEUR-WS, 2009.

[23] D. Romero, R. Rouvoy, L. Seinturier, S. Chabridon, C. Denis, and P. Nicolas. Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments. In Michael Sheng, Jian Yu, and Schahram Dustdar, editors, *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, pages 113–135. Chapman and Hall/CRC, 07 2009.

[24] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275, Bangalore Inde, 2009. IEEE. IST FP7 IP SOA4All.

[25] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland. A Concise Introduction to Autonomic Computing. *Advanced Engineering Informatics*, 19(3):181–187, 2005.

[26] J. Strassner, S. A. Dobson, J. A. B. Fortes, and K. K. Goswami, editors. *2008 International Conference on Autonomic Computing, ICAC 2008, June 2-6, 2008, Chicago, Illinois, USA*. IEEE Computer Society, 2008.