Using Cloud-based Resources to Improve Availability and Reliability in a Scientific Workflow Execution Framework

Sergio Hernández, Javier Fabra, Pedro Álvarez, Joaquín Ezpeleta
Aragón Institute of Engineering Research (I3A)
Department of Computer Science and Systems Engineering
University of Zaragoza, Spain
Email: {shernandez, jfabra, alvaper, ezpeleta}@unizar.es

***Abstract*—Different mechanisms, such as checkpointing, task replication, alternative tasks execution or task migration among different resources, for instance, have been traditionally applied in (heterogeneous) grid environments for fault-tolerance. Cloud based resources can easily improve both availability and reliability of a given system when used for recovering faulty tasks. In this paper we present how cloud resources have been included in a framework for the execution of scientific workflows and how this has helped in improving the framework in two different aspects: making it more scalable and more reliable, facilitating the application of very effective fault recovery policies.**

Keywords–Fault tolerance; Scalability; Cloud computing; Heterogeneous computing infrastructures; Resource management frameworks.

I. Introduction

Grid systems are prone to faults [1][2][3]. Different fault-tolerance mechanisms (checkpointing, task replication, alternative tasks, or task migration, for instance) have been traditionally integrated into Grid middlewares and management systems in order to handle and minimize the impact of these faults [4][5]. Nevertheless, these mechanisms do not prevent end-users jobs from experiencing high failure rates when they are executed in this type of distributed computing infrastructures [6]. For that reason, users must play a vital role in the course of detecting these faults: checking execution logs and job outputs, for instance [1]. Undesirable behaviour is then notified to Grid administrators so that they can adopt the necessary steps to restore the Grid.

In the last years, the Grid computing community has concentrated its research efforts on integrating several heterogeneous Grids in order to generate more powerful computing infrastructures. Resource management frameworks have been developed to provide a transparent and easy-to-use access to the set of integrated computing infrastructures. Consequently, these heterogeneous infrastructures are viewed as a whole from the end-users' point of view. Some relevant examples of these solutions are GJMF [7], P-GRADE [8], SWAMP [9], GridWay [10], eNANOS [11], EMPEROR [12], or GMBS [13]. Obviously, this new model of solution requires new fault-tolerance mechanisms at the global level because frameworks consist of internal services (schedulers, state monitors, resource registries, etc.) that are also prone to faults. These mechanisms must be compatible with the ones integrated into each local Grid. Currently, resource management frameworks use the monitoring and notification capabilities of their middlewares to detect faults. Then, resubmission techniques are integrated into their fault management components to recover the execution of failed jobs.

In [14], authors proposed an open framework for the flexible deployment of scientific workflows in heterogeneous Grid environments. From an architectural point of view, the framework was organized as a set of components connected through a central bus, which was used by the components as the mean to send and receive messages. At the beginning, the fault management was very simple and consisted of re-submitting the faulty task (either to the same computing resource or to an alternative one). In this paper, we try to improve framework availability and reliability by using cloud-based resources. The experience gained by solving complex computational problems has also allowed us to understand a wide variety of faults suffered by this type of distributed computing infrastructures. The use of cloud resources can help solve some of these faults or at least reduce their effects.

The paper is organized as follows. Section II briefly describes the architecture of the proposed framework for scientific workflows execution. The description is mainly focused on the components involved in fault management. Section III introduces the suggested fault classification and a discussion about their corresponding effects. Sections IV and V present two cloud-based solutions for solving availability and reliability problems. We have concentrated on situations caused by a large performance degradation of computing resources and bottlenecks in the common bus. Section VI describes the main related work. Finally, Section VII summarizes the main contributions of the paper.

II. Background

As it was mentioned earlier, we proposed a framework for the flexible deployment and execution of scientific workflows. The flexibility has been achieved at different levels:

from a computing point of view, the framework is able to integrate heterogeneous computing infrastructures to create more powerful execution environments; from a programming point of view, workflows can be programmed independently of the computing infrastructures where related jobs will be executed and using different high-level languages widely accepted by the scientific community; and, finally, from a configuration point of view, new functionalities can be dynamically added/removed to the framework in order to meet the different needs of each application and user.

An integration model based on a *message bus* is key to achieve the flexibility of the proposed solution. More specifically, the cornerstone of the proposal is a bus inspired by the Linda coordination model [15]. This component provides an application interface (API) for sending and receiving messages in an asynchronous way, coding them as Linda tuples. The rest of system components offer their capabilities through the common bus, and collaborate by exchanging messages using the bus as the communication channel. This integration model has several advantages compared to other more traditional approaches: (1) a bus reduces the coupling between system components (they are connected by making use of an asynchronous message passing mechanism); (2) components can be dynamically added or removed without disturbing the execution of other existing ones (to adopt new functionalities, for example); (3) a bus favours the scalability and distribution of the solution; and, finally, (4) a bus supports complex message exchange patterns (publish and subscribe mechanism, content-based message routing, etc.) that facilitate more flexible integration strategies.

In this communication model, framework components are not aware of other components connected to the message bus. Each message is assigned an exclusive tag to identify the receiver and each component identifies the messages addressed to it with that tag. Thus, management components and mediators can be easily replicated to improve framework performance and reliability. Replicated components compete for the same messages and the message bus decides which mediator gets each message. As a consequence, components can be easily replaced to adopt new functionalities, change them or fix bugs.

Figure 1 shows the high-level system architecture which is composed of three different layers. At the top, the *User interface layer* is composed of the different programming tools that can be used to program scientific workflows (Taverna, Triana, Kepler, Pegasus, etc.). Resulting workflows are submitted to the framework for their execution. The components of the *Execution layer* are responsible to manage the workflow execution life-cycle. Internally, this layer is composed of the *message bus* and the components that provide the core functionalities. In order to provide this functionality, two types of components have been connected through the bus: *management components* and *mediators*. The first ones offer extra functionalities to enhance workflows, task life-cycle and framework capabilities (meta-scheduling, fault-tolerance, monitoring, etc.). On the other hand, mediators encapsulate

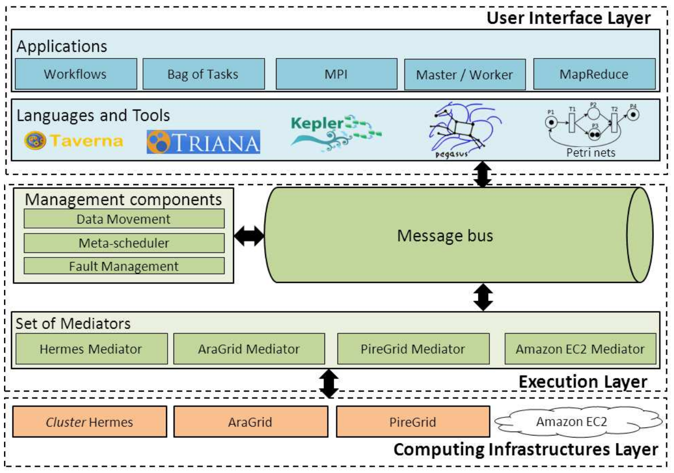


Figure 1: Architecture of the proposed framework for the execution of scientific workflows in multiple heterogeneous computing infrastructures.

the heterogeneity of each specific computing infrastructure to facilitate its integration into the framework (in more detail, a mediator must interact with a specific infrastructure to submit jobs, move input/output data, monitor job executions, detect undesirable states, etc.). Finally, at the bottom of the architecture, the *Computing infrastructures layer* is formed by different and heterogeneous computing infrastructures. At the beginning, three computing environments were integrated: the HERMES cluster hosted by the Aragón Institute of Engineering Research (I3A) [16], which is managed by the HTCondor middleware [17]; and two research and production grids managed by the gLite middleware [18] and hosted by the Institute for Biocomputation and Physics of Complex Systems (BIFI) [19], namely AraGrid [20] and PireGrid [21]. A more detailed description of the architecture can be found in [14][22].

In the first implementation of the framework, availability and reliability issues were deliberately ignored. In this paper, we propose the use of Cloud computing to add these requirements. In this Cloud-based approach the selected integration model plays a relevant role as it will be shown in the following sections.

III. INTEGRATION OF FAULT HANDLING MECHANISMS INTO THE FRAMEWORK

As we have already discussed, grids and computing clusters are prone to faults. In this section, we present various types of faults that can locally occur in these infrastructures and the techniques usually used to detect and handle them. Our discussion focuses on the user perspective and considers the effects produced by these faults in terms of availability (the ability of the system to be ready for successful job submission) and reliability (the ability of the system to successfully execute jobs even in the presence of failures during job execution). Other fault classifications can be found in [1][2][3].

Additionally, the proposed execution framework could also be affected by faults. The message bus is the most critical component of the architecture: if the bus fails, the whole system fails. Besides, the bus can become a bottleneck and, as a consequence, degrade performance (for instance, when a large number of application jobs are being executed by the framework). For this reason, mechanisms that improve the reliability and scalability of the framework must also be integrated.

Let us briefly describe the faults that can affect computing resources and introduce solution mechanisms. A more detailed description will be presented in the two following sections.

A. Faults at the computing infrastructures level

In this work, we have considered the followinf set of faults, identified from our experience in scientific workflows solving:

- *Computing resource failures*: A computing resource may suffer hardware, network or operating system faults that affect the jobs that are being executed on it. These faults are not critical because they only involve individual resources and can be easily repaired.
- *Hardware upgrades and maintenance*: These actions typically require shutting down the computing infrastructure causing unavailability periods. They involve the failure or cancellation of all jobs submitted to the infrastructure.
- *Software upgrades and maintenance*: Depending on the nature of the software upgrade, it may be transparent, it may cause some resources to be unavailable and some job failures, or it may cause total unavailability and the failure of all jobs. Also, it may affect only certain users. Additionally, these actions often lead to periods when the infrastructure is unreliable due to misconfiguration.
- *Environmental failures*: These faults are provoked by causes external to the computing infrastructure (power outages or cooling issues, for instance). The affected computing infrastructure can become totally unavailable and all running jobs may fail.
- *Deployment and configuration of new software*: The execution of some applications may need to install and configure new software and services. These operations must be performed by administrators and may take a large amount of time. Although this situation does not strictly involve any failure, it prevents users from executing jobs due to the deployment of new software and potential misconfiguration. During this period, the user views the computing infrastructure as totally unavailable.
- *Application-dependant problems*: When a service required for the execution of an application fails or is not available, administrators are responsible for restarting the service (users do not have the required privileges [1]). While the failure is being fixed, applications using the broken service fail. As a consequence, the resource is seen as unavailable for some users, while others remain unaffected.
- *Middleware failures*: Due to the distributed nature of grid middlewares, failures can involve different components and their effects may vary. A failure in key components, which represent a single point of failure, may cause total unavailability and the failure of all executing jobs, whereas a failure in a secondary component may have no effect on users. In our particular case, since the framework could be seen as a meta-middleware, these failures may appear at the framework level and the computing infrastructures level.

The previous faults involve different availability problems ranging from situations where less resources are available to states where the complete infrastructure becomes unavailable. From the reliability point of view, there could be no effect at all or failures in all executing jobs. To detect and repair some of these faults, grid middlewares integrate different fault-tolerant mechanisms. In general, they are only able to detect the most simple ones (computing resource failures) and they cannot recover from all detected faults [3]. Additionally, some middlewares provide techniques to mitigate the effect of faults, such as checkpointing [5] or over-provisioning [23][24]. In any case, these techniques are only useful to recover from computing resource failures where some resource becomes unavailable and a few jobs fail. More critical problems involving total unavailability and unreliability are much more difficult to manage. These problems lead to situations where users cannot execute any job and lose a valuable time waiting for the fault to be fixed.

B. A hierarchical strategy for handling grid/cluster faults

Once the effects caused by these faults are understood, a strategy to handle them can be implemented and integrated. We propose a solution based on the hierarchical management of faults. Firstly, when the execution of a job fails, the fault is locally managed by the computing infrastructure where the job was being executed (a kind of local strategy). The local fault tolerance mechanisms are responsible for detecting and handling this kind of failures. In some cases, these mechanisms can collaborate with the mediator component that manages the access to the infrastructure in order to react to the fault: for instance, if the execution of a job has failed, the mediator can locally submit it to a different computing resource of the same infrastructure.

If the fault persists after taking corrective actions at the level of the computing infrastructure, it is dispatched to the execution layer. More specifically, the mediator of the faulty infrastructure sends a fault message to the message bus. A new management component for fault handling has been integrated into the framework execution layer. This component is responsible for catching fault messages and guarantees the successful execution of jobs using reliable computing resources. In our approach, the job could be submitted to another computing infrastructure or, as a last option, cloud computing resources could be used by the component to execute the job.

Therefore, for these types of faults the proposed solution consists of two levels of fault handling: firstly, at the specific computing infrastructure level, and, secondly, at the execution framework level.

C. Improving the framework reliability and scalability

The architecture of the proposed framework favours the management of faults at the software components level (mediators, management components and the message bus). When a mediator or a management component fails, its functionality is disabled. In the case of a mediator, the access to the computing infrastructure managed by it is closed; whereas in the case of a management component, the capabilities of the framework (scheduling, data movement, fault tolerance, etc.) are reduced. Both situations can be solved using the same solution: integrating into the framework multiple instances of the same component. Let us remember that in the proposed solution components can be added or removed without disturbing the execution of other existing ones and multiple instances can work together without interfering with each other.

On the other hand, the message bus is the core component of the framework. How can we make this component reliable and scalable? In order to deal with the first issue, the message bus has been deployed in a virtual machine provided by Amazon EC2 [25]. For the scalability issue, a new version of the message bus has been implemented. Now, the bus is distributed through several computing nodes (virtual machines) and new elastic capabilities (inspired by cloud behaviour) have been integrated into it. The bus is able to monitor its internal state (number of messages, response time, throughput, etc.) and predict when its performances or capabilities might be compromised. When some of these undesired states is detected, new computing nodes can be dynamically added to host message exchanges.

In the following sections, we go deeper into these aspects.

IV. MANAGEMENT OF AVAILABILITY AND RELIABILITY ISSUES

The characterization presented in Section III shows that, in large-scale distributed computing infrastructures, there are a lot of problems leading to temporal or permanent unavailability states and job failures due to reliability issues. As a result, users experience severe delays in both submission and termination of their jobs and unexpected end statuses. To tackle this problem, we have extended the mediators with monitoring capabilities. A hierarchical fault management mechanism is proposed, enabling the framework to manage faults at different levels using several fault recovery policies. This reduces the overhead of the message bus and the time required to handle failures. We also propose the use of public clouds as reliable computing infrastructures for the execution of jobs that systematically fail in the integrated computing infrastructures.

A. Solution design

Mediators have been extended with an *Infrastructure Monitor* and a *Local Fault Manager*. Figure 2 shows the mediator architecture for this approach. The *Job Submission* process and its related components have been simplified (for more details about the job submission, please refer to [14]), as we will focus on monitoring and fault management.

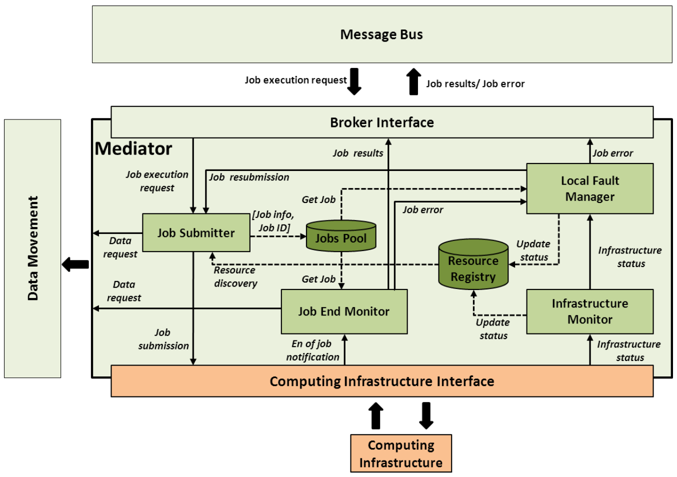


Figure 2: Simplified Architectural design of a generic mediator to lead with unavailability and failure events.

Let us briefly depict the process carried out in the mediator. First, the *Job Submitter* receives job execution requests, retrieves the input data required for the job execution (if necessary) and commands its execution to the computing infrastructure. After that, the job description and job identifier provided by the infrastructure are stored in the *Jobs Pool*. When a job finishes its execution, the *Job End Monitor* fetches the job description from the *Jobs Pool* using the job identifier. Then, it checks the log and error stream files as well as the existence of the output files defined in the job description. If an error is detected or the output have not been generated, the information about the error is passed to the *Local Fault Manager*. Otherwise, the output data are moved to the destination specified in the job description and the results are sent to the message bus.

The mediator can detect failures and unexpected job terminations. However, in order to avoid such situations, the *Infrastructure Monitor* periodically checks the status for resource availability. With this information, it updates the *Resource Registry* and notifies the *Local Fault Manager* if any availability problem is detected. The *Local Fault Manager* is the component responsible for taking decisions when a job fails or an unavailability state is detected. Its design is similar to the *Global Fault Manager* presented in [14]. A rule-based engine is used as the decision maker. The set of rules can be modified at runtime, providing adaptation capabilities for specific scenarios. Therefore, different policies can be used depending on the underlying computing infrastructures, execution traces or system load, for instance.

When a job fails or an unavailable state is detected, the *Local Fault Manager* can decide to either re-execute the involved jobs or notify the *Global Fault Manager*. In the first case, the re-execution process remains internal to the mediator. This approach reduces the overhead of the message bus and the time required to handle the failure. In the last case, a message with error information and the job description is sent to the *Global Fault Manager* (via the message bus).

Finally, the *Global Fault Manager* (namely *Fault Manager* in Figure 1) retrieves messages with information about faulty jobs and chooses a computing infrastructure to re-execute them on or notifies to the user if the fault is not recoverable (for example, because the server hosting input data is down). In case of a recoverable fault, the following approach is used: if it is the first failure, another computing infrastructure is selected; if it is the second failure, a reliable infrastructure is selected; finally, if the third failure is reached, the error is propagated and the user is notified.

We propose the use of public cloud resources as reliable infrastructures because they provide the opportunity of executing jobs in a well controlled and previously defined environment. Cloud resources are less sensitive to resource failures through virtualization and migration techniques. Clouds also provide high availability and reliability, and they supply ”infinite” on-demand resources in a pay-per-use model.

B. Evaluation

In collaboration with the Intelligent Systems Group of the University of Santiago de Compostela (Spain), we have solved a computing-intensive problem in the field of linked data. The problem consists of extracting a set of significant terms from learning units. Each set of terms must be semantically annotated with relevant contextual information extracted from the DBPedia [26]. This problem requires the execution of about 20000 jobs for a whole week. As a consequence, it is very sensitive to faults. We have used this experiment as a benchmark for the proposed hierarchical fault management system.

Figure 3 shows the failure rates obtained using different policies in the local fault manager (no fault recovery, one resubmission and two resubmissions) and the global fault manager (no fault recovery, resubmission on an alternative computing infrastructure, resubmission on an Amazon EC2 resource and a combination of the two last ones). As it can be observed, using public cloud resources allows us to recover from any failure (except failures due to unreachable input data or bad definition of jobs). Otherwise, if we only use the integrated local infrastructures, there are some jobs that still fail after several executions due to unavailability and unreliability states of computing infrastructures.

Besides, the hierarchical approach presented reduces both the message bus overhead and the time required to handle the fault. In the experiments, we have observed that the average time required to handle a fault with our previous design was 1071.23 milliseconds, and the hierarchical design reduces this time to 143.21 milliseconds. When a job fails for the first time in the hierarchical approach, its management remains internal to the mediator. In the previous (non-hierarchical) design, a message was introduced into the message bus and then retrieved by the Global Fault Manager, which would take the decision of resubmitting the job to the same infrastructure (so a new message was written in the message bus and then retrieved by the corresponding mediator in order to submit the job again).

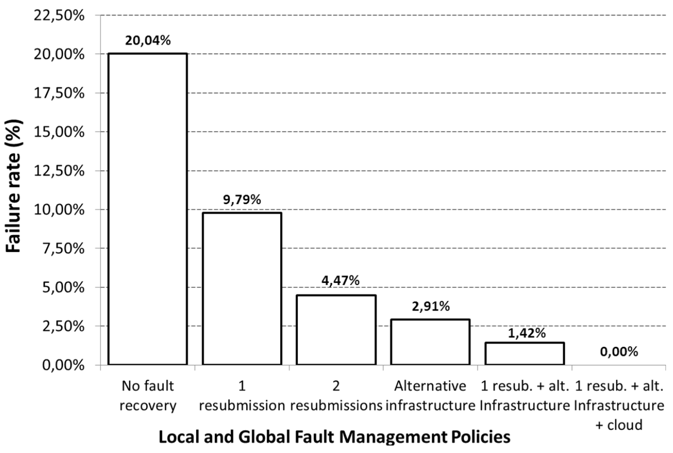


Figure 3: Failure rate for different fault management policies.

The percentage of faults detected by the mediator with respect to the number of total job failures has been also measured. Without infrastructure monitoring, some failures were not detected because the management middleware did not notify them, the middleware itself failed or the computing infrastructure was down. Currently, the Infrastructure Monitor is able to detect these situations and help mediator handle all failures. As a result, the percentage of job failures detected has increased from a 91.92% to a 99.99%.

V. IMPROVING FRAMEWORK SCALABILITY THROUGH AN ELASTIC MESSAGE BUS

Scalability is one of the main challenges of any distributed system. In cluster and grid computing, scalability focuses on the number of computing resources available as well as the flexibility to integrate new ones. The scalability of the management system plays a very important role in the improvement of the quality of service experienced by end-users in terms of response times and system crashes.

The message bus is the backbone of the proposed architecture. In order to make the system more scalable, we propose an elastic design, taking advantage of the dynamic scaling provided by cloud systems. As it will be shown, with respect to the previous message bus version, the use of a cloud-based solution improves both scalability and reliability.

A. Solution design

To deal with scalability issues, we have extended the original design of DRLinda, a distributed message bus based on the Linda coordination model [27]. The main idea behind DRLinda is the use of several nodes implementing message repositories to host messages in a distributed way. We have extended this approach to dynamically scale the number of nodes depending on the number of messages and message access frequency when the system is running.

The previous implementation of DRLinda could dynamically vary the number of nodes used to lead with bursts of requests. However, these changes must be performed manually and only local resources can be used. A cloud-based

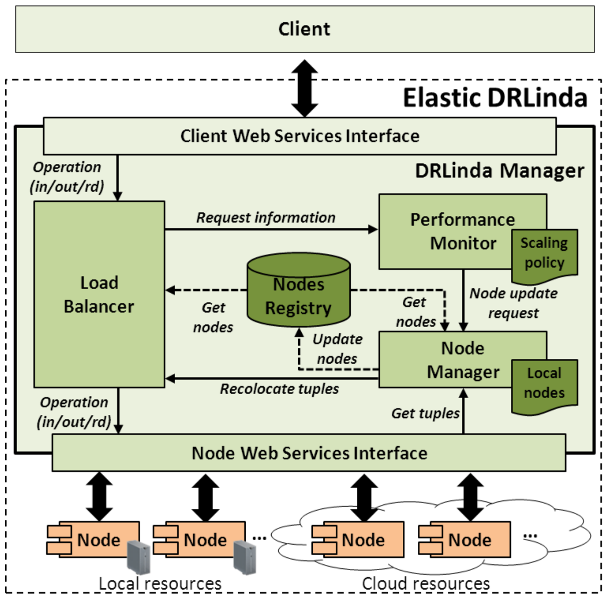


Figure 4: Message bus architecture overview.

elastic design allows self-configuration and auto-scaling of the number of nodes used at any moment. The new architectural message bus design is sketched in Figure 4. The approach includes two new components in addition to the existing *Load Balancer* [27]: a *Performance Monitor* and a *Node Manager*.

The *Performance Monitor* is a component that receives and processes information on client requests and collects metrics such as number of requests, response time or throughput, averaged for the last requests. The results of these metrics and time conditions (for example, time since the last scaling request) can then be used to define the scaling conditions. When a condition is satisfied (scaling up/scaling down), this component communicates with the Node Manager to deploy or release a node.

The *Node Manager* is responsible for allocating new resources and releasing unnecessary ones. When a new resource is requested, the Node Manager looks for a new local resource that becomes a DRLinda node. If there are no available local resources, it gets a cloud instance. In this way, physical local resources and virtualized cloud resources can be used at the same time to provide good quality of service. Also, when a resource must be released, the component selects the most appropriate one and manages message transfer between the involved nodes, via the Load Balancer. To reduce costs, cloud resources are only released when they are about to fulfil an entire hour of use (due to the hourly billing model of the cloud provider we have used). Consequently, if there is a pending release request when a cloud resource is going to complete an entire hour, that resource is released. Also, if a cloud resource can be released but there is no request, a local resource (if available) is used to replace it.

B. Evaluation

To measure the efficiency and scalability of both architectural designs, we have used the methodology proposed in [27][28]. In these experiments, a set of clients access the message bus. Every client warms-up the message bus by inserting a random number of messages (between 1500 and 5000) and then iterates 2000 times through the following sequence of operations: first, it executes an *out* operation (send a new message), then waits for a random time ($T_{delay} \in [200, 250]$ ms), and then retrieves the same message (operation *in*). When completed, the client terminates. A detailed justification of the parameters used for the experiment can be found in [28]. The size of the messages has been set accordingly to the problem we are managing. JSDL messages extracted from the experiments presented in [22] have been used, which an average size of 63 Kbytes.

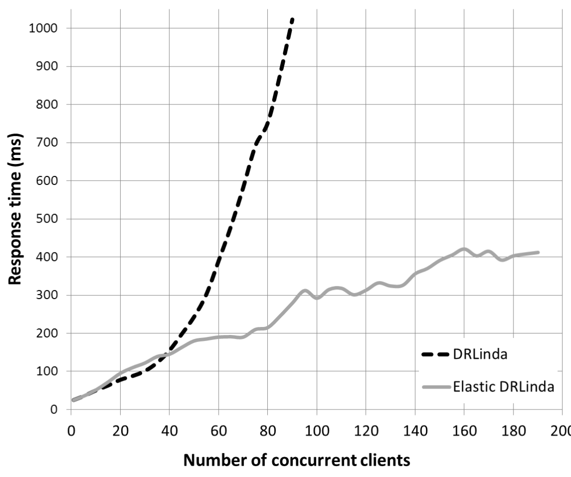
Figure 5a shows the average response time observed both in the original DRLinda implementation and the new elastic design. In both cases, we have used m1.medium Amazon Elastic Compute Cloud (Amazon EC2) [25] instances as resources to host message bus components. For the experiments, the former DRLinda was deployed over 25 nodes. On the other hand, in the case of the elastic solution, only two nodes were initially used, and then new nodes were added under request (up to 70 nodes were registered during the experiment). Obviously, the dynamic scalability introduces an overhead as the message space must be redistributed. However, the overhead is not significant compared to the response time and the throughput in terms of Input/Output Operations Per Second (IOPS). As it can be seen in Figure 5a, the response time improves very significantly when using the cloud-based solution. This is due to a more efficient load balancing in every node. While in the case of the former DRLinda the nodes have to support a higher load, the use of an elastic approach allows to keep nodes at optimum levels of occupation and CPU and memory loads.

On the other hand, the results in Figure 5b depict the throughput in terms of IOPS. As shown, the use of a cloud-based elastic approach reports several benefits. First, the number of concurrent clients supported by the bus scales with no problem over the maximum number of clients. Moreover, the IOPS only decrease because of the overhead of space distribution, but remain in the range of [1100,1200] milliseconds for a huge number of simultaneous clients.

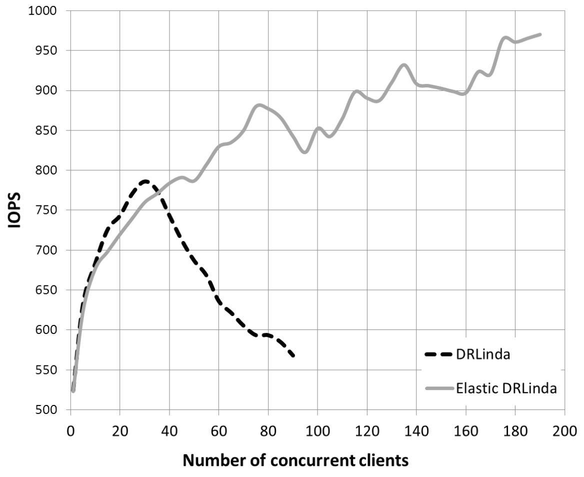
The experiments have also shown how the use of an elastic solution allows to extend the number of concurrent clients without suffering severe delays or service interruptions. Therefore, it is a successful mechanism to avoid bottlenecks in the message bus.

VI. Related work

There are several works seeking to improve understanding of failures in Grid environments. However, none of these studies analyse failure impact on end users. In [2], a taxonomy for the classification of Grid faults is proposed. The taxonomy presents several perspectives for the classification of Grid failures (origin, duration, consequences, etc.) but



(a) Performance comparison



(b) Throughput comparison

Figure 5: Performance comparison between former DRLinda and elastic DRLinda message bus in terms of: (a) response time and (b) Input/Output Operations Per Second.

it lacks an analysis of causes and effects that could help in handling failures. In [1] and [3], different surveys about Grid failures are presented. On the one hand, in [1], failures are classified as configuration, middleware, application and hardware failures. The main concerns and problems regarding fault management are studied showing that end users are highly involved in fault detection and recovery, failures are mostly due to misconfiguration and recovery mechanisms are application-dependant. On the other hand, in [3], hardware, operating system, middleware, task, workflow and user related failures are identified. Also, detection, prevention and recovering capabilities of several workflow management systems are analysed concluding that current systems are not able to properly manage faults.

With regard to scalability and dynamic autoscaling of resources, [29] analyses existing mechanisms to dynamically scale applications in clouds at three different levels: server, network and platform. [30] shows a technique to dynamically scale cloud resources up and down considering performance and budget information. This technique is based on acquiring enough instances to met application deadlines and shutting down unnecessary instances when an hour is going to be fulfilled. In [31], look-ahead optimizations are used to predict future workloads and scaling applications while cost remains low. However, results are limited to scenarios with few resources and accurate predictions. On the contrary, in [32], profiles are used to provide just-in-time scalability for cloud applications in environments with unpredictable workloads. Profiles capture application characteristics, architecture and topology, scaling conditions and mechanisms to automate the deployment and release of new resources.

Finally, different proposals use public Cloud resources to improve job completion rates and to meet the deadline of QoS-constrained jobs. In [33], a rescheduling algorithm is used to deal with grid performance fluctuations. When a job ends after its estimated finish time, a job waiting for execution in the same Grid resource is selected to be executed in a cloud resource. Meanwhile, in [23], task replication is used to reduce the makespan and cost of workflows executed in Grids and Clouds. An unreliable pool of resources is used to execute jobs in first instance, while reliable resources (formed by public Cloud instances and own resources) are used to execute replicated jobs in the tail phase of BoTs (Bag of Tasks). A similar approach is used in [24], where jobs are first scheduled in clusters and Grids, then some jobs are replicated to increase their success probability and finally public cloud resources are used as backup if additional replication is required.

VII. Conclusion and Future Work

In this paper, we have identified and analysed several availability and reliability problems from the users' point of view in the context of a framework to execute scientific workflows in heterogeneous computing environments. This analysis has allowed us to identify common situations where job fails and users cannot execute any job.

To increase framework availability and reliability, two cloud-based solutions have been proposed: an elastic design of the message bus and a hierarchical fault management. On the one hand, the elastic design of the message bus allows the framework to deal with bursts of requests providing high quality of service at a low cost. On the other hand, managing faults hierarchically results in a better treatment of faults by applying different policies at different levels, faster fault-recovery and less overhead in the framework. Also, using public clouds as reliable computing infrastructures allows the framework to execute jobs even in total unavailability and total unreliability situations, reducing the failure rate experienced by end-users.

As future work, we will study techniques to reduce the cost of the proposed solutions without decreasing the quality of service and job completion rates. Also, we will define reliable scheduling policies to increase the number of jobs successfully completed in their first execution. Finally, we will explore the use of Amazon Simple Queue Service (Amazon SQS) [34] in replacement of the Linda-based message bus to improve performance, availability and reliability of the proposed framework.

ACKNOWLEDGMENT

This work has been supported by the research project TIN2010-17905, granted by the Spanish Ministry of Science and Innovation, and the regional project DGA-FSE, granted by the European Regional Development Fund (ERDF).

REFERENCES

[1] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauvé, "Faults in grids: Why are they so bad and what can be done about it?" in *Proceedings of the 4th International Workshop on Grid Computing*, ser. GRID '03, vol. 0, 2003, pp. 18–24.

[2] J. Hofer and T. Fahringer, "A multi-perspective taxonomy for systematic classification of grid faults," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '08, 2008, pp. 126–130.

[3] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertész, and P. Kacsuk, "Fault-tolerant behavior in state-of-the-art grid workflow management systems." CoreGrid, Tech. Rep. TR-0091, 2008.

[4] C. Dabrowski, "Reliability in grid computing systems," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 8, pp. 927–959, 2009.

[5] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *J. Grid Comput.*, vol. 3, no. 3-4, pp. 171–200, 2005.

[6] O. Khalili, J. He, C. Olschanowsky, A. Snavely, and H. Casanova, "Measuring the performance and reliability of production computational grids," in *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID 2006)*, 2006, pp. 293–300.

[7] P.-O. ÖStberg and E. Elmroth, "GJMF - a composable service-oriented grid job management framework," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 144–157, 2013.

[8] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás, "P-grade: A grid programming environment," *J. Grid Comput.*, vol. 1, pp. 171–197, 2003.

[9] Q. Wu, M. Zhu, Y. Gu, P. Brown, X. Lu, W. Lin, and Y. Liu, "A distributed workflow management system with case study of real-life scientific applications on grids," *J. Grid Comput.*, vol. 10, no. 3, pp. 367–393, 2012.

[10] E. Huedo, R. S. Montero, and I. M. Llorente, "A framework for adaptive execution in grids," *Softw. Pract. Exper.*, vol. 34, no. 7, pp. 631–651, 2004.

[11] I. Rodero, J. Corbalán, R. M. Badia, and J. Labarta, "eNANOS grid resource broker," in *Proceedings of the 2005 European conference on Advances in Grid Computing*, ser. EGC '05, 2005, pp. 111–121.

[12] L. Adzigogov, J. Soldatos, and L. Polymenakos, "EMPEROR: An OGSA grid meta-scheduler based on dynamic resource predictions," *J. Grid Comput.*, vol. 3, no. 1-2, pp. 19–37, 2005.

[13] A. Kertész and P. Kacsuk, "GMBS: A new middleware service for making grids interoperable," *Futur. Gener. Comp. Syst.*, vol. 26, no. 4, pp. 542–553, 2010.

[14] J. Fabra, S. Hernández, P. Álvarez, and J. Ezpeleta, "A framework for the flexible deployment of scientific workflows in grid environments," in *Proceedings of the Third International Conference on Cloud Computing, GRIDs, and Virtualization*, ser. CLOUD COMPUTING '12, 2012, pp. 1–8.

[15] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[16] Aragón Institute of Engineering Research (I3A). (2013) http://i3a.unizar.es. Accessed 15 March 2013.

[17] HTCondor Middleware. (2013) http://research.cs.wisc.edu/htcondor/. Accessed 15 March 2013.

[18] gLite Middleware. (2013) http://glite.cern.ch/. Accessed 15 March 2013.

[19] Institute for Biocomputation and Physics of Complex Systems (BIFI). (2013) http://bifi.es/en/. Accessed 15 March 2013.

[20] AraGrid. (2013) http://www.aragrid.es/. Accessed 15 March 2013.

[21] PireGrid. (2013) http://www.piregrid.eu/?idioma=english. Accessed 15 March 2013.

[22] S. Hernández, J. Fabra, P. Álvarez, and J. Ezpeleta, "A Simulation-based Scheduling Strategy for Scientific Workflows," in *Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, ser. SIMULTECH '12, 2012, pp. 61–70.

[23] O. A. Ben-Yehuda, A. Schuster, A. Sharov, M. Silberstein, and A. Iosup, "ExPERT: Pareto-Efficient Task Replication on Grids and a Cloud." in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '12, 2012, pp. 167–178.

[24] L. Ramakrishnan et al., "VGrADS: enabling e-Science workflows on grids and clouds with fault tolerance," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 47:1–47:12.

[25] Amazon Elastic Compute Cloud (Amazon EC2). (2013) http://aws.amazon.com/ec2/. Accessed 15 March 2013.

[26] M. Lama, J. C. Vidal, E. Otero-García, A. Bugarín, and S. Barro, "Semantic linking of learning object repositories to DBpedia," *Educ. Technol. Soc.*, vol. 15, no. 4, pp. 47–61, 2012.

[27] J. Fabra, P. Álvarez, and J. Ezpeleta, "DRLinda: A Distributed Message Broker for Collaborative Interactions Among Business Processes," in *Proceedings of the 8th International Conference E-Commerce and Web Technologies*, ser. EC-Web '07, 2007, pp. 212–221.

[28] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis, "Towards the measurement of tuple space performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 3, pp. 51–62, 2005.

[29] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, 2011.

[30] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, ser. GRID '10, 2010, pp. 41–48.

[31] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proceedings of the IEEE 4th International Conference on Cloud Computing*, ser. CLOUD '11, 2011, pp. 500–507.

[32] J. Yang, J. Qiu, and Y. Li, "A profile-based approach to just-in-time scalability for cloud applications," in *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, ser. CLOUD '09, 2009, pp. 9–16.

[33] Y. C. Lee and A. Y. Zomaya, "Rescheduling for reliable job completion with the support of clouds," *Future Gener. Comput. Syst.*, vol. 26, no. 8, pp. 1192–1199, 2010.

[34] Amazon Simple Queue Service (Amazon SQS). (2013) http://aws.amazon.com/sqs/. Accessed 15 March 2013.