



A path to achieving a self-managed Grid middleware

Ramon Nou^{a,b,*}, Ferran Julià^b, Kevin Hogan^a, Jordi Torres^{a,b}

^a Barcelona SuperComputing Center, Nexus II Building, c/ Jordi Girona, 29, 08034 Barcelona, Spain

^b Computer Architecture Department, Technical University of Catalonia (UPC), C/Jordi Girona 1-3, 08034 Barcelona, Spain

ARTICLE INFO

Article history:

Received 12 July 2007

Received in revised form

30 June 2010

Accepted 10 July 2010

Available online 18 July 2010

ABSTRACT

Tantamount to the overall performance delivered by a Grid environment is the quality of the middleware on which distributed Grid applications can run. Due to its complex nature, this middleware can be difficult to investigate in full detail and can also be problematic to tune efficiently, especially when running on a production type environment.

Thanks to the BSC Monitoring Framework, a set of tools that can instrument and analyze Java applications as well as the entire system, we were able to undertake both global and fine-grained investigation into one of the most popular Grid middleware of the moment, Globus Toolkit 4. The steps taken, revealed some interesting findings and resulted in the detection of some job management problems in this middleware. Primarily, the main issue was that it was possible to reach a situation which caused jobs to be lost on the node due to an overloading amount of jobs being processed by the system. Again, the BSC-MF was used to investigate this issue further and helped extract a possible solution to prevent the node becoming a point of contention in the architecture. A simple but effective policy was formulated, which prioritized the finishing and acceptance of jobs over the response time and throughput, and was evaluated as a solution to the problem.

It was determined that, due to the dynamic nature of the problem, it could be best resolved by adding self-managing capabilities to the middleware. Using the new policy, a prototype of an autonomous system was built and succeeded in allowing more jobs to be accepted and finished correctly. The improvement over the original GT4 middleware was significant and resulted in better performance by a factor of 30%.

The path from investigation to development, as described in this paper, might serve as a guide to others involved in the field who are interested in extracting knowledge about a Grid node, extending the Grid middleware or adding self-managing behaviour to their applications.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

As an enabling mechanism, Grid technologies have been very successful in recent years in improving the provisioning and clustering of a wide variety of geographically distributed resources and services. An important part of this technology is the Grid middleware, which provides a set of extremely complex pieces of software that can be used as the base for providing Grid services. This middleware has to be considered a crucial part of the Grid architecture since it can have an immediate effect on the quality of service provided by the distributed system. Due to its complexity, however, it can be incredibly difficult to analyze effectively. There are almost innumerable factors and parameters that could come into play in this intricate environment: anything from network

devices and services to systems resources and schedulers could be impacting on the quality of the overall service provided by the middleware layer.

Luckily, we have at our disposal a set of tools known collectively as a Monitoring Framework (BSC-MF) that can help us with investigation in this area. The tools are able to analyze and visualize any Java application, combining the middleware layer plus the underlying system layer together as one. This novel feature allows for in-depth, highly detailed, performance analysis of the components of a Grid environment and can give some perspective of their relation to each other as well as their relation to the execution platform. For the scope of this paper, we focus on the performance of a single Grid node (default Globus Toolkit 4, GT4, installation without a Job Queue), with a view to detecting its performance problems as well as their root causes. This node can be considered a critical part of the Grid topology as it could be responsible for accepting all the jobs from clients and distributing them through a cluster server. It should be able to accept the maximum number of jobs possible without reaching a state of unavailability.

* Corresponding author at: Barcelona SuperComputing Center, Nexus II Building, c/ Jordi Girona, 29, 08034 Barcelona, Spain. Tel.: +34 93 405 42 81; fax: +34 93 413 77 21.

E-mail address: rnou@ac.upc.edu (R. Nou).

Since jobs are dynamic in nature, it is hard to formulate concrete policies that will be able to last the lifetime of a running Grid middleware. To provide a better approach with regards to this matter, an idea that is investigated in this paper is the use of Autonomic Computing in the Grid middleware. Autonomic Systems and self-managed environments provide a way for systems to react to (and prevent) specific states when they detect undesirable behaviour. For example, certain outcomes might need to be prevented if they contravene a service level agreement (SLA) or cause instability in the system. A system built in this way could communicate with a virtualized environment to provide a variable amount of resources and increase the return of the system.

As a summary, in this paper we present an extension for a monitoring framework suitable to analyze Java middleware plus the underlying Operating System. We use it to monitor and track the problems that appear on a basic node installation of Globus Toolkit 4 when submitting a large number of jobs. We select one single node as this is the basic unit of GT4 accepting jobs. The problems come from the nonexistent relation/collaboration between the Grid middleware and the underlying Operating System. Finally, we present a static solution and a dynamic one using concepts of Autonomic Computing. The dynamic solution is able to reduce states where the middleware is unable to work normally (accept jobs/finish jobs) and reduces to zero the number of jobs canceled or lost.

The remainder of this article is structured as follows. Section 2 describes the issues that are of concern when trying to extract knowledge about the performance of Java middleware and how the BSC-MF achieves this. Section 3 shows the steps we took to investigate the middleware and Section 4 describes the proposal that was formulated to solve an issue we identified during investigation. Section 5 reveals an evaluation of the prototype of that proposal, and finally Section 6 mentions some of the conclusions that can be drawn from all of this work.

2. Tracing and profiling Java middleware

Since middleware is typically run over a complex environment, there can often be problems trying to measure and analyze its performance accurately and in depth [1]. We are only focusing on Java middleware for now due to its popularity and the fact that it has been successfully used to provide services, such as the Grid, over heterogeneous environments.

Since the quality of the service being provided by the middleware node depends not only on the middleware layer, but also the underlying system layer, information from both layers needs to be obtained to get the full picture. System tracing, such as that provided by Sun's Dtrace, or the open-source tracing tool for Linux, Linux Trace Toolkit [2], can provide sufficient details on the system layer but cannot be relied on solely, as it does not have sufficient scope to provide any details about a Java application running in a JVM (Java Virtual Machine).

A requirement often missed by profiling tools is the fact that they should be able to be run in a standard production-type environment. If there are excessive overheads introduced by a monitoring tool, then this automatically rules it out as being able to provide accurate readings for the performance of the middleware. Using the standard profiling interface, JVMPi (Java Virtual Machine Profiler Interface) [3] (or JVMTI, Java Virtual Machine Tracing Interface [4] which replaces it), in the JVM supplied by Sun can be particularly heavy on the system and is therefore not recommended for exclusive monitoring of an application. Some JVMs, such as Sun's JFluid [5] or BEA's JRockit, provide more efficient mechanisms for monitoring applications, but they are only really intended to be used during the development cycle. They can be helpful when trying to prevent things such as memory leaks

and ensuring that code is written solidly, but unfortunately the performance achieved by these JVMs may bear no correlation to what is actually possible on a production machine running in the real world.

There are also some commercial offerings that can provide useful information on applications but may be too limited in some respects. Borland's OptimizeIT [6] and Quest's Performance Management Suite [7] are geared towards developing new applications while Wily's Introscope [8] aims to provide a full monitoring environment for certain application servers, such as WebSphere or Oracle, for example. To reduce the impact on performance, many of these tools use a sampling method of tracing which records the state repeatedly after a set interval. This is good at on-the-fly monitoring by does not provide a full picture, and therefore is of no use in scenarios which would require a model to be extracted from the trace.

To provide a way of measuring and analyzing the performance of Java middleware in a production environment, and taking the concerns just described into consideration, the BSC Monitoring Framework was developed. It is ideal at helping to determine the best deployment settings to use in a given environment, and it can isolate any problems that may exist either in the middleware or on the system. It is able to provide full insight on a system by combining traces from both the system level and the Java application level, thus providing a vertical view. To keep overheads low, expensive JVMPi operations were kept to a minimum, and a method of filtering the scope of the trace was provided, as well as other things, which we will go into subsequently.

The BSC-MF, which is built on the ideas given in [9], is composed of two main modules: one is an instrumentation and monitoring framework, and the other is an analysis and visualization tool. Fig. 1 shows the structure of the framework and how it works together to extract knowledge on the performance of an application on a system.

To start a trace of an application or middleware, first a Bootstrap class is used to launch it. This bootstrap replaces the default ClassLoader with our own modified one before starting the application or middleware and prepares the InstrumentationModule class for use. An alternative to this set-up is available with Java 5.0 and above, where the BSC-MF can be launched as a Java agent which prepares and calls the InstrumentationModule class and uses a ClassFileTransformer, rather than a custom ClassLoader, to catch the classes as they are defined by the JVM.

Regardless of which launch mechanism is used, the InstrumentationModule class is central to the tracing process. It initially reads in the filter.xml file (with the help of the jdom package), which allows the user to configure which classes and methods are of importance. This removes the need to trace everything blindly, which would introduce more overheads than needed. Once the filter file is read in, any of the classes and methods included in it are instrumented by inserting bytecode at important points (such as the entry and exit of methods). The instrumentation of the bytecode is facilitated by the Javassist package [10], a bytecode modifying API, and the code that it inserts calls the JIS (Java Instrumentation Suite) tracing interface, which in turn uses the JNI (Java Native Interface) to call the JIS native library. Native code can provide better precision and greater writing speed than currently available in Java, and is the reason it was chosen for the task here. The native code makes very limited use of JVMPi/JVMTI to capture events that cannot be obtained in any other way: namely, the creation and destruction of Java threads and the start and finish of the JVM. If available, the native library will also launch the Linux Tracing Toolkit (LTT) for system tracing. The LTT is able to monitor system processes on a Linux operating system that is running with a patched kernel. It has received much praise for its high-speed buffer, relays, which enables it to write large amounts of data without sacrificing much in terms of performance.

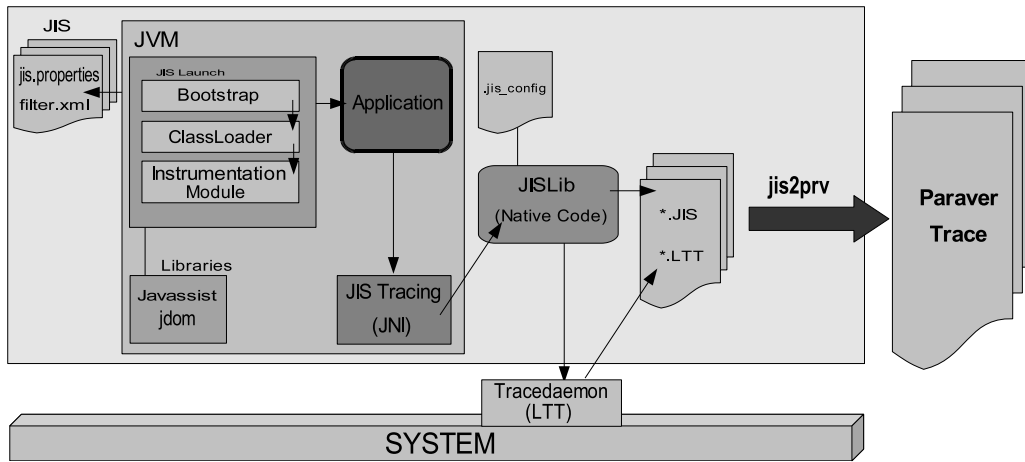


Fig. 1. The BSC-MF structure.

During the running of the application or middleware, system details will be recorded in the LTT trace format and operations running in the JVM will be recorded in a different format by the JIS library. This means that there is one last thing to do before the complete view can be achieved, and that is to combine all of the traces together. This stage is intended to be run offline, after the application has finished running, so that it will not incur any cost in performance. A utility called *Jis2prv* performs this operation and will output a single trace for use with Paraver.

The BSC visualization and analysis tool, Paraver, is very effective at providing detailed quantitative analysis, and is flexible enough to obtain an enormous number of different views on the trace [11]. An important feature of the tool is that it can handle very large trace files, which may result from the fine-grained analysis of an application or middleware.

3. Initial investigations into GT4 middleware

To begin investigating the performance of the GT4 middleware [12], two machines were set up to run as a client and a server using GT4.0.1. The client was programmed to simultaneously submit a large number of jobs (Fig. 2 shows the job workflow inside Globus) at the same time to the server node, and the jobs were written to perform a for-loop that created 5 s of CPU load. Although this workload is not realistic, it reduces the requirements of resources needed to test a stressed middleware/system. The problems presented in this paper could appear in production Grids with other kind of jobs, if they arrive at the same time at the server node.

When measuring the job finishing ratio, it was discovered that, when there were more than 150 jobs submitted at the same time, some of them started to get dropped and the response time to the client also began to increase. We can see this behaviour in Fig. 3.

Since the problem could originate in the Operating System or it could be at the application level, the system-wide tracing capabilities of the BSC-MF were paramount in uncovering what exactly was going wrong: unbalanced CPU power distribution between threads accepting jobs and threads running jobs. In the following subsection, we will introduce the GT4 middleware.

3.1. Globus Toolkit 4

Globus Toolkit is an open-source implementation of a Grid middleware multilevel architecture. It also has a set of software services and libraries for resource monitoring, discovery and management, as well as security and file management. Its core services, interfaces and protocols allow users to access remote

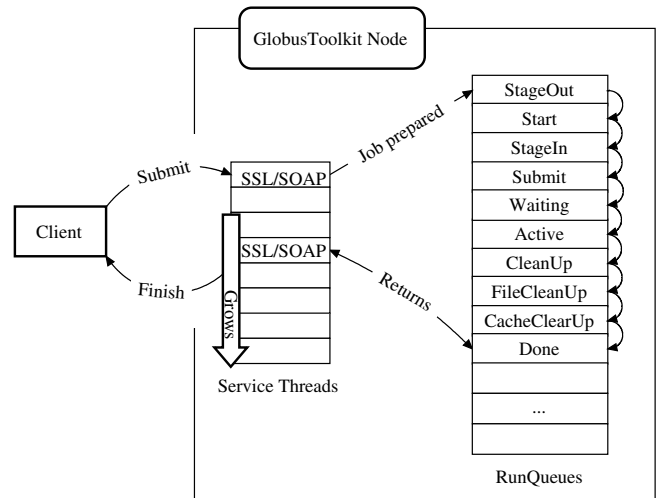


Fig. 2. Job workflow inside a GT4 node.

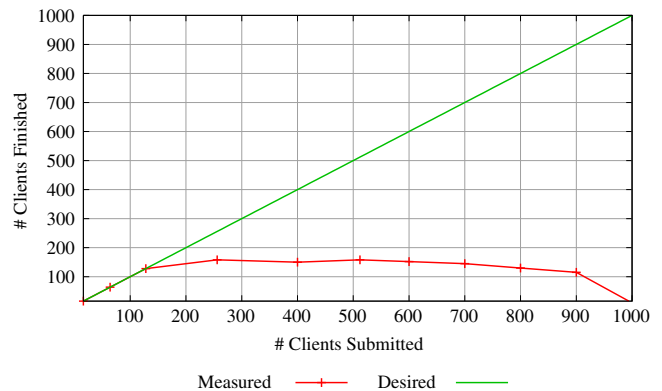


Fig. 3. Behaviour of a GT4 node that is overloaded.

resources as if they were located within their own machine space while simultaneously preserving local control over who can use resources and when. The software is composed of five different components, which provide Security, Data Management, Execution Management, Information Services and a Common Runtime. We focus on two of these components. WSRF is a WSRF-compliant implementation of a protocol for communicating with a range of different local resource schedulers using a standard message format, and it provides execution management. Java WSCore takes care of the common runtime and is a set of Java

libraries and tools that allows the GT4 Web services to be platform independent.

3.2. Identifying the problem

Initially, the BSC-MF traces were made of different workload levels so that they could be compared to one another, both globally and in fine-grained detail. The representative workloads were selected as a 1-job submission workload, a 128-job submission workload and a 256-job submission workload. The trace of the 1-job workload could be considered the most desirable behaviour, since there was no identifiable problem with it running with such light demand. The traces of the 128-job (Fig. 4) and 256-job workloads were taken to capture the middleware under normal load conditions and under overloaded conditions, respectively.

Paraver, the visualization and analysis component of the BSC-MF, allowed us to easily compare the two workloads and see the distribution of the CPU power across the different layers. On a global level, the first thing that stood out was the high use of the CPU in both the 128-job and 245-job cases. This is to be expected, because the submitted jobs run a for-loop consuming 5 s of CPU power, but it appears that the management of the available resources may not be adequate for the situation under test.

The next thing to note was the behaviour of the key processes and threads that were being run in the node during the experiment. At the middleware level, in the JVM, the ServiceThreads and Runqueues were identified as two important threads, which are supplied by the WS-core and WS-GRAM services of GT4 [13,14]. ServiceThreads are responsible for dealing with incoming requests to GT4 and provides functions such as socket operations and SOAP processing, while the Runqueue's role is to move the job from stage to stage in Globus until it is completed. At the OS level it was necessary to look at the for-loop job being submitted and how it ran on the system.

Filtering these three important processes and comparing the difference between the normal running 128-job workload and the overloaded 256-job one showed that there was interference in the usual operation of the JVM threads introduced by the processes running at the system level of the OS. It could be seen that the default scheduling policy for newly arriving jobs was to execute them as quickly as possible on the system. Unfortunately this means that under high loads it can reach a situation where nearly all the available CPU is being used by the jobs on the OS, and this means that there is not enough free CPU power to receive any more incoming jobs. At the beginning of a job request, the ServiceThread can have high demands on the CPU while it establishes new SSL connections. This has been identified before as a problem in some systems [15] and looks like it could be the cause of the job rejection in our tests.

Lastly, we also performed some statistical analysis on the trace of the 1-job workload, paying particular attention to the running of the process at the system level. It was striking that the number of context switches for a 1-job workload was 92, but when we compared the average number of context switches for the 128-job workload it increased to 218. Although context switching may not be the root cause of the problem, it is very indicative of the situation.

3.3. Proposing and evaluating a solution

After thoroughly analyzing the behaviour of the middleware under stress, the next step was to come up with a solution to the problem. To sum up, our primary goal is to get more jobs successfully finished on the middleware and we have already identified the main obstacle in accomplishing this as being the lack of free CPU power to handle new incoming job requests. A

symptom of this is that, as the number of jobs in the system increases, so too does the number of context switches for each job. In other words, when we send more jobs to the node, there is an increase in the average completion time for the jobs, which also increases the chance that there will not be enough CPU power available for new requests.

To make sure that there is always enough CPU power available for incoming requests, it was decided to limit the number of job processes running at any one time on the OS. A minimum value for the number of processes allowed to be run at any one time was specified with regard to the number of CPUs available on the node. In our test set-up, the server had two processors, so we limited the number of jobs that could run on that system to two. This clearly will have a negative effect on the performance of the middleware node, but the expectation is that there will be a gain in the overall throughput and more jobs will be able to be completed successfully. In an attempt at reducing the high number of context switches of the job processes, the priority of these processes was also increased on the system.

After the middleware node was modified with the recommendations just described and tested, we confirmed that this proposal was in fact able to deliver more completed jobs, as we can see in Table 1. The modifications also showed that, as suspected, it slowed down the throughput.

While we have discovered a solution to getting more jobs completed, it appears that there needs to be more of a balance to try to maintain both good reliability and good throughput, if possible. It might be possible to achieve this by separating the solution into two different stages. One stage could handle the situation where the middleware node is being bombarded by multiple job requests, and it would implement the changes that we identified were necessary to get better throughput. The other stage would be used by the node when it is not receiving any more new jobs, and it would use the original default GT4 container policy.

A simple implementation of the proposed two-stage policy was created, by modifying the original middleware, and then used to test how valid this solution was. Initially it started off in the first stage, guaranteeing that new jobs would be received, and then after an arbitrary time it would go into the second stage and prioritize the processing of the jobs on the system. The modified middleware was run with a 128-job workload and the BSC-MF captured a trace of it to investigate the viability of the resource management policy. The resulting visualization of the trace can be seen in Fig. 5, where the first dotted line indicates the start of the test and the third dotted line shows when it changed from stage one to stage two of the policy.

The characteristics of the first phase were a medium level of CPU usage and a high density of ServiceThread activity. The second phase shows the RunQueues working through the other steps of the jobs in the system, and there is a peak of activity. However, since it is not a critical stage, this works well and results in the CPU needs of the middleware being fulfilled for the entire run. Most significantly, there was no noticeable slowdown in the amount of time it took to complete the jobs. In the original middleware we were able to process 128 jobs (which were submitted in parallel) in just less than 450 s. In the modified middleware we were able to complete the same workload in about 450 s as well.

Running the modified middleware with workloads containing a larger number of jobs also produced a positive result. We were able to reduce the average response time for the jobs and increase the overall number of them executed. Taking the case where 256 jobs are submitted as an example, in the original middleware only 150 of the submitted jobs are completed, while in the modified middleware all of them get completed successfully. Fig. 6 shows some views that were extracted from the trace of the two-stage policy running 256 jobs. As in the previous Fig. 5, the first dotted

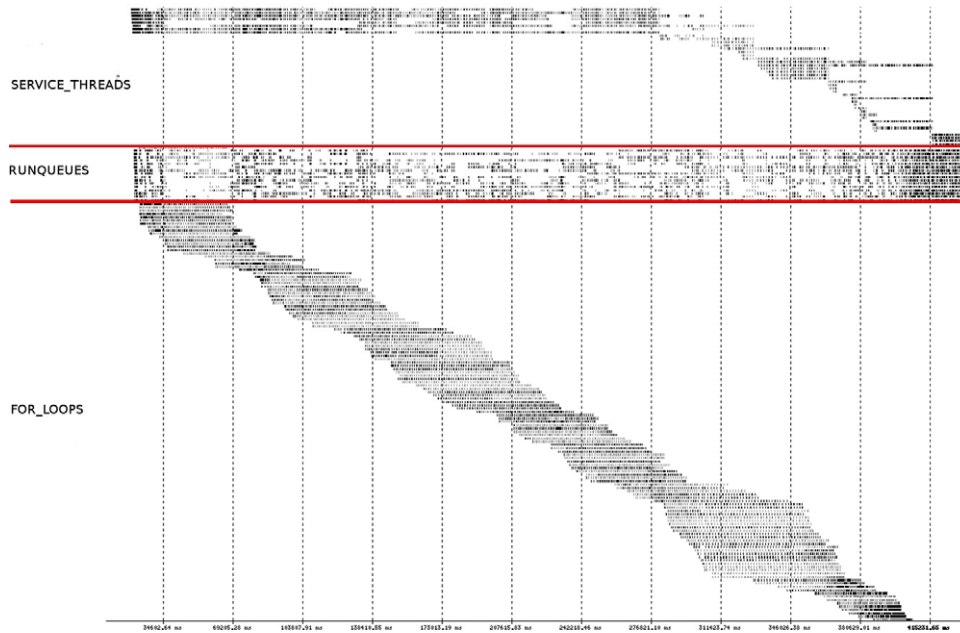


Fig. 4. Paraver view of the 128-job trace. The X-axis represents time, and the Y-axis represents threads.

Table 1

Summary of results obtained with the two-jobs limitation.

Number of jobs	Average response time	Jobs executed (original/modified)
16	40, 73/5, 34	16/16
32	43, 83/5, 74	32/32
64	47, 44/10, 65	64/64
128	52, 67/15, 8	128/128
256	N/A	158/201
512	N/A	158/238

line indicates the start of the test and the third dotted line shows when the middleware changed from using the first stage of the policy to the second stage. The overall time to complete 256 jobs was around 900 s, which is encouraging, and suggests that the performance is comparable to the original middleware, albeit with better and more stable throughput.

While this two-stage policy appears to be the ideal solution to the problem, there is still one thing lacking: a useful method and trigger for switching between the different stages. This is mandatory to be able to use the proposed mechanism on different workloads. We already know that the CPU usage information from the ServiceThreads could be used to self-adjust and obtain a high-quality middleware in terms of finished job, but what readings of this variable should be used and how can the policy be incorporated in a truly dynamic manner?

Due to the overloaded nature of the environment, we cannot explicitly rely on analytical methods such as Queueing Networks [16], although, if we simplify the scenario, some queueing networks theory could be used to extract preliminary conclusions. The original GT4 middleware distributed the CPU power equally to all threads so, with a high number of jobs, we have a small part of the available CPU power to process incoming requests and a much larger part of it to process the jobs themselves.

A better solution, keeping in mind our aim of increasing the completion ratio of the jobs, would be to provide, for example, 1/4 CPU, namely a small part of the available CPU power, for processing jobs and more to process incoming requests (e.g. 3/4 of CPU), when appropriate. Taking an approach to the problem in this manner would end up providing a general purpose self-managed system as a solution. This kind of approach falls into the field of Autonomic Computing, which we will briefly describe in the next section, and which has been proposed for use with a Grid by others [17, 18].

4. Prototyping autonomic middleware

Autonomic Computing is a term coined by IBM, and it describes their ideas on how a self-managed system can be implemented [19,20]. Their original proposal describes four basic components which work together in a lifecycle to adapt and efficiently run a system in constant flux. These components combine to provide a service in accordance with the policies of the application or system and they can continuously adjust themselves, thereby conforming to dynamically changing factors throughout the runtime. The next section gives a brief description of Autonomic Computing to clarify the concept. This simple but powerful idea has attracted a lot of attention recently [21–23] as it can provide a solution to help operators control and run modern-day servers, which have become increasingly complex and intricate environments over time.

4.1. Self-managed architecture and lifecycle

The four components that IBM describes in an Autonomic System are a General Manager, an Autonomic Manager, Touchpoints and Managed Resources. The General Manager decides which policies should be used, constructs an overall plan and later uses it to guide the application or system, telling it what it has to do to reach a desired healthy state.

The Autonomic Manager is very similar to the General Manager, with an analogous lifecycle, and the same goal of producing and executing a plan according to predefined policies. However, the Autonomic Manager component performs this at a lower level and is therefore often considered the “core” of the autonomic system. It takes care of the self-management lifecycle whereby it reads the system and manages it according to the changes in the readings

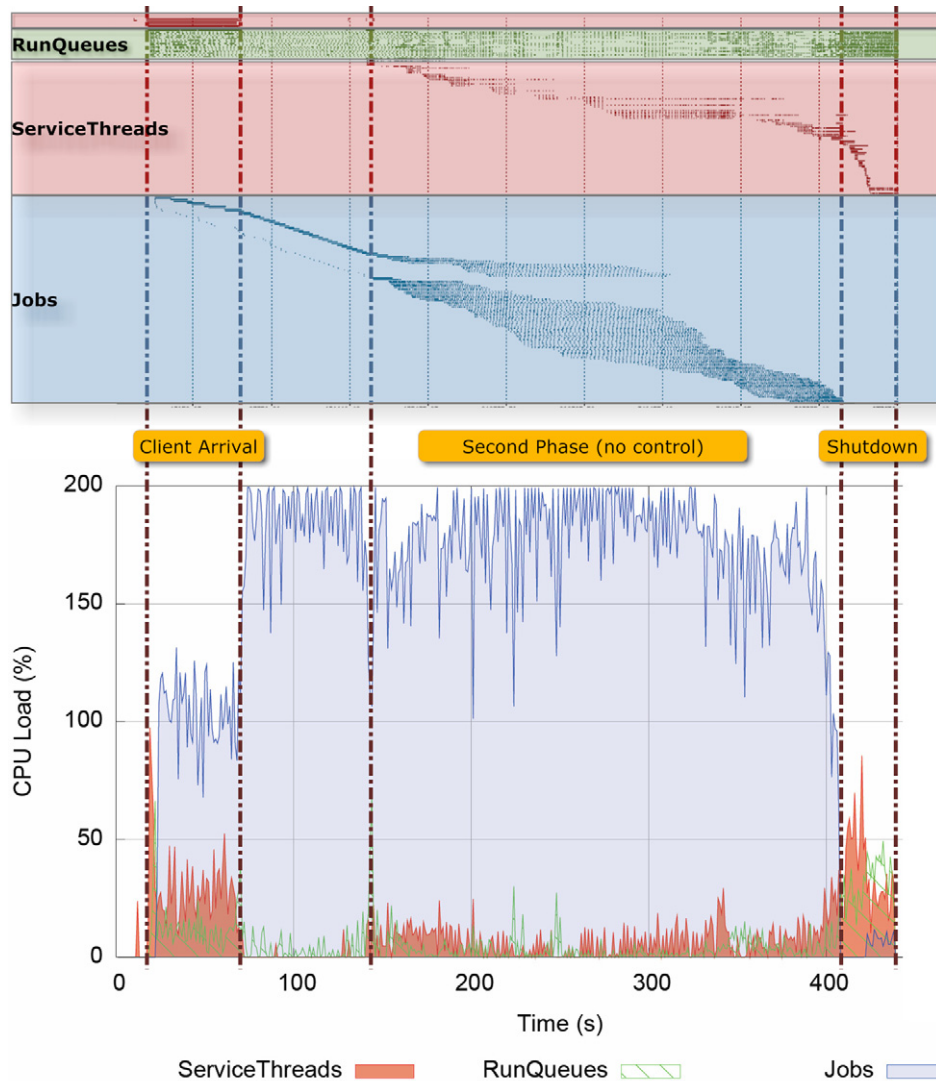


Fig. 5. Two-stage policy applied when we are submitting 128 jobs at once.

and sees how they compare with the identified policies of the system and application.

Managed Resources are those resources that the self-managed system is able to control. Such a resource could map directly to a physical resource such as a hard drive or it could be a logical resource such as a communications channel.

A Touchpoint is essentially an interface which is used to link the Autonomic Manager to the Managed Resource, and there are two different methods for providing interaction between these components. The first, sensors, enable us to consult and check the system's behaviour. The second, effectors, actually let us modify the behaviour of that resource. We can calculate and change the system state using both of these.

The self-managed lifecycle is a general mechanism with which any application can manage itself, and it consists of four distinct phases: monitoring, analyzing, planning and executing. Initially, knowledge is required of the different possible states of the system and how they can be determined using the values available from the sensors. At start-up, the Autonomic Manager can load this along with the policies which will be used to plan the running of the application. Once it is up and running, the monitoring phase is where it calls the sensors of the resources and reads their values. Having completed monitoring, it moves on to the analyzing stage, where it compares the values obtained in the monitoring phase

with the possible states to calculate the current state. The planning stage is entered next, and a plan is formulated based on the current state we are in so that the system can be led to its desired state. Finally, the Autonomic Manager executes this plan by making calls to the effectors of the Managed Resources. The manager repeats the entire cycle every X seconds to capture any changes and adapt its policies.

4.2. Our prototype

Our own prototype was built for a Grid and uses the self-managed architecture to deliver better management of the system resources for a Grid entry node. The prototype was built on top of Globus Toolkit 4.0.1 and is essentially the same as the original middleware but with an additional top-level manager (Fig. 7).

As stated before, the Autonomic Manager is in charge of reading and controlling the node's behaviour using the sensors and effectors of the resources. The primary goal is to avoid the issue that we discovered earlier, where the node exhibits self-destructive behaviour and tries to run all the submitted jobs at once. The number of jobs exceeds the safe levels of the middleware and the system starts failing. Losing jobs at the entry node of a Grid is a serious failing of the middleware and really should not happen under any circumstances. Autonomic

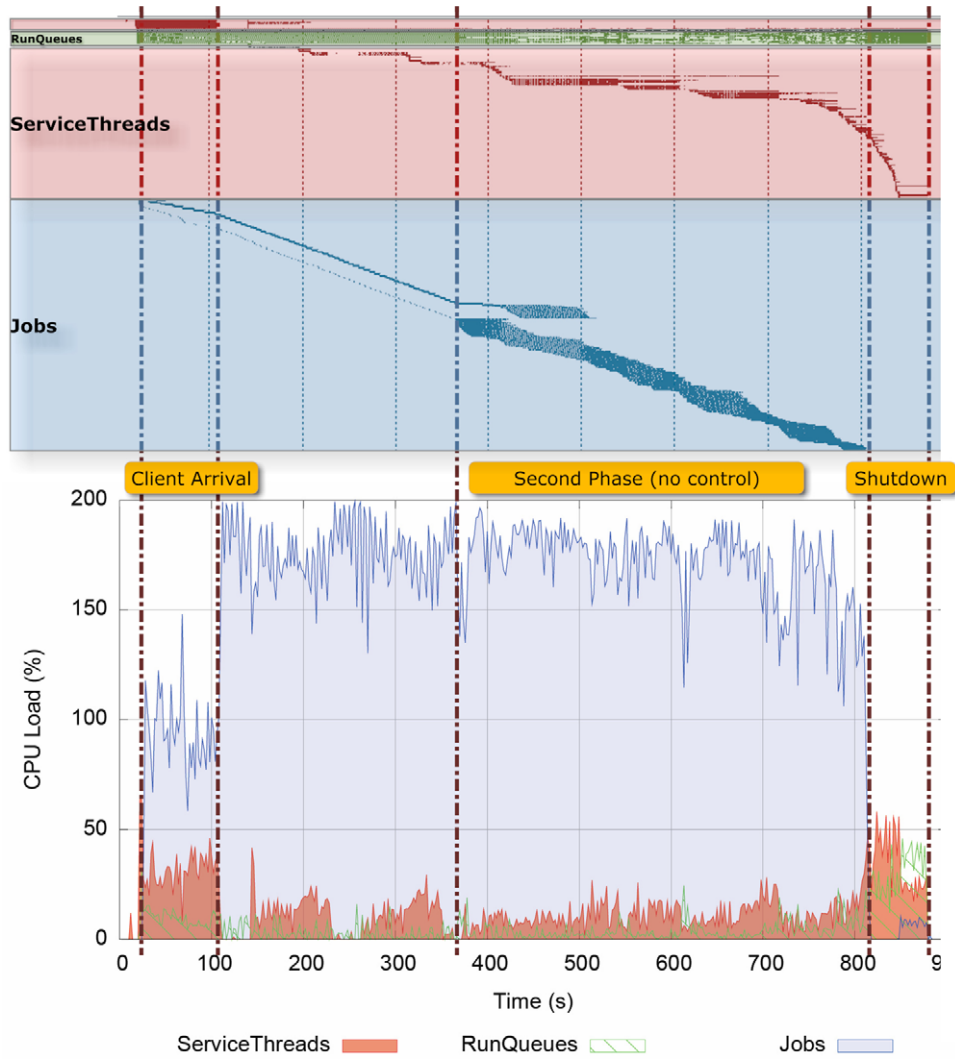


Fig. 6. Two-stage policy applied when we are submitting 256 jobs at once, all jobs executed.

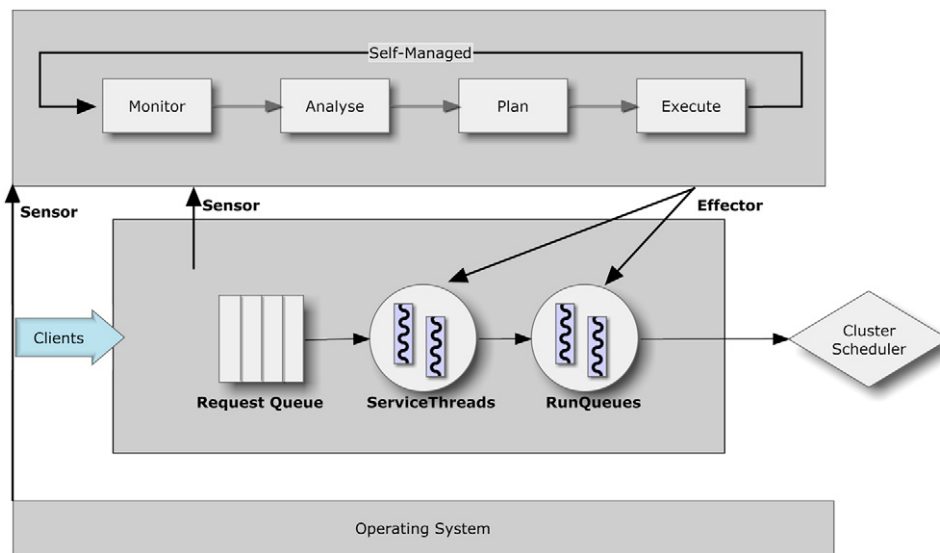


Fig. 7. Diagram of a Grid middleware, interacting with our self-managed prototype.

Computing principles applied to the scenario can allow it to detect this dangerous situation and adapt itself before crashing, thereby wasting fewer resources and managing the node more efficiently.

Since the only policy to be incorporated in our self-managed middleware is to lose the smallest number of jobs possible at the entry point, there is no need to use a fully featured General

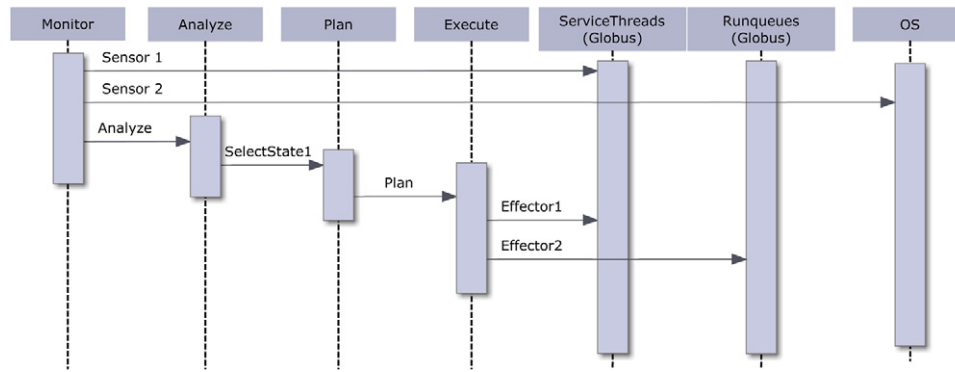


Fig. 8. Sequence diagram of self-managed Globus.

Manager. Therefore, for the purposes of our prototype, this was simply hardcoded. The Autonomic Manager in the prototype loads the knowledge about the possible states of the system, enables the policy and performs the self-management lifecycle every 5 s (selectable). We consider this to be the essence of our self-managed node.

There are only two Managed Resources in the modified node: the GT4 server and the Operating System. Each of these resources has one Touchpoint consisting of a single sensor and a single effector. The sensors and effectors of importance to this situation were isolated during our preliminary investigation of the overloading problem using the BSC-MF. The GT4 server's sensor is the number of ServiceThreads that currently accept requests and its effector is a variable which sets the number of ServiceThreads in the thread pool. The Operating Systems sensor corresponds to the CPU load and its effector allows it to set the number of jobs that Globus can execute on the CPU at the same time. In Fig. 8, we show the interaction through a sequence diagram of Globus and the self-managing layer.

4.3. Evaluation

To evaluate the prototype, a job generator was built to test many different workloads on the entry node. We generate an arrival rate (λ) of jobs per second very big and count the jobs that finished correctly (β). In this scenario we can reduce λ to α ; α counts the number of jobs sent to the system in the same instant. The ideal scenario will be the one with $\alpha = \beta$. An overloaded one will have $\alpha > \beta$. In this test we need another variable, time (t). If we are going to limit the execution priority/time to reduce the number of lost jobs, we also will need more time to run all of them (T). To extract some measurements, we will count β for certain t , where $0 < t \leq T$. All this information will be plotted in a three-dimensional (3D) plot. In these experiments we consider that we can execute all the jobs with a t of 100 s. The workload generated tries to submit from 10 to 100 jobs at once, in 10 jobs steps, and count the number of jobs finished, β , from 10 to 100 s. For completeness, all possible combinations of the variables in this range are submitted to the node by the job generator. The acceptance mechanism is really of utmost interest in these tests, so the jobs in this case are different from the jobs described earlier and this time do not perform any work; they only actually test the job's acceptance. As a side note, we could use other kinds of job, but we will end up with the same problem on the acceptance.

For our test, we wanted to see how both the original Globus middleware and our own modified middleware would perform under different situations. Our test machine, running as an entry node, has four CPUs, so we can reduce the resources available to the middleware by binding the middleware to only use a certain

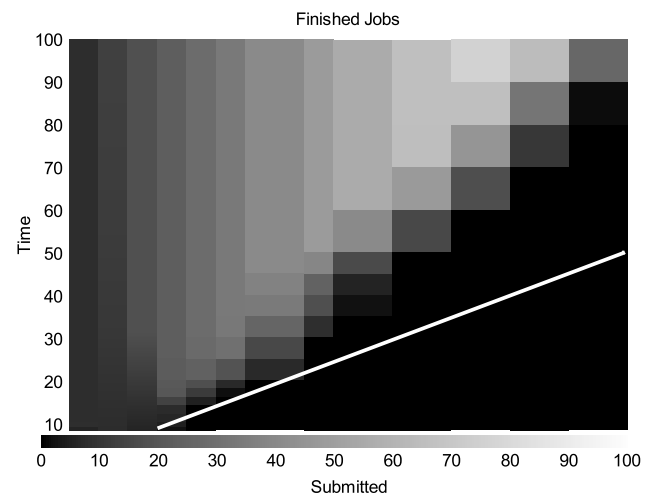


Fig. 9. 100% of CPU available using middleware without self-management. The white line is the ideal frontier from queue theory. The X-axis represents the number of submitted jobs at once. The Y-axis is the time allowed to run the jobs and finish them. The grey scale indicates the finished jobs.

number of the processors. For example, if we only allow it to use two out of the four CPUs then we only have access to 50% of the resources. A reduced amount of resources is important in virtualized environments and environments with heterogeneous workloads.

After running the job generator on a test machine, a 3D plot can be produced showing the number of finished jobs on a grey scale, the number of submitted jobs on the X-axis and the amount of time (in seconds) on the Y-axis. The undesirable situation where the middleware is unable to finish any jobs at all is clearly identified by black zones.

In total, we ran the job generator eight different times to capture the results of the original GT4 middleware and our modified middleware running with 25%, 50%, 75% and 100% of the CPU available. Figs. 9 and 10 show the results of the original middleware running with 100% and 25% CPU, respectively. Figs. 11 and 12 show the results of our prototype running with 100% and 25% CPU. It can be easily seen that the number of jobs being finished by the prototype is higher when comparing the two graphs. There are fewer black zones in the results from the prototype than in the corresponding results from the original. A more detailed analysis of these results can be found in [24].

5. Conclusions

By undertaking the tests and analysis described in this article we have been able to reveal many different things of use, both in

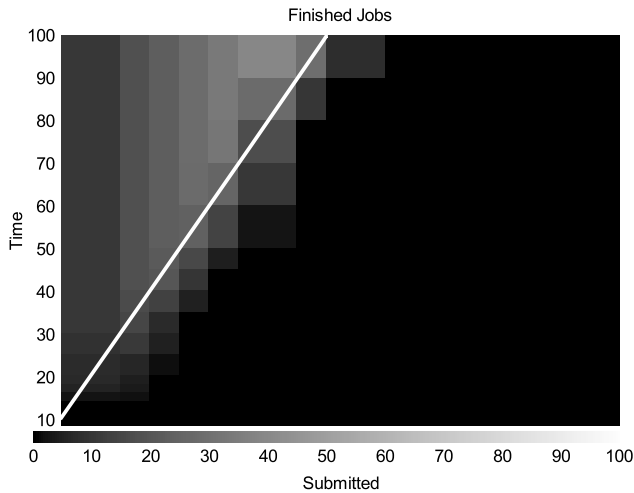


Fig. 10. 25% of CPU available using middleware without self-management.

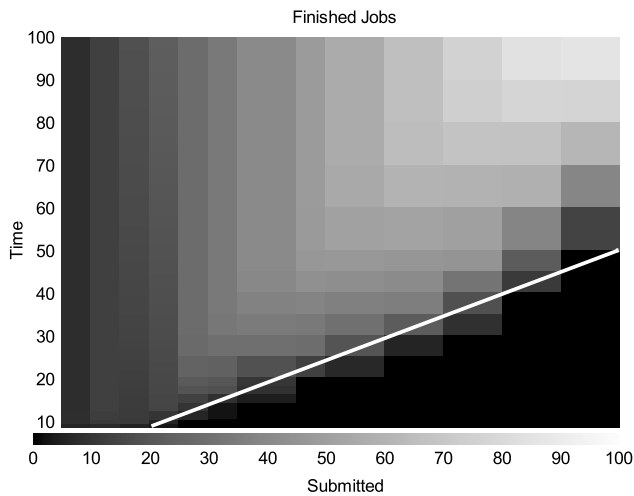


Fig. 11. 100% of CPU available using middleware with self-management.

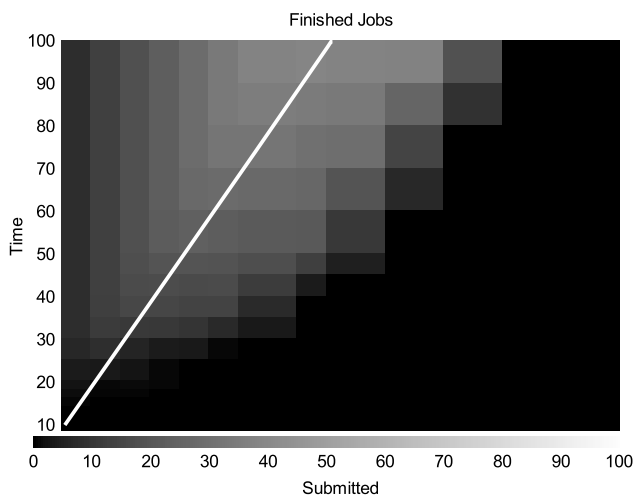


Fig. 12. 25% of CPU available using middleware with self-management.

the general field of performance analysis in middleware and, more specifically, in our aim of producing a robust and efficient Grid middleware.

On the analysis side of things, we discovered how difficult it can be to effectively analyze the production environment and

extract the best deployment settings to use in complex systems such as those typically found on modern-day servers. We identified that, to gain full insight into the system, the data needs to be as complete as possible and monitoring tools must look further than simply monitoring a single application on its own. Other processes on the system could well be using resources required by the application or middleware and cause bottlenecks which possibly degrade the performance. It may be possible to solve some of these issues by reprogramming part of the application or configuring the application in a different way, although it is also likely that the problem would need to be solved by changing things within the system configuration or possibly even in the operation of a third-party application.

Thankfully, the BSC Monitoring Framework can provide a solution in this tricky type of environment as it supports real-time correlation of data from several different layers (namely the system, middleware and application layers), and can help experts extract knowledge that is invaluable to the creation of effective management policies. The problematic issue discovered, where the GT4 middleware gets overloaded when hit with a large number of jobs requests, could be difficult if not impossible to diagnose and solve without our global monitoring framework.

While investigating the specifics of the Grid middleware using the BSC-MF, we realized the usefulness of Autonomic Computing principles to this area. The dynamic nature of the environment makes it extremely hard to come up with one single configuration that will serve it well throughout its entire life, regardless of the circumstantial changes to its workload or available resources. We consider our prototype of a self-managed Grid middleware (which can adapt itself to changing requirements at runtime) a success because it reduced the average response time per job, while also reducing the number of context switches, and ultimately achieving our goal of increasing the number of jobs that can be finished on an overloaded server. This is an important improvement on the original, as the loss of jobs on a Grid middleware is not desired when flash crowds appear on the node, or at any other time for that matter. The wide range of loads and scenarios that we tested allowed us to verify, using 3D plots of the results, that it was able to lose a lot fewer jobs using the same resources.

6. Extended work

Our work in this area resulted in the reduction of this kind of stressed environment by using a prediction layer in front of a cluster of Grid nodes. Using this kind of layer improved the reliability, performance and response time of the system while maintaining the Grid middleware unmodified. The prediction layer is built using OMNeT++, and introduces QoS (quality of service) and SLA (service level agreement) management on a Grid Cluster using online simulation [25]. More details about this extended work can be found at [26]. Finding the exact behaviour of GT4 to be able to model it was only possible with the work explained in this paper. Other kinds of work exploring this kind of prediction layer are found in [27,28], but using simulation is more flexible.

Acknowledgements

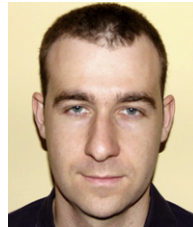
This work is supported by the Spanish Ministry of Science and Technology under grant TIN2007-60625, and by the Catalan Government under grant 2009-SGR-980.

References

- [1] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, J. Labarta, Complete instrumentation requirements for performance analysis of web based technologies, in: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, Austin, USA, March 2003.
- [2] Karim Yaghmour, Michel R. Dagenais, Measuring and characterizing system behavior using Kernel-level event logging, Usenix Annual Technical Conference, San Diego, CA, June 2000.
- [3] Sun Microsystems, JVMPI user manual, 2006. <http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html>.
- [4] Sun Microsystems, JVMTI user manual, 2006. <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html>.
- [5] M. Dmitriev, Design of JFluid: profiling technology and tool bases on dynamic bytecode instrumentation, Sun Microsystems Technical Report 2003-0820, 2003.
- [6] Borland, Borland's OptimizeIT product, 2006. <http://www.borland.com/us/products/optimizeit/index.html>.
- [7] Quest, Quest profiler, 2006. <http://www.quest.com>.
- [8] Wilytech, Wily technology, 2006. <http://www.wilytech.com>.
- [9] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, J. Labarta, An instrumentation tool for threaded java application servers, XIII Jornadas de Paralelismo, Lleida, Spain, September 2002.
- [10] S. Chiba, Javassist—a reflection-based programming wizard for java, in: OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.
- [11] Gabriele Jost, Hoaquian Jin, Jesús Labarta, Judit Gimenez, Jordi Caubet, Performance analysis of multilevel parallel applications on shared memory architectures, in: International Parallel and Distributed Processing Symposium, IPDPS, Nice, France, 2003.
- [12] Borja Sotomayor, Lisa Childers, Globus Toolkit 4: programming java services, 2005.
- [13] Globus Alliance, Globus Toolkit Documentation JavaWS-Core Component, 2005. <http://www.globus.org/toolkit/docs/4.0/common/javawscore>.
- [14] Globus Alliance, Globus Toolkit Documentation, WS-GRAM, 2005. <http://www.globus.org/toolkit/docs/4.0/execution/>.
- [15] J. Guitart, V. Beltran, D. Carrera, J. Torres, E. Ayguadé, Session-based adaptative overload control for secure dynamic web application, in: ICPP-05, Oslo, Norway, June 14–17, 2005.
- [16] G. Bolch, S. Greiner, H. De Meer, K.S. Trivedi, Queueing Networks and Markov Chains, in: Modelling and Performance Evaluation with Computer Science Applications, Wiley, New York, 1998.
- [17] M. Agarwal, V. Bhat, H. Liu, V. Matossi, Automate: enabling autonomic applications on the Grid, in: Proceedings of Active Middleware Services, AMS, 2003.
- [18] J. Hau, W. Lee, S. Newhouse, Autonomic service adaptation in ICENI using ontological annotation, in: Grid Computing, 2003, Proceedings, Fourth International Workshop on, 2003, pp. 10–17.
- [19] IBM, An architectural blueprint for autonomic computing. <http://www-128.ibm.com/developerworks/autonomic>.
- [20] J.O. Kephart, Research challenges of autonomic computing, ICSE, 2005, pp. 15–22.
- [21] D.M. Chess, G. Pacifici, M. Spreitzer, M. Steinder, A. Tantawi, Experience with collaborating managers: node group manager and provisioning manager, in: Autonomic Computing, 2005, ICAC 2005, Proceedings, Second International Conference on, 2005, pp. 39–50.
- [22] D. Menasce, M. Bennani, H. Ruan, On the use of online analytic performance models in self-managing and self-organizing computer systems, in: Lecture Notes in Computer Science, vol. 3460, 2005, pp. 128–142.
- [23] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, M. Trubian, Resource management in the autonomic service-oriented architecture, in: ICAC, 2006, pp. 84–92.
- [24] R. Nou, F. Julià, J. Torres, The need for self-managed access nodes in grid environments, in: EASE'2007, Tucson, Arizona, March 2007.
- [25] S. Kounev, R. Nou, J. Torres, Autonomic QoS-aware resource management in Grid computing using online performance models, in: 2nd International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2007, Nantes, France, October 23–25th, 2007. ISBN: 978-1-59593-819-0.
- [26] R. Nou, S. Kounev, F. Julià, J. Torres, Autonomic QoS control in enterprise Grid environments using online simulation, Journal of Systems and Software 82 (2009) 486–502. vol. 82, no. 3, 2009. ISSN: 0164-1212.
- [27] Y. Gao, H. Rong, J. Zhaxue, Adaptive grid job scheduling with genetic algorithms, Future Generation Computer Systems 21 (1) (2005) 151–161.
- [28] J. Cao, D.P. Spooner, S.A. Jarvis, G.R. Nudd, Grid load balancing using intelligent agents, Future Generation Computer Systems 21 (1) (2005) 135–149.



Ramon Nou worked at BSC in the Autonomic System and e-Business Platforms group of BSC from 2006 until 2008, when he switched to the Storage-system group led by Dr. Toni Cortés. He has been working on SORMA EU project and in the XtreamOS EU Project. In 2008, he obtained his Ph.D., with Prof. Jordi Torres as advisor, on the topic "Using online simulation to improve QoS on middleware". He has published 15 papers at international conferences and workshops, and has one journal paper. Ramon has a wide view on all computer levels, with expertise on optimization, performance measurements and simulation/modeling of complex systems. Now is also collaborating with the Autonomic System and e-Business Platforms group in the simulation of large cloud systems focusing on energy efficiency.



Ferran Julià has a master's degree in Physics from the University of Barcelona (UB) and a bachelor's degree in Computer Science from the Technical University of Catalonia (UPC), where he is currently finishing a master's degree in Computer Architecture and Network Systems (CANS). In addition, he works as a researcher at Barcelona Supercomputing Center (BSC). His research interests are simulation and power-efficient computing in cloud systems.



Kevin Hogan achieved an honors B.A. Mod degree in ICT from Trinity College, Dublin, in 2001. He worked for several years as a Software Engineer with Skillsoft, a leading company in the field of e-Learning, and then joined BSC in 2005 to pursue his interests in research. He was involved in the monitoring and analysis of middleware since 2008.



Jordi Torres is a full professor at UPC and is a manager for the Autonomic Systems and e-Business Platforms research line in Barcelona Supercomputing Center (BSC). His current principal interest as a researcher is making IT resources more efficient to obtain more sustainable IT. He has worked in a number of EU and industrial research projects. He has more than a hundred publications. He has been Vice-dean of Institutional Relations at the CS School, and currently he is member of the UPC Board of Governors.