Should the grid middleware look to self-managing capabilities?

Ramon Nou, Ferran Julià
Computer Architecture Department
Technical University of Catalonia
Campus Nord, Jordi Girona 1-3, E08034 Barcelona, Spain
{ rnou, fjulia }@ac.upc.edu

Jordi Torres
Barcelona Supercomputing Center (BSC)
Centro Nacional de Supercomputación
Jordi Girona, 29, E08034 Barcelona, Spain
jordi.torres@bsc.es

Abstract

Grid technologies have enabled the clustering of a wide variety of geographically distributed resources and services. While providing this, the grid middleware layers that make up the supporting platform for grid applications have taken an increased level of importance in the overall performance of such distributed applications.

In this paper we discuss the necessity of introducing self-managing capabilities to the core functionalities of the grid middleware. Our opinions are based on a simple example that shows how Globus Toolkit 4 (GT4) can be driven to a state of unavailability under certain overloading conditions, and how a simple but effective self-managing policy applied to its resource management mechanisms could overcome such an unfavourable scenario. We present an approach showing the benefits that a resource management introduced in the middleware can provide to the user. This approach is the base of a self-managing layer that is being developed for use under more generic conditions.

1 Introduction

The middleware layer is crucial because it can have an immediate impact on the quality of the service. At the moment, with rising complexity, it is becoming harder to effectively analyse the middleware layer. In this paper we aim to present our proposals for a specific middleware, Grid [24, 25], although the results could be applied to other platforms.

Inside Grid, the infrastructure provides some integrated technologies to enable the creation of distributed applications. There are several efforts to consolidate solutions for these issues such as Open Source initiatives, government funded projects or proprietary solutions [20]. One of them is the Globus Alliance [25], which is a joint project between certain research centers, universities and other organisations to create fundamental technologies supporting the Grid. One of their contributions was the specification of the Open Grid Services Architecture [12] and the Open Grid Services Infrastructure [13]. These specifications define a global architecture and a set of interfaces for the Grid. The first implementations of the OGSI specification were in early versions of Globus Toolkit [25] and the new WSRF are implemented in GT4. This is composed of several services and libraries which offer a development and execution middleware to create and deploy Grid applications easily. There are still significant challenges at various different levels facing the use of a global Grid infrastructure [22]. One of them is the performance of the middleware. For example, when a Grid node receives several jobs, the overhead introduced by the middleware can have a noticeable impact on the overall execution. Managing connections, parsing requests (XML) and dealing with the Web Services protocols are tasks that require a significant amount of computing resources. Tuning the environment, and especially the application server supporting the Grid middleware framework, is a crucial step in order to improve the quality of service offered by a Grid node [11] but is not easy because the jobs requested are dynamically modified at run-time. The most appropriate solution to this is to offer the ability for middleware components to manage themselves. This can be solved using proposals from the Autonomic Computing research area, which draw on an enormous diversity of fields within and beyond the boundaries of traditional research in computer science [18, 21]. To succeed in this approach, knowledge needs to be extracted from the observation of complex systems to allow a detailed analysis of their behaviour and this can be a tedious task.

This paper addresses the question of a need for an autonomic Grid middleware in order to fulfill the requirements for attaining these goals. Work on Autonomic Middleware is less common than desired [10], and the lack of work in this area is particularly noticeable when working with Grid middleware. There are some works [1, 16] in the area that take care of some of the issues written down in this paper, but they don't do self-management at a low level (system).

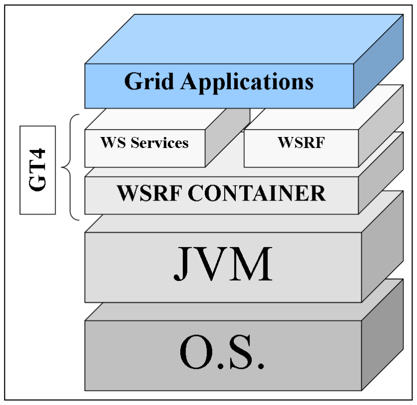

Figure 1. Structure of a simple system with Grid middleware

To show how important self-management can be, we used a simple workload as an example to overload a Globus Grid Node. In that example we sent a huge number of jobs at once, just to stress test the job acceptance capabilities of this particular Grid middleware. We will see that there was a lack of resources available on the server to fulfill the request, because not all the jobs were executed. With a deeper analysis using eDMF(eDragon Monitoring Framework) [19] we can see what is happening in the whole system. Using that information we came to the assumption that we should control server resources in order to fulfill the job and middleware needs. We then built a very simple resource management system that allowed us to test several workloads and showed some big improvements over the original middleware. This approach is being generalised to run under more generic workloads, and we are getting good results. The goal of this paper is to discuss the necessity of introducting self-managing capabilities into the core functionalities of the Grid Middleware. Well-known mechanisms such as dynamic provisioning [5, 8] can be applied to the Grid middleware to improve their QoS.

This paper is divided into the following sections : Section 2 describes the framework used to analyse the whole system. Section 3 shows the system being tested and exposes the experimental tests that allow us to test our hypothesis. Section 4 shows how our proposal works and describes some analysis on it using the monitoring framework.

2 Analyzing the performance of Grid Middleware

2.1 GT4 Organisation

The open source Globus Toolkit 4 (GT4) is a fundamental enabling technology for the Grid. This toolkit includes 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 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 5 different components which provide: Security, Data Management, Execution Management, Information Services and a Common Runtime. For the scope of this paper, we will only focus on two of these components. WS-GRAM [3] is a WSRF-compliant implementation of a protocol for communicating with a range of different local resource schedulers using a standard message format, and provides execution management. Java WS-Core [2] 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.

Figure 1 shows a simplified view of GT4 system like the one studied in this paper. When a job is submitted to a GT4 node, it is first processed by the WSRF container which catches the request and identifies the service to be executed. Once that is determined, the service is invoked, and then the job is dispatched to a system's local scheduler which will later put the task indicated by the job into execution on the system.

Each service contains a number of threads (called Runqueues) that move jobs through their life-cycle states [4]. The number of threads assigned to a service is static in GT4, and they are created the first time the service is invoked. Once a job reaches the submit state, the Runqueues send a request for execution to a local scheduler which will run them on the underlying operating system. In our experiments, we have used the default fork scheduler distributed with GT4, which provides no kind of system level resource management.

2.2 EDMF

With an increasing amount of Java applications being released into highly complex environments, it has become harder and harder to work out the best deployment settings to use and to effectively analyse the production environment. Bearing this in mind, the eDragon Monitoring Framework (eDMF) was developed to monitor and measure the performance of Java applications, track the underlying system running them and display all of this data in an easily digestible format for analysis.

The idea is to first of all use eDMF to spot any inefficiencies or issues of contention on the system as a whole and then to later use it to test different settings/changes which could solve the issue at hand. Solutions could involve reprogramming routines, changing configuration files for the application, or changing the system configuration, amongst other things. To give full insight into the running of an application on a particular environment the trace needs to be as complete as possible and should include any relevant details from the system resources (these have been identified as network, processing, and disk usage). This approach is in stark contrast to other Java profilers/performance monitors since they generally focus on the application only and view things from that perspective [6] [23] [27].

Things such as the memory usage, the thread processing times, thread synchronisation calls, object creation and the number of method invocations are the kinds of factors that are important to an application itself. While profilers providing this kind of information are useful during the development cycle of applications, they often do not have much usefulness during the deployment and running of them in the real world. In complex application servers, where external processes may be sapping resources or causing bottlenecks, this view is too simplistic and ultimately inadequate to determine the applications performance correctly.

The eDMF consists of three separate parts that all work together to achieve our aims. The first part is called the Java Instrumentation Suite (JIS) [7] and contains some tools to trace running Java applications on a JVM as well as parsing tools for merging different trace files. To trace the system level processes running on a Linux OS, a well known application called the Linux Trace Toolkit (LTT) [28] was used. Paraver [17] is a flexible parallel program visualisation and analysis tool and was selected as the best way to visualise the final traces.

In tandem with the goals of the eDMF, it is important for it to have a low overhead and allow the monitored application to be run in a normal environment (as opposed to in a development or debug mode) [7]. This placed certain constraints on how the tools could be developed and restricts the use of many of the standard features supplied with most modern JVMs to measure performance, as they can be very heavy on the system.

Using Paraver, we can analyse traces obtained by the eDMF from the system. Interpreting all the information can be overwhelming, so we are able to select a subset of information that we want to get a graphical view of. We can correlate every subset of information with each other in order to find and understand the impact of several parameters. The main view of a Paraver trace window shows a timeline; on the X-axis we have time and on the Y-axis we have the different threads (or gathered threads in groups such as CPU, TASK or APPLICATION). The data drawn inside this view, can be states, events or even communication lines. One of the main uses of Paraver is to allow us to create a set of operations to calculate a performance index or value, and then to be able to repeat this user-defined test as a configuration file over all the traces. 2D and 3D tools within Paraver make it possible to apply statistical calculations on the information contained in the trace files and compare them to the other data in the file (e.g. the average CPU burst time for a thread in relation to the number of concurrent jobs being executed at each moment in time). The 3D tool extends the 2D tool by adding a third level of correlation among the different statistical metrics calculated over the collected data.

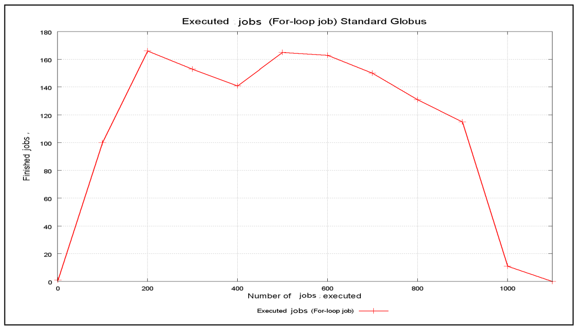

Figure 2. Plot of job completion rate

3 Effects of overloading a GT4 server

In the scenario studied in this paper we consider a client which submits several GT4 jobs at the same time to a single node. Although it's not a realistic test, it's enough to show the effectiveness of a resource management policy. Each job is a CPU consuming task, and the client can submit a variable (but high) number of simultaneous jobs to the server. We submitted all the jobs at the beginning of the execution and created the desired number of parallel clients on the client machine to simplify the testing and analysis. Note that the general proposal that we are currently developing should work for any behaviour. On the server side, we have the GT4 Java container which accepts the clients secure connections and sends the requests to the corresponding services. As each connection is assigned to one thread in the server container, the number of concurrent threads in the server dynamically adapts to the workload conditions.

As will be shown, when the number of concurrent jobs overloads the server, it can reach a state where it becomes unavailable due to being too busy processing the in-execution jobs. The result of this process is that the server starts rejecting new incoming jobs even when the CPU power to process all the requests would be there if an adequate resource management policy were introduced to the system.

3.1 Testing platform and experimental methodology

Our test environment consists of two machines, one a client and the other a server, both with similar features: 2-

way 2.4 Ghz Pentium Xeons, with 2 Gb of memory. There is a 2.6.8 Linux kernel installed on the client and a 2.6.9 kernel, enabled with LTT [28], on the server.

At the application level, we have an installation of the GT4 client and server pair which both use a 1.5 Java Virtual Machine. The installation is simple, so it cannot match a real installation but it can provide us with a way to see possible points of improvement. If we can get better performance on the server then performance of the whole system can be improved.

To emulate the problematic scenario which we want to study, we submit two batches of concurrent jobs to the server. In the first batch we try to keep the system under a high load but without actually overloading it. In the second batch, we try to overload the system. The synthetic jobs used for the experiment each consist of a five second CPU consuming loop. As our system under test is a two-way SMP server, it should be possible to run two jobs in parallel at the same time.

3.2 Job processing: overloaded vs non-overloaded scenario

Our first objective at this point was to find the load threshold that separates an overloaded GT4 server from one operating under normal conditions. From our point-of-view, the server gets clearly overloaded when it starts rejecting new incoming jobs because no more CPU power is available to attend to them.

With this objective in mind, an out-of-the-box GT4 installation was tested with an increasing series of workloads. The results obtained from these experiments can be seen in figure 2, where the X axis represents the number of concurrent jobs submitted to the GT4 server, and the Y axis indicates the number of jobs that where successfully completed at the end of the test. From this experiment, it can be derived that our testing platform running an out-of-the-box GT4 server is able to successfully process up to 128 concurrent jobs without problems. Beyond this point, some jobs start to fail and never more than 170 jobs are completed. When the overloading level becomes extreme, with more than 900 concurrent jobs to be processed, the output level of the server becomes obviously unacceptable.

3.3 Identifying the causes of the job rejection

As we've already found out, under severe overload conditions, the server starts to reject new incoming jobs. We need to determine what is causing the job rejection and then, see if it is possible to overcome or at least lighten this situation.

To achieve this, we make use of the eDMF (see section 2 for more details) so that we can perform an in-depth analysis of the overloaded GT4 server and compare its behaviour with a server operating under normal load conditions. To do this, we chose two representative load levels as seen in figure 2; one for a non-overloaded server (128 concurrent jobs) and one for an overloading scenario (256 concurrent jobs).

As a result of this process, we obtained two trace files representing the execution of 128 and 256 concurrent jobs respectively on an out-of-the-box GT4 server. From the study of the 128 and 256 job submission traces, we extracted information about the CPU consumption produced during the execution period for both workloads. This information was classified according to the source of CPU consumption: the for-loop (system layer), the ServiceThreads (on the server, usually corresponding to socket operations and SOAP processing) and the Runqueues that go through the different stages of the job. Figures 3 and 4 show the plotted results obtained after analysis of the trace file (using eDMF). These figures show the distribution of CPU power across the different layers, which make up the Grid middleware execution stack, when the system is operating under normal conditions and when it is operating under

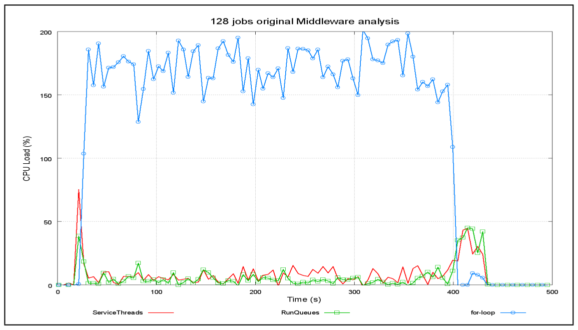

Figure 3. Plot showing CPU usage for 128 jobs on original Middleware

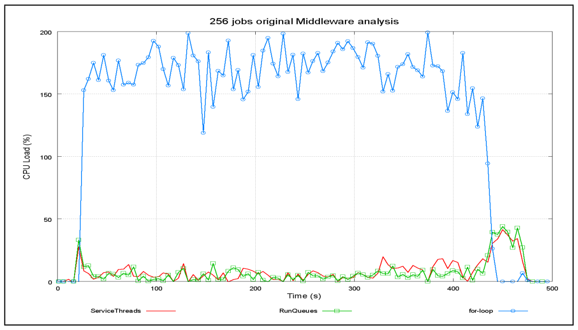

Figure 4. Plot showing CPU usage for 256 jobs submitted on original Middleware (only 150 jobs are actually executed)

overloaded conditions. From these figures, it can be observed that the computing power requirements for the ServiceThreads are quite similar for a workload of 128 concurrent jobs to that of 256 jobs. A deeper analysis of the trace files using the statistical calculations provided by Paraver shows that the problem resides in the way the system level interferes with the normal way that RunQueues and ServiceThreads work. This behaviour generates a loss of jobs inside the system, reducing the number of finished jobs to 150. With this in mind, further analysis from Figure 3 and 4 shows that the CPU needs of the GT4 layers didn't increase (and they should, because we're increasing the number of jobs submitted) because they are restricted by the CPU power needs at the system level. Executing the jobs as soon as they come into the system is not always the wisest policy. We exhaust resources at other levels and we have a higher number of jobs relative to the threads in the Globus middleware. The effect of this behaviour is an inefficient or inadequate management of the system by the Globus middleware. In other words, the new incoming jobs (and their corresponding network connections) are not processed since there is no more CPU power available to do it.

More concretely, it appears that the cause of the unavailability situation is due to the lack of CPU power which is required by the server in order to establish a new SSL connection between it and the client submitting a new job. This is an already identified problem [14], studied in the past.

4 Resource Management Proposal

Our objective at this point is to adjust the behaviour of the whole system by modifying some of its configuration parameters, such as the way threads in the container are created and destroyed, the number of RunQueues or the local scheduler. Hence, a new resource management policy derived from the results discussed in the previous section was introduced to the GT4 container in order to alleviate the overloading symptoms identified above when submitting a large number of jobs at once.

4.1 Resource Provisioning Strategy

The strategy followed by our new resource management proposal is based on a two-stage policy. In order to balance all the tasks at each server layer (executing jobs, managing RunQueues and managing connections) we need to divide the processing into two phases :

1. Phase 1: Receiving new jobs. While jobs are being sent to the ServiceThread level, we try to reduce the CPU needs of the system level (i.e. the job execution in our test case). In order to achieve this behaviour, we will reduce the number of jobs being executed, forcing the parallel execution to a specified number of jobs at once (we limit it to two at the one time because we have a dual CPU machine). This means that, the entry of new jobs in the system will be guaranteed, at the cost of lightly reducing the overall throughput level of the server during a transient period. This ensures that less (or none if possible) jobs will be rejected before reaching the GT4 container.

2. Phase 2: No new jobs being received. Now we switch to the original GT4 container policy (execute as soon as possible) and all the CPU power in the system is available for executing the jobs. In this paper we used a fixed policy to show that we could get some benefits from a more general strategy. We are developing a more general approach using some learning algorithms to take the right decision at the right time. We describe it briefly (although it is still a Work in Progress), and explain how it will work in Subsection 4.4.

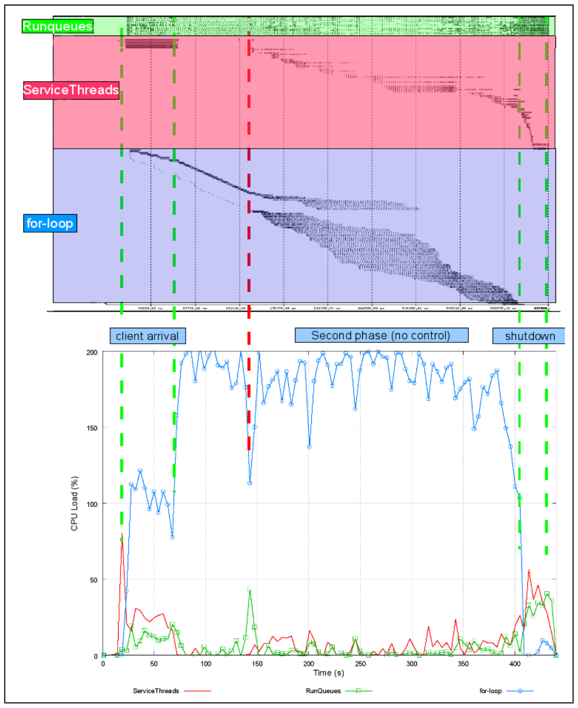

Figure 5. Paraver trace and plot showing CPU usage for 128 jobs submitted on a modified Middleware (450 seconds of execution)

4.2 Evaluation

Our proposal aims to increase the number of jobs being finished by finding an effective trade-off between the re-

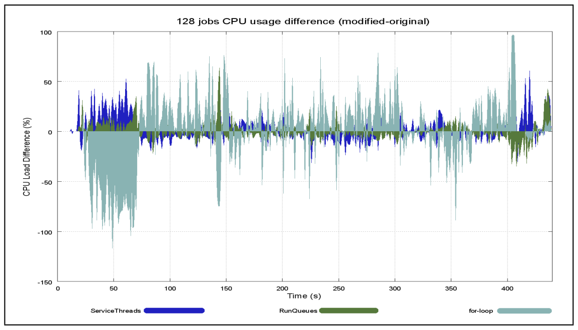

Figure 6. CPU usage difference between modified and original Middleware on a 128 job test

sponse time and the throughput offered by a GT4 container which was modified with the resource management policy discussed above.

Applying the previously presented resource management strategy to a workload consisting of 128 concurrent jobs produces the output shown in Figure 5. This shows a Paraver trace window view on top (with information about the tasks of each thread overlayed) and the corresponding CPU consumption plot below. The two-phases implemented in our resource management policy can be seen, separated by a vertical line (third line from left). It can be observed (between the two first vertical lines) how the ServiceThreads work in order to acquire all the jobs into the Runqueue system.

When the middleware is in the first phase we only submit 2 jobs to the system in order to give more CPU to the ServiceThreads. In this phase, the client submits all the jobs (128) at once, and we can see a high density of ServiceThreads in the Paraver view and a medium level of CPU usage in the plot.

When no more new jobs arrive at the server, since no CPU resources must be reserved for other tasks, the resource management control gets deactivated and all jobs reach the system as soon as possible (second phase). At this moment the Runqueues start to work through other steps (there is a peak on Runqueue usage), but this is not a critical moment,

Comparing the trace and plot views with Figure 3 and Figure 4 we can conclude that the difference in CPU usage comes at the beginning, as can be seen in Figure 6. The difference between the CPU usage on the modified middleware, less the CPU usage of the original middleware are plotted (128 jobs submitted, for 256 jobs submitted it is not applicable). A negative CPU difference means that the CPU usage on the original is higher. We noticed the reduction of CPU usage on the system layer (for-loop) at the start of the test (when jobs enter the system) and there is no noticeable time overhead.

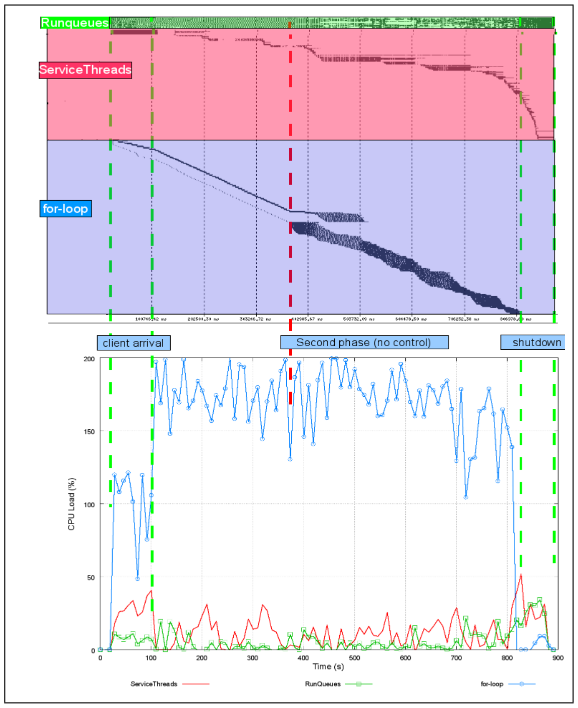

Figure 7. Paraver trace and plot showing CPU usage for 256 jobs submitted on a modified Middleware (900 seconds of execution)

Repeating the last test with 256 jobs (after tuning the start of the second phase for 256 job submissions), produces the trace and plots found in Figure 7.

From a quick comparison between the previous test (X and Y scale on trace is different due to 256 test duration), we can see that the system behaviour is similar. If we compare the first phase, when the client submits all the jobs , with the one in Figure 4, we find that the CPU usage needs of the ServiceThreads are fulfilled with our resource management. When we enter the second phase, the CPU usage of the middleware levels are similar to the ones found before. With these results we can say that the CPU needs of the middleware are fulfilled for all of the test (although it may not be optimal).

With our proposal we are able to execute all 256 jobs in nearly 900 seconds, and 128 jobs in 450 seconds. In the original middleware we executed 128 jobs in nearly 450 seconds, including startup and shutdown time, so it didn't introduce a noticeable slowdown or much inefficiencies.

4.3 Proposal Analysis

In order to show that our proposal could be necessary for the Globus middleware, we ran a batch test that executes a range of 100-1000 submitted jobs (all at once) using both

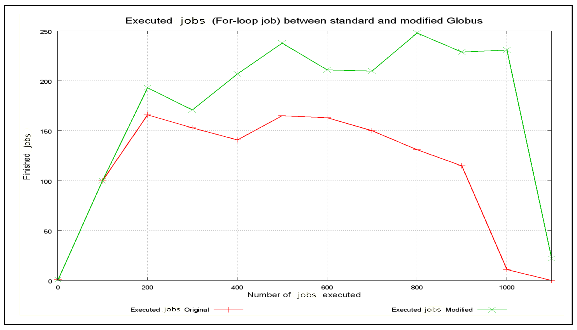

Figure 8. Comparison between the original and our resource management proposal

the original and the modified middleware. The modified middleware is a simplified one, where we are always in the phase one stage (i.e. we are always executing only 2 jobs at once). This is the same resource management system as before, but it is more inefficient in performance terms (because it lacks the phase two stage). While this is a naive modification, it shows us that the system will produce more finished jobs when running as a resource managed system. We can see the results in Figure 8, which shows that when we submit 1000 jobs at once, the original middleware can finish less than 50 of them but the modified one can finish more than 200.

4.4 General Proposal

In order to provide a more general approach we should provide a mechanism that learns from the system. We can use preliminary data and information obtained from the traces (e.g service times, CPU overload levels, ...) to initially feed the system. The eDMF framework provides us with a way to show, analyse and validate metrics that could be used in the are of Autonomic Computing. Now we know that we should get the CPU usage information from the Service Threads in order to self-adjust and obtain a high quality middleware (in terms of finished jobs). Using a self-managed system like this, we could obtain better performance than the one used in Figure 8. We can take decisions and get data from the system at different rates to reduce the intrusiveness of the self-managing system.

In Globus we can use other kind of schedulers, but we should feed them with information about the middleware layers in order to obtain optimal performance.

5 Conclusions

The emerging standards of the new WSRF adds a new dimension to web services. GT4 is a good example of this and useful in showing how important it is to monitor and coordinate all of the middleware layer to achieve better performance in WSRF. A good monitoring framework can help us to see problems where there doesn't seem to be any and the problem that we highlighted in this paper is actually quite difficult to identify without eDMF, our own global system monitoring framework. We need information from the application level, the JVM level and the system level because GT4 is a middleware that finishes running jobs on the system level. Our monitoring framework has allowed us to find and validate a self resource-management proposal which is able to increase the performance of the Grid Middleware. The results achieved in this paper by making a preliminary attempt at self-management on Grid Middleware were successful. We were able to increase the number of jobs finished on a server overloaded due to submitting a huge numbers of jobs at once. This is an important improvement as loss of jobs on a Grid Middleware is not desired. Introducing a layer of self-managing can improve the global system, so it can be deduced that there is a need for proposals from the Autonomic Computing research area in order to improve the QoS of Grid Middleware. These self-managed capabilities must adapt the systems behaviour to changing requirements at run-time, since the jobs can be submitted at any time.

Currently our group is working on a prototype based on well-known mechanisms such as admission control [9, 15] and dynamic provisioning [5, 8]. While Dynamic provisioning enables additional resources to be allocated to an application on demand and handle resource increases, the admission control mechanism maintains the QoS of admitted requests by dynamically limiting the number of jobs in the runqueue.

Acknowledgement

This work is supported by the Ministry of Science and Technology of Spain and the European Union under contract TIN2004-07739-C02-01. Special thanks to Kevin Hogan, David Carrera, Jordi Guitart and Judit Gimenez for their work and support. More information about eDragon Group can be found at [26]

References

[1] M. Agarwal, V. Bhat, H. Liu, and V. Matossi. Automate: Enabling autonomic applications on the grid. 2003.

[2] G. Alliance. Globus toolkit documentation java WS-Core component. `http://www.globus.org/toolkit/docs/4.0/common/javawscore/`, 2005.

[3] G. Alliance. Globus toolkit documentation, WS-GRAM. `http://www.globus.org/toolkit/docs/4.0/execution/`, 2005.

[4] G. Alliance. Globus toolkit WS-GRAM developer documentation. `http://www.globus.org/toolkit/docs/4.0/execution/wsgram/developer-index%.html`, 2005.

[5] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. *Technical Report HPL-2002-339, HP Labs*, December 2002.

[6] Borland. Borland's optimizeit product. `http://www.borland.com/us/products/optimizeit/index.html`, 2006.

[7] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta. Complete instrumentation requirements for performance analysis of web based technologies. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, USA*, March 2003.

[8] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. *11th International Workshop on Quality of Service (IWQoS 2003), Berkeley, California,USA*, pages 381–400, June 2-4, 2003.

[9] X. Chen, H. Chen, and P. Mohapatra. Aces: An efficient admission control scheme for qos-aware web servers. *Computer Communications*, 26(14):1581–1593, September 2003.

[10] D. M. Chess, G. Pacifici, M. Spreitzer, M. Steinder, and A. Tantawi. Experience with collaborating managers: Node group manager and provisioning manager. *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 39–50, 2005.

[11] D.Carrera, J. Guitart, V. Beltran, J. Torres, and E. Ayguadé. Performance impact of the grid middleware. *Engineering the Grid: Status and Perspective, B. DiMartino, J.Dongarra, A. Hoisie, L. Yang and H. Zima. Nova Science Publisher 2005, ISBN:1-58883-038-1*, pages 220–242, 2005.

[12] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum, June*, 22:2002, 2002.

[13] GridForum. Open Grid Services Infrastructure Working Group (OGSI-WG). `http://forge.gridforum.org/projects/ogsi-wg`, 2006.

[14] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Characterizing secure dynamic web applications scalability. *19th International Parallel and Distributed Processing Symposium, Denver, Colorado, USA*, pages 166–176, April 4-8 2005.

[15] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. Session-based adaptive overload control for secure dynamic web applications. *34th International Conference on Parallel Processing (ICPP-05), Oslo,Norway*, pages 341–349, June 14-17, 2005.

[16] J. Hau, W. Lee, and S. Newhouse. Autonomic service adaptation in ICENI using ontological annotation. *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 10–17, 2003.

[17] G. Jost, H. Jin, J. Labarta, J. Gimenez, and J. Caubet. Performance analysis of multilevel parallel applications on shared memory architectures. *International Parallel and Distributed Processing Symposium (IPDPS), Nice, France*, 2003.

[18] J. O. Kephart. Research challenges of autonomic computing. *ICSE*, pages 15–22, 2005.

[19] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Caubet, J. Labarta, and J. Torres. Monitoring and analysing a grid middleware node. *GRID 2006, Barcelona, Spain*, 2006.

[20] M. Parashar and J. Browne. Conceptual and implementation models for the grid. *Proceedings of the IEEE, Special Issue on Grid Computing, IEEE Press*, 93, 3:653–668, March 2005.

[21] M. Parashar and S. Hariri. Autonomic computing: An overview. *UPP 2004, Mont Saint-Michel, France, Springer Verlag*, 3566:247–259, 2005.

[22] M. Parashar and C. Lee. Grid computing - an evolving vision. *Proceedings of the IEEE, Special Issue on Grid Computing*, 93, 3:479–484, March 2005.

[23] Quest. Quest profiler. `http://www.quest.com`, 2006.

[24] M. Romberg. The unicore grid infrastructure. `http://www.unicore.org`, 2002.

[25] B. Sotomayor and L. Childers. *Globus Toolkit 4 : Programming Java Services*. 2005.

[26] WebPage. Barcelona eDragon research group. `http://www.bsc.es/autonomic`, 2005.

[27] Wilytech. Wily technology. `http://www.wilytech.com`, 2006.

[28] K. Yaghmour and M. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.