The need for self-managed access nodes in grid environments.

Ramon Nou, Ferran Julià, and Jordi Torres
Barcelona Supercomputing Center
Computer Architecture Department
Technical University of Catalonia
Barcelona, Spain
{ramon.nou, ferran.julia, jordi.torres}@bsc.es

Abstract

The Grid is constantly growing and it is being used by more and more applications. In this scenario the entry node is an important component in the whole architecture and will become a contention point. In this paper we will demonstrate that the use of a self-managed layer on the entry node of a grid is necessary. A self-managed system can allow more jobs to be accepted and finished correctly. Since it's not acceptable for a grid middleware layer to lose jobs, we would normally need to prioritize the finishing/acceptance of jobs over the response time or the throughput. A prototype of, what could be considered an autonomous system, is presented and tested over an installation of Globus Toolkit (GT4) and shows that we can greatly improve the performance of the original middleware by a factor of 30%. In this paper GT is used as an example but it could be added to any grid middleware.

1 Introduction

Grids [15, 16] are complex, since they can vary over a huge number of parameters: from the network (topology, speed, and security policies), the resources (number of machines, number of processors, speed, and usage of the machine) to the scheduler (fork or more complex ones). On top of this, we need an entry node where we can submit our jobs to. In a normal grid, this node can be a processing node too, but as the complexity of the grid increases and the arrival rate of jobs (λ) increases, we really need a dedicated entry point to be able to process and distribute as many jobs as possible over the grid. Some of the previous studies of this phenomenon [14] show how a node involved in accepting and executing jobs can reach a state where there are more jobs lost than executed. Loosing jobs is an important issue in real life [7] and becomes more important when we are sharing resources with other people. As far as we know, there is no related work using a Grid node, but there are a number of important papers where the self-managed system makes use of other techniques (analytic models [1, 2, 9]) or target Web Servers [10]. With these techniques, we have some of the typical limitations of mathematical models: time to get a result, inflexibility and the lack of modeling certain issues of the software. We started analyzing some of the problems on a Grid Node in [14] and improved the solution presented with more intelligent resource management and made it more suitable for becoming autonomic.

Other studies [5] on systems like Tomcat involving security and high acceptation rates show that part of the problem comes from the SSL handshake. Similarly, security policies are an important part of grid systems, because since we're executing jobs on another system we need to establish trust between the server and the client. Autonomic systems and self-managed environments provide a way in which the system reacts to (and prevents) specific states when it starts to become unstable or its behavior is intolerable. These systems could communicate with a virtualized [11] environment to provide a variable amount of resources (like virtual CPUs) and increase the return of the system [2, 4].

Presented here is a proposal for self-management in grid middleware which greatly decreases the number of jobs lost on a grid middleware node.

The paper is structured as follows: Section 2 shows how we considered trying a self-managed environment on the access node with several previous experiments. Section 3 shows our prototype and how it works. Finally, section 4 presents our conclusions and a guide for future work.

2 Problem Analysis

In this section, we are going to present a problem with this type of middleware, where jobs get lost when a server is overloaded. We start first of all by analyzing the behavior of Globus and showing how jobs get lost.

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

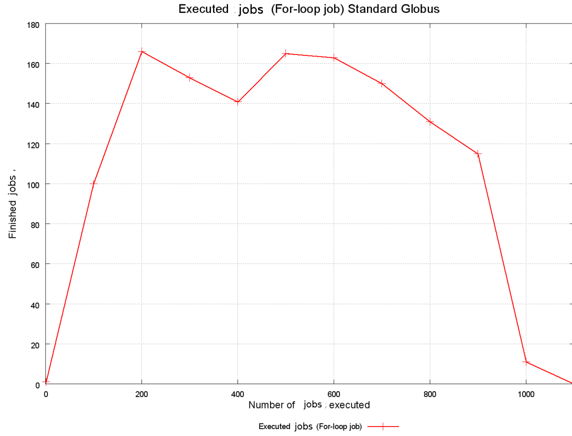


Figure 1. Number of Finished Jobs using a standard GT4

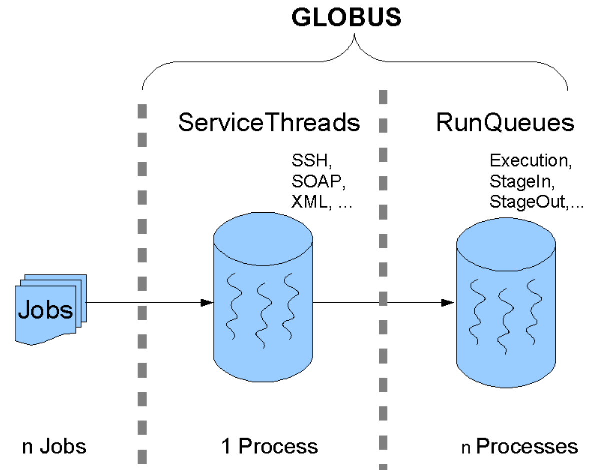


Figure 2. Globus Middleware workflow

# submitted	# finished
16	16
32	32
64	64
128	128
256	158
512	158

Table 1. Jobs finished in preliminary tests

The original behavior of Globus, investigated using eDMF [13] and Paraver [8], is to execute all jobs at once. To test the job finishing ratio, we submitted a huge number of jobs at once, and measured how many jobs finished when t = ∞. We can see in the plot in Figure 1 that we find that we start to lose jobs when we are submitting more than 150 jobs (Table 1 has a summary of the data).

When working in an overloaded environment like this, we can't use analytic methods like Queuing Networks [3]. Nevertheless, we can reduce and simplify the scenario, applying some queue analysis theory to extract some preliminary conclusions. Put simply, in order to process a job on a Grid system, we first need to accept the connection (serviceThreads), and finally process the job (RunQueues and OS, so that it can be submitted to another grid node). All of this processing work is done on the same node. This two-phase workflow is shown in Figure 2. Although we have more than one ServiceThread, for the sake of simplicity we will represent them as 1 in terms of processes in the following formulas.

2.1 Queue Network Analysis

Looking at Figure 3, the first QN system (*M/M/n* queue) shows what happens on the system when we have a huge number of submitted jobs at an entry node. As we have a PS (Processor Sharing) server, all the jobs are executed at once using a quantum (typically *100 Hz*) to switch between tasks and jobs on the system. Moreover, we need to process the arriving jobs on the same server, so we only get for this service a percentage of the CPU equal to the value in equation (1) and the limit is in equation (2). Jobs are discarded or lost at the entry process when there is no CPU for them (i.e. the server gets overloaded).

$$\frac{1}{numJobs + 1} \quad \text{CPU \% per process} \tag{1}$$

$$\lim_{numJobs \to \infty} \frac{1}{numJobs + 1} = 0 \quad \text{CPU \% per process} \tag{2}$$

$$\frac{1}{2+1} = 0.33 \quad \text{CPU \% per process} \tag{3}$$

$$\frac{1}{limit + 1} \quad \text{CPU \% per process} \tag{4}$$

We can improve this scenario by introducing a limitation on the number of jobs that are on the system at once (it can be simulated using a *M/M/n/k* queue). If we push two jobs onto the CPU, they are going to finish at 2*(service time) seconds. In this case, we increase the amount of CPU assigned to process the arriving jobs and we now get equation (3). This value, which is not dependent on NumJobs, needs to be sufficient to process the arriving jobs, or else we are going to end with the same problem as before. We can generalize this value to a variable we called "limit", and then we have the equation (4)

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

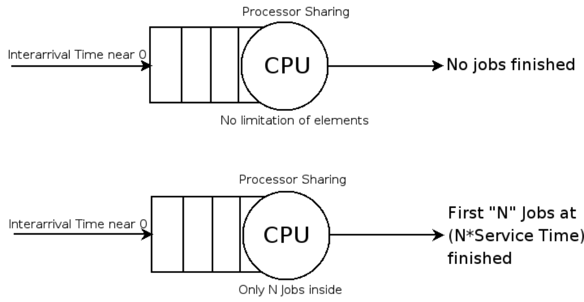


Figure 3. Simplified queue with arrival rate $(\lambda) = \infty$

Another issue arises with this modification. Due to a constantly increasing queue size, we need to include some admission control at the end. We are going to ignore this issue for the moment in the prototype and hopefully, the system we have created is going to be adequate to process flash crowds easily.

What is a good value for "*limit*"? We probably won't have the same arrival rate all the time, and also our resources could be modified at any time, so we need to dynamically change this parameter. We need to get a value for "*limit*" where equation (4) is greater or equal to the processing power requirements needed to accept jobs. It's clear that this value, "*limit*", is somehow related to the number of CPUs on the system.

Let's try another scenario with a different submission ratio. This time, we are submitting jobs with $\lambda = \frac{1}{2}$ jobs/second and a service time of 5 seconds. Using a QN without limitation (Figure 4 (a)), the time between finished jobs increases and the service is going to be overloaded at some point. Also, since there is a timeout, jobs will start to be dropped at some stage. This timeout limits the size of the queue (like an admission control).

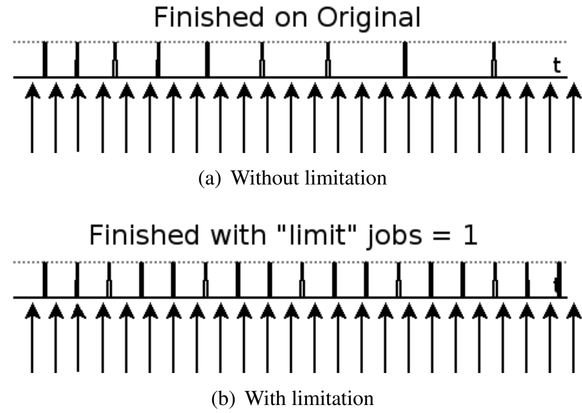
On the other hand, with a QN with a limitation of 1 job at once (Figure 4 (b)), the time between jobs finished is constant. In this case, the service won't become overloaded, but we should control the waiting queue as we won't have the timeout control.

2.2 Job and CPU usage decomposition

Taking a deeper approach, we can consider that at a certain time we have $\tau\%$ of the CPU available. We also know the request's CPU requirements, *p*, and a single job's CPU needs for processing, *j*. So if we process *n* requests and *n'* jobs, we have a total CPU need of Γ where:

$$\Gamma = np + n'j$$

We can consider 3 cases:



(a) Without limitation

(b) With limitation

Figure 4. Cronogram when submitting jobs with $\lambda = \frac{1}{2}$ **jobs/second**

$$\Gamma = \begin{cases} < \tau & \surd \\ = \tau & \surd \\ > \tau & \text{X} \end{cases}$$

In the first and second cases our CPU requirements are lower or equal to the available CPU, so there is no problem. But what happens when we have requirements for more CPU than that? In the standard case, CPU power is distributed equally to all threads, so we will have $\tau/2$ (half of the available CPU) for each processing part.

$$np + n'j > \tau$$

$$\begin{cases} np > \frac{\tau}{2} & \text{lose jobs} \\ n'j > \frac{\tau}{2} & \text{delay jobs} \end{cases}$$

Let's focus on the request process. In this case the system is spending too much time processing the request so we will have timeouts due to the SSL processing. On the other hand, if we have more requirements for CPU on the job processing side, we will have a delay in the job execution. Our self-managed prototype proposes a different scheduling of the available CPU in a similar way with the following expression:

$$\begin{cases} np \leq \frac{3\tau}{4} & \text{don't lose jobs} \\ n'j \gg \frac{\tau}{4} & \text{delay jobs} \end{cases}$$

Therefore, the request processing part requires less CPU than is available and no jobs are rejected.

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

3 Self-managed prototype

In this section, we are going to explain the design of our Self-Management proposal.

To build and test our prototype, we're using two machines; one is a client, which has a job generator submitting jobs to the other, a server, which is running as an entry node using Globus Toolkit 4 (GT4) [16].

3.1 Job Generator

A job is a small piece of code that can be executed on a grid. In our case, we are testing the submission acceptance on an entry node, so a job actually does nothing and only the acceptatance mechanism is used. Since we are attempting to overload the server, we are generating jobs at a rate of λ (where λ is as big as possible) and counting how many jobs are finished (β). In this scenario we can reduce λ to α. α will count the number of jobs submitted to the system at the same time. The ideal scenario will provide a result where $\alpha = \beta$. An overloaded scenario will give a result whereby $\alpha > \beta$. We also need to factor in another component, the time (t). If we are going to limit the execution rate to reduce the amount of lost jobs, we need more time to execute them (T). To be able to compute and get measurements of this issue we are going to count β for some t, where ($0<t<=$T). All of this information gives us a 3D plot, which we are going to analyze later.

3.1.1 Workload

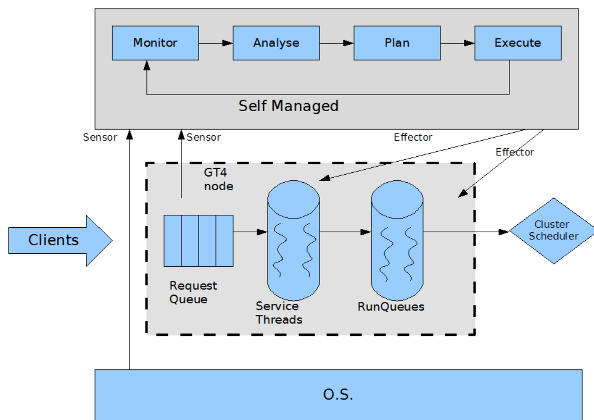


Figure 5. Prototype diagram

The workload can be divided in various different parts. First we try to submit (all at the same time) between 10 to 100 jobs (this is done several times using increments of 10 [10,20,30,40...90,100]). We can represent the number of jobs finished as $F^{10}...F^{100}$. As we said, we will also get the number of jobs finished when we send it to either the prototype or the original middleware with a run time of between 10 to 100 seconds (again, in increments of 10 seconds). This is going to be represented by F^{10}_{20} (number of finished jobs when 10 are submitted and 20 seconds of time is provided on the system). To provide us with all the data range, we execute all the workloads, modifying the number of jobs submitted and the number of seconds given for the test (we include some points where we consider we need more detail).

With this workload we can get some useful values. Firstly, we have the total number of submitted jobs (S = sum of submitted jobs for every time) and finally the finished jobs ($F = \sum_{x,y=10}^{100} F^x_y$). In our case S is equal to 12605 ($100 * 10 + 90 * 10 + 80 * 10... +$ points) and F depends on the result of the test.

3.2 Prototype

Our prototype is based on the ideas shown in [6], and following that architecture we developed a self-managed node for GT 4.0.1. Although we have only made a very simple self-managed system, it's enough to demonstrate the importance of such types of systems in grid environments. The architecture details are explained in the following sections.

3.2.1 Small brief

When we talk about a self-managed grid system (or more exactly a self-managed grid node), we need to consider some of the issues that we found while studying the behavior of grid nodes in an overloaded environment [14]. We concluded that we need a self-managed system to avoid "self-destructive" behaviors of the node in such situations. Therefore, we developed a simple system that is able to detect those situations and adapt itself to them before crashing. The self-managed node simply delivers good management of a system's resources.

The difference between the new and the old node is what we call the autonomic manager, which is in charge of controlling the node's behavior using sensors and effectors (explained in detail in the following sections). With the autonomic manager, we have a node that can manage itself to waste less resources. We see in Figure 5 how the prototype works.

3.2.2 Self-managed Architecture

The architecture of the new node is basically the same as before but with an additional top level manager. We can divide the software into 4 different parts: General Manager, Autonomic Manager, Touchpoints and finally Managed Resources

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

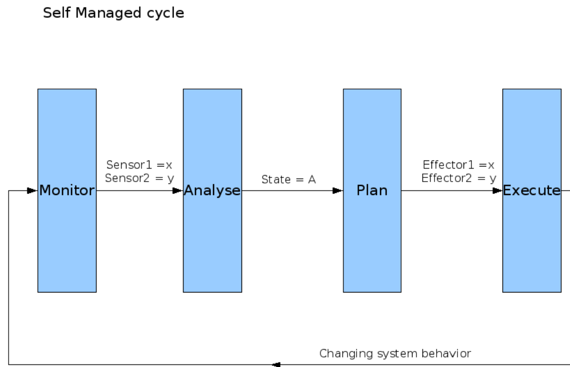


Figure 6. Autonomic system structure

General Manager This decides what type of policy has to be applied to construct the plan (for the Autonomic Manager). At the moment, this is hardcoded, since we only have one policy; which is to lose the minimum amount of jobs as possible.

Autonomic Manager The manager is the "core" of the autonomic system (Figure 6). This layer controls the monitoring, analyzing, planning and execution of tasks. The monitoring phase is done through the touchpoints that link the Autonomic manager with the manager resources. The analyze phase determines which state the system is in at any one time and once we know where we are, we can get a plan. This follows the policy determined by the general manager, and will guide the system to the desired state. The last phase executes the plan, and is done by using the touchpoints.

System states By comparing the values from the sensors, we can identify what state the system is in. In the actual implementation, these sensors are the number of ServiceThreads that Globus uses to process client requests and the CPU load. As the states are loaded from xml, we can change the number of the states and the sensors' values before Globus starts. In the analyze phase, the Autonomic Manager compares the values obtained in the monitor phase with the possible states to get the current one. Now the system is ready to jump to the next phase (plan).

Planning Once we know where we are, we have to know where we want to go; the plan phase is the one that will do that. The way to do this depends on the policy, which is determined by the General Manager. The different policies are loaded at the start and the decisions will be taken in a similar way that the Autonomic Manager does. Basically the General Manager and the Autonomic Manager have the same life cycle but at different levels. The plan basically indicates *what to do* with regard to the state we are in, so the system is led to the desired state. As we change the policy, the *what to do* will change and the system will be driven to different states. In the actual implementation of the self-managed container, as we only have only one policy, we didn't implement a General Manager.

Touchpoints The touchpoints are the interfaces which enable us to consult and modify the sensors and effectors. Going through that interface, we get an extra abstraction level. Through the get/set operations we can calculate and change the system state.

Managed Resources The managed resources, are those resources (OS and Globus in our case) that the self-managed system controls. The Autonomic manager reaches them using the touchpoints. In our case, we have two Managed Resources; one for the server and another for the OS. In each one we define the sensors (with which we can check the system's status) and the effectors (which let us modify the behavior of that resource). The simple version of self-managed GT4 implements 1 sensor and 1 effector for each resource. The sensors are the number of ServiceThreads accepting client requests and the CPU load. With those variables we can determine the state of the system.

The effectors are the number of ServiceThreads and the number of jobs that Globus can execute on the OS at once (this is done by controlling Globus' fork scheduler).

3.2.3 Self-managed Lifecycle

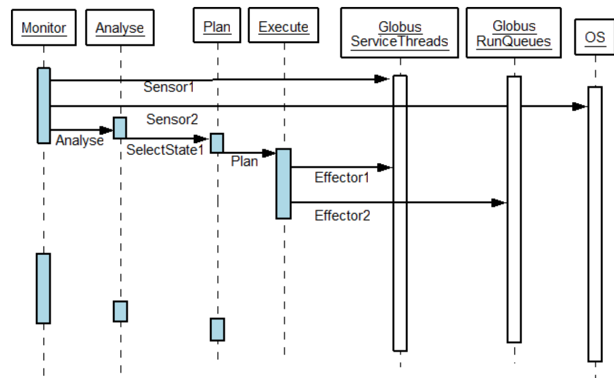


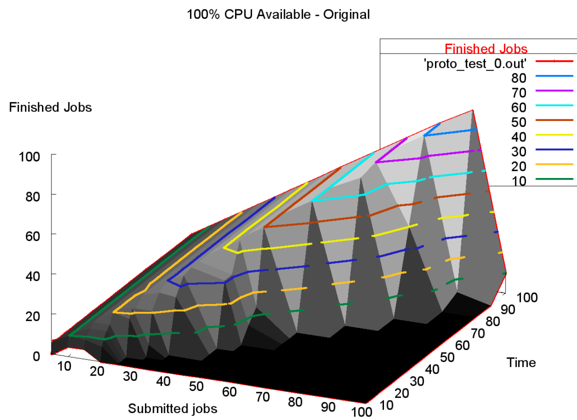
Figure 7. Sequence diagram for prototype

When Globus starts, the Autonomic Manager loads the knowledge (the different states of the system based on the sensor values, and the policy for constructing the plan). Once the startup is completed, every x seconds (5 by default) the manager gets the sensor values of the manager resources and analyzes them. The result from that analysis

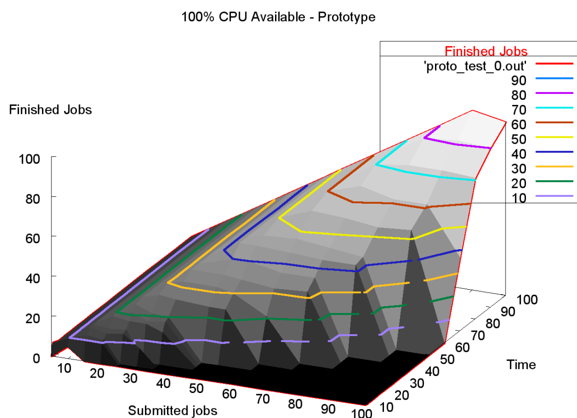
Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

is the actual state of the system. Then we have to make a plan according to the current policy(s) and once we have that plan, the manager executes it. This cycle is performed until the system goes down, and the plan is changed every time depending on the systems state. A sequence diagram can be found in Figure 7

3.3 Analyzing the Standard Middleware



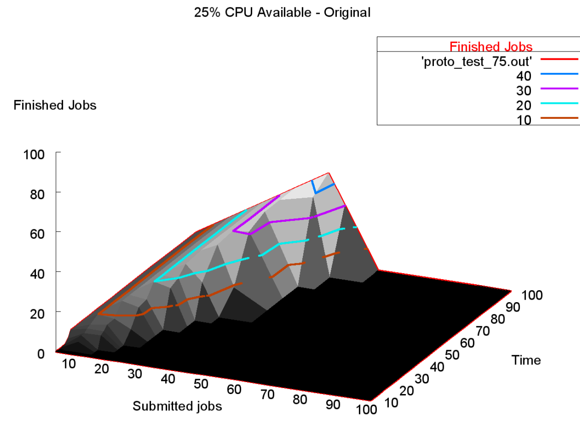
(a) Standard



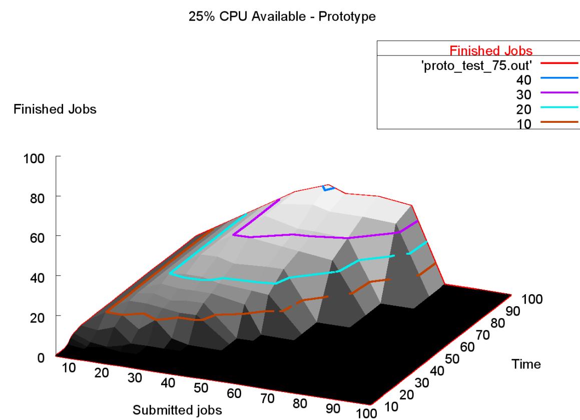
(b) Prototype

Figure 8. Finished jobs using 4 CPUs on the original middleware and with our prototype

With the help of eDMF and Paraver we were able to extract knowledge [14] about what is happening inside a Grid Node. In Globus, massive simultaneous submission of jobs can produce a state where no jobs are accepted. When we executed the job generator on our test machine we obtained the results shown in Figure 8 (the plot from figure (a) is



(a) Standard



(b) Prototype

Figure 9. Finished jobs using only 1 CPU on the original middleware and with our prototype

from the unmodified middleware). These 3D-plots show the number of finished jobs on the Y-axis (height), while the X-axis shows the number of submitted jobs. Finally, the Z-axis shows the time (in seconds) of the test. We have also some contours marked to show heights with same value. In these plots, the higher the better and we don't want black zones since these zones show that the middleware hasn't been able to finish any jobs. To demonstrate, let's take a look at a particular point in Figure 8. When we have 30 jobs submitted and a time of 20 seconds, we finished 21 jobs (the contour point that represents 20 finished jobs is goes through this point). If we decrease the time, we get 0 finished jobs (black area) and if we start increasing the time we get more jobs finished. All 30 jobs are executed at point

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

# CPU	(F) total finished jobs Standard/Prototype		% finished jobs $\frac{F}{S}$ Standard/Prototype)	
1	1192	1692	9,45	13,43
2	2376	3410	18,85	27,05
3	3220	4429	25,56	35,13
4	3504	5004	27,79	39,69

Table 2. Number of Jobs finished and the completion ratio with both the original and the prototype

(30,30).

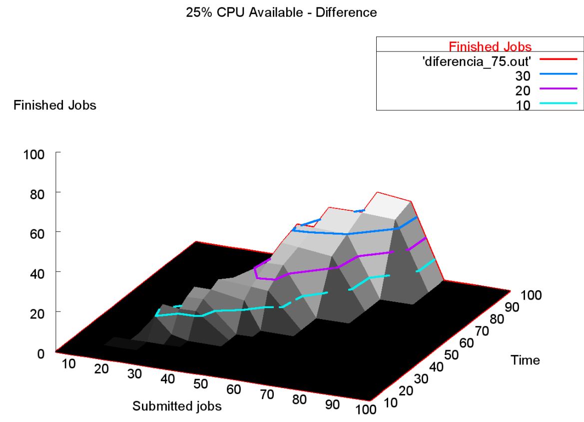
As our test machine has 4 CPUs, we can repeat the test using 1 (Figure 9), 2 or 3 CPUs (Table 2) to provide us with information about how the system works when the resources are reduced. Table 2 has information about the total number of jobs finished in all implementations and its % when compared to the total number of jobs submitted (12605).

Reduced resources is an important matter when we are using the middleware on a virtualized environment or in environments with heterogeneous workloads. The resources we are increasing or reducing for this test are the number of processors. We bind the middleware and all of it's processes to use one, two, three or four CPUs.

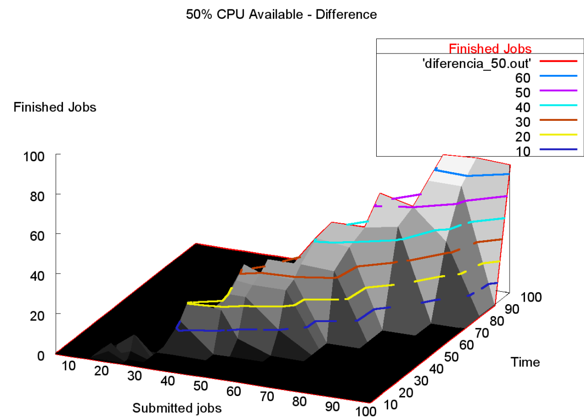
As we can see, limiting the resources produces big changes with the number of jobs that can be finished. Our main aim is reduce the black zones (zones where no jobs are finished). Table 2 shows that we are finishing 3504 jobs (27,79 %) with 4 CPUs and 1192 (9,45%) with 1 CPU. A peek at Figure 9 (a) shows how submitting more than 50 jobs leads to a state where no job is accepted.

3.4 Analyzing Prototype

We repeated the last test using our prototype and reduced the resources provided to the middleware in the same way we did with the original middleware. As we can see by comparing the two plots (top and bottom) in Figure 8, we were able to increase the number of jobs accepted using the same resources. Giving more time using 1, 2, and 3 CPUs in the modified middleware results in having the jobs finally accepted. The results correspond to part b) of Figures 9 and 8. Information from these tests is summarized in Table 2. With the prototype we increased the percentage finished to 39,69 % using 4 CPUs, as there were more than 5000 jobs finished. Analysing the 3D plot from Figure 9 we find that with the prototype we are able to process until there were 80 jobs submitted (original middleware only was able to handle up to 50). Finally in Figure 8 we can see how there are less black zones than in the original Middleware.



(a) 1 CPU



(b) 2 CPU

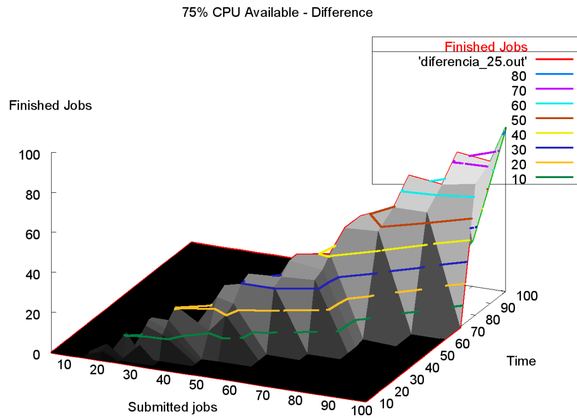
Figure 10. The difference in the number of finished jobs between our prototype and the original middleware (1 and 2 CPUs)

3.5 Results Comparison

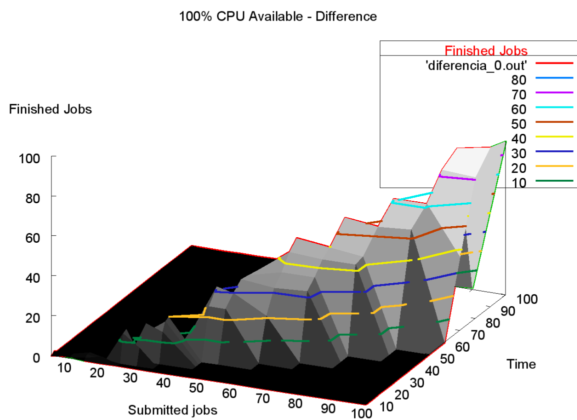
Comparing the results in Table 2, we can see how there is a large number of jobs that didn't execute either in the original or in the prototype. For the original middleware, this number is bigger because there is a timeout on the client side that gets affected by the Processor Sharing policy as shown in Section 2. Looking closer at the numbers, we can see how the number of jobs not executed when using the prototype is around 60% and with the original one, it's around 70%.

Table 3 has the % of finished jobs compared to the maximum number of finished jobs 5004 (achieved using 4 CPUs with the prototype). Finally, the last column gives the scala-

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY



(a) 3 CPU



(b) 4 CPU

Figure 11. The difference in the number of finished jobs between our prototype and the original middleware (3 and 4 CPUs)

bility of every implementation using the maximum number of finished jobs in each implementation.

As we can see in Table 3, 70% of the jobs are accepted with the original middleware using 4 CPUs, but when 3 CPUs are used this value only falls 6 points (64,34%). This could be a scalability problem because it happens on the two implementations and probably could be solved by changing the implementation of the acceptor code.

Finally Figures 10 and 11 visually show the difference in the number of jobs finished (the prototype's values minus the original middleware's values). With the lowest number of resources (1 CPU), the prototype is getting more jobs finished, where the original one can't get any.

# CPU	% $\frac{F}{max(F_{prototype})}$ Standard/Prototype		% $\frac{F}{max_{local}(F)}$ Standard/Prototype	
1	23,82	33,81	34,00	33,81
2	47,48	68,15	67,00	68,15
3	64,34	88,51	88,51	91,90
4	70,00	100,00	100,00	100,00

Table 3. Comparison of original, prototype and scalability (last column)

4 Conclusions

We showed in this paper how introducing a self-managed layer can provide more return. One important thing to take care of in this kind of work is the question of how to check if the prototype is doing something useful. If our aim is to decrease the number of rejected jobs (or increase the number of finished jobs), we need to be consistent and, as we typically have more jobs to execute, give more time to give a chance to the prototype. We cannot test the original implementation for 300 seconds, and then test the prototype for the same amount of time, we need to give extra time while jobs are still alive. To solve this situation we introduced 3D plots using time; with this data we can obtain useful information. Firstly, we can tune the initial data of the prototype using the behavior of the original middleware, and finally we can check the expected behavior of the system. Introducing our prototype to the grid middleware we were able to obtain some important improvements over the original one; we lose less jobs with the same resources. However this also means that we can't increase the throughput of the system in some cases, because we have more work to do.

Although this prototype works well for flash crowds and similar submission scenarios as can be seen in Table 2 (increasing the raw number of jobs finished in 4243 jobs), we need to take care with the queue sizes. In the future, some kind of admission control policy should be implemented and activated when needed. Using the sensors and system states implemented here, we can successfully work in heterogeneous environments where we are sharing resources with other tasks. As we said in Section 3.5, the unfitness of get more jobs finished could be a scalability problem because it happens on the two implementations and should be solved by changing the implementation of the acceptor code.

To finish, it's important to introduce a more intelligent mechanism to this prototype: Firstly, we should be able to predict the next [7] steps based on analysis of the arrival rate. Finally, we need to remove the predefined states and generate them using simulation or learning techniques (like

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

genetic algorithms). In our work, we are studying how to fill up the gaps using simulation. Some work has been done in simulating similar environments in [12] where we simulated admission control for Tomcat with great speed and success.

Acknowledgment

This work is supported by the Ministry of Science and Technology of Spain and the European Union under contract TIN2004-07739-C02-01 and the European Union under contract 034286 (SORMA). Thanks to Kevin Hogan for his help.

References

[1] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian. Resource management in the autonomic service-oriented architecture. In *ICAC*, pages 84–92, 2006.

[2] M. Bennani and D. Menasce. Resource Allocation for Autonomic Data Centers using Analytic Performance Models. *ICAC 2005. Proceedings. Second International Conference on Autonomic Computing, 2005*, pages 229–240, 2005.

[3] G. Bolch, S. Greiner, H. D. Meer, and K. S. Trivedi. Queueing networks and markov chains. *Modelling and Performance Evaluation with Computer Science Applications. Wiley, New York*, 1998.

[4] R. Buyya, D. Abramson, and S. Venugopal. The grid economy. *Proceedings of the IEEE*, 93(3):698–714, 2005.

[5] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Session-based adaptative overload control for secure dynamic web application. *ICPP-05, Oslo, Norway*, June 14-17 2005.

[6] IBM. An architectural blueprint for autonomic computing. `http://www-128.ibm.com/developerworks/autonomic/`.

[7] A. Iosup, C. Dumitrescu, D. Epema, H. Li, and L. Wolters. How are real grids used? the analysis of four grid traces and its implications. In *7th IEEE/ACM International Conference on Grid Computing (Grid2006)*, 2006.

[8] 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.

[9] D. Menasce and M. Bennani. On the Use of Performance Models to Design Self-Managing Computer Systems. *Proc. 2003 Computer Measurement Group Conference*, pages 7–12, 2003.

[10] D. Menasce, M. Bennani, and H. Ruan. On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems. *Lecture notes in computer science 3460*, pages 128–142, 2005.

[11] D. A. Menascé. Virtualization: Concepts, applications, and performance modeling. In *Int. CMG Conference*, pages 407–414. Computer Measurement Group, 2005.

[12] R. Nou, J. Guitart, D. Carrera, and J. Torres. Experiences with simulations - a light and fast model for secure web applications. In *12th International Conference on Parallel and Distributed Systems (ICPADS 2006)*, pages 177–186, 2006.

[13] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Caubet, J. Labarta, and J. Torres. Monitoring and analysis framework for java middlewares. *Research Report UPC-DAC-RR-2006-33*, 2006.

[14] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Caubet, J. Labarta, and J. Torres. Monitoring and analysing a grid middleware node. *GRID2006*, September 2006.

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

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

Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'07)
0-7695-2809-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY