

SHARING RESOURCES DYNAMICALLY IN SMP PLATFORMS: SELF-ADAPTIVE TOMCAT & GRID WORKLOADS

Keywords: Autonomic Computing, resource provisioning, heterogeneous workloads.

Abstract: We introduce a novel approach that allows heterogeneous applications run together on a shared hosting platform, dynamically sharing the platform's resources. The proposed approach has been validated by a proof-of-concept prototype which uses a global processor manager to distribute the platform's processors among two (or more) heterogeneous applications, i.e. a Tomcat application server and a Globus grid middleware. Our evaluation demonstrates the benefit of including bidirectional communication between applications and the OS for efficiently managing the resources and preventing the degradation of an applications performance, especially when the hosting platform is fully overloaded. For the sake of simplicity, we have modified the applications so that they communicate with the resource manager, although other techniques can be applied to avoid these modifications. Running different applications in a shared platform and being able to assign priorities between them provides important benefits.

1 Introduction

The consolidation of distributed and grid computing has been accompanied with the appearance of new computing models oriented to these environments. One of them is the utility computing model, in which applications run on hosting platforms that rent their resources to them. Application owners pay for platform resources, and in return, the application is provided with guarantees of resource availability and quality of service (QoS), which can be expressed in the form of a service level agreement (SLA). The hosting platform is responsible for providing sufficient resources to each application to meet its workload, or at least to satisfy the agreed QoS.

These hosting platforms must be able to provide resources to a heterogeneous set of applications, which range from web applications (e.g. an application server attending a transactional workload) to traditional scientific computations in the grid community. The traditional approach used by hosting platforms for provision of resources to heterogeneous applications is to consider a separate set of the cluster nodes for each application (dedicated model) (Appleby et al., 2001). In this model, resource allocation is performed with the granularity of a full cluster node and the provisioning technique must determine how many nodes to allocate to each application. However, economic reasons of space, power, cooling and cost can encourage the use of the shared model (Chandra et al., 2003a), in which node resources can be shared among multiple applications and the provisioning technique needs to determine how to partition resources on each node among competing applications.

We introduce a novel approach to allow heterogeneous applications to run together in a shared hosting platform, dynamically sharing the platform's resources (we focus on CPUs in this proof-of-concept prototype) while maintaining good performance. This paper extends the work performed in (Guitart et al., 2006). The previous paper proposed a global strategy for preventing the overloading of web applications and efficiently utilizing a platform's resources in a shared hosting platform running homogeneous web applications. The proposed strategy exploits the benefits of dynamically reallocating resources among hosted applications based on the variations in their workloads. These benefits have been described in recent studies (Appleby et al., 2001; Chandra et al., 2003a; Chandra et al., 2003b). The goal is to meet the applications requirements on demand and adapt to their changing resource needs. This requires an accurate collaboration between dynamic resource provisioning and admission control mechanisms.

The proposed approach has been validated using a proof-of-concept prototype built from the environment presented in (Guitart et al., 2006). The prototype includes a global cpu manager (ECM) that distributes the processors between the applications in the hosting platform, and two heterogeneous applications, e.g. a Tomcat application server and a Globus Grid middleware. We also have extended these applications with self-management capabilities, so that they are able to prevent performance degradation by adapting their behavior to the allocated resources that ECM has selected for them.

The paper is structured as follows: Section 2 shows our prototype and how it works. Section 3 describes how we have adapted the applications to be-

come self-managed. Section 4 describes the experimental environment used in our evaluation. Section 5 evaluates the results we obtained. Section 6 explores some work done in the area of dynamic resource provisioning for variable workloads. Finally, section 7 presents our conclusions and future work.

2 Resource provisioning strategy

In this section, we present a summary of the basic guidelines of our resource provisioning strategy. Our proposal is based on a global processor manager, called eDragon CPU Manager (ECM), responsible for periodically (configurable) distributing the available processors among the different applications running in a hosting platform. Further details can be found in (Guitart et al., 2006). We manage only processors because they are the more limited resource in the scenario we are studying. The ECM cooperates with the applications to efficiently manage the processors and prevent applications getting overloaded using bi-directional communication. On one side, the applications periodically request from the ECM the number of processors needed to handle their incoming load while avoiding degradation in the QoS. We define the number of processors requested by an application i as R_i . On the other side, the ECM can be requested at any time by the applications to inform them about their processor assignments. We define the number of processors allocated to application i as A_i . With this information, the applications can adapt their behavior to the allocated processors, avoiding in this way the degradation of their QoS. Figure 1 shows a diagram describing our resource provisioning strategy.

2.1 eDragon CPU Manager

The eDragon CPU Manager (ECM) is responsible for the distribution of processors among applications in the hosting platform. The ECM is implemented as a user-level process that wakes up periodically at a fixed time quantum, defined as k_{ECM} , examines the current requests of the applications and distributes processors according to a scheduling policy. With this configuration, direct modification of the native kernel is not required to show the usefulness of the proposed environment.

Traditionally, resource allocation policies have considered conventional performance metrics such as response time, throughput and availability. However, the metrics that are of utmost importance to the management of an e-commerce site are revenue and profits and should be incorporated when designing poli-

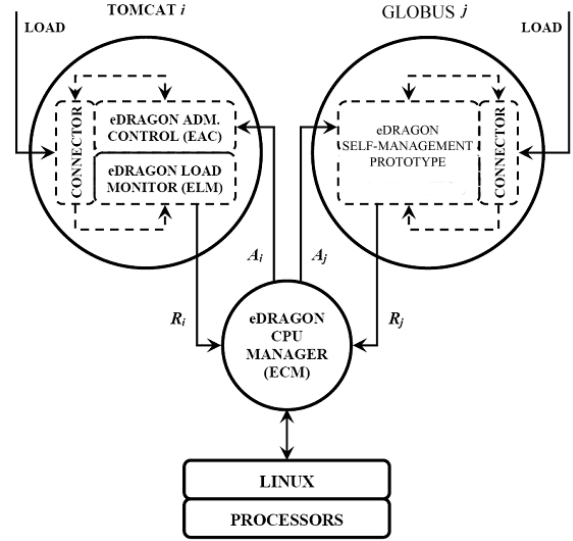


Figure 1: Prototype structure with ECM and two applications

cies (Cherkasova and Phaal, 2002). For this reason, the ECM can implement policies considering conventional performance metrics as well as incorporating e-business indicators, also policies like the ones in Section 6 can also be incorporated to the ECM. Our sample policy includes priority classes. The priority class P_i indicates a customer domain's priority in relation to other customer domains. It is expected that high priority customers will receive preferential service respect low priority customers. In our policy, at every sampling interval k_{ECM} , each application i receives a number of processors ($A_i(k_{ECM})$) that is proportional to its request ($R_i(k_{ECM})$) pondered depending on the application's priority class (P_i) and the number of processors in the hosting platform ($NCpus$), and inversely proportional to the total workload of the system ($\sum P_j * R_j(k_{ECM})$), expressed as the sum of requests of all applications in the system. The complete algorithm can be found at Algorithm 2.1.

The scheduling policy should also allow us to achieve the highest resource utilization in the hosting platform. Our proposal to accomplish this with the ECM is based on sharing processors among the applications under certain conditions (minimizing the impact on performance isolation). The current policy will decide to share a processor from application i to application j if the processor distribution policy has already assigned to application i all the processors it requested and the number of processors assigned to application j is lower than the number it requested. The current ECM implementation only con-

Algorithm**2.1: RESOURCE PROVISIONING()**

comment: Executed at every k_{ECM}
 $resourcesUsed \leftarrow 0$
for each $A_i, R_i \in Application$
 do $\begin{cases} A_i \leftarrow R_i \\ resourcesUsed \leftarrow R_i + resourcesUsed \end{cases}$

 if $resourcesUsed > resourcesAvailable$
 then $A_i \leftarrow Round \left[\frac{P_i * R_i * NCpus}{\sum_{j=1}^N P_j * R_j} \right]$
 EXECUTEASSIGNATION(k_{ECM})

siders sharing a processor between two applications with 0.5 processor granularity but could be lower.

The ECM not only decides how many processors to assign to each application, but also which processors to assign to each application. In order to accomplish this, the ECM configures the CPU affinity mask of each application (using the Linux *sched_setaffinity* function) so that the processors allocations to the different applications do not overlap (except if one processor is shared), in this way minimizing the performance interference among applications. For the applications (Tomcat and Globus) we are analysing, performance isolation for the CPU is almost equivalent to performance isolation for the whole system because these applications are CPU-intensive.

2.2 Applications/ECM Communication

The communication between the ECM and the applications is implemented using a shared memory region. The shared information includes the number of processors on which each application wants to run at any moment (R_i and R_j in Figure 1) and the number of processors currently allocated by the ECM to each application (A_i and A_j in Figure 1). Using JNI (Chen and Mohapatra, 2003) we can communicate with ECM in order to request (R_i) or consult the number of processors allocated.

3 Self-managed Applications

For the purpose of this paper, we present a managed middleware prototype that allows the execution of heterogeneous applications in the same host-

ing platform. We show how the communication between applications and OS layer can provide great improvements in performance terms. For this proof-of-concept we are considering web and grid applications. For simplicity in the prototype, we are using Tomcat for the web workload and the Globus platform for the grid workload; all of them are well known platforms and widely used. Next we will present the components that we have considered in our prototype. In this prototype we are modifying the applications in order to communicate with ECM, but we could use other mechanism to avoid these modification (i.e. use a proxy)

3.1 Self-managed Tomcat

In this section, we briefly describe the Tomcat adaptation to become self-managed. Further details can be found in (Guitart et al., 2006).

3.1.1 ELM

In order to be self-managed, Tomcat must be able to determine the number of processors it needs (R_i) to handle its incoming workload avoiding QoS degradation. Our proposal achieves this capability by adding a new component, called eDragon Load Monitor (ELM), within the server (see Figure 1). At every sampling interval, the ELM calculates the number of processors that the application will need to handle its incoming workload during that interval and informs to the ECM. For performing this calculation, the ELM continuously monitors the incoming secure connections to the server and performs on-line measurements distinguishing new SSL connections from resumed SSL connections. The differentiation of resumed SSL connections from new SSL connections is performed with the mechanism described in (Guitart et al., 2005).

3.1.2 EAC

In order to become fully self-managed, Tomcat must also adapt its behavior according to the resources provided by the ECM in order to prevent overload and maintain in this way the QoS of admitted requests. This is accomplished by adding a new component, called eDragon Admission Control (EAC), within the server (see Figure 1)

The EAC implements an admission control mechanism that prioritizes the acceptance of resumed SSL connections and only accepts the maximum number of new SSL connections that do not overload the server (they can be served with the available computational capacity without degrading the QoS) depend-

ing on the number of processors provided by the ECM (A_i). The rest of the new SSL connections arriving at the server will be refused.

3.2 Self-managed Globus

In this section we present an overview of the implementation we have made to add self-managed capabilities to a globus node. All the algorithms and the decisions of the Manager are driven by a global policy that enables the node to avoid overloaded states as seen on (Nou et al., 2007b). Further details on the implementation and architecture can be found in (Nou et al., 2007a), for these proof-of-concept we changed only the Managed Resources module.

3.2.1 Architecture Overview

Self-manager capabilities have been added using the architecture proposed by IBM on (IBM-Corporation, 2004). We can divide the architecture in two groups, the resources, which will be self-managed, and the intelligent part, the autonomic manager. In our prototype we have two managed resources; one for the server and another for the ECM (to report the number of CPUs needed). In each one we define sensors (to check the system's status) and the effectors (to modify the behavior of that resource). The sensors are the number of client requests waiting in the accept queue, and the CPU load of our assigned CPUs from ECM. With those variables we can determine the state of the system. Effectors are the number of ServiceThreads and the number of jobs that Globus can process (execution is done in other node) on the OS at once.

Once the startup is completed, the manager gets the CPUs assigned by ECM and the rest of sensor values every x seconds (the duration of the self-managed life cycle depends on many factors). Then we have to analyze this data, obtain the actual state and make a plan according to the current policy (i.e. accept the maximum number of jobs). When we have that plan, the manager executes it. At this point is when the manager change the effectors' values and notifies ECM the CPU needs of the globus node.

Although we have used this policy, we could change it to do, for example, some admission control like in Tomcat.

4 Experimental Environment

We have Tomcat v5.0.19 (Amza et al., 2002) and a Globus GT 4.0.1 (Sotomayor and Childers, 2005) in

the same node. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and to be a quality production servlet container too. Tomcat can work as a standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). In this paper, we use Tomcat as a standalone server. We have configured Tomcat by setting the maximum number of HttpProcessors to 100 and the connection persistence timeout to 10 seconds. The experimental environment also includes a deployment of version 1.4.2 of the RUBiS (Rice University Bidding System) (Coarfa et al., 2002) benchmark servlets on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httpperf (Crovella et al., 1999). The Tomcat instance has a variable input load throughout the run time, which is shown in the top subfigure of Figure 2 which displays the number of new clients per second that hit the server as a function of the time. Input load distribution has been chosen in order to represent the different processor requirement combinations when running with the Grid workload in the hosting platform.

The Globus server is a standard *de facto* of Grid middleware. We didn't make any modifications of its parameters (number of ServiceThreads or number of Runqueues) on the standard tests (with ECM they can be dynamically modified). The Globus workload generator is sending jobs that overload or stress the management code for a job. We assume that the job will be executed on another node or cluster. From the Globus workload generator we generate and submit jobs with an increasing throughput and try to execute them on the nodes of the cluster (simulated, so CPU requirements are only the ones to prepare the job). We then measure the output throughput. This gives a wide range of situations that are summarized in Figure 2. In our case, we selected different submission levels for Globus and several different levels of arrival rates for Tomcat to give a wide view of configurations and situations to show the benefits of our approach. The hosting platform is a 4-way Intel XEON 1.4 GHz with 2 GB RAM. The workloads generators are running on other machines in the same cluster. As a side note running the two applications at once is not necessary, ECM can manage new applications (and manage machine resources as well) whatever time they come into the system (or they left).

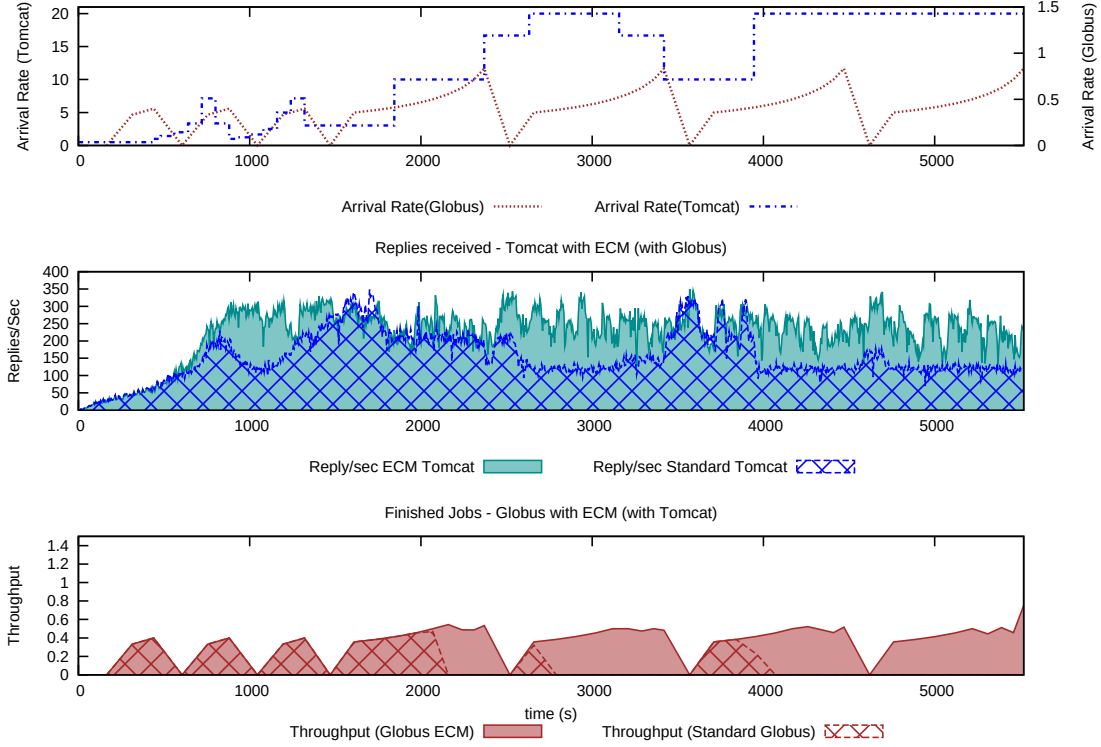


Figure 2: From top to bottom; workload of Tomcat and Globus, replies/sec of Standard Tomcat compared with the replies/sec of Tomcat using ECM, throughput in Standard Globus compared with throughput using ECM with Globus. The two applications are running at once using ECM facilities

5 Evaluation

Our evaluation will show the benefits of our proposal for managing the resources efficiently and preventing server overload on a 4-way multiprocessor Linux hosting platform. The CPU requirements to prepare a globus job are ten times the requirements to process a Tomcat request (with SSL handshake) or 100 times a Tomcat request without SSL handshake.

5.1 Standard Tomcat and Standard Globus, without ECM

In Figure 2, we can see in grid style how the system evolves (bottom two plots) when we are submitting different workloads (top plot) to Tomcat and Globus. We have divided the test in order to check different CPU requirement scenarios in the two middleware. Globus is submitting the jobs with the increasing workload explained in Section 4, while Tomcat is generating several load levels from overloaded to non-overloaded arrival rates. As we can see in the top plot, we can divide the test into two parts; one

workload with low load and another workload where the system is under a heavy load. If we take a closer view of Tomcat we can see that when the server is overloaded the reply/rate falls (i.e. around 4000 seconds in grid style). We can find another zone in the Tomcat plot of similar behaviour: in the 1500 seconds point the throughput obtained from Tomcat is very high, however when we look at the 3000 seconds zone we can see that the throughput obtained is lower with an higher input workload. Switching to the Globus plot there are a lot of zones where no jobs are finished (2800-3600 or 4000-5000). Globus is getting a very high workload, but as we will show in the next subsection we can increase its performance using ECM. The server needs to share the resources between the two applications, but such kinds of applications don't know in which environment and with what kinds of resources they are being executed. They are competing for a set of limited resources and getting worse performance than if they were divided onto two machines with the resources divided in half. Giving and getting information about the resources that the application consumes and that the system have available

Table 1: Summary of data obtained; We can find the total number of requests finished (in Tomcat) and the total throughput obtained in Globus when we are using ECM and when we are using the original environment (without ECM). Finally we consider a scenario without priorities assigned and another with priorities.

		Orig	ECM	% improvement
Without priorities	Globus	397 jobs finished	1162 jobs finished	292%
	Tomcat	129886.3 replies	153710.4 replies	118%
With priorities	Globus	397 jobs finished	1043 jobs finished	262%
	Tomcat	129886.3 replies	181230.4 replies	139%

for it should be necessary in order to avoid this situations.

5.2 Adaptative Middleware with ECM

If we repeat the last test with ECM (without priorities), we obtain the solid style results in Figure 2. We can see how the system overall is working better. And the most important thing is that we didn't get the low levels of replies/sec on Tomcat and the low number of finished jobs on Globus that we obtained in the previous subsection.

In some zones we can see how Tomcat is working better than before, as long as ECM receives a request of processing power from Tomcat, ECM tells Tomcat how many CPUs has assigned. Tomcat is able to overcome these situations and start its admission control to stabilize itself at the desired level.

It is the case of the zones near 1000 seconds and the ranges from 2200-3500 and 4000-5500 where the system without ECM is obtaining lower performance than with ECM. In this zones as long as Tomcat (and Globus) knows how many resources they have available can adapt his behaviour to the new scenario. We can notice this looking at the globus side, where we are getting more throughput than before also. Getting into these situations, on 1000 seconds zone Globus is working at the same level than before but Tomcat as it knows how many resources it has available can adapt his load to the new scenario. The reverse situations happens on 2500 seconds zone, Globus is improving its performance. When the two applications are overloaded a communication with the OS to know the available resources can provide an improvement over the two applications as we can see after 4000 seconds zone.

In some cases we may want or may need to increase the performance of an application. Applications are able to be prioritized to overcome these situations; maybe at some points we are more interested in Tomcat's revenue than in Globus'. We can do this using ECM with priorities depending on the revenue of the application. Giving more priority to Tomcat than Globus, produces a lose of performance

on Globus to gain it on Tomcat. As we can see on the workload plot, we are requesting more resources than the ones we have, so a priority policy can modify the fairness of the resource assignment, giving less resources to Globus (and more to Tomcat). As long as we prioritize an application over the other we are getting (if there aren't enough) more resources for the more prioritized application. When the applications are in the same priority category inside ECM, they are sharing resources without any preference between applications.

A summary of the data (in terms of reply/sec for Tomcat and total throughput for Globus) of some experiments obtained is shown in Table 1. We can see how under an overloaded (or high load) scenario ECM is producing benefits. If in some cases we are interested in give more revenue to Tomcat (a more important client, for example) we can do it with ECM. We can see in this case how the benefits are more biased to Tomcat (ECM provides a high range of configurations, it's just an example of priority assignation. In the case of low loads, the benefits of ECM are low compared to a shared environment (but we can have CPU isolation).

In an entry node with a secure connection scenario, like the one we are testing, it's crucial to provide more fairness to the several middleware that share the resources on the node to get better results as a whole.

6 Related Work

Recent studies (Andrzejak et al., 2002; Chandra et al., 2003b; Chandra and Shenoy, 2003) have reported the considerable benefit of dynamically adjusting resource allocations to handle variable workloads. This premise has motivated the proposal of several techniques to dynamically provision resources to applications in on-demand hosting platforms. Depending on the mechanism used to decide the resource allocations, these proposals can be classified into: control theoretic approaches with a feedback element (Abdelzaher et al., 2002), open-loop approaches

based on queuing models to achieve resource guarantees (Chandra et al., 2003a; Doyle et al., 2003; Liu et al., 2001) and observation-based approaches that use runtime measurements to compute the relationship between resources and a QoS goal (Pradhan et al., 2002). Control theory solutions require training the system at different operating points to determine the control parameters for a given workload. Queuing models are useful for steady state analysis but do not handle transients accurately. Observation-based approaches are most suited for handling varying workloads and non-linear behaviors. Resource management in a single machine has been covered in (Banga et al., 1999), where authors proposed to use resource containers as an operating system abstraction to embody a resource. In (Liu et al., 2005) authors propose the design of online feedback control algorithms to dynamically adjust entitlement values for a resource container on a server shared by multiple applications. The problem of provisioning resources in cluster architectures has been addressed in (Appleby et al., 2001; Ranjan et al., 2002) by allocating entire machines (dedicated model) and in (Chandra et al., 2003a; Pradhan et al., 2002; Uragonkar and Shenoy, 2004) by sharing node resources among multiple applications (shared model). Cataclysm (Sotomayor and Childers, 2005) performs overload control by bringing together admission control, adaptive service degradation and dynamic provisioning of platform resources, demonstrating that the most effective way to handle overloading must consider a combination of techniques. In this aspect, that work is similar to our proposal. There are also approaches (Menascé, 2005) that use virtualized environments and analytical methods to adjust the resources allocated to the virtualized systems. R-Opus (Cherkasova and Rolia, 2006) works on a different layer and scale of time. In our approach we focus on a single server machine which shares different applications and has a low time scale. Also, giving more processing power to an application, such as Tomcat (for example), will not directly produce better performance. The application needs to know how many resources it has available.

7 Conclusions & Future Work

In this paper we have presented a proof-of-concept prototype for demonstrating that bidirectional communication between applications and OS can provide that heterogeneous applications (running in overloaded conditions) can run together in a shared hosting platform and at the same time maintain their performance. Using a shared hosting platform reduces

important costs like space and power.

Our approach is based on implementing a global resource manager, responsible for periodically distributing the available processors between the applications following a determined policy. The resource manager can be configured to implement different policies, and consider traditional indicators (i.e. response time) as well as e-business indicators (i.e. customer's priority). In our proposal, the resource manager and the applications cooperate to manage the resources, in a manner totally transparent to the user, using bi-directional communication. On one side, the applications request from the resource manager the number of processors needed to handle their incoming load without QoS degradation. On the other side, the resource manager can be requested at any time by the applications to inform them about their processor assignments. With this information, applications can adapt their behavior to the allocated processors.

Our evaluation demonstrates the benefit of our approach for managing resources efficiently and for preventing degradation of an applications performance on shared hosting platforms. Although our implementation targets Tomcat and Globus, the proposed strategy can be applied with any other platform or application. Further improvements can be made on this proof-of-concept work: more fine grained assignments of CPU or a fairer Globus self-management objective.

Our future work considers the use of virtualization technologies in order to provide better performance isolation to the applications running in the hosting platform. In this approach, each application will run within a dedicated virtual machine, and the global resource manager will distribute the resources among these virtual machines.

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 Commission of the European Communities under IST contract 034286 (SORMA). Thanks to Mario Macias for his help. For additional information about the authors, visit the Barcelona eDragon Research Group web site (WebPage, 2007).

REFERENCES

- Abdelzaher, T., Shin, K., and Bhatti, N. (2002). Performance guarantees for web server end-

- systems: A control-theoretical approach. *IEEE TPDS*, 13(1):80–96.
- Amza, C., Cecchet, E., Chanda, A., Cox, A., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., and Zwaenepoel, W. (2002). Specification and implementation of dynamic web site benchmarks. *WWC-5, Austin, Texas, USA*.
- Andrzejak, A., Arlitt, M., and Rolia, J. (2002). Bounding the resource savings of utility computing models. *HPL-2002-339, HP Labs*.
- Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. (2001). Oceano :SLA-based management of a computing utility. *IM 2001, Seattle, Washington, USA*, pages 855–868.
- Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. *OSDI'99, New Orleans, Louisiana, USA*, pages 45–58.
- Chandra, A., Gong, W., and Shenoy, P. (2003a). Dynamic resource allocation for shared data centers using online measurements. *IWQoS 2003, Berkeley, California, USA*, pages 381–400.
- Chandra, A., Goyal, P., and Shenoy, P. (2003b). Quantifying the benefits of resource multiplexing in on-demand data centers. *Self-Manage 2003, San Diego, California, USA*.
- Chandra, A. and Shenoy, P. (2003). Effectiveness of dynamic resource allocation for handling internet flash crowds. *TR03-37, Department of Computer Science, University of Massachusetts, USA*.
- Chen, X. and Mohapatra, P. (2003). Overload control in QoS-aware web servers. *Computer Networks*, 42 (1):119–133.
- Cherkasova, L. and Phaal, P. (2002). Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51 (6):669–685.
- Cherkasova, L. and Rolia, J. (2006). R-Opus: A composite framework for application performability and qos in shared resource pools. In *DSN'06*, pages 526–535, Washington, DC, USA.
- Coarfa, C., Druschel, P., and Wallach, D. (2002). Performance analysis of TLS web servers. *NDSS'02, San Diego, California, USA*.
- Crovella, M., Frangioso, R., and Harchol-Balter, M. (1999). Connection scheduling in web servers. *USITS'99, Boulder, Colorado, USA*.
- Doyle, R., Chase, J., Asad, O., Jin, W., and Vahdat, A. (2003). Model-based resource provisioning in a web service utility. *USITS'03, Seattle, Washington, USA*.
- Guitart, J., Beltran, V., Carrera, D., Torres, J., and Ayguadè, E. (2005). Session-based adaptive overload control for secure dynamic web application. *ICPP-05, Oslo, Norway*.
- Guitart, J., Carrera, D., Beltran, V., Torres, J., and Ayguadè, E. (2006). Preventing secure web applications overload through dynamic resource provisioning and admission control. *UPC-DAC-RR-2006-37*.
- IBM-Corporation (2004). An architectural blueprint for autonomic computing. <http://www.ibm.com/autonomic>.
- Liu, X., Zhu, X., Singhal, S., and Arlitt, M. (2005). Adaptive entitlement control to resource containers on shared servers. *IM 2005, Nice, France*.
- Liu, Z., Squillante, M., and Wolf, J. (2001). On maximizing service-level-agreement profits. *EC 2001, Tampa, Florida, USA*, pages 213–223.
- Menascé, D. A. (2005). Virtualization: Concepts, applications, and performance modeling. *Int. CMG Conference, Orlando, Florida, USA*, pages 407–414.
- Nou, R., Julià, F., and Torres, J. (2007a). The need for self-managed access nodes in grid environments. *EASE 2007, Tucson, Arizona, USA*.
- Nou, R., Julià, F., and Torres, J. (2007b). Should the grid middleware look to self-managing capabilities? *ISADS 2007, Sedona, Arizona, USA*.
- Pradhan, P., Tewari, R., Sahu, S., Chandra, A., and Shenoy, P. (2002). An observation-based approach towards self-managing web servers. *IWQoS 2002, Miami Beach, Florida, USA*, pages 13–22.
- Ranjan, S., Rolia, J., Fu, H., and Knightly, E. (2002). Qos-driven server migration for internet data centers. *IWQoS 2002, Miami Beach, Florida, USA*, pages 3–12.
- Sotomayor, B. and Childers, L. (2005). *Globus Toolkit 4 : Programming Java Services*. Morgan Kaufmann.
- Uragonkar, B. and Shenoy, P. (2004). Cataclysm: Handling extreme overloads in internet services. *TR03-40, Department of Computer Science, University of Massachusetts, USA*.
- WebPage (2007). Barcelona eDragon Research Group. <http://research.ac.upc.edu/eDragon>.