Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 52 (2008) 1390–1409
Computer Networks
www.elsevier.com/locate/comnet

Dynamic CPU provisioning for self-managed secure web applications in SMP hosting platforms

Jordi Guitart *, David Carrera, Vicenç Beltran, Jordi Torres, Eduard Ayguadé

Barcelona Supercomputing Center (BSC), Computer Architecture Department, Technical University of Catalonia, C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034 Barcelona, Spain

Received 14 March 2007; received in revised form 3 December 2007; accepted 8 December 2007
Available online 1 February 2008

Responsible Editor: C. Westphal

Abstract

Overload control mechanisms such as admission control and connection differentiation have proven effective for preventing overload of application servers running secure web applications. However, achieving optimal results in overload prevention is only possible when some kind of resource management is considered in addition to these mechanisms.

In this paper we propose an overload control strategy for secure web applications that brings together dynamic provisioning of platform resources and admission control based on secure socket layer (SSL) connection differentiation. Dynamic provisioning enables additional resources to be allocated to an application on demand to handle workload increases, while the admission control mechanism avoids the server's performance degradation by dynamically limiting the number of new SSL connections accepted and preferentially serving resumed SSL connections (to maximize performance on session-based environments) while additional resources are being provisioned.

Our evaluation demonstrates the benefit of our proposal for efficiently managing the resources and preventing server overload on a 4-way multiprocessor Linux hosting platform, especially when the hosting platform is fully overloaded.

Keywords: Internet servers; Dynamic resource provisioning; Self-adaptation; Overload prevention; Admission control; Connection differentiation; Security

1. Introduction

In recent times, e-commerce applications have become commonplace in current web sites. In these applications, all the information that is confidential or has market value must be carefully protected when transmitted over the open Internet. Security between network nodes over the Internet is traditionally provided using HTTPS [1]. With HTTPS, which is based on using HTTP over SSL (secure socket layer [2]), you can perform mutual authentication of both the sender and receiver of messages and ensure message confidentiality. Although providing these security capabilities does not introduce

* Corresponding author. Tel.: +34 93 405 40 47; fax: +34 93 401 70 55.
E-mail addresses: jguitart@ac.upc.edu (J. Guitart), dcarrera@ac.upc.edu (D. Carrera), vbeltran@ac.upc.edu (V. Beltran), torres@ac.upc.edu (J. Torres), eduard@ac.upc.edu (E. Ayguadé).

1389-1286/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2007.12.009

a new degree of complexity in web applications structure, it increases remarkably the computation time needed to serve a connection, due to the use of cryptographic techniques, becoming a CPU-intensive workload.

At the same time, current sites are subject to enormous variations in demand, often in an unpredictable fashion, including flash crowds that cannot be easily processed. For this reason, the servers that host the sites can encounter situations with a large number of concurrent clients. Dealing with these situations and/or with workloads that demand high computational power (for instance secure workloads) can lead a server to overload (i.e. the volume of requests for content at a site temporarily exceeds the capacity for serving it and renders the site unusable). During overload conditions, the response times may grow to unacceptable levels, and exhaustion of resources may cause the server to behave erratically or even crash, causing denial of services. In e-commerce applications, which are heavily reliant on security, such server behavior could translate to sizable revenue loss.

Overload prevention is a critical goal so that a system can remain operational in the presence of overload even when the incoming request rate is several times greater than the system's capacity. At the same time it should be able to serve the maximum number of requests during such overload, while maintaining the response times at acceptable levels.

Additionally, in many web sites, especially in e-commerce, most of the applications are session-based. A session contains temporally and logically related request sequences from the same client. Session integrity is a critical metric in e-commerce. For an online retailer, the higher the number of sessions completed, the higher the amount of revenue that is likely to be generated. The same statement cannot be made about individual request completions. Sessions that are broken or delayed at some critical stages, like checkout and shipping, could mean loss of revenue to the web site. Sessions have distinguishable features from individual requests that complicate the overload control. For this reason, simple admission control techniques that work on a per request basis, such as limiting the number of threads in the server or suspending the listener thread during overload periods, may lead to a large number of broken or incomplete sessions when the system is overloaded (despite the fact that they can help to prevent server overload).

Taking into account these considerations, several techniques have been proposed to deal with overload, such as admission control, request scheduling, service differentiation, service degradation, and resource management. All of them have proven to be effective at preventing overload to a certain extend. For instance, in our previous work [3], we have demonstrated the benefit of using admission control based on SSL connection differentiation to prevent the overload of application servers running secure web applications. However, our following research indicates that some kind of dynamic resource management must be considered in addition to admission control for achieving optimal results in overload prevention. The reasoning for this is two fold.

Firstly, due to the variability of web workloads, it is desirable that web applications are able to increase their capacity dynamically by allocating more resources in order to deal with extreme overloads. Secondly, web applications typically 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 on resource availability and QoS. The hosting platform is responsible for providing sufficient resources to each application to meet its workload, or at least to satisfy the agreed QoS. Previous literature [4–6] has demonstrated that this is better accomplished by dynamically reallocating resources among hosted applications according to the variations in their workloads instead of statically overprovisioning resources. In this way, better resource utilization can be achieved and the system can react to unexpected workload increases.

According to this, in this paper, we propose a global overload control strategy for secure web applications that brings together admission control based on SSL connection differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads while avoiding the QoS degradation. We have implemented a global resource manager for a Linux hosting platform that distributes the available resources among the application servers running on it. These servers incorporate self-management facilities that adapt the accepted load to the assigned resources by using an admission control mechanism. All the process is transparent to the final user. Although our implementation targets the Tomcat [7] application server, the proposed overload control strategy can be applied on any other server.

The rest of the paper is organized as follows: Section 2 presents some experiments for motivating our work. Section 3 details the implementation of the different components of our overload control strategy. Section 4 describes the experimental environment used in our evaluation. Section 5 presents the evaluation results of our overload control strategy. Section 6 describes the related work and finally, Section 7 presents the conclusions of this paper.

2. Motivation and contributions

2.1. Motivating experiments

In this section, we will justify the convenience of integrating dynamic resource provisioning with admission control using two experiments, which consist of the execution of two application servers (*Tomcat 1* and *Tomcat 2*) running secure web applications for 90 minutes in a 4-way Linux hosting platform. In the first experiment, two processors are statically assigned to each server. In the second one, there is not any static allocation of processors. In this case, the default Linux scheduler is responsible for dynamically distributing the available processors between the two servers according to its scheduling policy. In both cases, the admission control mechanism described in [3] is used.

Fig. 1 shows the results obtained for these two experiments. Each application server has a variable input load over time, which is shown in the top part of the figure representing the number of new sessions per second initiated with the server as a function of the time (in seconds). The figure also shows the throughput achieved by each server as a function of the time when running with static processor allocation (middle) and with dynamic processor allocation (bottom), in a way that can be easily correlated with the input load.

These experiments confirm that dynamic resource provisioning allows exploiting the resources more efficiently. For example, as shown in Fig. 1 in the zone labeled *B* (between 1800 s and 2400 s), *Tomcat 1* achieves higher throughput with dynamic provisioning (bottom) rather than with static one (middle) (150 vs. 110 replies/s), while *Tomcat 2* achieves the same throughput in both cases. This occurs because when using static resource provisioning *Tomcat 1* is under-provisioned (it has less processors allocated than needed to handle its incoming workload) while

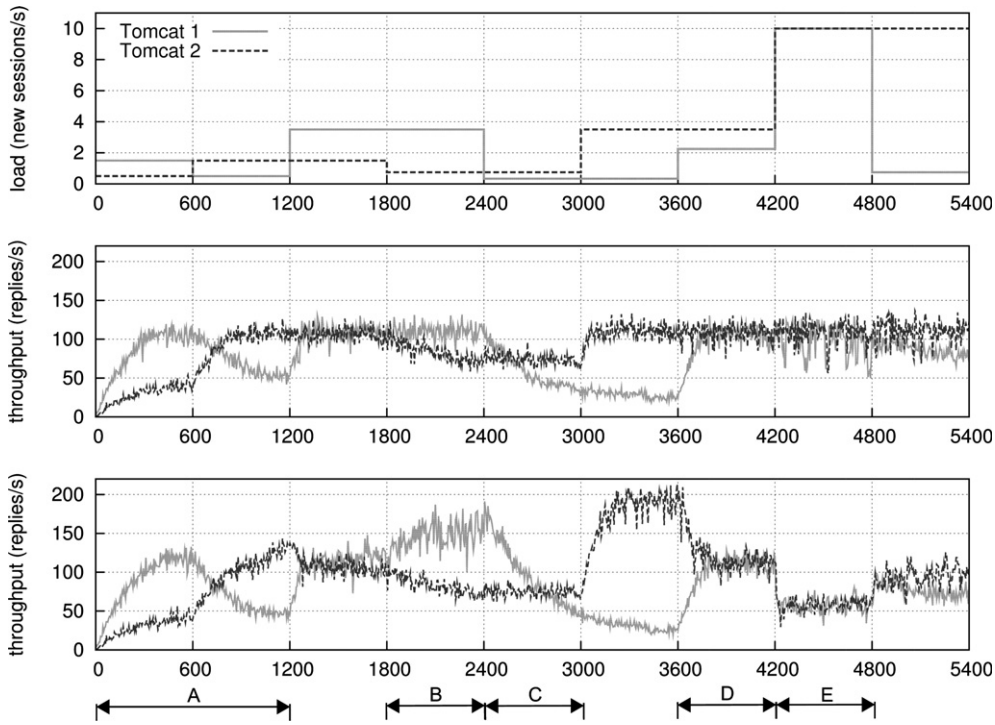


Fig. 1. Incoming workload (top) and achieved throughput of two Tomcat instances in a 4-way hosting platform with admission control and static resource provisioning (middle) and Linux default dynamic resource provisioning (bottom).

Tomcat 2 is over-provisioned (it has more processors allocated than needed to handle its incoming workload), denoting an inefficient processor distribution. On the other side, using dynamic resource provisioning allows the processors not used by *Tomcat 2* to be used by *Tomcat 1*.

As commented before, in these motivation experiments we use the admission control mechanism proposed in [3] to avoid the throughput degradation occurred when secure web applications are under-provisioned, as described in [8]. This mechanism limits the accepted load for each server so that it can be handled with the processors assigned to that server. When using static resource provisioning, the servers can easily determine the number of processors that they have allocated, because this is a fixed number (e.g. two in our experiment). However, when using dynamic resource provisioning, the servers cannot accurately apply the admission control mechanism because they do not have information about the number of processors assigned to them in a given moment (this number varies over time). In this case, the admission control mechanism becomes useless, neutralizing its benefit in preventing the throughput degradation occurred when secure web applications are under-provisioned. This effect can be appreciated in the zone labeled *E* (between 4200 s and 4800 s) of Fig. 1. In this zone, static provisioning (middle) allows both servers to achieve considerably higher throughput than dynamic provisioning (bottom) (110 vs. 55 replies/s).

From these experiments, we can conclude that, in order to manage the resources efficiently and prevent web applications overload in hosting platforms where applications compete for the available resources, we must design a global strategy involving an accurate collaboration between dynamic resource provisioning and admission control mechanisms.

2.2. Contributions

In this paper, we propose a novel global overload control strategy for secure web applications that brings together admission control based on SSL connection differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads avoiding the QoS degradation. Dynamic provisioning enables additional resources

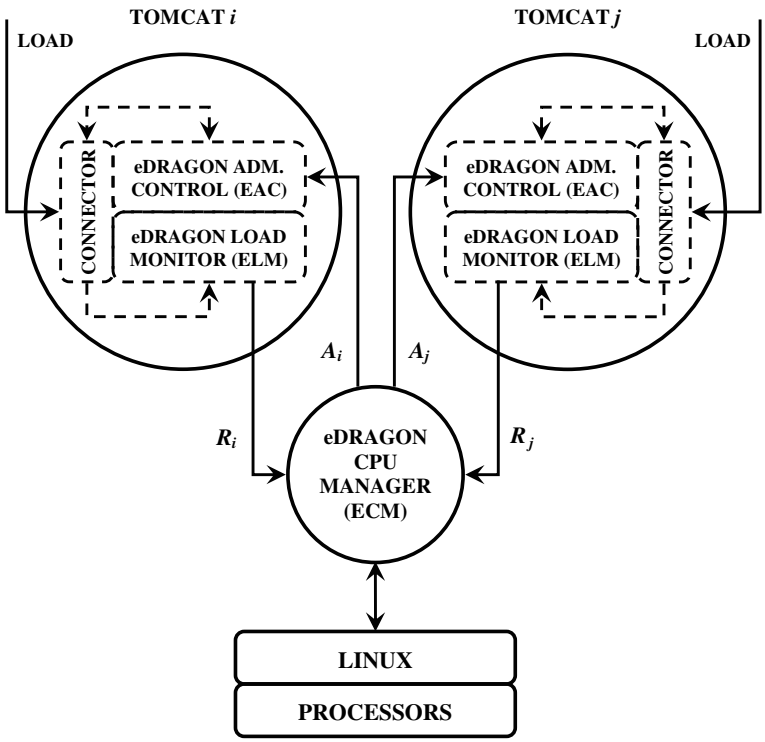


Fig. 2. Global overload control strategy.

to be allocated to an application on demand to handle workload increases, while the admission control mechanism maintains the QoS of admitted requests by turning away excess requests and preferentially serving preferred clients (to maximize the generated revenue) while additional resources are being provisioned.

Fig. 2 shows a diagram describing our overload control strategy. It is based on a global processor manager, called eDragon CPU Manager (ECM), responsible for periodically distributing the available resources (i.e. processors[1]) among the different applications running in a hosting platform applying a given policy (which can consider e-business indicators).

Our approach includes enabling self-management capabilities on the applications, allowing them to cooperate with the ECM to efficiently manage the processors and prevent the applications overloading. This cooperation includes 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 of the QoS. We define the number of processors requested by 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.

Given the number of processors assigned to them, the applications can apply an admission control mechanism to limit the number of admitted requests; accepting only those that can be served with the allocated processors without degrading their QoS. The admission control mechanism is based on SSL connection differentiation depending on if the SSL connection reuses an existing SSL connection on the server or not. Prioritizing resumed SSL connections maximizes the number of sessions successfully completed, allowing e-commerce sites based on SSL to increase the number of transactions completed, and thereby generating more revenue.

3. Overload control strategy

3.1. 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 periodically wakes up at a fixed time quantum, defined as k_{ECM} and with a value of 2 s in the current prototype, examines the current requests of the applications and distributes processors according to a scheduling policy. Using a user-level process avoids the direct modification of the native kernel in order to show the usefulness of the proposed environment.

The choice of a particular time quantum depends on the desired responsiveness from the system. We have chosen a small value in order to achieve a system that is able to react to unexpected workload changes in a very short time. Values in the same order of magnitude as current k_{ECM} will provide similar performance results and system responsiveness.

The communication between the ECM and the applications is implemented using a shared memory region. We have defined a new Java class, which contains the following Java methods for calling, through the Java Native Interface (JNI) [9], the ECM services for requesting and consulting processors:

- `cpus_assigned()`: returns the current number of processors allocated to the calling application i (A_i).
- `cpus_request(num)`: the calling application i requests to the ECM `num` processors (R_i).

3.1.1. ECM scheduling policy

Traditionally, resource allocation policies have considered conventional performance metrics such as the 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 policies [10]. For this reason, the ECM can implement policies which consider conventional performance metrics as well as incorporating e-business indicators.

As an example, we have implemented a simple but effective policy in order to demonstrate the efficiency of our overload prevention strategy. Policies that are more complex (see Related Work) can be incorporated to the ECM, though the ECM does not offer a user interface to update the policy at the moment. Nevertheless, the code of the ECM has been developed with this possibility in mind, and for this reason, the implementation of the sample policy is encapsulated within a single function. According to this, changing the policy should

[1] Since we focus on secure workloads, which are CPU-intensive, our resource provisioning strategy only considers processors.

be as easy as replacing this function with a new one implementing the desired policy.

Our sample policy considers an e-business indicator in the form of customers of different priority classes (such as Gold, Silver, or Bronze). 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 with respect to low priority customers. In our policy, at every sampling interval k_{ECM}, each application i receives a number of processors $(A_i(k_{\mathrm{ECM}}))$ that is proportional to its request $(R_i(k_{\mathrm{ECM}}))$ and to the application's priority class (P_i) according to the following equation, where NCpus is the number of processors in the hosting platform:

$$A_i(k_{\mathrm{ECM}}) = \mathrm{Round}\left[\frac{P_i \times R_i(k_{\mathrm{ECM}}) \times \mathrm{NCpus}}{\sum_{j=1}^{N} P_j \times R_j(k_{\mathrm{ECM}})}\right]. \quad (1)$$

The scheduling policy should also allow the highest resource utilization to be achieved 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 application isolation). The current policy will decide to share a processor from application i to application j if the processor distribution policy has assigned application i all the processors it requested and the number of processors assigned to application j is lower than the number it requested. Notice that, in this situation, it is possible that a fraction of a processor allocated to application i is not used, for example, if application i needs 2.5 processors, its processor request (R_i) will be 3. If the ECM allocates 3 processors to application i, 0.5 of a processor may be not used. With the processor sharing mechanism, this fraction can be utilized by the application j.

The current ECM implementation only considers sharing a processor between two applications with 0.5 processor granularity. In our opinion, this is enough to show that processor sharing can be an option to achieve good resource utilization in the hosting platform. However, we are working on a finer granularity implementation in which both the processor requests from the applications and the processor assignments decided by the ECM consider fractional utilization of the processors.

The processor sharing mechanism can also consider e-business indicators, such as the application's priority class. As an example, we have also modified the ECM policy in order to benefit high priority applications with respect to low priority applications when sharing processors. With this modification, a processor can only be shared from low priority applications to high priority applications, but not the other way around. This allows high priority applications to achieve higher throughput, but resource utilization in the hosting platform may be worse. Detailed results obtained with this modification can be found in the evaluation (see the third multiprogrammed experiment in Section 5.2).

Additionally, the ECM performs the complete accounting of all the resource allocations decided. This feature is very valuable in hosting platforms that earn money from applications depending on their resource usage, because in this case hosting platforms need to know exactly how many resources have been used by each application.

3.1.2. Application isolation

The ECM aims to minimize the interferences between applications by providing application isolation, that is, given a resource distribution among applications, an application always achieves the same performance with that resource allocation irrespective of the load on other applications. This is an important issue in shared hosting platforms, because a malicious or overloaded application could grab more than its allocated resources, affecting the performance of other applications.

Our proposal provides application isolation by limiting the resources consumed by each application to its reserved amount. This is accomplished in two complementary ways. First, we consider self-managed applications, which are able to use the admission control mechanism for limiting their accepted load so that it can be handled with their assigned processors. Second, in order to prevent the possibility that any application can affect the performance of the other applications because it does not use the admission control mechanism, the ECM not only decides how many processors to assign to each application but also which processors. In order to accomplish this, the ECM configures the CPU affinity mask of each application (using the Linux `sched_setaffinity` function) so that the processor allocations of the different applications do not overlap (except when applications share a processor).

3.2. Application components

Our overload control approach includes adding self-managing capabilities to the applications

running in a hosting platform with ECM. First, the applications must be able to determine the number of processors they need to handle their incoming workload while avoiding QoS degradation, that is R_i, and provide this value to the ECM. Our proposal achieves this capability by adding a new component, called the eDragon Load Monitor (ELM), within the server that runs each web application (see Fig. 2). Second, the applications must be able to adapt their behavior according to the resources provided by the ECM in order to prevent overload and in this way maintain the QoS of admitted requests. This is accomplished by adding another component, called the eDragon Admission Control (EAC), within the server that runs each web application (see Fig. 2).

Notice that, although we have modified the server that runs the applications (not the web applications), this is only as a proof-of-concept for demonstrating the feasibility of our proposal. In a production system, the proposed functionality could be easily implemented in a proxy, and in this way avoiding modifying the applications.

Both the ELM and the EAC are active components that continuously monitor the incoming SSL connections reaching the server. The ELM uses the monitored information to estimate the resource requirements of the application. The EAC is responsible for deciding on the acceptance of the incoming SSL connections depending on the allocated resources. Every sampling interval, the ELM and the EAC communicate with the ECM in order to update the resource provisioning depending on the current application workload. If desired, the ELM and the EAC can use a different sampling interval from the ECM. We define this value as k_{APL}. However, in the current implementation k_{ECM} and k_{APL} have the same value (i.e. 2 s).

As shown in Fig. 3, both the ELM and the EAC are based on the differentiation between new and resumed SSL connections, which is performed with the differentiation mechanism described in [3]. When a client establishes a new SSL connection with the server, a full SSL handshake is negotiated, including parts that need a great amount of computation time. On the other hand, when a client establishes a new HTTP connection with the server but using an existing SSL connection (it resumes a SSL connection), a SSL resumed handshake is negotiated. As the SSL session ID is reused, part of the SSL handshake negotiation can be avoided, considerably reducing the computation time for performing a resumed SSL handshake. Our measurements in a 1.4 GHz Xeon machine reveal that the computational demands of a full SSL handshake and a resumed SSL handshake are around 175 and 2 ms, respectively. Notice the big difference existing between both types of handshake.

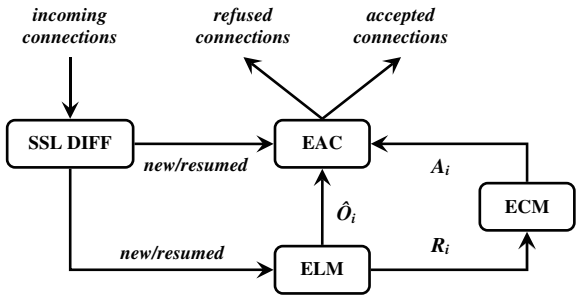


Fig. 3. Components interaction in our control strategy.

3.2.1. eDragon load monitor (ELM)

The ELM continuously monitors the incoming secure connections to the server and performs online measurements distinguishing new SSL connections from resumed SSL connections. For every sampling interval k_{APL}, the ELM calculates the number of resumed SSL connections (defined as $O_i(k_{\mathrm{APL}})$) and the number of new SSL connections (defined as $N_i(k_{\mathrm{APL}})$) received at application i during that interval. These measurements consider not only the accepted connections but all the connections received.

The average computation time entailed by a resumed SSL connection (defined as CTO_i) and the average computation time entailed by a new SSL connection (defined as CTN_i) have been measured using static profiling on the application. Each measurement includes not only the average computation time spent in the negotiation of the SSL handshake, but also the average computation time spent for generating all the dynamic web content requested using that connection and the average computation time spent for SSL encryption/decryption during the data transferring phase.

Using the previously defined values, at the beginning of every sampling interval k_{APL} the ELM calculates $R_i(k_{\mathrm{APL}})$, which is the number of processors that the application i will need to handle its incoming workload during the interval (k_{APL}). Then, it informs the ECM using the `cpus_request` function from the communication path between the

applications and the ECM (as shown in Fig. 3). The complete equation is as follows:

$$R_i(k_{\mathrm{APL}}) = \left\lceil \frac{\widehat{O}_i(k_{\mathrm{APL}}) \times \mathrm{CTO}_i + \widehat{N}_i(k_{\mathrm{APL}}) \times \mathrm{CTN}_i}{k_{\mathrm{APL}}} \right\rceil. \tag{2}$$

Notice that $R_i(k_{\mathrm{APL}})$ depends on predicting the number of new and resumed SSL connections that will hit the server during the sampling interval k_{APL}. The ELM predicts these values from previous observations using an exponentially weighted moving average (EWMA) with parameter $\alpha = 0.7$, in order to prevent sudden spikes in the sample from causing large reactions in the prediction. The complete equations are as follows:

$$\begin{aligned} \widehat{O}_i(k_{\mathrm{APL}}) &= \alpha \times O_i(k_{\mathrm{APL}} - 1) + (1-\alpha) \\ &\quad \times \widehat{O}_i(k_{\mathrm{APL}} - 1), \quad k_{\mathrm{APL}} \geqslant 3, \\ \widehat{O}_i(k_{\mathrm{APL}}) &= O_i(k_{\mathrm{APL}} - 1), \quad k_{\mathrm{APL}} = 2, \end{aligned} \tag{3}$$

$$\begin{aligned} \widehat{N}_i(k_{\mathrm{APL}}) &= \alpha \times N_i(k_{\mathrm{APL}} - 1) + (1-\alpha) \\ &\quad \times \widehat{N}_i(k_{\mathrm{APL}} - 1), \quad k_{\mathrm{APL}} \geqslant 3, \\ \widehat{N}_i(k_{\mathrm{APL}}) &= N_i(k_{\mathrm{APL}} - 1), \quad k_{\mathrm{APL}} = 2. \end{aligned} \tag{4}$$

3.2.2. eDragon admission control (EAC)

The EAC extends the session-oriented adaptive mechanism described in [3] which performs admission control based on SSL connection differentiation. This mechanism prevents applications overload by limiting the acceptation of new SSL connections to the maximum number acceptable by the application without overloading. In addition, it accepts all the client connections that resume an existing SSL session, in order to increase the probability for a client to complete a session, and in this way maximizing the number of sessions successfully completed.

The EAC extends the original mechanism by performing admission control when the number of processors assigned to the server varies along time. In addition, the EAC improves the mechanism functioning by using EWMA to make the predictions.

The EAC uses the measurements of incoming SSL connections performed by the ELM to calculate, at the beginning of every sampling interval k_{APL}, the maximum number of new SSL connections that can be accepted by application i during that interval without overloading. We define this value as $\mathrm{AN}_i(k_{\mathrm{APL}})$. This value depends on the number of processors allocated by the ECM to the application during the interval k_{APL} (previously defined as $A_i(k_{\mathrm{APL}})$), which can be requested from the ECM using the `cpus_assigned` function from the communication path between the applications and the ECM.

In addition, as resumed SSL connections have preference with respect to new SSL connections (all resumed SSL connections are accepted), $\mathrm{AN}_i(k_{\mathrm{APL}})$ depends also on the computation time required by the resumed SSL connections that will be accepted during the interval k_{APL}. This value can be calculated as $\widehat{O}_i(k_{\mathrm{APL}}) \times \mathrm{CTO}_i$, using the value of $\widehat{O}_i(k_{\mathrm{APL}})$ predicted by the ELM through EWMA (as shown in Fig. 3). According to this, the complete equation for calculating $\mathrm{AN}_i(k_{\mathrm{APL}})$ is as follows:

$$\mathrm{AN}_i(k_{\mathrm{APL}}) = \left\lfloor \frac{k_{\mathrm{APL}} \times A_i(k_{\mathrm{APL}}) - \widehat{O}_i(k_{\mathrm{APL}}) \times \mathrm{CTO}_i}{\mathrm{CTN}_i} \right\rfloor. \tag{5}$$

The EAC will only accept the maximum number of new SSL connections that do not overload the server ($\mathrm{AN}_i(k_{\mathrm{APL}})$) (they can be served with the available computational capacity without degrading their QoS). The rest of the new SSL connections arriving at the server will be refused. Notice that, although connection classification and rejection is done at the application level, these tasks are performed prior to any handling of the incoming connection, even prior to the negotiation of the SSL handshake. For this reason, the related rejection overhead is not noticeable.

4. Experimental environment

We use Tomcat v5.0.19 [7] as the application server. 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. 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 setting the maximum number of HttpProcessors to 100 and the connection persistence timeout to 10 s.

The experimental environment also includes a deployment of the servlets version 1.4.2 of the RUBiS (Rice University Bidding System) [11]

benchmark on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. We decided to use RUBiS since it models a prototypical e-commerce application. However, since our overload control proposal is oriented to secure environments, in order to demonstrate its effectiveness, we need to modify the original implementation of RUBiS to use secure connections. We firmly believe that, thought we are not using a benchmark in its strict meaning, the results demonstrating that our approach is able to deal with overload are still valid. RUBiS defines 26 interactions. Five of the 26 interactions are implemented using static HTML pages. The remaining 21 interactions require data to be generated dynamically.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [12]. This tool allows the creation of a continuous flow of HTTP or HTTPS requests issued from one or more client machines and processed by one server machine. The workload distribution generated by Httperf was extracted from the RUBiS client emulator. RUBiS client emulator defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read–write interactions. The experiments in this paper use the bidding mix, which is more representative of an auction site workload.

Although the RUBiS client emulator follows a closed system model, the combination of Httperf with a RUBiS workload allows the use of a partly-open model (as defined in [13]), which is a more realistic alternative to represent web applications behavior. In this model, new clients initiate a session with the server (implemented as a persistent HTTPS connection) according to a certain arrival rate. This rate can be specified as one of the parameters of Httperf. Emulated clients use the established persistent connection to issue RUBiS interactions to the server.

Every time a client receives the server response to an interaction, there is a probability p that the client simply leaves the system, and there is a probability $1\text{-}p$ that the client remains in the system and makes another interaction, possibly after some think time. The think time emulates the"thinking" period of a real client who takes a period of time before clicking on the next link. The think time is generated from a negative exponential distribution with a mean of 7 s. The next interaction in the session is decided using a Markov model. This model uses a transition probability matrix with probabilities attached to transitions from each RUBiS interaction to the others. In particular, this matrix defines for each interaction the probability of ending the session (i.e. p), which is 0.2 for 10 of the 26 interactions and 0 for the rest. This value determines the duration of a RUBiS session, which is 50 interactions per session on average (with a maximum of 100 interactions per session). In addition, each interaction includes a request to the server of all the images embedded in the HTML page returned as a response to the interaction (1,4 images per interaction on average). Considering this, we measure the server throughput as the number of HTTP replies per second returned by the server to the client's interactions (including the images requested).

Httperf also allows the configuring of a client's timeout. If this timeout is elapsed and no reply has been received from the server, the current persistent connection with the server is discarded. This emulates the fact that the real client can get bored of waiting for a response from the server and leave the site. For the experiments in this paper, Httperf has been configured by setting the client timeout value to 10 s.

The hosting platform is a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We use MySQL v4.0.18 [14] as our database server with the MM.MySQL v3.0.8 JDBC driver. The MySQL server runs on another machine, which is a 2-way Intel XEON 2.4 GHz with 2 GB RAM. We also have a 2-way Intel XEON 3.0 GHz with 2 GB RAM machine running the workload generator (Httperf 0.8.5). All the machines are connected through a 1 Gbps Ethernet interface and run the 2.6 Linux kernel. For our experiments, we use the Sun JVM 1.4.2 for Linux, using the server JVM instead of the client JVM, using the concurrent low pause collector and setting the initial and the maximum Java heap size to 1024 MB. All the tests are performed with the common RSA-3DES-SHA cipher suit, using 1024 bit RSA key.

5. Evaluation

In this section we present the evaluation of the results of our proposal divided into two parts. First, we evaluate the effectiveness of the self-managed Tomcat when using the eDragon Admission Control (EAC) to prevent application overload and the accuracy of the self-managed Tomcat when

using the eDragon Load Monitor (ELM) to estimate its processor requirements by comparing the performance of a single instance of the self-managed Tomcat server with respect to the original Tomcat. Second, we evaluate our overload control strategy which combines dynamic resource provisioning and admission control by running several experiments with two self-managed Tomcat instances in a multiprocessor hosting platform with the ECM.

5.1. Original Tomcat vs. self-managed Tomcat

In this section, we compare the performance of a single instance of the self-managed Tomcat server with respect to the original Tomcat by running a set of experiments with different client workloads in the environment previously described. In every experiment, the server receives client requests for 10 minutes and after that we measure some average metrics. Fig. 4 shows Tomcat's average throughput as a function of the number of new sessions per second initiated with the server comparing the original Tomcat server with respect to the self-managed Tomcat server. Notice that the server throughput increases linearly with respect to the input load (the server scales) until a determined number of clients hit the server. At this point, the throughput achieves its maximum value. Until this point, the self-managed Tomcat behaves in the same way as the original Tomcat. When the number of sessions that overloads the server has been achieved, the original Tomcat's throughput begins to degrade while the number of sessions increases until it reaches approximately 35% of the maximum achievable throughput when the server is fully overloaded. The self-managed Tomcat avoids this degradation by applying the admission control mechanism, maintaining the throughput at the maximum achievable level, as shown in Fig. 4.

As well as degrading the server throughput, overload also affects the server response time, as shown in Fig. 5. This figure shows Tomcat's 95th-percentile response time (a metric typically used in the definition of SLA contracts) as a function of the number of new sessions per second initiated with the server comparing the original Tomcat server with respect to the self-managed Tomcat server. Notice that when the original Tomcat is overloaded the response time increases while the workload increases. On the other hand, the self-managed Tomcat can maintain the response time at levels that guarantee a good quality of service to the clients, even when the number of sessions that would overload the server has been achieved, as shown in Fig. 5.

Overload has another undesirable effect, especially in e-commerce environments where session completion is a key factor. As shown in Fig. 6, which shows the number of sessions successfully completed comparing the original Tomcat server

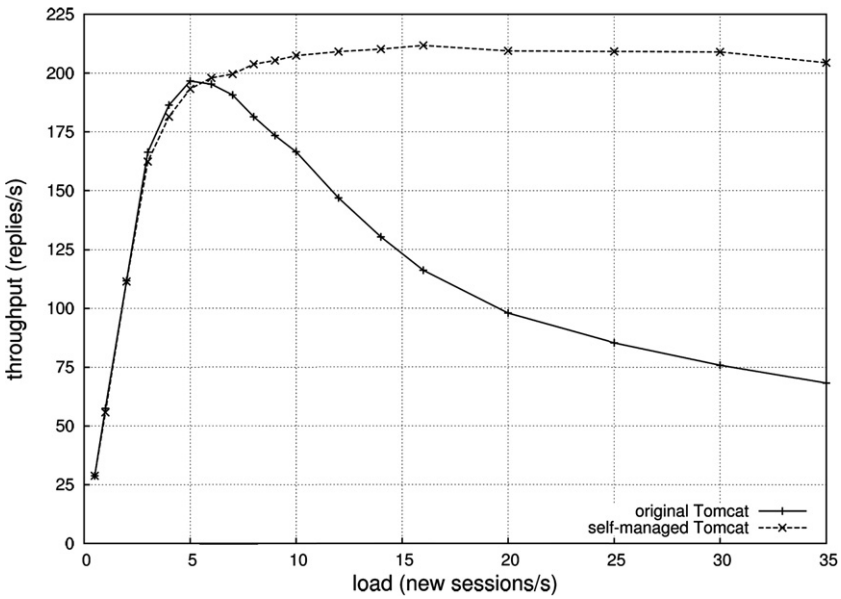


Fig. 4. Throughput of the original Tomcat vs. self-managed Tomcat.

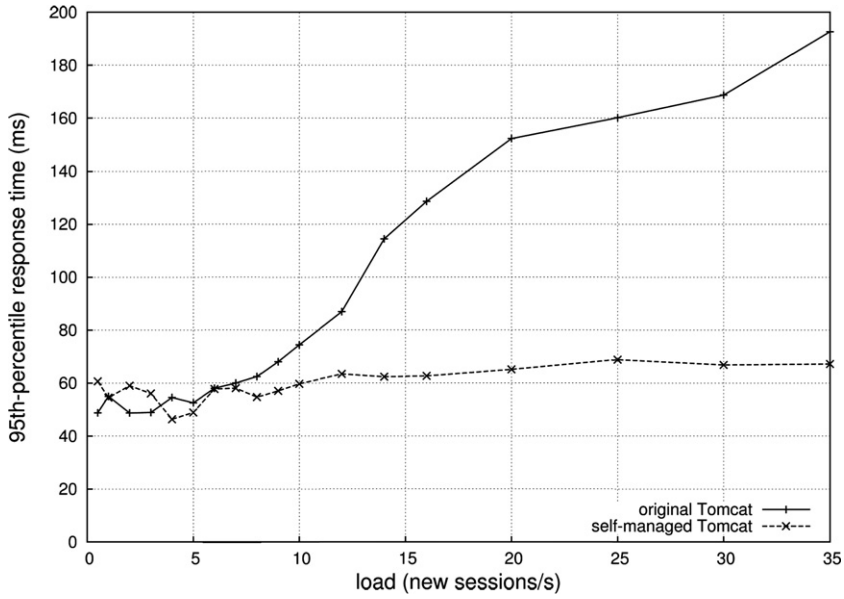


Fig. 5. 95th-percentile response time of the original Tomcat vs. self-managed Tomcat.

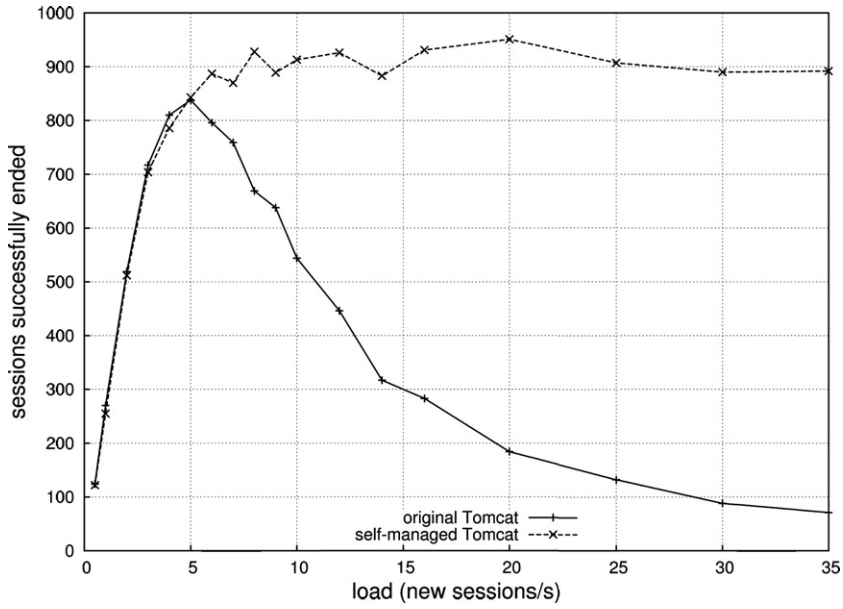


Fig. 6. Number of sessions successfully completed of the original Tomcat vs. self-managed Tomcat.

with respect to the self-managed Tomcat server, when the original Tomcat is overloaded only a few sessions can completely finalize. Consider the great revenue loss that this fact can provoke for example in an online store, where only a few clients can finalize the acquisition of a product. On the other hand, the self-managed Tomcat can maintain the number of sessions that can completely finalize, even when the number of clients that would overload the server has been achieved.

Previous results demonstrate that the self-managed Tomcat is able to attend to more clients while maintaining good QoS. As commented, this occurs as a result of refusing new SSL sessions that cannot be attended. Refusing these sessions can also have an effect on the QoS that clients perceive. In order

to quantify this effect we have measured the number of sessions refused by the server and the number of sessions that time out waiting for a server response as a function of the number of new sessions per second initiated with the server. Fig. 7 shows the results obtained comparing the original Tomcat server with respect to the self-managed Tomcat server, demonstrating that the self-managed Tomcat does not deteriorate the perceived QoS with respect to the original behavior, since the total number of sessions unattended by the server is lower. Notice that when the server is overloaded, surplus sessions tend to time out while waiting for the response of the original Tomcat, while the self-managed Tomcat directly refuses these sessions.

Finally, Fig. 8 shows the average number of processors allocated to the server during the execution as a function of the number of new sessions per second initiated with the server comparing the original Tomcat server with respect to the self-managed Tomcat server. When running the original Tomcat, the hosting platform must statically over-provision the server with the maximum number of processors (four in this case) in order to achieve the maximum performance, because it has no information about its processor requirements. However, this provokes poor processor utilization when the original Tomcat requires fewer processors. On the other side, the ELM implemented within the self-managed Tomcat is able to accurately calculate its processor requirements and communicate them to the hosting platform, which can dynamically allocate the server only the required processors it needs, as shown in Fig. 8, avoiding processor under-utilization but ensuring good performance.

5.2. Multiple Tomcat instances

In this section, we evaluate our overload control strategy by running three multiprogrammed experiments with two self-managed Tomcat instances in a multiprocessor hosting platform with the ECM. Our first multiprogrammed experiment consists of two Tomcat instances with the same priority running for 90 minutes in a 4-way hosting platform. Each Tomcat instance has a variable input load over time, which is shown in the top part of Fig. 9 representing the number of new sessions per second initiated with 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 two Tomcat instances in a hosting platform. For example, as shown in Fig. 4, when the total workload in the hosting platform is lower or equal to that of 3 new sessions/s, the hosting platform is not overloaded, i.e it can satisfy the processor requirements of all the application instances. In Fig. 9 this occurs in the zones labeled *A* (between 0 s and 1200 s) and *C* (between 2400 s and 3000 s). The rest of the time, the requirements of the

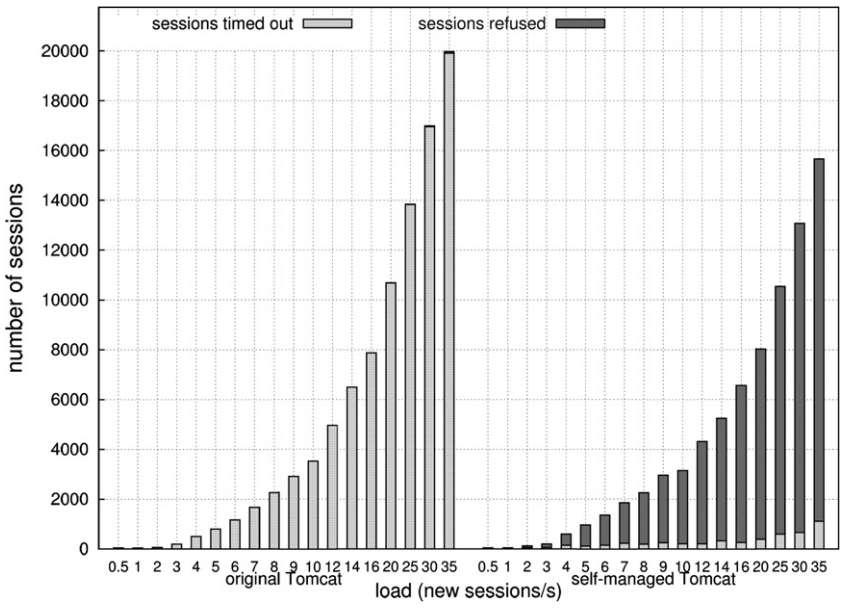


Fig. 7. Number of sessions refused and timed out of the original Tomcat vs. self-managed Tomcat.

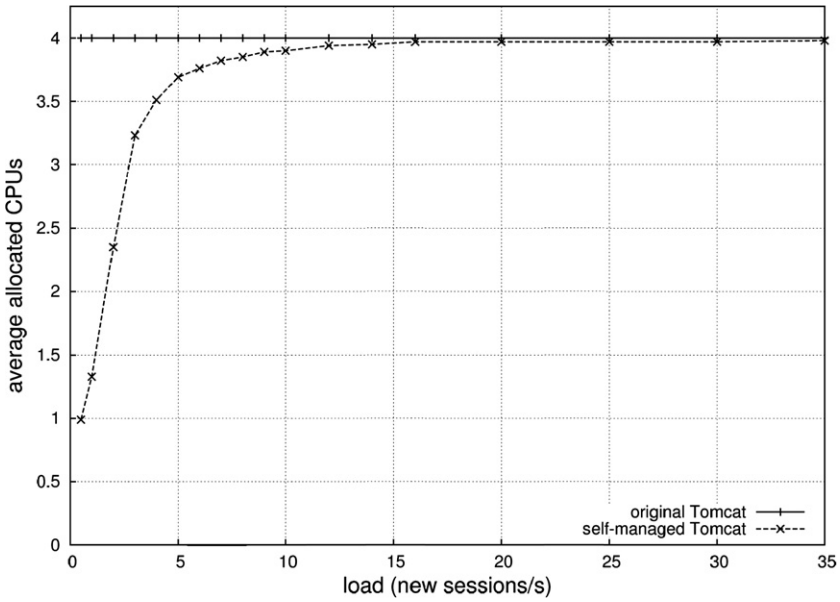


Fig. 8. Average number of allocated processors of the original Tomcat vs. self-managed Tomcat.

two Tomcat instances exceed the number of processors of the hosting platform, therefore the hosting platform is overloaded. The hosting platform can be only slightly overloaded, provoking a low performance degradation if overload is uncontrolled (i.e. this occurs when the total workload is lower than 8 new clients/s), as for instance in the zones labeled *B* (between 1800 s and 2400 s) and *D* (between 3600 s and 4200 s); or completely overloaded, provoking a severe performance degradation if overload is uncontrolled, as for instance in the zone labeled *E* (between 4200 s and 4800 s).

As well as the input load over time, Fig. 9 also shows the allocated processors for each Tomcat

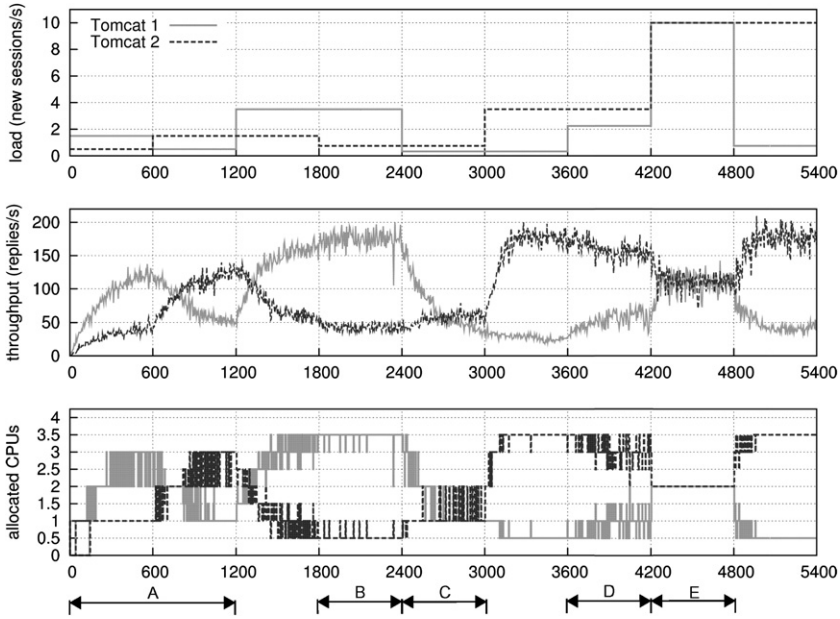


Fig. 9. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances in a 4-way Linux hosting platform with ECM if they have the same priority.

instance (bottom part) and the throughput achieved with these processors (middle part), presenting this information in a way that eases the correlation of the different metrics. Notice that, when the hosting platform is not overloaded, the two Tomcat instances receive all the processors they have requested, obtaining the corresponding throughput.

When the hosting platform is overloaded, since the two instances have the same priority, the ECM distributes the available processors depending only on each Tomcat request, which depends on each Tomcat's input load. Therefore, the Tomcat instance with the higher input load (that is, with more processor requirements) receives more processors and hence achieves higher throughput. For example, in the zone labeled *B* (between 1800 s and 2400 s), 3.5 new sessions per second are initiated with *Tomcat 1* while only 0.75 new sessions per second are initiated with *Tomcat 2*. In this case, input load from *Tomcat 1* is higher than input load from *Tomcat 2*, thus *Tomcat 1* will receive more processors than *Tomcat 2*. In particular, *Tomcat 1* receives 3.5 processors on average (achieving a throughput of around 175 replies/s) while *Tomcat 2* receives only 0.5 processors on average (achieving a throughput of around 45 replies/s). Notice that a processor is being shared between *Tomcat 1* and *Tomcat 2*.

In the same way, in the zone labeled *D* (between 3600 s and 4200 s), 3.5 new sessions per second initiate in *Tomcat 2* while only 2.25 new sessions per second initiate in *Tomcat 1*. In this case, input load from *Tomcat 2* is higher than input load from *Tomcat 1*, thus *Tomcat 2* will receive more processors than *Tomcat 1*. In particular, *Tomcat 2* receives between 2.5 and 3.5 processors on average (achieving a throughput of around 155 replies/s) while *Tomcat 1* receives only between 0.5 and 1.5 processors on average (achieving a throughput of around 60 replies/s).

Finally, when the input load is the same for *Tomcat 1* and *Tomcat 2* (for instance in the zone labeled *E* (between 4200 s and 4800 s)), the two instances receive the same number of processors (two in this case), obtaining the same throughput (around 115 replies/s).

Notice that, the EAC is ensuring that although the number of required processors is not supplied, the server throughput is not degraded and the response time of admitted requests is not increased, as demonstrated in Section 5.1. This cannot be guaranteed by the Linux default dynamic provisioning strategy, and for this reason, our proposal achieves considerably higher throughput when the hosting platform is fully overloaded, as for instance in the zone labeled *E* (compare middle part of Fig. 9 vs. the bottom part of Fig. 1).

Our second multiprogrammed experiment has the same configuration as the previous one but, in this case, *Tomcat 1* has higher priority than *Tomcat 2* (2 vs. 1). As the two instances have different priorities, the ECM distributes the available processors depending on each Tomcat request and on its priority, following the equation presented in Section 3.1.1. Fig. 10 shows the results obtained for this experiment presenting these results in the same way as Fig. 9. Notice that now in the zone labeled *B*, processors allocated to *Tomcat 1* have increased, oscillating between 3.5 and 4 on average, while processors allocated to *Tomcat 2* have decreased, oscillating between 0 and 0.5 on average, because as well as having higher input load, *Tomcat 1* has also higher priority than *Tomcat 2*.

In the same way, in the zone labeled *D*, processors allocated to *Tomcat 2* have decreased, oscillating between 1.5 and 2.5 on average, while processors allocated to *Tomcat 1* have increased, oscillating between 1.5 and 2.5 on average, because although *Tomcat 2* has a higher input load, *Tomcat 1* has higher priority than *Tomcat 2*.

Finally, in the zone labeled *E*, although the input load is the same for *Tomcat 1* and *Tomcat 2*, *Tomcat 1* now receives more processors than *Tomcat 2* (3 vs. 1), because *Tomcat 1* has a higher priority than *Tomcat 2*. With this processor allocation, *Tomcat 1* obtains higher throughput than *Tomcat 2* (around 160 replies/s vs. 60 replies/s).

Again, the EAC prevents the throughput degradation that occurred with the default Linux dynamic provisioning strategy, thus the throughput achieved when the hosting platform is fully overloaded (as for instance in zone labeled *E*) is higher (compare the middle part of Fig. 10 vs. the bottom part of Fig. 1).

In our last multiprogrammed experiment, we have the same configuration as in the previous one, but with a slightly different behavior of the ECM in order to benefit the execution of high priority applications. In this experiment, a processor can be only shared from low priority applications to high priority applications, but not the other way around. Fig. 11 shows the results obtained for this experiment presenting these results in the same way as Fig. 9. As shown in this figure, in the zone labeled *B*, processors allocated to *Tomcat 1* have

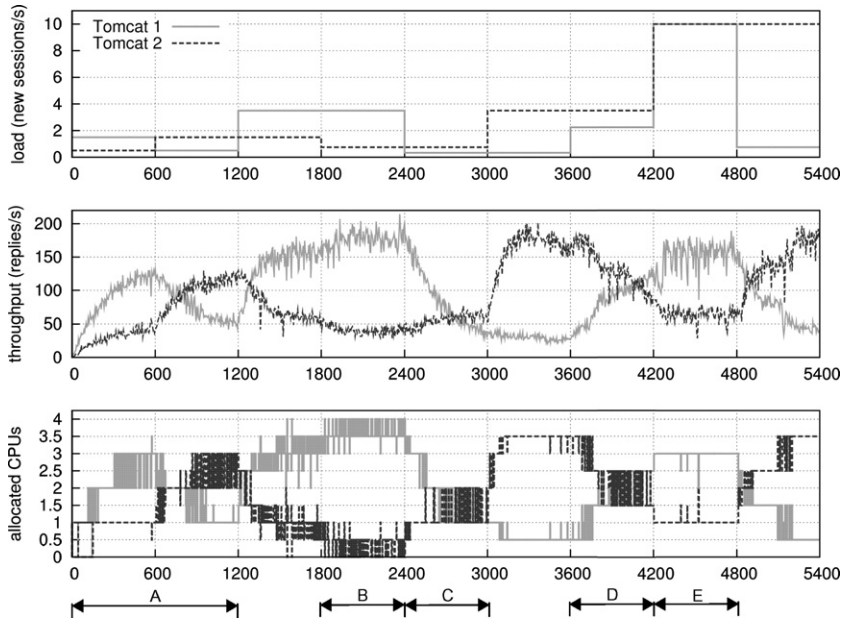


Fig. 10. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances in a 4-way Linux hosting platform with ECM if *Tomcat 1* has higher priority than *Tomcat 2*.

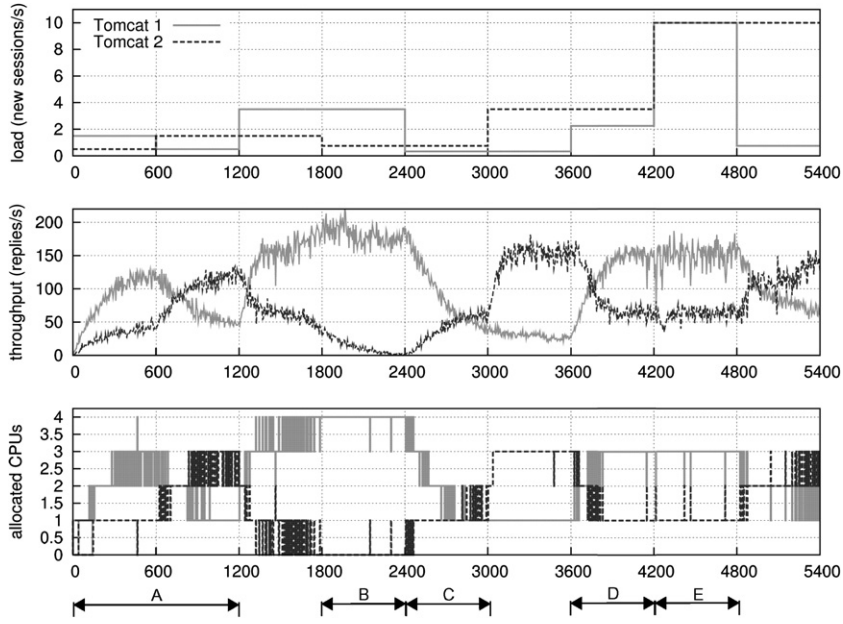


Fig. 11. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances in a 4-way Linux hosting platform with ECM if *Tomcat 1* has higher priority than *Tomcat 2* and *Tomcat 1* does not share processors with *Tomcat 2*.

increased to almost 4 on average while processors allocated to *Tomcat 2* are now nearly 0, because *Tomcat 1* has higher priority than *Tomcat 2* and does not share processors with lower priority applications.

In the same way, in the zone labeled *D*, processors allocated to *Tomcat 2* have decreased to 1 on average while processors allocated to *Tomcat 1* have increased to 3 on average, because although *Tomcat 2* has a higher input load, *Tomcat 1* has a higher

priority than *Tomcat 2* and does not share processors. With this processor allocation, *Tomcat 1* now obtains higher throughput than in the previous experiment (around 150 replies/s vs. 90 replies/s) while *Tomcat 2* now achieves lower throughput (around 60 replies/s vs. 125 replies/s).

Notice that with this modification in the processor sharing mechanism, high priority applications can achieve higher throughput, but resource utilization in the hosting platform is a little worse.

6. Related work

Literature dealing with overload in web applications has widely considered the techniques used in this paper; that is admission control, service differentiation and dynamic resource management.

Admission control and service differentiation have been typically combined to prevent server overload. However, some of these works [15,16] are based on web systems serving only static content, and for this reason, these solutions are not directly applicable on multi-tiered sites based on dynamic web content. Because of this, some works (including our proposal) have focused specifically on these kinds of systems. For example, [17] describes an adaptive approach to overload control in the context of the SEDA [18] Web server. SEDA decomposes services into multiple stages, each one of which can perform admission control based on monitoring the response time through the stage. In [19], the authors present an admission control mechanism for e-commerce sites that externally observes the execution costs of requests, distinguishing different request types.

Some of these works implement admission control by means of control theory. For instance, Yaksha [20] implements a self-tuning proportional integral controller for admission control in multitier e-commerce applications using a single queue model. Quorum [21] is a non-invasive software approach to QoS provisioning for large-scale Internet services that ensures reliable QoS guarantees using traffic shaping and admission control at the entrance of the site, and monitoring service at the exit. In [22], the authors propose a scheme for autonomous performance control of Web applications. They use a queuing model predictor and an online adaptive feedback loop that enforces admission control of the incoming requests to ensure the desired response time target is met.

However, on most of the described works, overload control is performed on a per request basis, which may not be adequate for many session-based applications, such as e-commerce applications. This has motivated some proposals [23,24] to prevent overload by considering the particularities of session-based applications. However, none of them consider complex e-commerce environments and a combinations of techniques as our approach does.

As commented, recent studies [4–6] have reported the considerable benefits of dynamically adjusting resource allocations to handle variable workloads instead of statically over-provisioning resources in a hosting platform. This premise has motivated the proposal of several techniques to dynamically provision resources to applications in on-demand hosting platforms.

These proposals can be based on either a dedicated or a shared model [4]. In the dedicated model, some cluster nodes are dedicated to each application and the provisioning technique must determine how many nodes to allocate to the application. In the shared model, which we consider in this paper, 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.

Hosting platforms based on a dedicated model use to be expensive in terms of space, power, cooling, and cost. For this reason, shared model constitutes an attractive low-cost choice for hosting environments when dedicated model cannot be afforded. For instance, distributed resource management technology of IBM's WebSphere Extended Deployment [25] is based on hosting platforms that support multiple web applications, with each web application deployed and replicated on different but overlapping subsets of machines.

Shared model can be implemented as a cluster of servers where several applications can run in the same server, or using a multiprocessor machine for hosting all the applications. Clusters of servers are widely extended and are easily scalable but resource provisioning in these systems is more complex and has some efficiency problems. First, software installation and configuration overheads entail some latencies when switching a server from one application to another [26]. The second problem arises in session-based environments when the session state must be transferred between servers. These problems have motivated some proposals for reducing these latencies [27]. However, as we are considering e-commerce applications, which are typically session-based, and we want to implement an overload

control strategy able to react to unexpected workload changes in a very short time, we focus on SMP hosting platforms.

Depending on the hosting platform architecture considered, the problem of provisioning resources in cluster architectures has been addressed in [26,28–30] by allocating entire machines (dedicated model) and in [31–33,25] by sharing node resources among multiple applications (shared model).

Depending on the mechanism used to decide the resource allocations, dynamic resource management proposals can be classified as: control theoretic approaches with a feedback element [34–36], open-loop approaches based on queuing models to achieve resource guarantees [31,37,38,30] and observation-based approaches that use runtime measurements to compute the relationship between resources and QoS goal [33].

However, 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 (such as our proposal) are most suited for handling varying workloads and non-linear behaviors. In addition, our proposal is fully adaptive to the available resources and to the incoming load in the server instead of using predefined thresholds (e.g. [28]).

Nevertheless, most of the described works fail at achieving a global solution for preventing overload. Only the works presented in [39,25] demonstrate that the most effective way to handle overload must consider a combination of techniques instead of considering each technique in isolation. In this aspect, these works are similar to our proposal.

In addition, other novelties of our proposal with respect to previous work include the inclusion of e-business indicators to the resource allocation process (suggested only in a few works such as [32]) or the consideration of the particularities of secure web applications when dealing with overload.

Other approaches for dealing with overload include request scheduling and service degradation. Request scheduling refers to the order in which concurrent requests should be served. Typically, servers have left this ordination to the operating system. But, as it is well known from queuing theory that shortest remaining processing time first (SRPT) scheduling minimizes queuing time (and therefore the average response time), some proposals [40,41] implement policies based on this algorithm to prioritize the service of short static content requests in front of long requests. This prioritized scheduling in web servers has proven effective in providing significantly better response time to high priority requests at a relatively low cost to lower priority requests. However, although scheduling can improve response times, under extreme overloads other mechanisms become indispensable. Anyway, better scheduling can always be complementary to any other mechanism.

Service degradation is based on avoiding refusing clients as a response to overload but reducing the service offered to clients [42,43,39,17], for example in the form on providing smaller content (e.g. lower resolution images).

7. Conclusions and future work

In this paper, we have proposed an overload control strategy for secure web applications that brings together admission control based on SSL connection differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads avoiding the QoS degradation.

Our approach is based on a global resource manager responsible for periodically distributing the available processors among the web applications following a determined policy. The resource manager can be configured to implement different policies, and can 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 which is totally transparent to the user, using bi-directional communication. On one hand, the applications request the resource manager for the number of processors needed to handle their incoming load without QoS degradation. On the other hand, the resource manager can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications can apply an admission control mechanism to limit the number of admitted requests so they can be served with the allocated processors without degrading their QoS. The admission control mechanism is based on SSL connection differentiation; depending on if the SSL connection reuses an existing SSL connection on the server or not. Prioritizing resumed SSL connections maximizes the number of sessions completed successfully, allowing e-commerce sites which are based

on SSL to increase the number of transactions completed, thus generating higher profit.

Our evaluation demonstrates the benefit of our approach for efficiently managing the resources on hosting platforms and for preventing servers overload on secure environments. Our proposal performs considerably better than the default Linux dynamic resource provisioning strategy especially when the hosting platform is fully overloaded. Although our implementation targets the Tomcat application server, the proposed overload control strategy can be applied on any other server.

Our future work goes towards the extension of our proposal in order to support heterogeneous applications in the hosting platform. With respect to this, we are currently working with the SPECWeb2005 benchmark. We are also considering the use of virtualization technologies in order to provide better performance isolation to the applications running in the hosting platform. In addition, we plan to improve the resource utilization in the hosting platform when using the ECM by considering fractional allocation of processors.

Overload control and resource management of web applications on shared hosting platforms are hot topics in current research in this area. In addition, security has appeared as an important issue that can heavily affect the scalability and performance of web applications. This work effectively faces all these issues in a global approach, creating a solid base for further research toward autonomic web systems.

Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under Contracts TIN2004-07739-C02-01 and TIN2007-60625 and by the BSC (Barcelona Supercomputing Center). For additional information about the authors, visit the Barcelona eDragon Research Group web site [44].

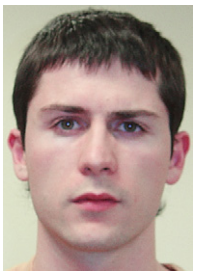
References

[1] E. Rescorla, HTTP over TLS. RFC 2818, May 2000.
[2] A.O. Freier, P. Karlton, C. Kocher, The SSL Protocol. Version 3.0. November 1996. URL <http://wp.netscape.com/eng/ssl3/ssl-toc.html>.
[3] J. Guitart, D. Carrera, V. Beltran, J. Torres, E. Ayguadé, Session-based adaptive overload control for secure dynamic web applications, in: 34th International Conference on Parallel Processing (ICPP'05), Oslo, Norway, June 14–17, 2005, pp. 341–349.
[4] A. Andrzejak, M. Arlitt, J. Rolia, Bounding the resource savings of utility computing models, Tech. Rep. HPL-2002-339, HP Labs, December 2002.
[5] A. Chandra, P. Goyal, P. Shenoy, Quantifying the benefits of resource multiplexing in on-demand data centers, in: 1st Workshop on Algorithms and Architectures for Self-Managing Systems (Self-Manage 2003), San Diego, California, USA, June 11, 2003.
[6] A. Chandra, P. Shenoy, Effectiveness of dynamic resource allocation for handling internet flash crowds, Tech. Rep. TR03-37, Department of Computer Science, University of Massachusetts, USA, November 2003.
[7] Jakarta Project. Apache Software Foundation, Jakarta Tomcat Servlet Container. URL <http://jakarta.apache.org/tomcat>.
[8] J. Guitart, V. Beltran, D. Carrera, J. Torres, E. Ayguadé, Characterizing secure dynamic web applications scalability, in: 19th International Parallel and Distributed Symposium (IPDPS'05), Denver, Colorado, USA, April 4–8, 2005.
[9] Sun Microsystems, Java Native Interface (JNI). URL <http://java.sun.com/j2se/1.5.0/docs/guide/jni/>.
[10] D. Menasce, V. Almeida, R. Fonseca, M. Mendes, Business-oriented resource management policies for e-commerce servers, Perform. Evaluation 42 (2–3) (2000) 223–239.
[11] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, W. Zwaenepoel, Specification and implementation of dynamic web site benchmarks, in: IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, Texas, USA, November 25, 2002.
[12] D. Mosberger, T. Jin, httperf: A tool for measuring web server performance, in: Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), Madison, Wisconsin, USA, June 23, 1998, pp. 59–67.
[13] B. Schroeder, A. Wierman, M. Harchol-Balter, Open versus closed: a cautionary tale, in: 3rd Symposium on Networked Systems Design & Implementation (NSDI'06), San Jose, CA, USA, May 8–10, 2006, pp. 239–252.
[14] MySQL. URL <http://www.mysql.com/>.
[15] X. Chen, H. Chen, P. Mohapatra, ACES: an efficient admission control scheme for QoS-Aware web servers, Computer Communications 26 (14) (2003) 1581–1593.
[16] T. Voigt, R. Tewari, D. Freimuth, A. Mehra, Kernel mechanisms for service differentiation in overloaded web servers, in: 2001 USENIX Annual Technical Conference, Boston, MA, USA, June 25–30, 2001, pp. 189–202.
[17] M. Welsh, D. Culler, Adaptive overload control for busy internet servers, in: 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, USA, March 26–28, 2003.
[18] M. Welsh, D. Culler, E. Brewer, SEDA: An architecture for well-conditioned, scalable internet services, in: 18th Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, October 21–24, 2001, pp. 230–243.
[19] S. Elnikety, E. Nahum, J. Tracey, W. Zwaenepoel, A method for transparent admission control and request scheduling in e-commerce web sites, in: 13th International Conference on

World Wide Web (WWW'04), New York, USA, May 17–22, 2004, pp. 276–286.
[20] A. Kamra, V. Misra, E. Nahum, Yaksha: a controller for managing the performance of 3-tiered websites, in: 12th International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada, June 7–9, 2004.
[21] J. Blanquer, A. Batchelli, K. Schauser, R. Wolski, Quorum: flexible quality of service for internet services, in: 2nd Symposium on Networked Systems Design and Implementation (NSDI'05), Boston, MA, USA, May 2–4, 2005, pp. 159–174.
[22] X. Liu, J. Heo, L. Sha, X. Zhu, Adaptive Control of Multi-Tiered Web Application Using Queueing Predictor, in: 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada, April 3–7,2006.
[23] H. Chen, P. Mohapatra, Overload control in QoS-aware web servers, Computer Networks 42 (1) (2003) 119–133.
[24] L. Cherkasova, P. Phaal, Session-based admission control: a mechanism for peak load management of commercial web sites, IEEE Transactions on Computers 51 (6) (2002) 669–685.
[25] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, A. Youssef, Managing the response time for multi-tiered web applications, Tech. Rep. RC23651, IBM, November 2005.
[26] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, B. Rochwerger, Oceano – SLA-based management of a computing utility, in: IFIP/IEEE Symposium on Integrated Network Management (IM 2001), Seattle, Washington, USA, May 14–18, 2001, pp. 855–868.
[27] G. Choi, J. Kim, D. Ersoz, M. Yousif, C. Das, Exploiting NIC memory for improving cluster-based webserver performance, in: IEEE International Conference on Cluster Computing (Cluster 2005), Boston, MA, USA, September 27–30, 2005.
[28] S. Bouchenak, N.D. Palma, D. Hagimont, C. Taton, Autonomic management of clustered applications, in: International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, September 25–28, 2006.
[29] S. Ranjan, J. Rolia, H. Fu, E. Knightly, QoS-driven server migration for internet data centers, in: 10th International Workshop on Quality of Service (IWQoS 2002), Miami Beach, Florida, USA, May 15–17, 2002, pp. 3–12.
[30] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, Dynamic provisioning of multi-tier internet applications, in: 2nd International Conference on Autonomic Computing (ICAC'05), Seattle, Washington, USA, June 13–16, 2005, pp. 217–228.
[31] A. Chandra, W. Gong, P. Shenoy, Dynamic resource allocation for shared data centers using online measurements, in: 11th International Workshop on Quality of Service (IWQoS 2003), Berkeley, California, USA, June 2–4, 2003, pp. 381–400.
[32] J. Norris, K. Coleman, A. Fox, G. Candea, OnCall: defeating spikes with a free-market application cluster, in: 1st International Conference on Autonomic Computing (ICAC'04), New York, New York, USA, May 17–18, 2004, pp. 198–205.
[33] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, P. Shenoy, An observation-based approach towards self-managing web servers, in: 10th International Workshop on Quality of Service (IWQoS 2002), Miami Beach, Florida, USA, May 15–17, 2002, pp. 13–22.
[34] T. Abdelzaher, K. Shin, N. Bhatti, Performance guarantees for web server end-systems: a control-theoretical approach, IEEE Transactions on Parallel and Distributed Systems 13 (1) (2002) 80–96.
[35] M. Karlsson, C. Karamanolis, X. Zhu, An adaptive optimal controller for non-intrusive performance differentiation in computing services, in: International Conference on Control and Automation (ICCA'05), Budapest, Hungary, June 26–29, 2005.
[36] X. Liu, X. Zhu, S. Singhal, M. Arlitt, Adaptive entitlement control to resource containers on shared servers, in: 9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005), Nice, France, May 15–19, 2005.
[37] R. Doyle, J. Chase, O. Asad, W. Jin, A. Vahdat, Model-based resource provisioning in a web service utility, in: 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, USA, March 26–28, 2003.
[38] Z. Liu, M. Squillante, J. Wolf, On maximizing service-level-agreement profits, in: 3rd ACM Conference on Electronic Commerce (EC 2001), Tampa, Florida, USA, October 14–17, 2001, pp. 213–223.
[39] B. Urgaonkar, P. Shenoy, Cataclysm: handling extreme overloads in internet services, Tech. Rep. TR03-40, Department of Computer Science, University of Massachusetts, USA, December 2003.
[40] M. Crovella, R. Frangioso, M. Harchol-Balter, Connection scheduling in web servers, in: 2nd Symposium on Internet Technologies and Systems (USITS'99), Boulder, Colorado, USA, October 11–14, 1999.
[41] M. Harchol-Balter, B. Schroeder, N. Bansal, M. Agrawal, Size-based scheduling to improve web performance, ACM Transactions on Computer Systems (TOCS) 21 (2) (2003) 207–233.
[42] T. Abdelzaher, N. Bhatti, Web content adaptation to improve server overload behavior, Computer Networks 31 (11–16) (1999) 1563–1577.
[43] S. Chandra, C. Ellis, A. Vahdat, Differentiated multimedia web services using quality aware transcoding, in: IEEE INFOCOM 2000, Tel-Aviv, Israel, March 26–30, 2000, pp. 961–969.
[44] Barcelona eDragon research group. URL <http://research.ac.upc.edu/eDragon>.

Jordi Guitart received the MS and PhD degrees in Computer Science at the Technical University of Catalonia (UPC), in 1999 and 2005, respectively. Currently, he is a collaborator professor and researcher at the Computer Architecture Department of the UPC. His research interests are oriented towards the efficient execution of multithreaded Java applications (especially Java application servers) on parallel systems.

David Carrera received the MS degree at the Technical University of Catalonia (UPC) in 2002. Since then, he is working on his PhD at the Computer Architecture Department at the UPC, where he is also an assistant professor. His research interests are focused on the development of autonomic J2EE Application Servers in parallel and distributed execution platforms.



Vicenç Beltran received the MS degree at the Technical University of Catalonia (UPC) in 2004. Since then, he is working on his PhD at the Computer Architecture Department at the UPC. Currently, he is member of the research staff of the Barcelona Supercomputing Center (BSC). His research interests cover the scalability of new architectures for high performance servers.

Jordi Torres received the engineering degree in Computer Science in 1988 and the PhD in Computer Science in 1993, both from the Technical University of Catalonia (UPC) in Barcelona, Spain. Currently, he is manager for eBusiness platforms and Complex Systems activities at BSC-CNS, Barcelona Supercomputing Center. He also is Associate Professor at the Computer Architecture Department of the UPC. His research interests include applications for parallel and distributed systems, web applications, multiprocessor architectures, operating systems, tools and performance analysis and prediction tools. He has worked in a number of EU and industrial projects.

Eduard Ayguadé received the engineering degree in telecommunications in 1986 and the PhD in Computer Science in 1989, both from the Technical University of Catalonia (UPC) in Barcelona, Spain. He has been lecturing at UPC on computer organization and architecture and optimizing compilers since 1987. He has been a professor in the Department of Computer Architecture at UPC since 1997. His research interests cover the areas of processor micro-architecture and ILP exploitation, parallelizing compilers for high-performance computing systems and tools for performance analysis and visualization. He has published more than 100 papers on these topics and participated in several multiinstitutional research projects, mostly in the framework of the European Union ESPRIT and IST programs.