

Preventing Secure Web Applications Overload through Dynamic Resource Provisioning and Admission Control

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

Abstract

Overload control mechanisms such as admission control and connections differentiation have been proven effective for preventing overload of application servers running secure web applications. However, achieving optimal results in overload prevention is only possible when considering some kind of resource management 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) connections differentiation. Dynamic provisioning enables additional resources to be allocated to an application on demand to handle workload increases, while the admission control mechanism maintains the QoS of admitted requests by limiting dynamically 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 managing the resources efficiently and preventing server overload on a 4-way multiprocessor Linux hosting platform, especially when the hosting platform is fully overloaded.

Index Terms

Web servers, security, dynamic resource provisioning, self-adaptation, quality of service, overload prevention, admission control, connections differentiation.

I. INTRODUCTION

DYNAMIC web content is becoming commonplace in current web sites due to the great diffusion that e-commerce applications are experimenting in the latter days. In these applications, all information that is confidential or has market value must be carefully protected when transmitted over the open Internet. These security capabilities among network nodes over the Internet are traditionally provided using HTTPS [1]. With HTTPS, which is based on using HTTP over SSL (Secure Socket Layer [2]), mutual authentication of both the sender and receiver of messages can be performed ensuring message confidentiality. Although using security does not introduce a new degree of complexity in web applications structure, it increases the computation resources necessary to serve a connection remarkably, due to the use of cryptographic techniques, becoming a CPU-intensive workload.

The increasing load that web sites must support and the widespread diffusion of dynamic web content and SSL increase the performance demand on application servers that host the sites. At the same time, the workload of Internet applications is known to vary dynamically over multiple time scales, often in an unpredictable fashion. These factors lead sometimes these servers to overload (i.e. the volume of requests for content at a site temporarily exceeds the capacity for serving them). 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 based on the use of security, such server behavior could translate to sizable revenue losses.

Overload prevention is a critical issue in order to get a system that remains operational in the presence of overload even when the incoming request rate is several times greater than system capacity, and at the same time is able to serve the maximum the number of requests during such overload, maintaining response times (i.e. Quality of Service (QoS)) in 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 the 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 example, admission control on per request basis may lead to a large number of incomplete sessions when the system is overloaded.

Taking into account these considerations, several techniques have been proposed to face with overload, such as admission control, request scheduling, service differentiation, service degradation or resource management.

In this paper we propose a global overload control strategy for secure web applications that brings together admission control based on SSL connections differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads 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 Tomcat [3] 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 the motivation of our work. Section 3 describes the SSL protocol used to provide security capabilities when accessing web content. Sections 4 and 5 detail the implementation of our resource provisioning strategy and SSL admission control component. Section 6 describes the experimental environment used in our evaluation. Section 7 presents the evaluation results of our overload control strategy. Section 8 describes the related work and finally, Section 9 presents the conclusions of this paper.

II. MOTIVATION

In our previous work [4], we have demonstrated the benefit of using admission control based on SSL connections differentiation for preventing overload of application servers running secure web applications. However, achieving optimal results in overload prevention is only possible when considering some kind of resource management in addition to the admission control mechanism. Two issues justify the need of this resource management.

First, due to the variability of web applications workloads, the resource requirements of web applications are also variable along time. For this reason, it is desirable that web applications are able to dynamically increase their capacity by allocating more resources in order to deal with extreme overloads.

Second, it is uncommon that web applications run isolated in dedicated environments. Instead of this, they 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 (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.

Resource provisioning in a hosting platform can be based on either a dedicated or a shared model [5]. 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.

Shared model constitutes an attractive choice for hosting environments due to economic reasons of space, power, cooling and cost when hosted applications cannot support dedicated model additional cost. For instance, distributed resource management technology of IBM's WebSphere Extended Deployment [6] 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 can be complex and inefficient. For example, traditional methods to switch a server from an underloaded to an overloaded application have entailed latencies of several minutes or more, due to software installation and configuration overheads [7]. In the same way, in session-based environments, transferring session state between servers is an inefficient task. As we consider 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 very short time, we focus on SMP hosting platforms.

Literature covering resource provisioning has typically proposed two different solutions: static and dynamic provisioning of resources. Recent studies have shown the considerable benefits of dynamically reallocate resources among hosted applications based on the variations in their workloads instead of statically over-provisioning resources in a hosting platform [5], [8], [9]. The goal is to meet the applications requirements on demand and adapt to their changing resource needs. In this way, better resource utilization can be achieved and the system can react to unexpected workload increases.

However, in overloaded hosting platforms where applications compete for the available resources, overload prevention of web applications and efficient utilization of platform resources can only be achieved by using a strategy involving an accurate collaboration between dynamic resource provisioning and admission control mechanisms.

A. 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 during 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 assignation of processors. In this case, the default Linux scheduler is the responsible for dynamically distributing the available processors among the two servers according to its scheduling policy. In both cases, the admission control mechanism described in [4] is used.

Fig. 1 shows the results obtained for these two experiments. Each application server has variable input load along time, which is shown in the top part of the figure representing the number of new clients per second that hit the server¹ as a function of the time (in seconds). The figure also shows the achieved throughput for 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 1800s and 2400s), *Tomcat 1* achieves higher throughput with dynamic provisioning (bottom) than with static provisioning (middle) (240 vs. 160 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 *Tomcat 2* is over-provisioned (it has more processors allocated than needed to handle its incoming workload), denoting an inefficient processor distribution. On

¹As well as the new clients that arrive every second, the number of concurrent clients in the system includes the clients that have already initiated a session with the server and are still performing requests

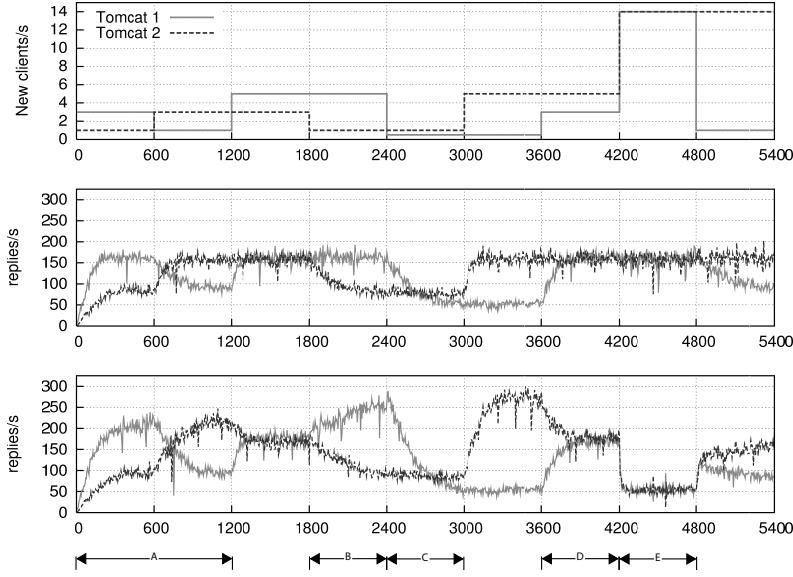


Fig. 1. Incoming workload (top) and achieved throughput of two Tomcat instances in a 4-way hosting platform with admission control and static resource provisioning (middle) and Linux default dynamic resource provisioning (bottom)

the other side, using dynamic resource provisioning allows that the processors not used by *Tomcat 2* can be used by *Tomcat 1*.

An important issue to consider when running secure web applications is the throughput degradation suffered when they are under-provisioned, as described in [10]. In order to avoid this degradation, in [4] the authors propose an admission control mechanism that limits the accepted load for each server so that it can be handled with the processors assigned to that server.

We have used this mechanism in these motivation experiments. 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 apply accurately the admission control mechanism to adapt the accepted load to the available resources 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 to prevent the throughput degradation occurred when secure web applications are under-provisioned. This effect can be appreciated in the zone labeled *E* (between 4200s and 4800s) of Fig. 1. In this zone, static provisioning (middle) allows both servers to achieve considerably higher throughput than dynamic provisioning (bottom) (160 vs. 50 replies/s).

From these experiments, we can conclude that to handle efficiently web applications in hosting platforms using dynamic resource provisioning, we must extend its functionality by fitting it within a global strategy in which the applications and the execution environment cooperate to manage the resources efficiently and prevent web applications overload.

B. Contributions

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

Our approach is based on a global resource manager responsible for distributing periodically the available resources (i.e. processors²) among web applications in a hosting platform applying a given policy (which can consider e-business indicators). We propose the cooperation between this resource manager and the applications to manage the resources, based on a bi-directional communication. On one side, the applications request to the resource manager the number of processors needed to handle their incoming load avoiding the 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, the applications can apply the 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 connections 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 to e-commerce sites based on SSL to increase the number of transactions completed, generating more revenue.

III. SSL PROTOCOL

The SSL protocol provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. To obtain these objectives it uses a combination of public-key and private-key cryptography algorithm and digital certificates (X.509).

The SSL protocol fundamentally has two phases of operation: SSL handshake and SSL record protocol. We will do an overview of the SSL handshake phase, which is the responsible for most of the computation time required when using SSL. The detailed description of the whole protocol can be found in RFC 2246 [11].

The SSL handshake phase features the negotiation between the server and the client for establishing a SSL connection. Two different SSL handshake types can be distinguished: The full SSL handshake and the resumed SSL handshake. The full SSL handshake is negotiated when a client establishes a new SSL connection with the server, and requires the complete negotiation of the SSL handshake. This negotiation includes parts that spend a lot of computation time to be accomplished. We have measured the computational demand of a full SSL handshake in a 1.4 GHz Xeon machine to be around 175 ms.

The SSL resumed handshake is negotiated when a client establishes a new HTTP connection with the server but using an existing SSL connection (it resumes a SSL connection). As the SSL session ID is reused, part of the SSL handshake negotiation can be avoided, reducing considerably the computation time for performing a resumed SSL handshake. We have measured the computational demand of a resumed SSL handshake in a 1.4 GHz Xeon machine to be around 2 ms. Notice the big difference between negotiate a full SSL handshake respect to negotiate a resumed SSL handshake (175 ms vs. 2 ms).

Based on these two handshake types, two types of SSL connections can be distinguished: the new SSL connections and the resumed SSL connections. The new SSL connections try to establish a new SSL session and must negotiate a full SSL handshake. The resumed SSL connections can negotiate a resumed SSL handshake because they provide a reusable SSL session ID (they resume an existing SSL session).

IV. RESOURCE PROVISIONING STRATEGY

Our proposal is based on a global processor manager, called eDragon CPU Manager (ECM), responsible for distributing periodically the available processors among the different applications running in a hosting platform. The ECM cooperates with the applications to manage efficiently the processors and prevent applications overload using a bi-directional communication. On one side, the applications request periodically to the ECM the number of processors needed to handle their incoming load avoiding the QoS degradation. We define the number of processors requested by application i as R_i . On the other

²As we focus on secure workloads, which are CPU-intensive, our resource provisioning strategy only considers processors

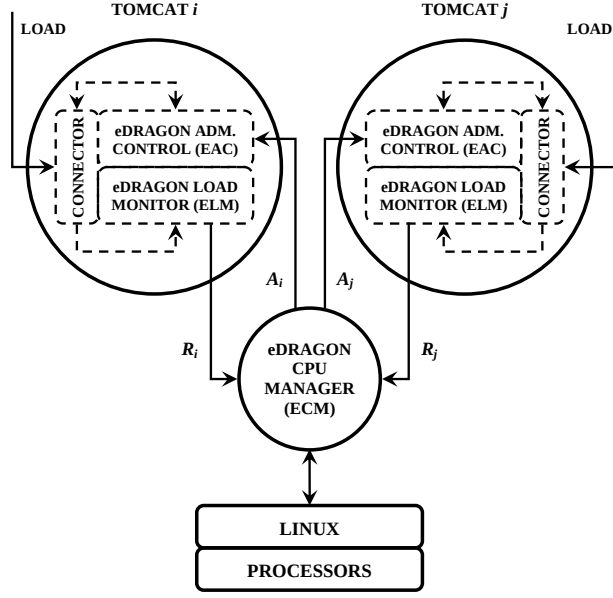


Fig. 2. Global overload control strategy

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 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 (see Section V). Fig. 2 shows a diagram describing our overload control strategy.

A. 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 Fig. 2) and the number of processors currently allocated by the ECM to each application (A_i and A_j in Fig. 2).

We have defined a new Java class, which contains the following Java methods for calling, through the Java Native Interface (JNI) [12], the ECM services for requesting and consulting processors:

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

In order to be self-managed, applications must be able to determine the number of processors they need to handle their incoming workload avoiding QoS degradation, that is R_i . Our proposal achieves this capability by adding a new component, called eDragon Load Monitor (ELM), within the server that runs each web application (see Fig. 2). At every sampling interval k_{ELM} , which is 2 seconds in current ELM implementation, the ELM calculates $R_i(k_{ELM})$ (the number of processors that the application i needs to handle its incoming workload at the interval k_{ELM}) and informs to the ECM using the `cpus_request` function from the communication path between the applications and the ECM.

For calculating $R_i(k_{ELM})$, the ELM continuously monitors the incoming secure connections to the server and performs online 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 [4]. For every sampling interval k_{ELM} , the ELM calculates the number of resumed SSL connections (defined as $O_i(k_{ELM})$) and the number of new SSL connections (defined as

$N_i(k_{ELM})$). Using these two values, the average computation time required by a resumed SSL connection (defined as CTO_i) and the average computation time required by a new SSL connection (defined as CTN_i), the ELM calculates $R_i(k_{ELM})$ in the following way:

$$R_i(k_{ELM}) = \left\lceil \frac{O_i(k_{ELM}) \times CTO_i + N_i(k_{ELM}) \times CTN_i}{k_{ELM}} \right\rceil$$

B. 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} and with value 2 seconds in the current prototype), 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.

1) *ECM scheduling policy*: 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 policies [13]. For this reason, the ECM can implement policies considering 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 usefulness of our overload prevention strategy. Nevertheless, more complex policies (see Related Work) can be incorporated to the ECM. 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 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 of the application priority class (P_i) and the number of processors in the hosting platform ($NCpus$), and inversely proportional to the total workload of the system ($\sum P_j \times R_j(k_{ECM})$), expressed as the sum of requests of all applications in the system. The complete equation is as follows:

$$A_i(k_{ECM}) = Round \left\lceil \frac{P_i \times R_i(k_{ECM}) \times NCpus}{\sum_{j=1}^N P_j \times R_j(k_{ECM})} \right\rceil$$

The scheduling policy should also allow 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). Current policy will decide to share a processor from application i to application j if the processor distribution policy has 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. 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 , a 0.5 processor may be not used. With the processor sharing mechanism, this fraction can be utilized by the application j .

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 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 be only shared from low priority applications to high priority applications, but not on the other side. 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 third multiprogrammed experiment in Section VII-B).

Additionally, the ECM performs a 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.

2) *Performance isolation*: There is another important issue that must be considered in hosting platforms where resources are shared by all the applications, because when an application overloads it can affect the performance of other applications. Consequently, it is a responsibility of the hosting platform to provide performance isolation, that is ensure that a minimal fraction of resources is available to serve requests from a certain application, and given a resource distribution among applications, an application should obtain the same performance independent of load generated by other applications. The first requirement is ensured by the ECM when allocating processors to the applications. The second requirement is accomplished by the ECM in two different ways. First, through the admission control mechanism that limits the applications accepted load so that it can be handled with the assigned processors, avoiding in this way affecting the other applications performance. The second one is necessary 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.

For this reason, 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 of the different applications do not overlap, guarantying in this way performance isolation for the CPU. For secure web applications, performance isolation for the CPU is almost equivalent to performance isolation for the whole system because these applications are CPU-intensive.

Finally, it is important to notice that it is possible that the ECM allocates a given subset of processors to an application in a given sampling interval and a different subset of processors in the next one, but this is not an issue because web applications are not especially sensitive to processor locality.

V. SSL ADMISSION CONTROL

In the previous section, we have already described how each ELM determines the application resource requirements to handle its incoming load without QoS degradation by monitoring the arrival rate of new and resumed SSL connections, trying to perform a self-managing behavior. In order to become fully self-managed applications, in this section, we describe how applications adapt their 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 that runs each web application (see Fig. 2) that applies the session-oriented adaptive mechanism described in [4] that performs admission control based on SSL connections differentiation.

This mechanism prevents applications overload by limiting the acceptance of new SSL connections to the maximum number acceptable by the application without overloading, depending on the number of processors provided by the ECM. In addition, it prioritizes the acceptance of client connections that resume an existing SSL session, in order to increase the probability for a client to complete a session, maximizing in this way the number of sessions successfully completed.

In order to prioritize the resumed SSL connections, the EAC accepts all the connections that supply a valid SSL session ID. The required verification to differentiate resumed SSL connections from new SSL connections is performed with the differentiation mechanism described in [4].

Overload is prevented and peak throughput is achieved by keeping the maximum amount of load just below the system capacity. As it has been demonstrated in [10], this is extremely important, because

reaching an overloaded state when running secure web applications causes the throughput to rapidly degrade and the response time to increase. For secure web applications, the system capacity depends on the available processors, as concluded in [10], due to the great computational demand of this kind of applications. Therefore, if the application can use more processors, it can accept more SSL connections without overloading.

The EAC calculates at every sampling interval k_{EAC} (currently defined in 2 seconds), the maximum number of new SSL connections that can be accepted by application i during this interval without overloading. We define this value as $AN_i(k_{EAC})$. This maximum depends on the computation time required by the already accepted resumed SSL connections (which can be calculated using previous definitions as $O_i(k_{EAC}) \times CTO_i$) and the number of processors allocated by the ECM to the application (previously defined as $A_i(k_{EAC})$ and that can be requested to the ECM using the `cpus_assigned` function from the communication path between the applications and the ECM). Considering these two values, the EAC can calculate the remaining computational capacity for attending new SSL connections and therefore the maximum number of new SSL connections that can be accepted, in the following way:

$$AN_i(k_{EAC}) = \left\lfloor \frac{k_{EAC} \times A_i(k_{EAC}) - O_i(k_{EAC}) \times CTO_i}{CTN_i} \right\rfloor$$

The EAC will only accept the maximum number of new SSL connections that do not overload the server ($AN_i(k_{EAC})$) (they can be served with the available computational capacity without degrading their QoS). The rest of new SSL connections arriving at the server will be refused.

VI. EXPERIMENTAL ENVIRONMENT

We use Tomcat v5.0.19 [3] 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, 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 setting the maximum number of `HttpProcessors` to 100 and the connection persistence timeout to 10 seconds.

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [14] benchmark servlets version 1.4.2 on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS defines 27 interactions. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 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 `Httpperf` [15]. This tool allows the creation of a continuous flow of HTTP/S requests issued from one or more client machines and processed by one server machine. We use `Httpperf` instead of the RUBiS client emulator because `Httpperf` can generate higher loads. One of the parameters of the tool represents the number of new clients per second initiating an interaction with the server. Each emulated client makes several requests using a persistent HTTP/S connection with the server. The workload distribution generated by `Httpperf` was extracted from the RUBiS client emulator. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. This emulates the "thinking" period of a real client who takes a period of time before clicking on the next request. The think time is generated from a negative exponential distribution with a mean of 7 seconds. `Httpperf` allows also configuring a client timeout. If this timeout is elapsed and no reply has been received from the server, the current persistent connection with the server is discarded, and a new emulated client is initiated. We have configured `Httpperf` setting the client timeout value to 10 seconds.

The hosting platform is a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We use MySQL v4.0.18 [16] as our database server with the MM.MySQL v3.0.8 JDBC driver. MySQL runs on a 2-way Intel

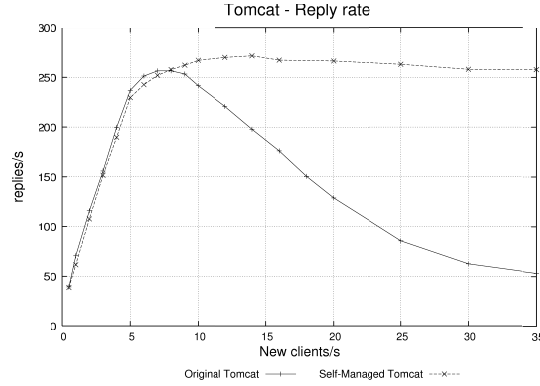


Fig. 3. Original Tomcat vs. self-managed Tomcat throughput

XEON 2.4 GHz with 2 GB RAM. We have also a 2-way Intel XEON 2.4 GHz with 2 GB RAM machine running the workload generator (Httpperf 0.8). Each client emulation machine emulates the configured number of clients performing requests to the server using the browsing mix (read-only interactions). 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 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.

VII. EVALUATION

In this section we present the evaluation results of our proposal divided in 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) for estimating 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 that 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.

A. 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 in the environment previously described with different client workloads. In every experiment, the server receives client requests during 10 minutes and after that we measure some average metrics. Fig. 3 shows the Tomcat average throughput as a function of the number of new clients per second initiating a session 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 than the original Tomcat. When the number of clients that overloads the server has been achieved, the original Tomcat throughput begins to degrade while the number of clients increases until approximately the 20% of the maximum achievable throughput when the server is fully overloaded. This degradation is avoided by the self-managed Tomcat by applying the admission control mechanism, maintaining the throughput in the maximum achievable throughput, as shown in Fig. 3.

As well as degrading the server throughput, the server overload also affects to the server response time, as shown in Fig. 4. This figure shows the Tomcat average response time as a function of the number of new clients per second initiating a session 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

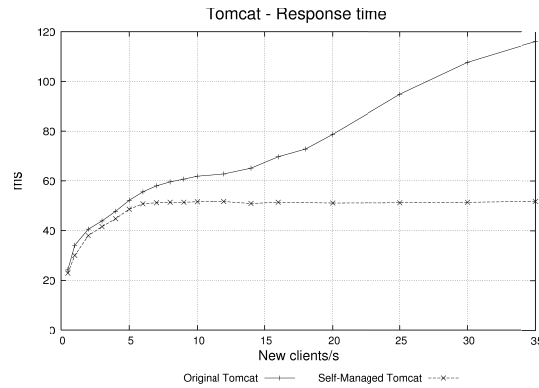


Fig. 4. Original Tomcat vs. self-managed Tomcat response time

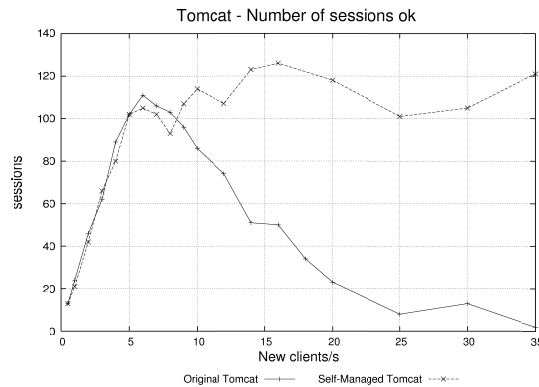


Fig. 5. Original Tomcat vs. self-managed Tomcat number of sessions completed successfully

increases while the number of clients increases. On the other side, self-managed Tomcat can maintain the response time in levels that guarantee a good quality of service to the clients, even when the number of clients that would overload the server has been achieved, as shown in Fig. 4.

Server overload has another undesirable effect, especially in e-commerce environments where session completion is a key factor. As shown in Fig. 5, which shows the number of sessions completed successfully comparing the original Tomcat server with respect to the self-managed Tomcat server, when the original Tomcat is overloaded only a few sessions can finalize completely. 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 side, self-managed Tomcat can maintain the number of sessions that can finalize completely, even when the number of clients that would overload the server has been achieved.

Notice that results in Fig. 5 have some variability. This is produced because RUBiS' sessions are in general very long, while our experiments last only 10 minutes, thus the number of completed sessions is relatively low, being sensitive to variability. Although having longer experiments will reduce the results variability, we consider that our evaluation is enough to derive the appropriate conclusions.

Finally, Fig. 6 shows the average number of processors allocated to the server during the execution as a function of the number of new clients per second 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 application 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 calculate accurately its processor requirements and communicate them to the hosting platform, which can dynamically allocate to the server only the required processors, as shown in Fig. 6, avoiding processor under-utilization but

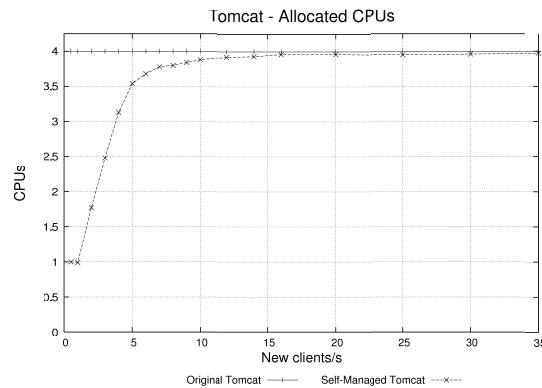


Fig. 6. Original Tomcat vs. self-managed Tomcat average number of allocated processors

ensuring performance.

B. 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 during 90 minutes in a 4-way hosting platform. Each Tomcat instance has variable input load along time, which is shown in the top part of Fig. 7 representing 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 two Tomcat instances in a hosting platform. For example, as shown in Fig. 3, when the total workload in the hosting platform is lower or equal than 5 new clients/s, the hosting platform is not overloaded, that is it can satisfy the processor requirements of all the application instances. In Fig. 7 this occurs in the zones labeled A (between 0s and 1200s) and C (between 2400s and 3000s). The rest of the time, the two Tomcat instances requirements 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 10 new clients/s), as for instance in the zones labeled B (between 1800s and 2400s) and D (between 3600s and 4200s); or completely overloaded, provoking a severe performance degradation if overload is uncontrolled, as for instance in the zone labeled E (between 4200s and 4800s).

As well as the input load along time, Fig. 7 also shows the processors allocation for each Tomcat instance (bottom part) and the throughput achieved with these processors allocations (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, as the two instances have the same priority, the ECM distributes the available processors depending only on each Tomcat request, which depend on each Tomcat input load. Therefore, the Tomcat instance with higher input load (that is, with more processor requirements) is receiving more processors and hence achieving higher throughput. For example, in the zone labeled B (between 1800s and 2400s), 5 new clients per second arrive to *Tomcat 1* while to *Tomcat 2* arrives only 1 new client per second. 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 around 260 replies/s) while *Tomcat 2* receives only 0.5 processors on average (achieving a throughput around 50 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 3600s and 4200s), 5 new clients per second arrive to *Tomcat 2* while to *Tomcat 1* arrive only 3 new clients per second. In this case, input load from *Tomcat 2*

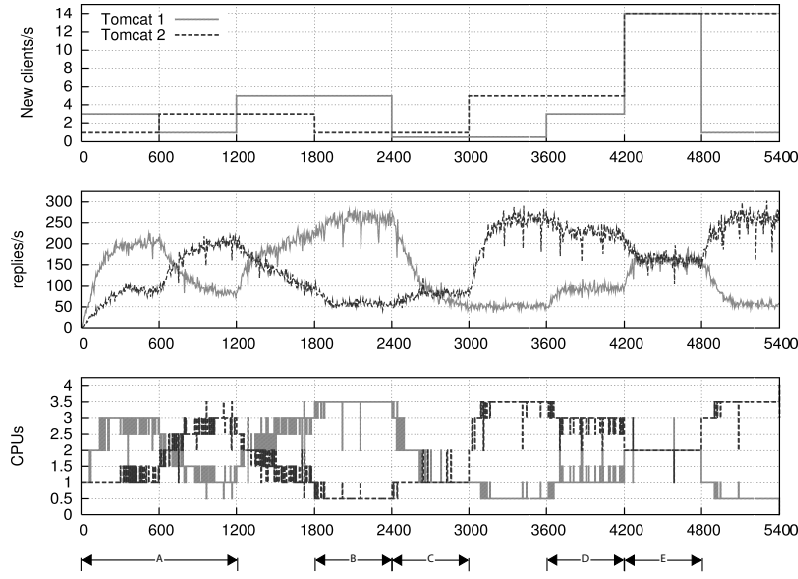


Fig. 7. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances in a 4-way Linux hosting platform with ECM if they have the same priority

is higher than input load from *Tomcat 1*, thus *Tomcat 2* will receive more processors than *Tomcat 1*. In particular, *Tomcat 2* receives 3 processors on average (achieving a throughput around 230 replies/s) while *Tomcat 1* receives only 1 processor on average (achieving a throughput around 50 replies/s).

Finally, when the input load is the same for *Tomcat 1* and *Tomcat 2* (for instance in the zone labeled *E* (between 4200s and 4800s)), the two instances receive the same number of processors (two in this case), obtaining the same throughput (around 150 replies/s).

Notice that, in any case, the EAC ensures that although the number of required processors is not supplied, the QoS of admitted requests is maintained, as demonstrated in Section VII-A. This cannot be guaranteed by default Linux dynamic provisioning strategy, and for this reason, our proposal achieves considerable higher throughput when the hosting platform is fully overloaded, as for instance in zone labeled *E* (compare middle part of Fig. 7 vs. bottom part of Fig. 1).

Our second multiprogrammed experiment has the same configuration that the previous one but, in this case, *Tomcat 1* has higher priority than *Tomcat 2* (2 vs. 1). As the two instances have different priority, the ECM distributes the available processors depending on each Tomcat request and on its priority, following the equation presented in Section IV-B.1. Fig. 8 shows the results obtained for this experiment presenting these results in the same way as Fig. 7. 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 2 and 2.5 on average, while processors allocated to *Tomcat 1* have increased, oscillating between 1.5 and 2 on average, because although *Tomcat 2* has 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* receives now more processors than *Tomcat 2* (3 vs. 1), because *Tomcat 1* has higher priority than *Tomcat 2*. With this processor allocation, *Tomcat 1* obtains higher throughput than *Tomcat 2* (around 230 replies/s vs. 100 replies/s).

Again, the EAC prevents the throughput degradation 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 middle part of Fig. 8 vs. bottom part of Fig. 1).

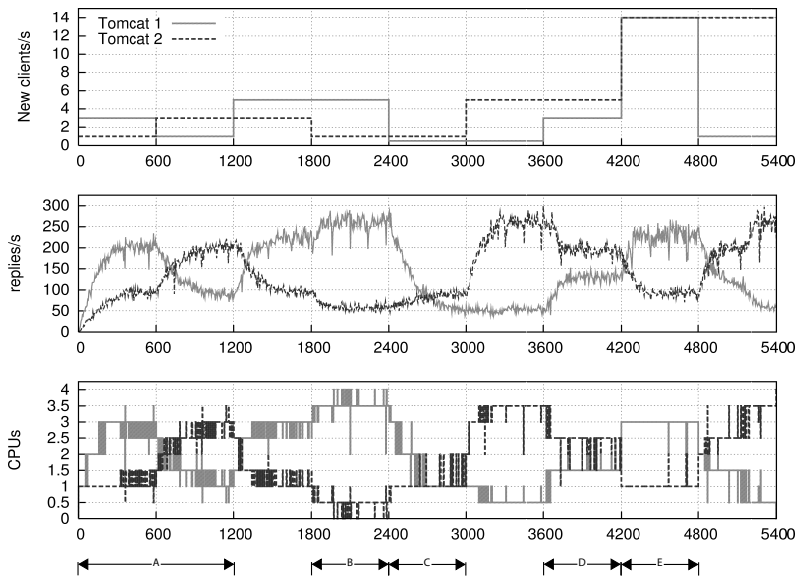


Fig. 8. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances in a 4-way Linux hosting platform with ECM if *Tomcat 1* has higher priority than *Tomcat 2*

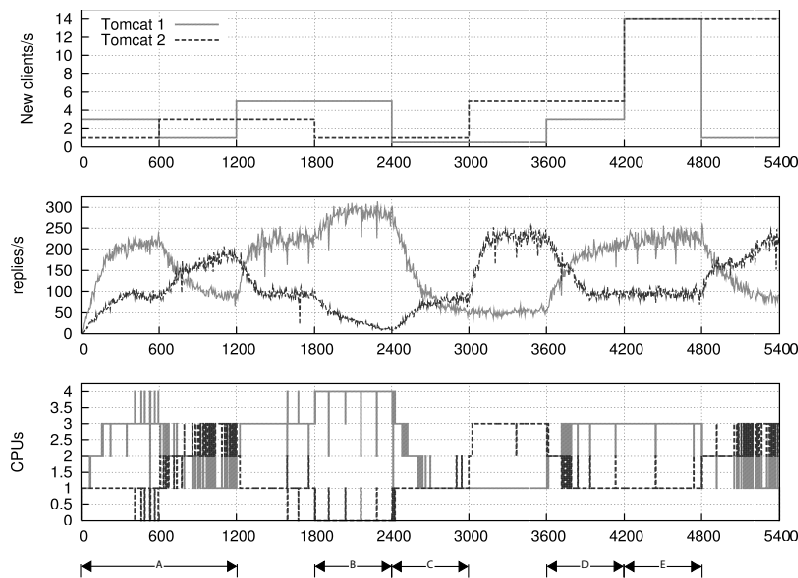


Fig. 9. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances in a 4-way Linux hosting platform with ECM if *Tomcat 1* has higher priority than *Tomcat 2* and *Tomcat 1* does not share processors with *Tomcat 2*

In our last multiprogrammed experiment, we have the same configuration that 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 on the other side. Fig. 9 shows the results obtained for this experiment presenting these results in the same way as Fig. 7. As shown in this figure, in the zone labeled *B*, processors allocated to *Tomcat 1* have 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

higher input load, *Tomcat 1* has higher priority than *Tomcat 2* and does not share processors. With this processor allocation, *Tomcat 1* obtains now higher throughput than in the previous experiment (around 200 replies/s vs. 130 replies/s) while *Tomcat 2* achieves now lower throughput (around 100 replies/s vs. 200 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.

VIII. RELATED WORK

The effect of overload on web applications has been covered in several works, applying different perspectives in order to prevent these effects. These different approaches can be resumed on request scheduling, admission control, service differentiation, service degradation, resource management and almost any combination of them.

Request scheduling refers to the order in which concurrent requests should be served. Typically, servers have been left this ordination to the operating system. But, as it is well know from queuing theory that shortest remaining processing time first (SRPT) scheduling minimizes queuing time (and therefore the average response time), some proposals [17], [18] 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 been proven effective in providing significantly better response time to high priority requests at 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 [19]–[22], for example in the form on providing smaller content (e.g. lower resolution images).

Admission control is based on reducing the amount of work the server accepts when it is faced with overload. Service differentiation is based on differentiating classes of customers so that response times of preferred clients do not suffer in the presence of overload. Admission control and service differentiation have been combined in some works to prevent server overload. For example, ACES [23] attempts to limit the number of admitted requests based on estimated service times, allowing also service prioritization. The evaluation of this approach is done based only on simulation. Other works have considered dynamic web content. An adaptive approach to overload control in the context of the SEDA [24] Web server is described in [22]. SEDA decomposes services into multiple stages, each one of which can perform admission control based on monitoring the response time through the stage. The evaluation includes dynamic content in the form of a web-based email service. In [25], the authors present an admission control mechanism for e-commerce sites that externally observes execution costs of requests, distinguishing different requests types. Yaksha [26] implements a self-tuning proportional integral controller for admission control in multi-tier e-commerce applications using a single queue model. Quorum [27] 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 [28], the authors propose a scheme for autonomous performance control of Web applications. They use a queueing 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.

It is important to notice that these admission control strategies can indirectly also be resource allocation strategies, because if a given application admits more requests, this will allocate more resources for that application if the system is not at its maximum capacity. However, when the system is overloaded, the different applications compete for the available resources. In this case, admitting more requests does not necessarily imply the allocation of more resources. For this reason, in this situation, some manager that decides the distribution of the available resources among the applications is required.

Recent studies [5], [8], [9] 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 on: control theoretic approaches with a feedback element [29]–[31], open-loop approaches based on queuing models to achieve resource guarantees [32]–[34] and observation-based approaches that use runtime measurements to compute the relationship between resources and QoS goal [35]. 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.

Depending on the hosting platform architecture considered, resource management in a single machine has been covered in [31], [36]. The first work proposes resource containers as an operating system abstraction that embodies a resource, while the second one describes 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 [7], [37] by allocating entire machines (dedicated model) and in [6], [32], [35], [38] by sharing node resources among multiple applications (shared model).

Cataclysm [21] performs overload control bringing together admission control, adaptive service degradation and dynamic provisioning of platform resources, demonstrating that the most effective way to handle overload must consider the combination of techniques. In this aspect, this work is similar to our proposal.

On most of the prior work, overload control is performed on per request basis, which may not be adequate for many session-based applications, such as e-commerce applications. A session-based admission control scheme has been reported in [39]. This approach allows sessions to run to completion even under overload, denying all access when the server load exceeds a predefined threshold. Another approach to session-based admission control based on characterization of a commercial web server log, discriminating the scheduling of requests based on the probability of completion of the session that the requests belong to is presented in [40].

Our proposal combines important aspects that previous work has considered in isolation or simply has ignored. First, we consider dynamic web content instead of simpler static web content. Second, we focus on session-based applications considering the particularities of these applications when performing admission control. Third, we combine several techniques as admission control, service differentiation and dynamic resource provisioning that have been demonstrated to be useful to prevent overload [21] instead of considering each technique in isolation. Fourth, our proposal is fully adaptive to the available resources and to the number of connections in the server instead of using predefined thresholds. Fifth, our resource provisioning mechanism incorporates e-business indicators instead of only considering conventional performance metrics such as response time and throughput. Finally, we consider overload control on secure web applications while none of the above works has covered this issue.

Although none of them has covered overload control, the influence of security on application server scalability has been covered in some works. For example, the performance and architectural impact of SSL on the servers in terms of various parameters such as throughput, utilization, cache sizes and cache miss ratios has been analyzed in [41], concluding that SSL increases computational cost of transactions by a factor of 5-7. The impact of each individual operation of TLS protocol in the context of web servers has been studied in [42], showing that key exchange is the slowest operation in the protocol.

IX. CONCLUSION

In this paper we have proposed an overload control strategy for secure web applications that brings together admission control based on SSL connections 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 distributing periodically the available processors among web applications following a determined policy. The resource manager can be

configured to implement different policies, considering 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 totally transparent to the user manner, using a bi-directional communication. On one side, the applications request to 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, 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 connections 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 to e-commerce sites based on SSL to increase the number of transactions completed, generating more benefit.

Our evaluation demonstrates the benefit of our approach for managing the resources efficiently 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 Tomcat application server, the proposed overload control strategy can be applied on any other server.

Our future work goes toward the extension of our proposal in order to support very heterogeneous applications in the hosting platform, as well as improving 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 faces effectively 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 contract TIN2004-07739-C02-01 and by the BSC (Barcelona Supercomputing Center). For additional information about the authors, visit the Barcelona eDragon Research Group web site [43].

REFERENCES

- [1] E. Rescorla, *HTTP over TLS. RFC 2818*, May 2000.
- [2] A. O. Freier, P. Karlton, and C. Kocher, "The SSL Protocol. Version 3.0. November 1996." [Online]. Available: <http://wp.netscape.com/eng/ssl3/ssl-toc.html>
- [3] Jakarta Project. Apache Software Foundation, *Jakarta Tomcat Servlet Container*. [Online]. Available: <http://jakarta.apache.org/tomcat>
- [4] J. Guitart, D. Carrera, V. Beltran, J. Torres, and 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.
- [5] A. Andrzejak, M. Arlitt, and J. Rolia, "Bounding the Resource Savings of Utility Computing Models," HP Labs, Tech. Rep. HPL-2002-339, December 2002.
- [6] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef, "Managing the Response Time for Multi-tiered Web Applications," IBM, Tech. Rep. RC23651, November 2005.
- [7] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and 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.
- [8] A. Chandra, P. Goyal, and 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.
- [9] A. Chandra and P. Shenoy, "Effectiveness of Dynamic Resource Allocation for Handling Internet Flash Crowds," Department of Computer Science, University of Massachusetts, USA, Tech. Rep. TR03-37, November 2003.
- [10] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé, "Characterizing Secure Dynamic Web Applications Scalability," in *19th International Parallel and Distributed Symposium (IPDPS'05)*, Denver, Colorado, USA, April 4–8, 2005.
- [11] T. Dierks, "The TLS Protocol, Version 1.0. RFC 2246," January 1999.
- [12] Sun Microsystems, "Java Native Interface (JNI)." [Online]. Available: <http://java.sun.com/j2se/1.5.0/docs/guide/jni/>

- [13] D. Menasce, V. Almeida, R. Fonseca, and M. Mendes, "Business-Oriented Resource Management Policies for e-Commerce Servers," *Performance Evaluation*, vol. 42, no. 2–3, pp. 223–239, September 2000.
- [14] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and 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.
- [15] D. Mosberger and T. Jin, "httpperf: 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.
- [16] MySQL. [Online]. Available: <http://www.mysql.com/>
- [17] M. Crovella, R. Frangioso, and 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.
- [18] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based Scheduling to Improve Web Performance," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 207–233, May 2003.
- [19] T. Abdelzaher and N. Bhatti, "Web Content Adaptation to Improve Server Overload Behavior," *Computer Networks*, vol. 31, no. 11–16, pp. 1563–1577, May 1999.
- [20] S. Chandra, C. Ellis, and A. Vahdat, "Differentiated Multimedia Web Services using Quality Aware Transcoding," in *IEEE INFOCOM 2000, Tel-Aviv, Israel*, March 26–30, 2000, pp. 961–969.
- [21] B. Urgaonkar and P. Shenoy, "Cataclysm: Handling Extreme Overloads in Internet Services," Department of Computer Science, University of Massachusetts, USA, Tech. Rep. TR03-40, December 2003.
- [22] M. Welsh and 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.
- [23] X. Chen, H. Chen, and P. Mohapatra, "ACES: An Efficient Admission Control Scheme for QoS-Aware Web Servers," *Computer Communications*, vol. 26, no. 14, pp. 1581–1593, September 2003.
- [24] M. Welsh, D. Culler, and 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.
- [25] S. Elnikety, E. Nahum, J. Tracey, and 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, New York, USA, May 17–22, 2004, pp. 276–286.
- [26] A. Kamra, V. Misra, and 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.
- [27] J. Blanquer, A. Batchelli, K. Schausser, and 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.
- [28] X. Liu, J. Heo, L. Sha, and 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.
- [29] T. Abdelzaher, K. Shin, and N. Bhatti, "Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 80–96, January 2002.
- [30] M. Karlsson, C. Karamanolis, and 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.
- [31] X. Liu, X. Zhu, S. Singhal, and 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.
- [32] A. Chandra, W. Gong, and 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.
- [33] R. Doyle, J. Chase, O. Asad, W. Jin, and 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.
- [34] Z. Liu, M. Squillante, and 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.
- [35] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and 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.
- [36] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana, USA, February 22–25, 1999, pp. 45–58.
- [37] S. Ranjan, J. Rolia, H. Fu, and 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.
- [38] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource Overbooking and Application Profiling in Shared Hosting Platforms," in *5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, Massachusetts, USA, December 9–11, 2002.
- [39] L. Cherkasova and P. Phaal, "Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 669–685, June 2002.
- [40] H. Chen and P. Mohapatra, "Overload Control in QoS-aware Web Servers," *Computer Networks*, vol. 42, no. 1, pp. 119–133, May 2003.
- [41] K. Kant, R. Iyer, and P. Mohapatra, "Architectural Impact of Secure Socket Layer on Internet Servers," in *2000 IEEE International Conference on Computer Design (ICCD'00)*, Austin, Texas, USA, September 17–20, 2000, pp. 7–14.
- [42] C. Coarfa, P. Druschel, and D. Wallach, "Performance Analysis of TLS Web Servers," in *9th Network and Distributed System Security Symposium (NDSS'02)*, San Diego, California, USA, February 6–8, 2002.
- [43] Barcelona eDragon Research Group. [Online]. Available: <http://www.bsc.es/eDragon>