Session-Based Adaptive Overload Control for Secure Dynamic Web Applications

Jordi Guitart, David Carrera, Vicenç Beltran, Jordi Torres and Eduard Ayguadé
European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department - Technical University of Catalonia
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034 Barcelona (Spain)
{jguitart, dcarrera, vbeltran, torres, eduard}@ac.upc.edu

Abstract

As dynamic web content and security capabilities are becoming popular in current web sites, the performance demand on application servers that host the sites is increasing, leading sometimes these servers to overload. As a result, response times may grow to unacceptable levels and the server may saturate or even crash. In this paper we present a session-based adaptive overload control mechanism based on SSL (Secure Socket Layer) connections differentiation and admission control. The SSL connections differentiation is a key factor because the cost of establishing a new SSL connection is much greater than establishing a resumed SSL connection (it reuses an existing SSL session on server). Considering this big difference, we have implemented an admission control algorithm that prioritizes the resumed SSL connections to maximize performance on session-based environments and limits dynamically the number of new SSL connections accepted depending on the available resources and the current number of connections in the system to avoid server overload. In order to allow the differentiation of resumed SSL connections from new SSL connections we propose a possible extension of the Java Secure Sockets Extension (JSSE) API. Our evaluation on Tomcat server demonstrates the benefit of our proposal for preventing server overload.

1. Introduction

Current web sites have to face three issues to keep clients satisfied. First, the web community is growing day after day, increasing exponentially the load that sites must support. Second, current sites are subject to enormous variations in demand, often in an unpredictable fashion, including flash crowds that cannot be processed. Third, dynamic web content is becoming popular on current sites. At the same time, all 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 [21]. With HTTPS, which is based on using HTTP over SSL (Secure Socket Layer [13]), you can perform mutual authentication of both the sender and receiver of messages and ensure message confidentiality. This process involves X.509 digital certificates that are configured on both sides of the connection. This widespread diffusion of dynamic web content and SSL increases the performance demand on application servers that host the sites, leading sometimes these servers to overload (i.e. the volume of requests for content at a site temporarily exceeds the capacity for serving them 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 based on the use of security, such server behavior could translate to sizable revenue losses. For instance, [26] estimates that between 10 and 25% of e-commerce transactions are aborted because of slow response times, which translates to about 1.9 billion dollars in lost revenue.

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 in acceptable levels. With these objectives, several mechanisms have been proposed to face with overload, such as admission control, request scheduling, service differentiation, service degradation or resource management.

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 broken or incomplete sessions when the system is overloaded.

In this paper we present an overload control mechanism based on SSL connections differentiation and admission control. First, we propose a possible extension of the Java Secure Sockets Extension (JSSE) API [22], which implements a Java version of the SSL protocol, to allow SSL connections differentiation depending on if the connection will reuse an existing SSL connection on the server or not. The SSL connections differentiation can be very useful in order to design intelligent overload control policies on server, given the big difference existing on the computational demand of new SSL connections versus resumed SSL connections. This differentiation is done with not significant additional cost. Second, we propose a session-based adaptive admission control mechanism for the Tomcat application server. This mechanism will allow the server to avoid throughput degradation and response time increments produced with SSL connections on server saturation, increasing the performance with good quality of service. Moreover, the admission control mechanism will maximize the number of sessions completed successfully, allowing to e-commerce sites based on SSL to increase the number of transactions completed, generating higher benefit.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 introduces security for Java web applications, describing the SSL protocol and its implementation for Java. Sections 4 and 5 detail the implementation of our SSL connections differentiation and SSL admission control mechanisms. Section 6 describes the experimental environment used in our evaluation. Section 7 presents the evaluation results of the overload control mechanism and finally, Section 8 presents the conclusions of this paper.

2. 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 [10][15] 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. Although scheduling can improve response times, under extreme overloads other mechanisms become indispensable. Anyway, better scheduling can always be complementary to any other mechanism.

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 [6] 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 Web server is described in [25]. 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 [12], the authors present an admission control mechanism for e-commerce sites that externally observes execution costs of requests, distinguishing different requests types. Yaksha [17] implements a self-tuning proportional integral controller for admission control in multi-tier e-commerce applications using a single queue model.

Some works have integrated the resource management with other approaches as admission control and service differentiation. For example, [3] proposes resource containers as an operating system abstraction that embodies a resource. [24] proposes a resource overbooking based scheme for maximizing

revenue generated by the available resources in a shared platform. [5] presents a prototype data center implementation used to study the effectiveness of dynamic resource allocation for handling flash crowds. Cataclysm [23] performs overload control bringing together admission control, service degradation and dynamic provisioning of platform resources.

Service degradation is based on avoiding refusing clients as a response to overload but reducing the service offered to clients [1][23][25], for example in the form on providing smaller content (e.g. lower resolution images).

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 [8]. 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 [7].

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, our proposal is fully adaptive to the available resources and to the number of connections in the server instead of using predefined thresholds. 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 servers 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 [18]. The impact of each individual operation of TLS protocol in the context of web servers has been studied in [9], showing that key exchange is the slowest operation in the protocol.

3. Security for Java Web Applications

3.1 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 does not introduce a new degree of complexity in web applications structure because it works almost transparently on top of the socket layer. However, SSL increases the computation time necessary to serve a connection remarkably, due to the use of cryptography to achieve their objectives. This increment has a noticeable impact on server performance, which has been evaluated in [14]. This study concludes that the maximum throughput obtained when using SSL connections is 7 times lower than when using normal connections. The study also notices that when the server is attending non-secure connections and saturates, it can maintain the throughput if new clients arrive, while if attending SSL connections, the saturation of the server provokes the degradation of the throughput.

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 of 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 allows the server to authenticate itself to the client using public-key techniques like RSA, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server. 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, including 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 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. 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 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).

3.2 JSSE API Limitations

The Java Secure Socket Extension (JSSE) [22] is a set of packages that enable secure Internet communications. It implements a Java technology version of Secure Sockets Layer (SSL) [13] and Transport Layer Security (TLS) [11] protocols.

The JSSE API provides the `SSLSocket` and `SSLServerSocket` classes, which can be instantiated to create secure channels. The JSSE API supports the initiation of a handshake on a SSL connection in one of three ways. Calling `startHandshake` that explicitly begins handshakes, or any attempt to read or write application data through the connection causes an implicit handshake, or a call to `getSession` tries to set up a session if there is no currently valid session, and an implicit handshake is done. After handshaking has completed, session attributes can be accessed by using the `getSession` method. If handshaking fails for any reason, the `SSLSocket` is closed, and no further communications can be done.

Notice that the JSSE API does not support any way to consult if an incoming SSL connection provides a reusable SSL session ID until the handshake is fully completed. Having this information prior to handshake negotiation could be very useful for example for servers in order to do overload control based on SSL connections differentiation, given the big difference existing on the computational demand of new SSL connections versus resumed SSL connections. It is important to notice that the verification about an incoming SSL connection provides a valid SSL session ID is already performed by the JSSE API prior handshaking in order to negotiate a full SSL handshake or a resumed SSL handshake. Therefore, the addition of a new interface to access this information would not involve additional cost.

4. SSL Connections Differentiation

As we mentioned in the previous section, there is no way in JSSE packages to consult if an incoming SSL connection provides a reusable SSL session ID until the handshake is fully completed. We propose the extension of the JSSE API to allow applications to differentiate new SSL connections from resumed SSL connections prior the handshaking has started.

This new feature can be useful in many scenarios. For example, a connection scheduling policy based on prioritizing the resumed SSL connections (that is, the short connections) will result in a reduction of the average response time, as described in previous works with static web content using the SRPT scheduling [10][15]. Moreover, prioritizing the resumed SSL connections will increase the probability for a client to complete a session, maximizing the number of sessions completed successfully. We have already commented the importance of this metric in e-commerce environments. Remember that the higher the number of sessions completed the higher the amount of revenue that is likely to be generated. In addition, a server could limit the number of new SSL connections that it accepts, in order to avoid throughput degradation produced if server overloads.

In order to evaluate the advantages of being able to differentiate new SSL connections from resumed SSL connections and the convenience of adding this functionality to the standard JSSE API, we have implemented an experimental mechanism that allows this differentiation prior to the handshake negotiation. We have measured that this mechanism does not suppose significant additional cost. The mechanism works at system level and it is based on examining the contents of the first TCP segment received on the server after the connection establishment.

After a new connection is established between the server and a client, the SSL protocol starts a handshake negotiation. The protocol begins with the client sending a SSL ClientHello message (see the RFC 2246 for more details) to the server. This message can include a SSL session ID from a previous connection if the SSL session wants to be reused. This message is sent in the first TCP segment that the client sends to the server. The implemented mechanism checks the value of this SSL message field to decide if the connection is a resumed SSL connection or a new one instead.

The mechanism operation begins when a new incoming connection is accepted by the Tomcat server, and a socket structure is created to represent the connection in the operating system as well as in the JVM. After establishing the connection but prior to the handshake negotiation, the Tomcat server requests to the mechanism the classification of this SSL connection, using a JNI native library that is loaded into the JVM process. The library translates the Java request into a new native system call implemented in the Linux kernel using a Linux kernel module. The implementation of the system call calculates a hash

function from the parameters provided by the Tomcat server (local and remote IP address and TCP port) which produces a socket hash code that makes possible to find the socket inside of a connection established socket hash table. When the system `struct sock` that represents the socket is located and in consequence all the received TCP segments for that socket after the connection establishment, the first one of the TCP segments is interpreted as a SSL ClientHello message. If this message contains a SSL session ID with value 0, it can be concluded that the connection tries to establish a new SSL session. If a non-zero SSL session ID is found instead, the connection tries to resume a previous SSL session. The value of this SSL message field is returned by the system call to the JNI native library that, in turn, returns it to the Tomcat server. With this result, the server can decide, for instance, to apply an admission control algorithm in order to decide if the connection should be accepted or rejected.

5. SSL Admission Control

In order to prevent server overload in secure environments, we have incorporated to the Tomcat server a session-oriented adaptive mechanism that performs admission control based on SSL connections differentiation. This mechanism has been developed with two objectives. First, to prioritize the acceptation of client connections that resume an existing SSL session, in order to maximize the number of sessions successfully completed. Second, to limit the massive arrival of new SSL connections to the maximum number acceptable by the server before overloading, depending on the available resources.

To prioritize the resumed SSL connections, the admission control mechanism 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 mechanism described in Section 4.

To avoid the server throughput degradation and maintain acceptable response times, the admission control mechanism must to avoid the server overload. By keeping the maximum amount of load just below the system capacity, overload is prevented and peak throughput is achieved. For secure web applications, the system capacity depends on the available processors, as it has been demonstrated in [14], due to the great computational demand of this kind of applications. Therefore, if the server can use more processors, it can accept more SSL connections without saturating.

The admission control mechanism calculates periodically, introducing an adaptive behavior, the maximum number of new SSL connections that can be accepted without overloading the server. This maximum depends on the available processors for the server and the computational demand required by the accepted resumed SSL connections. The calculation of this demand is based on the number of accepted resumed SSL connections and the typical computational demand of one of these connections.

After calculating the computational demand required by the accepted resumed SSL connections and with information relative to the available processors for the server, the admission control mechanism can calculate the remaining computational capacity for attending new SSL connections. The admission control mechanism will only accept the maximum number of new SSL connections that do not overload the server (they can be served with the available computational capacity). The rest of new SSL connections arriving at the server will be refused.

Notice that if the number of resumed SSL connections increases, the server has to decrease the number of new SSL connections it accepts, in order to avoid server overload with the available processors and vice versa, if the number of resumed SSL connections decreases, the server can increase the number of new SSL connections that it accepts.

Notice that this constitutes an interesting starting point to develop autonomic computing strategies on the server in a bidirectional fashion. First, the server can restrict the number of new SSL connections it accepts to adapt its behavior to the available resources (i.e. processors) in order to prevent server overload. Second, the server can inform about its resource requirements to a global manager (which will distribute all the available resources among the existing servers following a given policy) depending on the rate of incoming connections (new SSL connections and resumed SSL connections) requesting for service.

6. Experimental Environment

6.1 Tomcat Servlet Container

We use Tomcat v5.0.19 [16] 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.

Tomcat follows a connection service schema where, at a given time, one thread (an HttpProcessor) is

responsible of accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point, this HttpProcessor will be responsible of attending and serving the received requests through the persistent connection established with the client, while another HttpProcessor will continue accepting new connections.

Persistent connections are a feature of HTTP 1.1 that allows serving different requests using the same connection, saving a lot of work and time for the web server, client and the network, considering that establishing and tearing down HTTP connections is an expensive operation. A connection timeout is programmed to close the connection if no more requests are received.

We have configured Tomcat setting the maximum number of HttpProcessors to 100 and the connection persistence timeout to 10 seconds.

6.2 Auction Site Benchmark (RUBiS)

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [2] 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. Among the most important ones are browsing items by category or region, bidding, buying or selling items and leaving comments on other users. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [19]. This tool, which supports both HTTP and HTTPS protocols, allows the creation of a continuous flow of HTTP/S requests issued from one or more client machines and processed by one server machine. One of the parameters of the tool represents the number of new clients per second initiating an interaction with the server. Each emulated client opens a session with the server. Each session is a persistent HTTP/S connection with the server. Using this connection, the client repeatedly makes a request (the client can also pipeline some requests), parses the server response to the request, and follows a link embedded in the response. The workload distribution generated by Httperf was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. The think time is generated from a negative exponential distribution with a mean of 7 seconds. Httperf 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 Httperf setting the client timeout value to 10 seconds. RUBiS defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions.

6.3 Hardware & Software Platform

Tomcat runs on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We use MySQL v4.0.18 [20] as our database server with the MM.MySQL v3.0.8 JDBC driver. MySQL runs on a 2-way Intel 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 (Httperf 0.8). Client machine emulates the configured number of clients performing requests to the server during 10 minutes 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.

7. Evaluation

In this section we present the evaluation results of the overload control mechanism on Tomcat server, comparing the results obtained with the original Tomcat.

7.1 Original Tomcat

Figure 1 shows the Tomcat throughput as a function of the number of new clients per second initiating a session with the server when running with different number of processors. Notice that for a given number of processors, 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. Notice that running with more processors allows the server to handle more clients before saturating, so the maximum achieved throughput is higher. When the

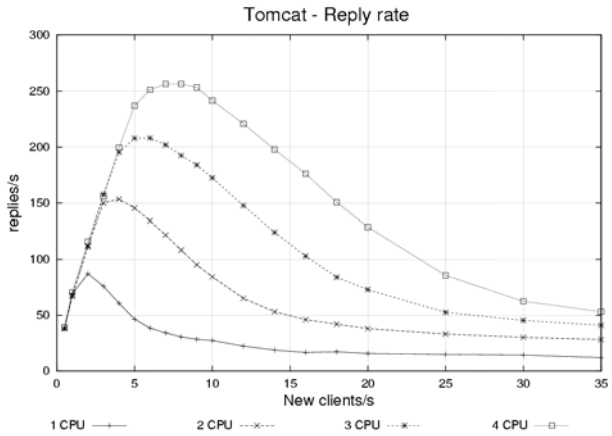

Figure 1. Original Tomcat throughput with different number of processors

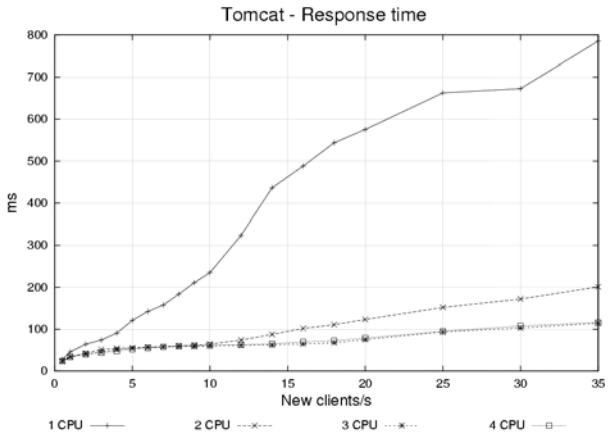

Figure 2. Original Tomcat response time with different number of processors

number of clients that overload the server has been achieved, the server throughput degrades until approximately the 20% of the maximum achievable throughput while the number of clients increases.

As well as degrading the server throughput, the server overload also affects to the server response time, as shown in Figure 2. This figure shows the server average response time as a function of the number of new clients per second initiating a session with the server when running with different number of processors. Notice that when the server is overloaded the response time increases (especially when running with one processor) while the number of clients increases.

Server overload has another undesirable effect, especially in e-commerce environments where session completion is a key factor. As shown in Figure 3, which shows the number of sessions completed successfully when running with different number of processors, when the server is overloaded only a few sessions can finalize completely. Consider the great revenue lost that this fact can provoke for example in an online store, where only a few clients can finalize the acquisition of a product.

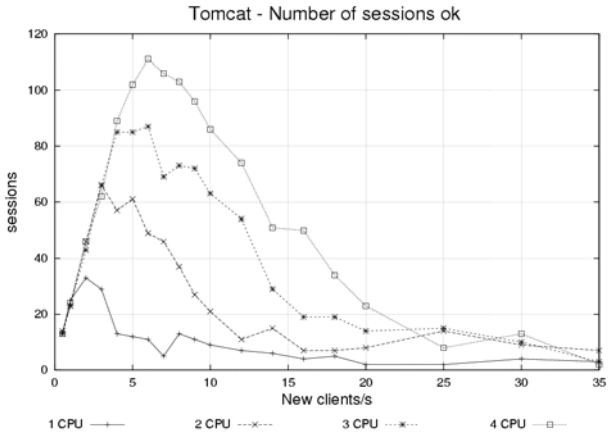

Figure 3. Completed sessions by original Tomcat with different number of processors

The cause of this great performance degradation on server overload has been analyzed in [14]. They conclude that the server throughput degrades when most of the incoming client connections must negotiate a full SSL handshake instead of resuming an existing SSL connection, requiring a computing capacity that the available processors are unable to supply. This circumstance is produced when the server is overloaded and it cannot handle the incoming requests before the client timeouts expire. In this case, clients with expired timeouts are discarded and new ones are initiated, provoking the arrival of a great amount of new client connections that need the negotiation of a full SSL handshake, provoking the server performance degradation.

Considering the described behavior, it makes sense to apply an admission control mechanism in order to improve server performance in the following way. First, to filter the massive arrival of client connections that need to negotiate a full SSL handshake that will saturate the server, avoiding the server throughput degradation and maintaining a good quality of service (good response time) for already connected clients. Second, to prioritize the acceptation of client connections that resume an existing SSL session, in order to maximize the number of sessions successfully completed.

7.2 Tomcat with Admission Control

Figure 4 shows the Tomcat throughput as a function of the number of new clients per second initiating a

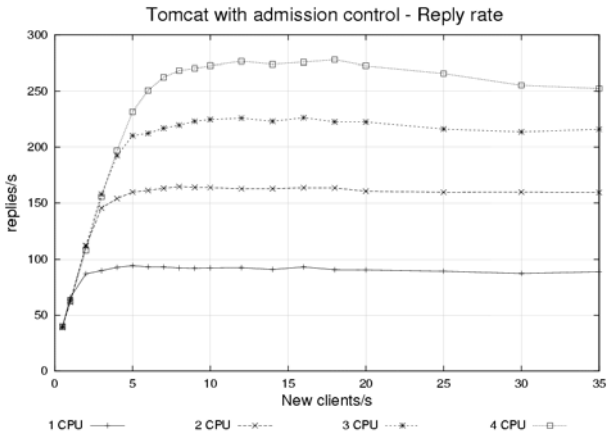

Figure 4. Tomcat with admission control throughput with different number of processors

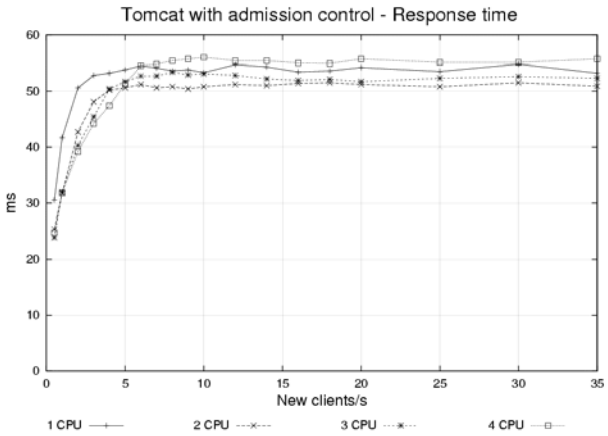

Figure 5. Tomcat with admission control response time with different number of processors

session with the server when running with different number of processors. Notice that for a given number of processors, 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 server with admission control behaves in the same way than the original server. However, when the number of clients that would overload the server has been achieved, the admission control mechanism can avoid the throughput degradation, maintaining it in the maximum achievable throughput, as shown in Figure 5. Notice that running with more processors allows the server to handle more clients, so the maximum achieved throughput is higher.

The admission control mechanism on Tomcat allows also maintaining 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 Figure 5. This figure shows the server average response time as a function of the number of new clients per second initiating a session with the server when running with different number of processors.

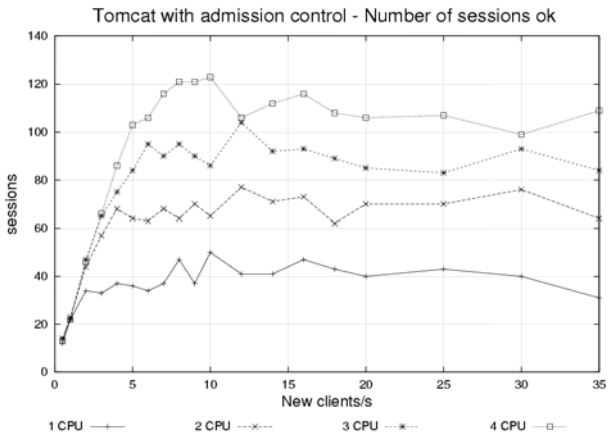

Figure 6. Sessions completed by Tomcat with admission control with different number of processors

Finally, the admission control mechanism has also a beneficial effect for session-based clients. As shown in Figure 6, which shows the number of sessions finalized successfully when running with different number of processors, the number of sessions that can finalize completely does not decrease, even when the number of clients that would overload the server has been achieved.

8. Conclusions

In this paper we have presented a session-based adaptive overload control mechanism based on SSL connections differentiation and admission control. First, we have proposed a possible extension of the JSSE API in order to allow the differentiation of resumed SSL connections (that reuse an existing SSL session on server) from new SSL connections. Second, we have incorporated to the Tomcat server a session-based adaptive admission control mechanism that prioritizes resumed SSL connections to maximize the number of sessions completed successfully (which is a very important metric on e-commerce environments). The admission control also limits dynamically the number of new SSL connections accepted depending on the available resources and the number of resumed SSL connections accepted, in order to avoid server overload.

Our evaluation demonstrates the benefit of our approach on overload prevention for servers on secure environments, and confirms that security must be

considered as an important issue that can heavily affect the scalability and performance of web applications.

9. 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 CEPBA (European Center for Parallelism of Barcelona). For additional information about the authors, please visit the Barcelona eDragon Research Group web site [4].

10. References

[1] T. Abdelzaher and N. Bhatti. *Web Content Adaptation to Improve Server Overload Behavior*. Computer Networks, Vol. 31 (11-16), pp. 1563-1577, May 1999.

[2] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. *Specification and Implementation of Dynamic Web Site Benchmarks*. IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, Texas, USA. November 25, 2002.

[3] G. Banga, P. Druschel and J. C. Mogul. *Resource Containers: A New Facility for Resource Management in Server Systems*. 3rd Symposium on Operating Systems Design and Implementation (OSDI'99), pp. 45-58, New Orleans, Louisiana, USA. February 22-25, 1999.

[4] Barcelona eDragon Research Group http://www.cepba.upc.es/eDragon

[5] A. Chandra and P. Shenoy. *Effectiveness of Dynamic Resource Allocation for Handling Internet Flash Crowds*. Technical Report TR03-37, Department of Computer Science, University of Massachusetts, USA. November 2003.

[6] X. Chen, H. Chen and P. Mohapatra. *ACES: An Efficient Admission Control Scheme for QoS-Aware Web Servers*. Computer Communications, Vol. 26 (14), pp. 1581-1593. September 2003.

[7] H. Chen and P. Mohapatra. *Overload Control in QoS-aware Web Servers*. Computer Networks, Vol. 42 (1), pp. 119-133. May 2003.

[8] 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 (6), pp. 669-685. June 2002.

[9] C. Coarfa, P. Druschel, and D. Wallach. *Performance Analysis of TLS Web Servers*. 9th Network and Distributed System Security Symposium (NDSS'02), San Diego, California, USA. February 6-8, 2002.

[10] M. Crovella, R. Frangioso and M. Harchol-Balter. *Connection Scheduling in Web Servers*. 2nd Symposium on Internet Technologies and Systems (USITS'99), Boulder, Colorado, USA. October 11-14, 1999.

[11] T. Dierks and C. Allen. *The TLS Protocol, Version 1.0*. RFC 2246. January 1999.

[12] S. Elnikety, E. Nahum, J. Tracey and W. Zwaenepoel. *A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites*. 13th International Conference on World Wide Web (WWW'04), pp. 276-286, New York, New York, USA. May 17-22, 2004.

[13] A. O. Freier, P. Karlton, and C. Kocher. *The SSL Protocol, Version 3.0*. November 1996.

[14] J. Guitart, V. Beltran, D. Carrera, J. Torres and E. Ayguadé. *Characterizing Secure Dynamic Web Applications Scalability*. 19th International Parallel and Distributed Symposium (IPDPS'05), Denver, Colorado, USA. April 4-8, 2005.

[15] 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 (2), pp. 207-233. May 2003.

[16] Jakarta Tomcat Servlet Container http://jakarta.apache.org/tomcat

[17] A. Kamra, V. Misra and E. Nahum. *Yaksha: A Controller for Managing the Performance of 3-Tiered Websites*. 12th International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada. June 7-9, 2004.

[18] K. Kant, R. Iyer, and P. Mohapatra. *Architectural Impact of Secure Socket Layer on Internet Servers*. 2000 IEEE International Conference on Computer Design (ICCD'00), pp. 7-14, Austin, Texas, USA. September 17-20, 2000.

[19] D. Mosberger and T. Jin. *httperf: A Tool for Measuring Web Server Performance*. Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), pp. 59-67. Madison, Wisconsin, USA. June 23, 1998.

[20] MySQL http://www.mysql.com

[21] E. Rescorla. *HTTP over TLS*. RFC 2818. May 2000.

[22] Sun Microsystems. Java Secure Socket Extension http://java.sun.com/products/jsse/

[23] B. Urgaonkar and P. Shenoy. *Cataclysm: Handling Extreme Overloads in Internet Services*. Technical Report TR03-40, Department of Computer Science, University of Massachusetts, USA. November 2004.

[24] B. Urgaonkar, P. Shenoy and T. Roscoe. *Resource Overbooking and Application Profiling in Shared Hosting Platforms*. 5th Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, Massachusetts, USA. December 9-11, 2002.

[25] M. Welsh and D. Culler. *Adaptive Overload Control for Busy Internet Servers*. 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, USA. March 26-28, 2003.

[26] T. Wilson. *E-Biz Bucks Lost under SSL Strain*. Internet Week Online. May 20, 1999. http://www.internetwk.com/lead/lead052099.htm