Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Computer Networks 51 (2007) 4492–4510

Computer Networks

www.elsevier.com/locate/comnet

Designing an overload control strategy for secure e-commerce applications

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

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

Received 19 October 2006; received in revised form 9 March 2007; accepted 29 May 2007
Available online 6 July 2007

Responsible Editor: E. Cohen

Abstract

Uncontrolled overload can lead e-commerce applications to considerable revenue losses. For this reason, overload prevention in these applications is a critical issue. In this paper we present a complete characterization of secure e-commerce applications scalability to determine which are the bottlenecks in their performance that must be considered for an overload control strategy. With this information, we design an adaptive session-based overload control strategy based on SSL (Secure Socket Layer) connection differentiation and admission control. The SSL connection 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 the server). Considering this big difference, we have implemented an admission control algorithm that prioritizes resumed SSL connections to maximize the performance in session-based environments and dynamically limits the number of new SSL connections accepted, according to the available resources and the current number of connections in the system, in order to avoid server overload. Our evaluation on a Tomcat server demonstrates the benefit of our proposal for preventing server overload.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Security; SSL; e-commerce; Application servers; Overload control; Service differentiation; Admission control; Session-based; Scalability characterization

1. Introduction

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

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

1389-1286/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2007.05.010

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

At the same time, current sites are subject to enormous variations in demand, often in an unpredictable fashion, including flash crowds that cannot be easily processed. For this reason, the servers that host the sites must be ready to handle situations with a large number of concurrent clients.

Dealing with situations with a large number of concurrent clients and/or with a workload that demands high computational power (for instance secure workloads) can lead a server to overload (i.e. the volume of requests for content at a site temporarily exceeds the capacity for serving 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 reliant on security, such server behavior could translate to sizable revenue losses.

Overload prevention is a critical goal so that a system can remain operational in the presence of overload even when the incoming request rate is several times greater than the system capacity, and at the same time is able to serve the maximum number of requests during such overload, while maintaining response times at acceptable levels. With these objectives in mind, several mechanisms have been proposed to deal with overload, such as admission control, request scheduling, service differentiation, service degradation and resource management.

However, the design of a successful overload prevention strategy must be preceded by a complete characterization of the application server scalability, which should consist of something more complex than simply measuring the application server performance with different number of clients and determining the load levels that overload the server. A complete characterization should also supply the causes of this overload, in order to identify which factors are the bottlenecks in the application server's performance that must be considered in an overload prevention strategy. For this reason, this characterization requires powerful analysis tools that allow an in-depth analysis of the application server's behavior and its interaction with the other system elements (including distributed clients, a database server, etc.). These tools must support and consider all the levels involved in the execution of web applications if they want to provide meaningful performance information to the administrators because the origin of performance problems can reside in any of these levels or in the interaction between them.

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

This paper contributes a complete characterization of the scalability of Java application servers running secure e-commerce applications. We have decided to focus on the Tomcat [3] application server, which serves as a reference implementation of the Sun Servlet and JSP specifications. The characterization is divided in two parts. The first part consists of measuring Tomcat's vertical scalability (i.e. adding more processors) when using SSL and determining the impact of adding more processors when the server is overloaded. The second part involves a detailed analysis of the server's behavior using a performance analysis framework, in order to determine the causes of the server overload when running with different numbers of processors. This analysis demonstrates the convenience of developing a strategy to prevent server overload, and identifies the factors that must be considered for its implementation.

With this information, we design an overload control strategy that is based on SSL connection differentiation and admission control. SSL connection differentiation is accomplished by proposing a possible extension of the Java Secure Sockets Extension [4] (JSSE) package, which implements a Java version of the SSL protocol, to distinguish between SSL connections depending on whether the connection

will reuse an existing SSL connection on the server or not. This differentiation can be very useful in order to design intelligent overload control policies for a server, given the big difference that exists in the computational demand of new SSL connections versus resumed SSL connections, especially in scenarios where the SSL computational demand dominates over the computational demand of the service itself, as typically occurs in e-commerce applications.

Based on the SSL connection differentiation, a session-based adaptive admission control mechanism for the Tomcat application server is implemented. This mechanism allows the server to avoid throughput degradation and response time increments occurred during overload conditions. The server differentiates full SSL connections from resumed SSL connections and limits the acceptance of full SSL connections to the maximum number possible without overloading the available resources, while it accepts all the resumed SSL connections. In this way, the admission control mechanism maximizes the number of sessions completed successfully, allowing e-commerce sites based on SSL to increase the number of transactions completed, thus generating higher benefit.

The rest of the paper is organized as follows: Section 2 introduces the SSL protocol and its implementation in Java. Section 3 describes our methodology for characterizing the scalability of e-commerce applications. Sections 4 and 5 detail the implementation of the SSL connection differentiation and SSL admission control mechanisms. Section 6 describes the experimental environment used in our evaluation. Section 7 presents the evaluation results of our overload control strategy. Section 8 presents the related work and finally, Section 9 presents the conclusions of this paper.

2. Security for e-commerce Java applications

2.1. SSL protocol

The SSL protocol provides communications privacy over the Internet using a combination of public-key and private-key cryptography and digital certificates (X.509). This 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 remarkably increases the computation time necessary to serve a connection, due to the use of cryptography to achieve its objectives. This increment has a noticeable impact on the server's performance, which has been evaluated in [5]. 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 overloads, it can maintain the throughput if new clients arrive, while if attending SSL connections, the overload of the server provokes degradation of the throughput.

Most of the computation time required when using SSL is spent during the SSL handshake phase, which features the negotiation between the client and the server to establish a SSL connection. Two different SSL handshake types can be distinguished, namely 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 need 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 resumes 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. Note the big difference between the time to negotiate a full SSL handshake compared to the time to negotiate a resumed SSL handshake (175 ms vs. 2 ms).

2.2. JSSE API limitations

The Java Secure Socket Extension (JSSE) [4] implements a Java technology version of the Secure Sockets Layer (SSL) [2] and Transport Layer Security (TLS) [6] 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` explicitly begins handshakes, any attempt to read or write application data through the connection causes an implicit handshake, and 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 whether an incoming SSL connection provides a reusable SSL session ID until the handshake is fully completed. Having this information prior to the handshake negotiation could be very useful, for example for servers doing overload control based on SSL connection differentiation, given the big difference that exists on the computational demand of new SSL connections versus resumed SSL connections. It is important to note that the verification to check if an incoming SSL connection provides a valid SSL session ID is already performed by the JSSE API prior to handshaking, because this verification is needed in order to negotiate both a full SSL handshake and a resumed SSL handshake. Therefore, the addition of a new interface to access this information would not involve any additional cost.

3. Scalability characterization methodology

The scalability of an application server is defined as its ability to maintain a site's availability, reliability, and performance while the amount of simultaneous web traffic, or load, hitting the application server increases [7]. Given this definition, the scalability of an application server can be represented by measuring the performance of the application server while the load increases. With this representation, the load that overloads the server can be detected. An application server is overloaded when it is unable to maintain the site's availability, reliability, and performance (i.e. the server does not scale). As derived from the definition, when the server is overloaded, the performance is degraded (lower throughput and higher response time) and the number of client requests refused is increased.

At this point, two questions should appear to the reader (and of course, to the application server administrator). Firstly, the load that overloads the server has been detected, but why is this load causing the server's performance to degrade? In other words, in which parts of the system (CPU, database, network, etc.) will a request be spending most of its execution time when the server is overloaded? In order to answer this question, this paper proposes to analyze the application server's behavior using a performance analysis framework, which considers all levels involved in the application server's execution, allowing a fine-grained analysis of dynamic web applications.

Secondly, the application server's scalability with given resources has been measured, but how would affect the addition of more resources to the application server's scalability? This adds a new dimension to the application servers scalability: the measurement of the scalability relative to the resources. This scalability can be done in two different ways: vertical and horizontal.

Vertical scalability (also called scaling up) is achieved by adding capacity (memory, processors, etc.) to an existing application server and requires few to no changes to the architecture of the system. Vertical scalability increases the performance (as long as the added resource were a bottleneck for the server's performance) and the manageability of the system, but decreases the reliability and availability (single failure is more likely to lead to system failure). This paper considers this kind of scalability relative to the resources.

Horizontal scalability (also called scaling out) is achieved by adding new application servers to the system, also increasing its complexity. Horizontal scalability increases the reliability, the availability and the performance (depending on load balancing), but decreases the manageability (there are more elements in the system).

The analysis of the application server's behavior will provide hints to help answer the question of how it would affect the application server's scalability if more resources were added. If the analysis reveals that a resource is a bottleneck for the application server's performance, then this encourages the addition of new resources of this type (vertical scaling) in order to improve the server's scalability. On the other hand, if a resource that is not a bottleneck for the application server's performance is upgraded, the added resources are wasted because the scalability is not improved and the causes of the server's performance degradation remain unresolved. This observation motivates the analysis of the application server's behavior in order to detect the causes of overload before adding new resources.

4. SSL connection differentiation

As we mentioned in Section 2.2, there is no way in the JSSE packages to query if an incoming SSL

connection provides a reusable SSL session ID until the handshake is fully completed. We propose extending the JSSE API to allow applications to differentiate new SSL connections from resumed SSL connections prior to when 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 (i.e. the short connections) will result in a reduction of the average response time, as described in previous works using SRPT and related policies for scheduling HTTP requests in static web servers [8,9], TCP flows in Internet routers [10], and download requests in peer-to-peer systems [11]. Moreover, prioritizing the resumed SSL connections will increase the probability for a client to complete a session, maximizing the number of sessions completed successfully. In addition, a server could limit the number of new SSL connections that it accepts, in order to avoid throughput degradation produced during overload conditions.

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 is not expected to add significant additional cost. The mechanism works at the system level and is based on examining the contents of the first TCP segment received on the server after the connection is established.

After a new connection is established between the server and the client, the SSL protocol starts a handshake negotiation. The protocol begins with the client sending a SSL ClientHello message (refer to the RFC 2246 [6] for more details) to the server. This message can include a SSL session ID from a previous connection if it wants the SSL session 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 determine if the connection is a resumed SSL connection or a new one.

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 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 it possible to find the socket inside 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 is trying to establish a new SSL session. If a non-zero SSL session ID is found instead, the connection is trying to resume a previous SSL session. The value of this SSL message field is returned by the system call to the JNI native library which, in turn, returns it to the Tomcat server. With this result, the server can decide, for instance, to apply an admission control algorithm 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 a new component into the Tomcat server, called eDragon Admission Control (EAC), which implements a session-oriented adaptive mechanism that performs admission control based on SSL connection differentiation. The EAC continuously monitors incoming secure connections to the server, performs online measurements distinguishing new SSL connections from resumed SSL connections and decides which incoming SSL connections are accepted. The differentiation of resumed SSL connections from new SSL connections is performed with the mechanism described in Section 4.

The EAC has two main objectives. The first objective consists of maximizing the number of sessions successfully completed. The EAC accomplishes this objective by accepting all the resumed SSL connections that hit the server. The second objective consists of preventing the server from getting overloaded, and in this way, avoiding server throughput degradation and maintaining acceptable

response times. This objective is accomplished by the EAC by keeping the maximum amount of load just below the system's capacity. For secure web applications, the system's capacity depends on the available processors, as it has been demonstrated in [5], due to the great computational demand of these kinds of applications. Therefore, if the server can use more processors, it can accept more SSL connections without overloading. Considering this, the EAC limits the massive arrival of new SSL connections to the maximum number acceptable by the server before it overloads, which depends on the available processors. The procedure used by the EAC for calculating this number is as follows:

For every sampling interval k (currently defined as 2 s), the EAC calculates the number of resumed SSL connections (defined as $\mathrm{O}(k)$) that arrive to the server during that interval. The average computation time entailed by a resumed SSL connection (defined as CTO) and the average computation time entailed by a new SSL connection (defined as CTN) have been measured using static profiling on the application. Each measurement includes not only the average computation time spent in the negotiation of the SSL handshake, but also the average computation time spent for generating all the dynamic web content requested using that connection and the average computation time spent for SSL encryption/decryption during the data transferring phase.

Using the previously defined values, the EAC periodically calculates, at the beginning of every sampling interval k, the maximum number of new SSL connections that can be accepted by the server during that interval without overloading. We define this value as $N(k)$. Since resumed SSL connections have preference with respect to new SSL connections (all resumed SSL connections are accepted), this maximum depends on the computation time required by the already accepted resumed SSL connections (which can be calculated using the previous definitions as $\mathrm{O}(k) \times CTO$) and the number of processors allocated to the server (defined as $A(k)$). Considering these two values, the EAC can determine the remaining computational capacity for attending new SSL connections and hence the value of $N(k)$, in the following way:

$$N(k) = \left\lfloor \frac{k \times A(k) - O(k) \times CTO}{CTN} \right\rfloor.$$

The EAC will only accept up until the maximum number of new SSL connections that do not overload the server ($N(k)$) (i.e. they can be served with the available computational capacity without degrading the QoS). The rest of the 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.

6. Experimental environment

6.1. Tomcat Servlet container

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. 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 (a HttpProcessor) is responsible for accepting a new incoming connection on the server listening port and assigning it to a socket structure. From this point, that HttpProcessor will be responsible for attending to and serving the received requests through the persistent connection established with the client, while another HttpProcessor will continue accepting new connections. HttpProcessors are commonly chosen from a pool of threads in order to avoid thread creation overheads.

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. The pattern of a persistent connection in Tomcat is shown in Fig. 1. In this example, three different requests are served through the same connection. The rest of the time (*connection (no request)*) the server is keeping the connection open waiting for another client request. A connection timeout is programmed to close the connection if no more requests are received. Notice that within every *request* the service (execution of the servlet implementing the demanded request) is distinguished

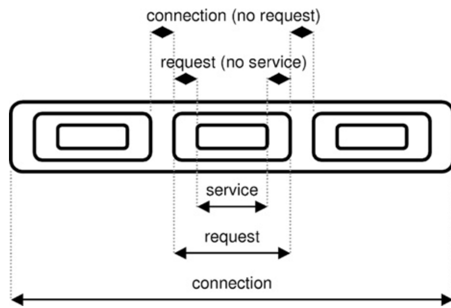


Fig. 1. Tomcat persistent connection pattern.

from the *request (no service)*. This is the pre and post process that Tomcat requires to invoke the servlet that implements the demanded request.

Fig. 2 shows the pattern of a secure persistent connection in Tomcat. Notice that when using SSL the pattern of the HTTP persistent connection is maintained, but the underlying SSL connection supporting this persistent HTTP connection must be previously established, by negotiating an SSL handshake, which can be full or resumed depending on whether an SSL Session ID is reused. For instance, if a client must establish a new HTTP connection because its current HTTP connection has been closed by the server due to a connection persistence timeout expiration, since it reuses the underlying SSL connection, it will negotiate a resumed SSL handshake.

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

6.2. Auction site benchmark (RUBiS)

The experimental environment also includes a deployment of the servlets version 1.4.2 of the RUBiS (Rice University Bidding System) [12] benchmark 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. Five of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using different mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJBs.

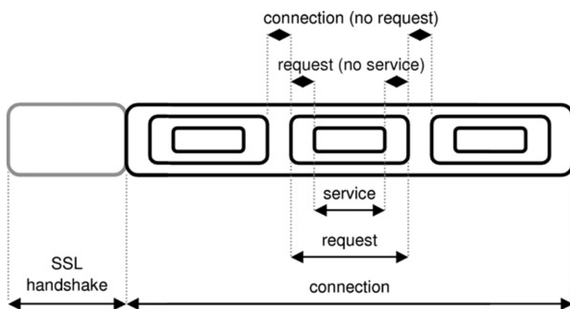


Fig. 2. Tomcat secure persistent connection pattern.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [13]. 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. The workload distribution generated by Httperf was extracted from the RUBiS client emulator. RUBiS client emulator defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read–write interactions. The experiments in this paper use the browsing mix.

Although the RUBiS client emulator follows a closed system model, the combination of Httperf with a RUBiS workload allows the use of a partly-open model (as defined in [14]), which is a more realistic alternative to represent web applications behavior. In this model, new clients initiate a session with the server (implemented as a persistent HTTP/S connection) according to a certain arrival rate. This rate can be specified as one of the parameters of Httperf. Emulated clients use the established persistent connection to issue requests to the server.

Every time a client receives the server's response to a request, there is a probability p that the client simply leaves the system, and there is a probability $1 - p$ that the client remains in the system and makes another request, possibly after some think time. The think time emulates the "thinking" period of a real client who takes a period of time before clicking on the next request. The think time is generated from a negative exponential distribution with a mean of 7 s. The next request in the session is decided using a Markov model. This model uses a transition probability matrix with probabilities attached to transitions from each RUBiS interaction to the others. In particular, this matrix defines for each interaction

the probability of ending the session (i.e. p), which is 0.2 for 10 of the 27 interactions and 0 for the rest. This value determines the sessions duration for a given workload, which is an important factor because overload is especially harmful for the performance of workloads with longer sessions, such as those in RUBiS. Notice that our overload prevention strategy can prevent throughput degradation for all SSL workloads, independently of the duration of the sessions, by limiting the number of accepted connections. However, the performance gains will be more noticeable for workloads with long sessions, due to the added benefit of giving priority to resumed SSL connections.

Httperf also allows 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. This emulates the fact that the real client gets bored of waiting for a response from the server and leaves the site. For the experiments in this paper, Httperf has been configured by setting the client timeout value to 10 s.

6.3. Performance analysis framework

In order to determine the causes of the server overload, we propose analyzing the application server's behavior using a performance analysis framework. This framework, which consists of an instrumentation tool called Java Instrumentation Suite (JIS [15]) and a visualization and analysis tool called Paraver [16], considers all levels involved in the application server's execution (operating system, JVM, application server and application), and allows a fine-grained analysis of dynamic web applications. For example, the framework can provide detailed information about the thread status, system calls (I/O, sockets, memory & thread management, etc.), monitors, services, connections, etc. Further information about the implementation of the performance analysis framework and its use for the analysis of dynamic web applications can be found in [15,17].

6.4. Hardware and software platform

The Tomcat server (with the RUBiS benchmark deployed on it) runs on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. Tomcat makes queries to a MySQL v4.0.18 [18] database server using the MM.MySQL v3.0.8 JDBC driver. The MySQL server runs on another machine, which is a 2-way Intel XEON 2.4 GHz with 2 GB RAM. In addition, we have configured a machine with a 2-way Intel XEON 2.4 GHz and 2 GB RAM to run the workload generator (Httperf 0.8). For each experiment performed in the evaluation section (each point in the graphs corresponds to an experiment), the Httperf tool is parametrized with the number of new clients per second that initiate a session with the server (this value is indicated in the horizontal axis of the graphs) and with the workload distribution that these clients must follow (extracted from the Markov model mentioned in the previous section). Then, Httperf emulates the execution of these clients by performing secure requests to the Tomcat server during a run of 10 min. All the machines are connected using 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 a 1024 bit RSA key.

7. Evaluation

In this section we present our characterization of the Tomcat server's scalability and the evaluation comparison of our overload control strategy with respect to the original Tomcat server.

7.1. Scalability characterization of the Tomcat server

This section presents the scalability characterization of the Tomcat application server when running the RUBiS benchmark using SSL. The characterization is divided in two parts. The first part is an evaluation of the vertical scalability of the server when running with different numbers of processors, determining the impact of adding more processors on the server overload (can the server support more clients before overloading?). The second part consists of a detailed analysis of the server's behavior using the performance analysis framework, in order to determine the causes of the server overload when running with different numbers of processors.

7.1.1. Vertical scalability of the Tomcat server

Fig. 3 shows the Tomcat throughput as a function of the number of new clients per second initiating a session with the server when running with

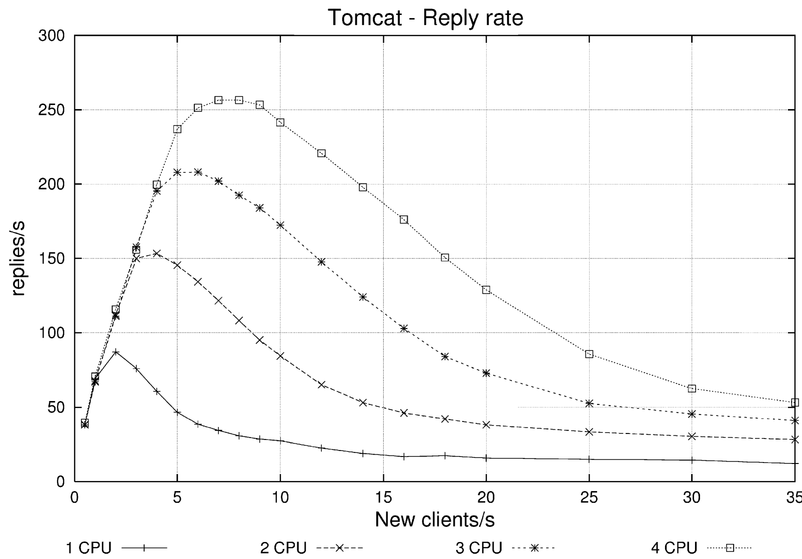


Fig. 3. Throughput of the original Tomcat with different numbers of processors.

different numbers of processors. Notice that for a given number of processors, the server's throughput increases linearly with respect to the input load (i.e. the server scales) until a determined number of clients hit the server. At this point, the throughput achieves its maximum value. Table 1 shows the number of clients that overload the server and the maximum throughput achieved before overloading when running with different numbers of processors. Notice that, since secure workloads are CPU-intensive, running with more processors allows the server to handle more clients before overloading, so the maximum throughput achieved is higher. In particular, running with 2 processors increases the maximum throughput achieved by a factor of 1.7, running with 3 processors by a factor of 2.3, and running with 4 processors by a factor of 2.8. Notice that the achieved improvement is not linear because processors are not the only component limiting the server's performance. When the number of clients that overload the server has been reached, the server's throughput degrades until approximately 20% of the maximum achievable throughput while the number of clients increases, as shown in Table 2. This table shows the average throughput obtained when the server is overloaded when running with different numbers of processors. Notice that, although the throughput obtained has decreased in all cases where the server has reached an overloaded state, running with more processors improves the throughput again. When the server is overloaded, running with 2 processors increases the maximum achieved throughput by a factor of 2.1, running with 3 processors by a factor of 3.1, and running with 4 processors by a factor of 4. In this case, a linear improvement is achieved, because processors are the main component limiting the performance.

Table 1
Number of clients that overload the server and maximum throughput achieved before overload occurs

Number of processors	New clients/s	Throughput (replies/s)
1	2	90
2	4	155
3	6	208
4	8	256

Table 2
Average server's throughput when overloaded

Number of processors	Throughput (replies/s)
1	13
2	28
3	41
4	53

As well as degrading the server's throughput, the server overload also affects the server's response time, as shown in Fig. 4. This figure shows the server's average response time as a function of the number of new clients per second initiating a session with the server when running with different numbers of processors. Notice that when the server

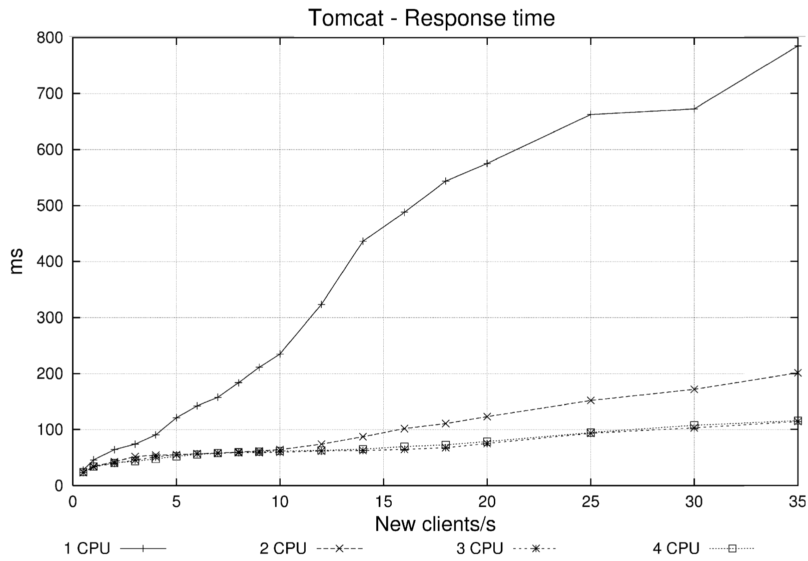


Fig. 4. Response time of the original Tomcat with different numbers of processors.

is overloaded the response time increases (especially when running with one processor) as the number of clients grow.

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 when running with different numbers of processors, when the server 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.

7.1.2. Scalability analysis of the Tomcat server

The analysis methodology consists of comparing the server's behavior when it is overloaded with that when it is not, using the performance analysis framework described in Section 6.3. We calculate a series of metrics representing the server's behavior, and we determine which of them are affected while

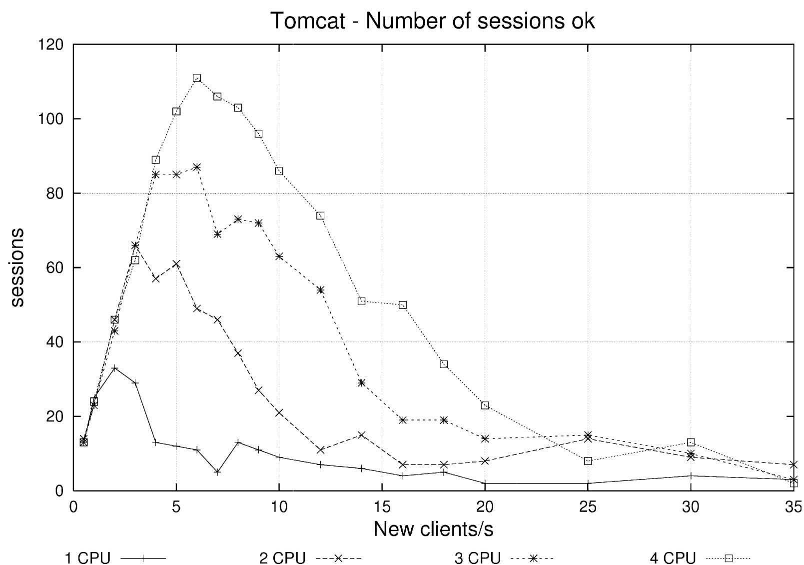


Fig. 5. Completed sessions by the original Tomcat with different numbers of processors.

increasing the number of clients. From these metrics, an in-depth analysis is performed looking for the causes of their dependence on the server load.

First of all, using the performance analysis framework, we calculate the average time spent by the server processing a persistent client connection, distinguishing the time devoted to each phase of the connection (connection phases have been described in Section 6.1) when running with different numbers of processors. This information is displayed in Fig. 6. Notice that when the server is not overloaded, the majority of the time spent to process a client connection is devoted to the *connection* (*no request*) phase. During this phase the connection remains established waiting for additional requests from the client (i.e. maintaining connection persistence) and for this reason CPU consumption is low.

On the other side, when the server overloads, the average time required to handle a connection increases considerably, mainly at the *SSL handshake* and the *connection (no request)* phases. The proportionally greater increment occurs in the *SSL handshake* phase. In particular, the time spent in this phase increases from 18 ms to 1050 ms when running with one processor, from 7 ms to 602 ms with two processors, from 6 ms to 610 ms with three processors and from 6 ms to 464 ms with four processors. The increment that occurs in the *connection (no request)* phase is proportionally lower, but also noticeable. In particular, the time spent in this phase has increased from 605 ms to 2711 ms when running with one processor, from 485 ms to 1815 ms with two processors, from 515 ms to 1134 ms with three processors and from 176 ms to 785 ms with four processors.

To determine the causes of the great increase in the time spent in the *SSL handshake* phase of the connection, we calculate the percentage of connections that perform a resumed SSL handshake (reusing the SSL Session ID) versus the percentage of connections that perform a full SSL handshake when running with different numbers of processors. This information is shown in Fig. 7. Notice that when the server runs with one processor without overloading, 97.3% of SSL handshakes reuse the SSL connection, but when it overloads, only 32.9% reuse it. The rest negotiate a full SSL handshake, overloading the server because it cannot supply the computational demand of these full SSL handshakes. Remember the big difference between the computational demand of a resumed SSL handshake (2 ms) and a full SSL handshake (175 ms). The same situation is produced when running with two processors (the percentage of full SSL handshakes increases from 2% to 67.2%), when running with three processors (from 2.4% to 62.5%), and when running with four processors (from 1.4% to 63.5%).

This lack of computational power also explains the increase in the time spent in the *connection (no request)* phase. The connection remains open waiting for additional requests from a given client, but when these requests arrive to the server machine, since there is no available CPU to accept them, they must wait longer in the operating system's internal network structures before being accepted.

We have determined that when running with any number of processors the server overloads when most of the incoming client connections need to negotiate a full SSL handshake instead of resuming an existing SSL connection, requiring a computing capacity that the available processors are unable

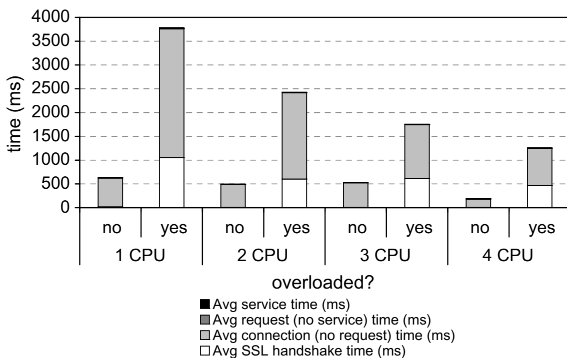


Fig. 6. Average time spent by the server processing a persistent client connection.

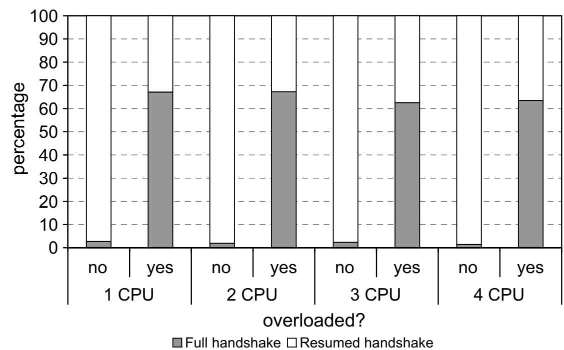


Fig. 7. Incoming server connections classification depending on the SSL handshake type performed.

to supply. Nevertheless, why does this occur from a given number of clients? In other words, why do incoming connections negotiate a full SSL handshake instead of a resumed SSL handshake when attending to a given number of clients? Remember that we have configured the client with a timeout of 10 s. This means that if no reply is received in this time, the client's connection will be discarded. When the server is overloaded, it cannot handle the incoming requests before the client timeouts expire. For this reason, in the long run most of the resumed connections with the server will be discarded, and only new clients will arrive at the server. Remember that the initiation of a new client requires the establishment of a new SSL connection, and therefore the negotiation of a full SSL handshake. Therefore, if the server is overloaded and it cannot handle the incoming requests before the client timeouts expire, this provokes the arrival of a great amount of new client connections that need the negotiation of a full SSL handshake, requiring a computing capacity that the available processors are unable to supply.

This shows that client timeouts have an important effect on the server's performance. One could think about raising client timeouts as the server load increases in order to avoid the degradation of server's performance. However, this is not an appropriate solution for two reasons. Firstly, client timeouts cannot be modified because they are out of the scope of the server administrator. Secondly, even if this modification were feasible, the server will overload anyway, although this will occur when attending to a higher number of clients. In addition, since this solution does not allow differentiating resumed SSL connections from new ones, the prioritization of resumed connections that our approach supports cannot be accomplished, and for this reason, the number of sessions completed successfully will be lower, losing one of the added values of our approach.

Considering the described behavior, it makes sense to design an overload control strategy that prevents the massive arrival of client connections that need to negotiate a full SSL handshake and will overload the server, avoiding in this way the server's throughput degradation and maintaining a good quality of service (good response time) for already connected clients.

7.2. Tomcat with overload control

Fig. 8 shows the throughput of Tomcat with overload control as a function of the number of new clients per second initiating a session with the server when running with different numbers of processors. Notice that for a given number of processors, the server's 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 overload control behaves in the same way as the original one.

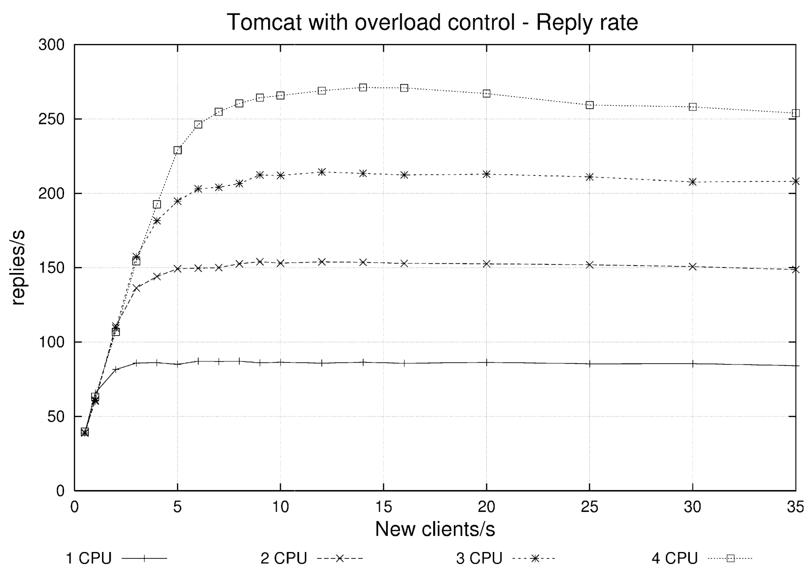


Fig. 8. Throughput of Tomcat with overload control with different numbers of processors.

However, when the number of clients that would overload the server has been achieved, the EAC can avoid degradation in the throughput, maintaining the maximum achievable throughput, as shown in Fig. 8. Notice that running with more processors allows the server to handle more clients, so the maximum achieved throughput is higher.

The overload control strategy also allows to maintain the response time at levels that guarantee a good quality of service to the clients, even when the number of clients that would normally overload the server has been reached, as shown in Fig. 9. This figure shows the server's average response time as a function of the number of new clients per second initiating a session with the server when running with different numbers of processors.

Finally, the overload control strategy also has a beneficial effect for session-based clients. As shown in Fig. 10, which shows the number of sessions successfully finished when running with different

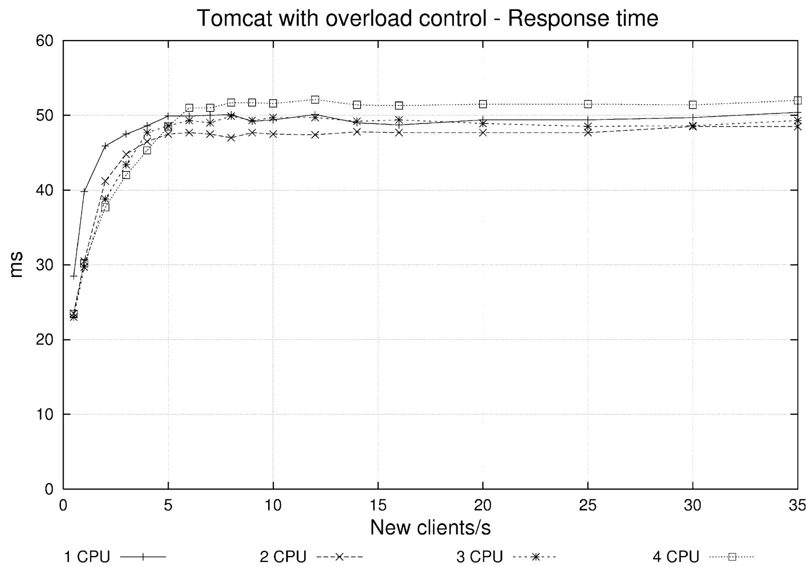


Fig. 9. Response time of Tomcat with overload control with different numbers of processors.

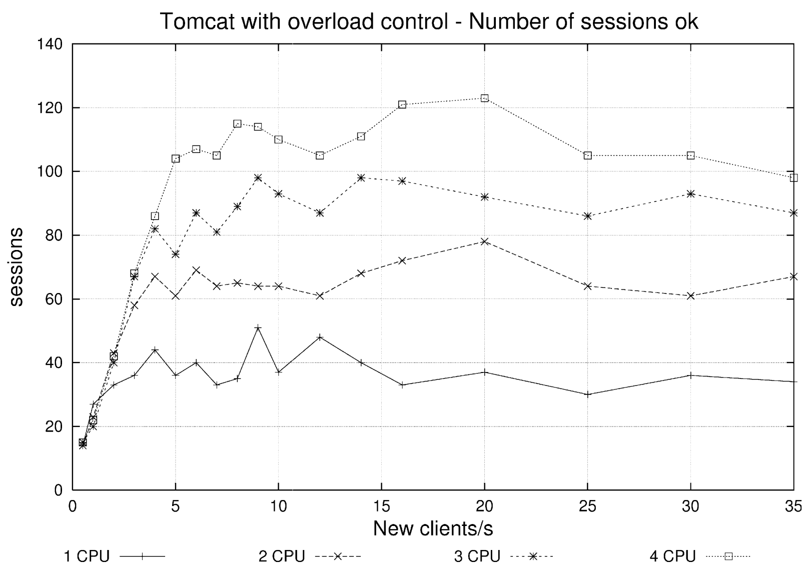


Fig. 10. Completed sessions by Tomcat with overload control with different numbers of processors.

numbers of processors, the number of sessions that can finalize completely does not decrease, even when the number of clients that would normally overload the server has been reached.

If we repeat the scalability analysis performed in Section 7.1.2 on a Tomcat server that uses the overload control strategy, we can confirm that even though the server is in an overloaded state, our admission control mechanism is able to maintain almost the same average time spent processing a persistent client's connection (as shown in Fig. 11) and a similar ratio between resumed and new SSL connections to the one observed when the server is not overloaded (as shown in Fig. 12).

As a final comment, we want to mention that our overload control approach adapts well to transient overload situations produced due to the inherent variability of web workloads. This has not been directly demonstrated in the evaluation, which shows experiments with a fixed number of new clients arriving at the server every second, but it has been demonstrated indirectly, because the number of requests that the server receives every second is not fixed, since in addition to the requests from new clients, it includes the requests from all the clients that are already in the system. The only requirement for our mechanism to work properly is to have information about the resources allocated to the server at every moment. This paper assumes that the server has a static number of resources allocated, thus having this information is straightforward. We have considered the case of having dynamically allocated resources in [19], where we have complemented our admission control strategy with a dynamic resource provisioning mechanism that decides on the resource distribution among the applications and informs them about their allocated resources. The evaluation in this paper (which includes experiments with a variable number of new clients per second arriving at the server) demonstrates that our solution can adequately prevent transient overloads.

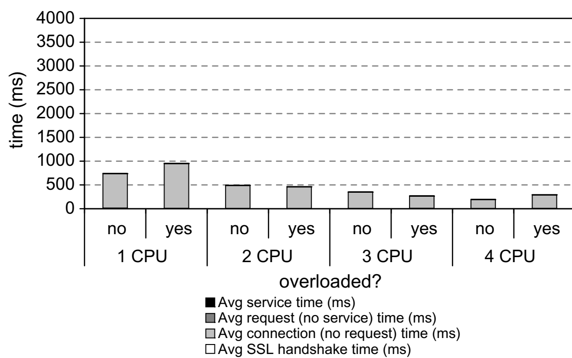


Fig. 11. Average time spent by the server processing a persistent client connection when using the overload control strategy.

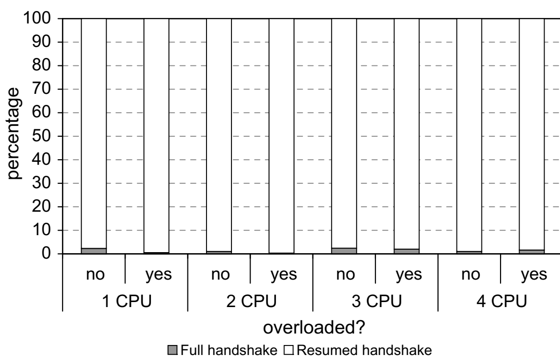


Fig. 12. Incoming server connections classification depending on the SSL handshake type performed when using the overload control strategy.

8. Related work

8.1. Characterization of application servers' scalability

Application servers' scalability constitutes an important aspect in supporting the increasing number of users of secure dynamic web sites. Although this work focuses on maintaining a server's scalability when running in secure environments by adding more resources (vertical scaling), the great computational demand of SSL protocol can also be handled using other approaches.

Major J2EE vendors such as BEA [20] or IBM [21,22] use clustering (horizontal scaling) to achieve scalability and high availability. Several studies evaluating server's scalability using clustering have been performed [21,23], but none of them considers security issues.

Scalability can also be achieved by delegating the security issues on a web server (e.g. Apache web server [24]) while the application server only processes dynamic web requests. In this case, the computational demand will be transferred to the web server, which can be optimized for SSL management.

Servers' scalability in secure environments can also be achieved by adding new specialized hardware [25] for processing SSL requests, thereby

reducing the processor demand, but increasing the cost of the system.

Related with the vertical scalability covered in this paper, some works have evaluated this scalability on web servers and application servers. For example, [26,27] only consider static web content, and in [28,26,27,29] the evaluation is limited to a numerical study without performing analysis to justify the scalability results obtained. None of these works evaluates the effect of security on an application server's scalability.

Other works try to improve the application server's scalability by tuning some server parameters and/or JVM options and/or operating system properties. For example, Tomcat's scalability while tuning certain parameters, including different JVM implementations, JVM flags and XML implementations has been studied in [30]. In the same way, the application server's scalability using different mechanisms for generating dynamic web content has been evaluated in [31]. However, none of these works considers any kind of scalability relative to resources (either vertical or horizontal), nor the influence of security on the application server's scalability.

Certain types of analysis has been performed in some works. For example, [12,31] provide a quantitative analysis based on general metrics of application server's execution by collecting system utilization statistics (CPU, memory, network bandwidth, etc.). These statistics may allow the detection of some application server bottlenecks, but this coarse-grained analysis is often not enough when dealing with more sophisticated performance problems.

The influence of security on application server's 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 [32], concluding that SSL increases the 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 [33], showing that the key exchange is the slowest operation in the protocol. [34] analyzes the impact of the full handshake in the establishment of a connection and proposes caching sessions to reduce it.

Security for Web Services can also be provided with SSL, but other proposals such as WS-Security [35], which use industry standards like XML Encryption and XML Signature, can be used instead. Coupled with WS-SecureConversation, the advantage WS-Security has over SSL over HTTP is twofold: firstly, it works independently of the underlying transport protocol and secondly, it provides security mechanisms that operate in end-to-end scenarios (across trust boundaries) as opposed to point-to-point scenarios (i.e. SSL). Anyway, since WS-Security also requires a great computational demand to support its encryption mechanisms, it makes most of the conclusions obtained in this paper valid in a Web Services environment too.

Our approach intends to achieve a complete characterization of dynamic web applications using SSL vertical scalability and determines the causes of server overload by performing a detailed analysis of the application server's behavior while considering all levels involved in the execution of dynamic web applications.

8.2. Overload control in web environments

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 focus 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 left this ordination to the operating system. But, as it is well known from queuing theory that the shortest remaining processing time first (SRPT) scheduling minimizes queuing time (and therefore the average response time), some proposals [8,9] implement policies based on this algorithm to prioritize the service of short static content requests before long requests. This prioritized scheduling in web servers has been proven effective in providing significantly better response time to high priority requests at a relatively low cost to lower priority requests. Although scheduling can improve response times, under extreme overloads other mechanisms become indispensable. Besides, 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 the 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. ACES [36] attempts to limit the number of admitted requests based on estimated service times, and also allows service prioritization. The evaluation of this approach is 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 [37] is described in [38]. SEDA decomposes services into multiple stages, each one of which can perform admission control based on monitoring the response time through that stage. The evaluation includes dynamic content in the form of a web-based email service. In [39], the authors present an admission control mechanism for e-commerce sites that externally observes the execution costs of requests, distinguishing different request types. Other approaches [40–42] use control theory with a feedback element for dealing with overload. Yaksha [40] implements a self-tuning proportional integral controller for admission control in multi-tier e-commerce applications using a single queue model. Quorum [41] 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 the service at the exit. In [42], 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 that the desired response time target is met.

Some works have integrated resource management with other approaches such as admission control and service differentiation. For example, [43] proposes using resource containers as an operating system abstraction that embodies a resource. [44] proposes a resource overbooking based scheme for maximizing the revenue generated by the available resources in a shared platform. [45] presents a prototype data center implementation which is used to study the effectiveness of dynamic resource allocation for handling flash crowds. Cataclysm [46] performs overload control by bringing together admission control, service degradation and the dynamic provisioning of platform resources.

Service degradation tries to avoid refusing clients as a response to the overload but instead reduces the level of service offered to clients [47,48,46,38], for example in the form of providing smaller content (e.g. lower resolution images).

On most of the prior work, overload control is performed on a 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 [49]. This approach allows sessions to run to completion even under overload, by denying all access when the server load exceeds a predefined threshold. Another approach to session-based admission control based on the characterization of a commercial web server's log, which discriminates the scheduling of requests based on the probability of completion of the session that the requests belong to, is presented in [50].

Our proposal combines important aspects that previous work has considered in isolation or has simply ignored. Firstly, we consider dynamic web content instead of simpler static web content. Secondly, we focus on session-based applications, considering the particularities of these applications when performing admission control. Thirdly, 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 in secure web applications while none of the above works have covered this issue.

9. Conclusions

In this paper we have designed an overload control strategy for secure e-commerce applications. Firstly, we have presented a complete characterization of the Tomcat server's scalability when executing secure e-commerce applications. This characterization is divided in two parts:

The first part consisted of measuring Tomcat's vertical scalability (i.e. adding more processors) when using SSL and analyzing the effect of this addition on the server's scalability. The results confirmed that, since secure workloads are CPU-intensive, running with more processors makes the server able to handle more clients before overloading, with the maximum achieved throughput improvement ranging from 1.7 to 2.8 for 2 and 4 processors, respectively. In addition, even when the server has reached an overloaded state, a linear improvement on throughput can be obtained by using more processors. The second part involved the analysis of the causes of server overload when running with different numbers of processors by using a performance analysis framework. The analysis revealed that the

server overloads due to the massive arrival of new SSL connections which demand computational power that the system is unable to supply, demonstrating the convenience of developing some kind of overload control strategy to filter this massive arrival of new SSL connections, avoiding in this way the degradation of the server's throughput.

Based on the conclusions extracted from this analysis, we have designed an adaptive session-based overload control strategy based on SSL connection differentiation and admission control. SSL connection differentiation has been accomplished using a possible extension of the JSSE package to allow distinguishing resumed SSL connections (that reuse an existing SSL session on server) from new SSL connections. This feature has been used to implement an admission control mechanism that has been incorporated into the Tomcat server. This admission control mechanism differentiates new SSL connections from resumed SSL connections and limits the acceptance of new SSL connections to the maximum number possible with the available resources without overloading the server, while accepting all the resumed SSL connections in order to maximize the number of sessions completed successfully, allowing e-commerce sites based on SSL to increase the number of transactions completed (which is a very important metric in e-commerce environments).

The experimental results demonstrate that the proposed strategy prevents the overload of application servers in secure environments. It maintains the response time at levels that guarantee good QoS and completely avoids throughput degradation (the throughput previously degraded until approximately 20% of the maximum achievable throughput when the server overloads), while maximizes the number of sessions completed successfully. These results confirm that security must be considered as an important issue that can heavily affect the scalability and performance of Java application servers.

Acknowledgements

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 BSC (Barcelona Supercomputing Center). For additional information about the authors, please visit the Barcelona eDragon Research Group web site [51].

References

[1] E. Rescorla, HTTP over TLS. RFC 2818, May 2000.

[2] A.O. Freier, P. Karlton, C. Kocher, The SSL Protocol. Version 3.0. November 1996. URL: http://wp.netscape.com/eng/ssl3/ssl-toc.html.

[3] Jakarta Project. Apache Software Foundation, Jakarta Tomcat Servlet Container. URL: http://jakarta.apache.org/tomcat.

[4] Sun Microsystems, Java Secure Socket Extension (JSSE). URL: http://java.sun.com/products/jsse/.

[5] J. Guitart, V. Beltran, D. Carrera, J. Torres, E. Ayguadé, Characterizing secure dynamic web applications scalability, in: 19th International Parallel and Distributed Symposium (IPDPS'05), Denver, Colorado, USA, April 4–8, 2005.

[6] T. Dierks, The TLS Protocol, Version 1.0. RFC 2246, January 1999.

[7] I. Haddad, Scalability issues and clustered web servers, Tech. rep., Concordia University (August) 13, 2000.

[8] M. Crovella, R. Frangioso, M. Harchol-Balter, Connection scheduling in web servers, in: 2nd Symposium on Internet Technologies and Systems (USITS'99), Boulder, Colorado, USA, October 11–14, 1999.

[9] M. Harchol-Balter, B. Schroeder, N. Bansal, M. Agrawal, Size-based scheduling to improve web performance, ACM Transactions on Computer Systems (TOCS) 21 (2) (2003) 207–233.

[10] I. Rai, E. Biersack, G. Urvoy-Keller, Size-based scheduling to improve the performance of short TCP flows, IEEE Network Magazine 19 (1) (2005) 12–17.

[11] Y. Qiao, D. Lu, F. Bustamante, P. Dinda, Looking at the server side of peer-to-peer systems, in: 7th Workshop on Languages, Compilers and Run-time Support for Scalable Systems (LCR 2004), Houston, Texas, USA, October 22–23, 2004, pp. 1–8.

[12] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, W. Zwaenepoel, Specification and implementation of dynamic web site benchmarks, in: IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, Texas, USA, November 25, 2002.

[13] D. Mosberger, T. Jin, httperf: A tool for measuring web server performance, in: Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), Madison, Wisconsin, USA, June 23, 1998, pp. 59–67.

[14] B. Schroeder, A. Wierman, M. Harchol-Balter, Open versus closed: a cautionary tale, in: 3rd Symposium on Networked Systems Design & Implementation (NSDI'06), San Jose, CA, USA, May 8–10, 2006, pp. 239–252.

[15] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, J. Labarta, Complete instrumentation requirements for performance analysis of web based technologies, in: 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), Austin, Texas, USA, March 6–8, 2003, pp. 166–176.

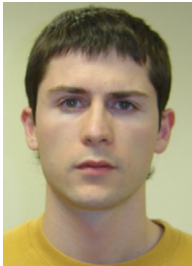
[16] European Center for Parallelism of Barcelona, Paraver. URL: http://www.cepba.upc.es/paraver.

[17] J. Guitart, D. Carrera, J. Torres, E. Ayguadé, J. Labarta, Tuning dynamic web applications using fine-grain analysis, in: 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05), Lugano, Switzerland, February 9–11, 2005, pp. 84–91.

[18] MySQL. URL: http://www.mysql.com/.
[19] J. Guitart, D. Carrera, V. Beltran, J. Torres, E. Ayguade, Preventing secure web applications overload through dynamic resource provisioning and admission control, Tech. Rep. UPC-DAC-RR-2006-37, Computer Architecture Department, Technical University of Catalonia, November 2006.
[20] BEA Systems, Inc, Achieving Scalability and High Availability for e-Business. BEA white paper, March 2003.
[21] Y. An, T.K.T. Lau, P. Shum, A Scalability Study for WebSphere Application Server and DB2. IBM white paper, January 2002.
[22] W. Chiu, Design for Scalability. IBM white paper, September 2001.
[23] I. Haddad, G. Butler, Experimental studies of scalability in clustered web system, in: Workshop on Communication Architecture for Clusters (CAC'04) (in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'04)), Santa Fe, New Mexico, USA, April 26, 2004.
[24] The Apache Software Foundation, Apache HTTP Server Project. URL: http://httpd.apache.org/.
[25] R. Mraz, SecureBlue: An architecture for a high volume SSL internet server, in: 17th Annual Computer Security Applications Conference (ACSAC'01), New Orleans, Louisiana, USA, December 10–14, 2001.
[26] V. Beltran, D. Carrera, J. Torres, E. Ayguadé, Evaluating the scalability of Java event-driven web servers, in: 2004 International Conference on Parallel Processing (ICPP'04), Montreal, Canada, August 15–18, 2004, pp. 134–142.
[27] I. Haddad, Open-source web servers: performance on carrier-class linux platform, Linux Journal 2001 (91) (2001) 1.
[28] S. Anne, A. Dickson, D. Eaton, J. Guizan, R. Maiolini, JBoss 3.2.1 vs. WebSphere 5.0.2 Trade3 Benchmark. SMP Scaling: Comparison report, October 2003.
[29] M. Malzacher, T. Kochie, Using a Web application server to provide flexible and scalable e-business solutions. IBM white paper, April 2002.
[30] P. Lin, So You Want High Performance (Tomcat Performance) (September 2003). URL: http://jakarta.apache.org/tomcat/articles/performance.pdf.
[31] E. Cecchet, J. Marguerite, W. Zwaenepoel, Performance and scalability of EJB applications, in: 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), Seattle, Washington, USA, November 4–8, 2002, pp. 246–261.
[32] K. Kant, R. Iyer, 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.
[33] C. Coarfa, P. Druschel, 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.
[34] A. Goldberg, R. Buff, A. Schmitt, Secure web server performance dramatically improved by caching SSL session keys, in: Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), Madison, Wisconsin, USA, June 23, 1998.
[35] IBM Corporation, Microsoft Corporation and VeriSign Inc, Web Services Security (WS-Security) Specification. Version 1.0.05 (April 2002). URL: http://www-106.ibm.com/developerworks/webservices/libra.
[36] X. Chen, H. Chen, P. Mohapatra, ACES: An efficient admission control scheme for QoS-aware web servers, Computer Communications 26 (14) (2003) 1581–1593.
[37] M. Welsh, D. Culler, E. Brewer, SEDA: An architecture for well-conditioned, scalable internet services, in: 18th Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, October 21–24, 2001, pp. 230–243.
[38] M. Welsh, D. Culler, Adaptive overload control for busy internet servers, in: 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, USA, March 26–28, 2003.
[39] S. Elnikety, E. Nahum, J. Tracey, W. Zwaenepoel, A method for transparent admission control and request scheduling in e-commerce web sites, in: 13th International Conference on World Wide Web (WWW'04), New York, New York, USA, May 17–22, 2004, pp. 276–286.
[40] A. Kamra, V. Misra, E. Nahum, Yaksha: A controller for managing the performance of 3-tiered websites, in: 12th International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada, June 7–9, 2004.
[41] J. Blanquer, A. Batchelli, K. Schauser, R. Wolski, Quorum: Flexible quality of service for internet services, in: 2nd Symposium on Networked Systems Design and Implementation (NSDI'05), Boston, MA, USA, May 2–4, 2005, pp. 159–174.
[42] X. Liu, J. Heo, L. Sha, X. Zhu, Adaptive control of multi-tiered web application using queueing predictor, in: 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada, April 3–7, 2006.
[43] G. Banga, P. Druschel, 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.
[44] B. Urgaonkar, P. Shenoy, 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.
[45] A. Chandra, P. Shenoy, Effectiveness of dynamic resource allocation for handling internet flash crowds, Tech. Rep. TR03-37, Department of Computer Science, University of Massachusetts, USA, November 2003.
[46] B. Urgaonkar, P. Shenoy, Cataclysm: Handling extreme overloads in internet services, Tech. Rep. TR03-40, Department of Computer Science, University of Massachusetts, USA, December 2003.
[47] T. Abdelzaher, N. Bhatti, Web content adaptation to improve server overload behavior, Computer Networks 31 (11–16) (1999) 1563–1577.
[48] S. Chandra, C. Ellis, A. Vahdat, Differentiated multimedia web services using quality aware transcoding, in: IEEE INFOCOM 2000, Tel-Aviv, Israel, March 26–30, 2000, pp. 961–969.
[49] L. Cherkasova, P. Phaal, Session-based admission control: a mechanism for peak load management of commercial web sites, IEEE Transactions on Computers 51 (6) (2002) 669–685.
[50] H. Chen, P. Mohapatra, Overload control in QoS-aware web servers, Computer Networks 42 (1) (2003) 119–133.
[51] Barcelona eDragon Research Group. URL: http://research.ac.upc.edu/eDragon.

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

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



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

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

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