Performance Impact of Using SSL on Dynamic Web Applications

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

Resumen— **Security requirements are becoming common on current Internet transactions. HTTPS connections are frequently used by application servers in order to host secure transactions. HTTPS connections are based on HTTP protocol over SSL connections, to provide authentication, confidentiality and integrity, using symmetric and asymmetric cryptographic algorithms (using private or public key). But the utilization of SSL connections involves higher computational and resource demand than non-secure connections. These new requirements must be analyzed and evaluated to determine the viability of a secure platform. In this paper we analyze the impact of using SSL connections on dynamic web applications' performance, showing the effect of having different SSL client patterns and the benefits of running secure application servers on multiprocessors.**

Palabras clave— **SSL, Web Servers, Performance Evaluation, Scalability, Dynamic Workload**

I. Introduction

Main goal of current e-commerce sites consists of serving as many concurrent clients as possible. The HTTPS protocol [18] is widely extended over these sites due to the security requirements associated with e-commerce transactions. The HTTPS protocol, which is based on SSL (Secure Socket Layer) protocol [10] from Netscape and was used as the basis of the standard protocol TLS (Transport Layer Security) [9], provides some fundamental characteristics for e-commerce transactions, like the confidentiality, the integrity and the authentication. SSL is based on cryptographic techniques, and for this reason, it becomes a computational demanding process, causing the concurrent number of clients a site can attend to decrease considerably. This fact motivates the evaluation of the impact of using this protocol in a realistic e-commerce environment, as for instance the eBay site.

In this paper we evaluate the impact of security on application servers performance. First, we study the general server behavior when using secure connections as a function of the number of clients, considering especially the server scalability on multiprocessors. Second, we evaluate the effect of some parameters, which determine different SSL client patterns, on server performance. Our experimental environment is composed of Tomcat application server [13] running RUBiS benchmark [7], which supplies a realistic dynamic web content environment simulating an auctions site.

This work is being developed by the Barcelona eDragon Research Group [4] in the way toward the self-optimization of Java-based application servers.

European Center for Parallelism of Barcelona (CEPBA), C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034, Barcelona (Spain). {vbeltran, jguitart, dcarrera, torres, eduard, jesus}@ac.upc.es

II. Related Work

Web server performance has been widely evaluated on the literature. K. Kant et al. [14] analyze 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. They conclude that SSL increases computational cost of transactions by a factor of 5-7. C. Coarfa [8] studies the impact of each individual operation of TLS protocol in the context of Web Servers and they show that key exchange is the slowest operation in SSL protocol. About web servers evaluation without SSL, we can mention V. Beltran et al. [5] focused on web server architecture, scalability on multiprocessors and performance evaluation without TLS utilization. About dynamic content, Emmanuel Cecchet et al. [7] [2] study web servers performance under dynamic workload using RUBiS. RUBiS is an auction site prototype modeled after eBay.com that is used to evaluate application design patterns and application servers performance scalability.

Other related work is focused on the analysis and optimization of the SSL protocol. Boneh and Shacham [19] propose a batch mechanism to process the complex RSA decryption that offers good speedups but introduces latency in the handshake setup. In [6] the same authors survey four variants of RSA designed to speed up RSA decryption and signing. Intel [12], Mraz [16] and Alteon [1] propose offload RSA processing of web servers into specialized hardware by different techniques. A variety of commercial TLS hardware accelerators are inspired in this approach. Other efforts are put optimizing the TLS

protocol, like Apostolopuolos et al. [3]. Goldberg et al. [11] analyze the impact of full handshake in connection establishment and caching sessions mechanism to avoid it.

III. Server Architecture and SSL

Application servers are I/O intensive applications. The operative core of a HTTP application server is commonly implemented as a continuous loop where incoming requests are served. To serve a new request, it's necessary to read from the client application what is being requested. Clients usually access the server through a network connection: a socket. After having received a request, it must be processed to determine which resource should be sent to the client. Once the accessed resource is located, it is written to the client socket. SSL protocol doesn't introduce a new degree of complexity in this 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.

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 this objectives it uses a combination of public-key and private-key cryptography algorithm and digital certificates (X.509). The SSL protocol fundamentally has two phases of operation: SSL handshake and SSL record protocol. In the next subsection we do an overview of the phases. The detailed description of the protocol can be found in RFC 2246 [9].

A. SSL handshake

The handshake allows the server to authenticate itself to the client using public-key techniques like RSA, 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. This process is detailed in Figure 1.

Two different handshake types can be distinguished: The initial handshake and the resumed handshake. The initial handshake is negotiated when a client establishes a new SSL connection with the server, and requires the negotiation of the full SSL handshake. The 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, the part of the SSL handshake negotiation can be avoided.

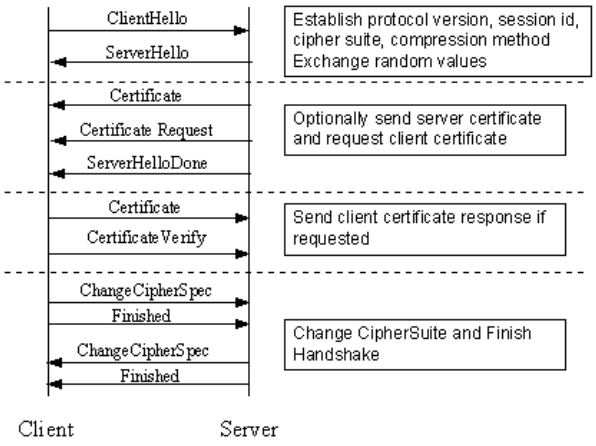

Fig. 1. SSL Handshake protocol

Initial handshake

The client sends a client hello message to which the server must respond with a server hello message. The client hello and server hello establish the following attributes: protocol version, session ID, cipher suite, and compression method. Additionally, two random values are generated and exchanged. Following the hello messages, the server will send its certificate. If the server is authenticated, it may request a certificate from the client. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. The client key exchange message is now sent. At this point, a change cipher spec message is sent by the client and then immediately sends the finished message. In response, the server will send its own change cipher spec message and Finished message. At this point, the handshake is complete and the client and server may begin to exchange application layer data. We have measured the computational demand of an initial handshake in a 1.4 GHz Xeon machine to be around 175 ms.

Resumed handshake

The client sends a client hello using the Session ID of the session to be resumed. The Server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a server hello with the same Session ID value. At this point, both client and server must send change cipher spec messages and proceed directly to finished messages.

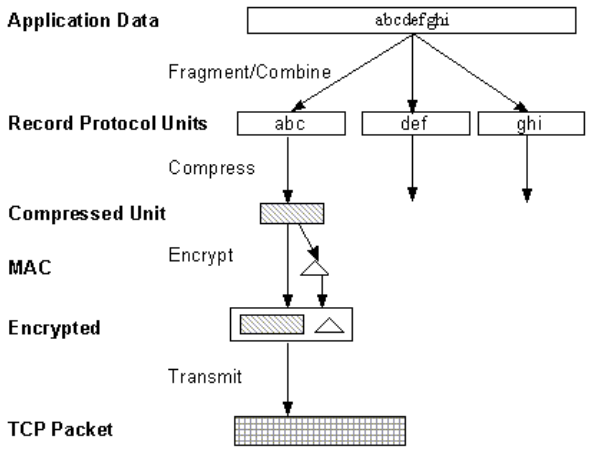

Fig. 2. SSL Record protocol

Once the re-establishment is complete, the client and server may begin to exchange application layer data. If a Session ID match is not found, the server generates new session ID and the SSL client and server perform a full handshake. We have measured the computational demand of a resumed handshake in a 1.4 GHz Xeon machine to be around 2 ms. Notice the big difference between negotiate a full SSL handshake respect to negotiate a resumed SSL handshake (175 ms versus 2 ms).

B. SSL Record Protocol

The SSL Record Layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size. Then the information blocks are fragmented into plain-text records of 214 bytes or less. All records are compressed using the compression algorithm defined in the current session state and protected using the encryption and MAC (Message Authentication Code) algorithms defined in the current CipherSpec. Finally encryption and MAC functions translate compressed units to encrypted data, ready to be send into TCP packet. This process is detailed in Figure 2.

IV. Experimental environment

We use Tomcat v5.0.19 [13] as the application server. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to be an accurate implementation and serve as a reference implementation of the Sun Servlet and JSP specifications, and also to be a quality production servlet container. 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.

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [7] benchmark servlets version 1.4 on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS defines 27 interactions. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB.

The workload for the experiments was generated using Httperf [15] which support both HTTP and HTTPS protocols. This workload generator and web performance measurement tool allows the creation of a continuous flow of HTTP requests issued from one or more client machines and processed by one server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for the experiments presented in this paper were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of emulated clients on the client machine. Each emulated client issues a number of requests, some of them pipelined, over a persistent HTTP/S connection. The workload distribution generated by Httperf was extracted from the RUBiS client emulator.

Tomcat runs on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We use MySQL v4.0.18 [17] 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). All the machines run the 2.6.2 Linux kernel. Server machine is connected with client machine through a 1 Gbps Ethernet interface. Database and server machine are direct connected through 100 Mbps fast Ethernet crossed-link. 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 512 MB.

V. Methodology and Experimental Results

In this section we analyze the truly impact of SSL in an application server environment. We have performed a set of tests to determine the behavior of the server in different scenarios. Each test shows server throughput as a function of workload intensity for a specific server configuration.

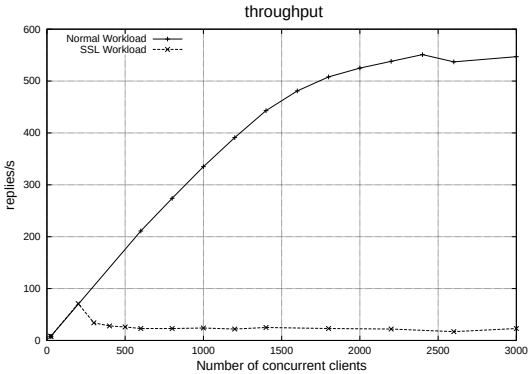

Fig. 3. Throughput comparison. SSL vs non SSL

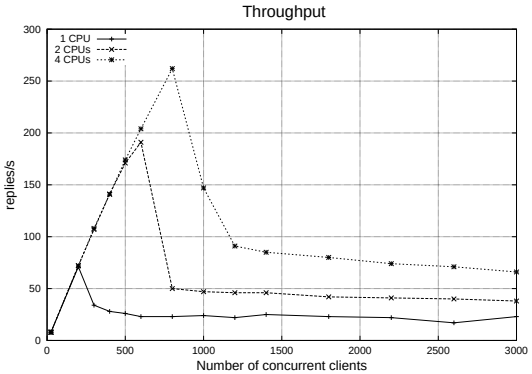

Fig. 4. Throughput comparison on SMP system

First, we study the general server behavior when using secure connections as a function of the number of clients, considering especially the server scalability on multiprocessors. Second, we evaluate the effect of some parameters, which determine different client patterns, on server performance.
All the tests are realized with the common RSA-3DES-SHA cipher suit. Handshake is performed with 1024 bit RSA key. Record protocol uses triple DES to encrypt all application data. Finally SHA digest algorithm is used to provide the message authentication code (MAC).

A. Scalability on uniprocessors

In this section we will evaluate the server performance measuring the throughput as a function of the number of clients. With this comparison we will show the performance gap between normal HTTP clients and HTTPS clients. The default size for the server thread pool is 100 and the timeout for the Httperf emulated clients is set to 10 seconds.
Figure 3 shows the big impact of using secure connections in server throughput. Until 250 concurrent clients the performance with normal clients and secure clients is the same. After this point, throughput with non-secure clients grows up until 2500 clients, whereas throughput with secure clients remains stable and very low.
The poor performance obtained when using secure connection is produced by two factors. First, the establishment of a SSL connection requires a lot of computation and second, when the server is heavy loaded and it cannot serve a client, this client reconnects immediately, increasing the computational demand (a new SSL connection has to be negotiated) and degrading the server performance even more.

B. Scalability on multiprocessors

As commented in previous sections, when using secure connections, computation demand is increased considerably. We have seen that when running in uniprocessors, the observed throughput when using secure connection is clearly lower that when using non-secure connections. At this point, and taking into account the great computational demand of SSL protocol, we want to know if it is makes sense to use a multiprocessor to host the application server in order to achieve better performance.
Figure 4 shows the server throughput when running in a system with 1, 2 and 4 processors. Notice that processor addition contributes to considerably improve server performance. In spite of this improvement, all configurations reach in any moment a saturation point in which throughput falls in a great way. This is produced because there are too much clients demanding the server to establish a secure connection, and the server is unable to handle them all simultaneously. Having more processors the same problem appears but when the number of clients attempting to connect the server is higher.

C. Influence of client behaviour with SSL

In this section we will evaluate the effect of different client behaviours relative to SSL on server performance. We want to show that different client patterns (do the clients reuse the SSL session IDs?, do the clients retry the erroneous requests? how much do the clients wait for a server response?) can heavily determine the server performance. Our methodology consists on varying one parameter and comparing the throughput obtained with the one obtained with the standard configuration used in previous section.
First, we will evaluate the effect of not reusing the

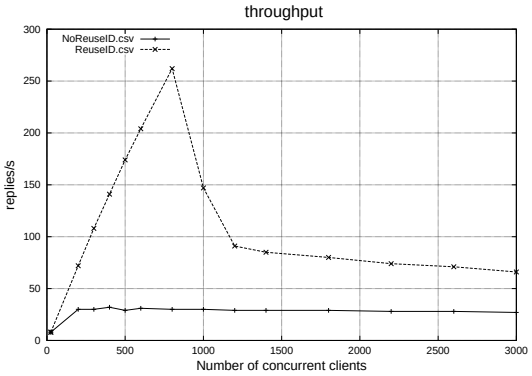

Fig. 5. Standard configuration vs NoReuseSessionID

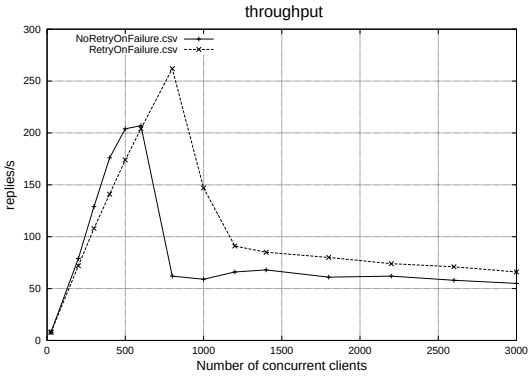

Fig. 6. Standard configuration vs NoRetryOnFailure

SSL connections. Second, we will evaluate the server throughput supposing that a new client is initiated when an error is produced (instead of retrying the failed requests). Finally, we will evaluate the server throughput when reducing the number of client connections by programming the clients to wait indefinitely the server responses.

C.1 Reuse session ID

As commented before, SSL connections can be reused to reduce the cost of handshake phase. But Httperf can be forced to do a full handshake every time a connection is established. This has a great effect of performance, as shown in Figure 5. If SSL session ID it is not reused, each client connection with the server has to negotiate a full SSL handshake. We have distinguished in previous sections the big difference of processing demand between negotiating a full SSL handshake (around 175 ms) and negotiating a resumed SSL handshake (around 2 ms). This fact justifies the great performance degradation showed in Figure 5.

C.2 Retry on failure

This parameter determines the client behavior when an error is produced. Is it enabled by default, producing that if an error is detected, the same URL is requested again (this implies that the SSL session ID is reused). On the other side, if this parameter is disabled, current client is aborted and a new client is initiated (provoking the establishment of a new SSL connection with the server). Figure 6 shows the effect on throughput when using this parameter. Notice that, as less SSL connections are established, throughput is better.

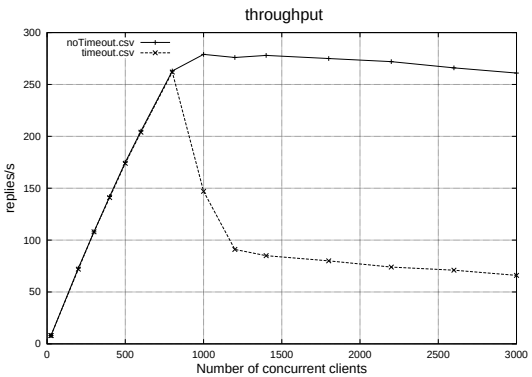

Fig. 7. Standard configuration vs Infinite Client Timeout

C.3 Client timeout

In this section we want to evaluate the server behavior when reducing the number of clients reconnections due to timeouts. Notice that when using secure connections, the cost of establishing reconnections is critical for performance. In order to achieve this reduction, we parametrize the client timeout to an infinite value, therefore the clients will wait for a server response indefinitely, reducing in this way the number of re-connections to the server. Figure 7 shows the throughput achieved. Notice that when clients wait for server response indefinitely, throughput is maintained, while when setting the client timeout, throughput degrades considerably due to the high number of client re-connections.
With these results, it makes sense to introduce the idea of having some control on the number of clients connections that the server is able to process. For example, we could introduce a connection manager prepared to detect situations of high number of con-

nections that are affecting negatively the server performance, and, at this point decide to limit the maximum number of connection establishments per unit of time to avoid performance degradation.

VI. Conclusions & Future Work

In this paper we have shown the impact of security on application servers performance. First, we have studied the general server behavior when using secure connection as a function of the number of clients. We have determined that the main problem of servers using SSL to provide security resides on the high number of full SSL connections established, because this kind of connections require a great amount of computation, producing a server throughput degradation when a lot of them are requested simultaneously. Second, we have evaluated the effect of some client behaviours relative to SSL on server performance. These evaluation has revealed that different client patterns (do the clients reuse the SSL session IDs?, do the clients retry the erroneous requests? how much do the clients wait for a server response?) can heavily determine the server throughput.

Our future work goes toward a resource manager able to determine when the server is near to initiate its performance degradation, and at this point limit the maximum number of SSL connections allowed to the server, in order to maintain the server throughput. We want to consider also the available system resources, so the resource manager will determine at every moment which is the optimal resource configuration, asking for more resources or releasing assigned resources depending on the number of requests.

Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-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].

Referencias

[1] Alteon. *Alteon Web Switching Portfolio*, 2002.

[2] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks, 2002.

[3] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security, how much does it really cost ? In *Proceedings of Eighteenth Conference on Computer Communications*, Mar. 1999.

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

[5] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the scalability of java event-driven web servers. In *Proceedings of the International Conference on Parallel Processing*, August 2004.

[6] D. Boneh and H. Shacham. Fast variants of rsa.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246–261, 2002.

[8] C. Coarfa, P. Druschel, and D. Wallach. Performance analysis of tls web servers, 2002.

[9] T. Dierks and C. Allen. The tls protocol, version 1.0., Jan. 1999.

[10] A. O. Freier, P. Karlton, and C. Kocher. The ssl protocol, version 3.0., Nov. 1996.

[11] A. Goldberg, R. Buff, and A. Schmitt. Secure web server performance dramatically improved by caching ssl session keys. In *Proceedings of the Workshop on Internet Server Performance*, June 1998.

[12] Intel. *Intel(R) ADD8125Y and AAD8120Y e-Commerce Directors*, 2002.

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

[14] K. Kant, R. Iyer, and P. Mohapatra. Architectural impact of secure socket layer on internet servers. In *International conference on computer design (ICCD)*, page 7, 2000.

[15] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[16] R. Mraz. An architecture for a high volume ssl internet server. In *Proceedings of Seventeenth Annual Computer Security Applications Conference*, page 7, Dec. 2001.

[17] MySQL. *http://www.mysql.com*.

[18] E. Rescorla. Http over tls, May. 2000.

[19] H. Shacham and D. Boneh. Improving SSL handshake performance via batching. *Lecture Notes in Computer Science*, 2020:28–32, 2001.