

# A Hybrid Web Server Architecture for Secure e-Business Web Applications

Vicenç Beltran, David Carrera, Jordi Guitart, Jordi Torres and Eduard Ayguadé  
{vbeltran, dcarrera, jguitart, torres, eduard}@ac.upc.es

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

## Abstract

*Nowadays the success of many e-commerce applications, such as on-line banking, depends on their reliability, robustness and security. Designing a web server architecture that keeps these properties under high loads is a challenging task because they are the opposite to performance. The industry standard way to provide security on web applications is the use the Secure Socket Layer (SSL) protocol to create a secure communication channel between the clients and the server. Traditionally, the use of data encryption has introduced a negative performance impact over web application servers because it is an extremely CPU consuming task, reducing the throughput achieved by the server as well as increasing its average response time. As far as the revenue obtained by a commercial web application is directly related to the amount of clients that complete business transactions, the performance of such secure applications becomes a mission critical objective for most companies. In this paper we evaluate a novel hybrid web server architecture (implemented over Tomcat 5.5) that combines the best aspects of the two most extended server architectures, the multithreaded and the event-driven, to provide an excellent trade-off between reliability, robustness, security and performance. The obtained results demonstrate the feasibility of the proposed hybrid architecture as well as the performance benefits that this model introduces for secure web applications, providing the same security level than the original Tomcat 5.5 and improved reliability, robustness and performance, according to both technical and business metrics.*

Keywords: Web Server Architectures, SSL, Performance Evaluation, Java, Network-Based Computing

## 1 Introduction

Many e-commerce applications must offer a secure communication channel to their clients in order to achieve the level of security and privacy required to carry out most commercial transactions. But the cost of introducing security mechanisms in on-line transactions is not negligible. The industry standard for secure web communications is the Secure Socket Layer (SSL) protocol, which is generally used as a complement to the Hypertext Transport Protocol (HTTP) to create the Secure HTTP protocol (HTTPS). The primary goal of the SSL protocol is to provide privacy and reliability between two applications in communication over the Internet. This is achieved using a combination of public and private cryptography to encode the communication between the peers.

The use of public key cryptography introduces an important computational cost to the web containers. Each time a new connection attempt is accepted by a server, a cryptographic negotiation takes place. This initial handshake is required by the peers in order to exchange the encryption keys that will be used during the communication. The cost of the initial handshake is high enough to limit enormously the maximum number of new connections than can be accepted by the server in a period of time, as well as degrading the performance of the server to unacceptable levels, as it can be seen in [5].

The architectural design of most existing web servers is based on the multithreaded paradigm (Apache[11] and Tomcat [7] are widely extended examples), which assigns one thread to each client connected to the server. These threads, commonly known as worker threads, are in charge of attending all the requests issued by their corresponding client until it gets disconnected. The problem associated to this model is that the maximum number of concurrent clients accepted by the server is lim-

ited to the number of threads created. In front of this situation, the solution adopted by most web servers is to impose an inactivity timeout to each connection client, forcing the connection to get closed if the client does not produce any work activity before the timeout expires.

The effect of closing client connections is relatively irrelevant when working with plain connections, but it becomes tremendously negative when dealing with secure workloads. Closing connections, especially when the server is overloaded, increases the number of cryptographic handshakes performed and reduces remarkably the capacity of the server, which results in an important impact over the maximum throughput achieved by the server.

On the other hand, an alternative architectural design for web servers is the event-driven model, already used in Flash[9] and in the SEDA[12] architecture. This model comes to solve the problems associated to the multithreading paradigm, especially in client-server environments. But this model lacks of the innate ease of programming associated to the multithreading model, making the task of developing web servers remarkably more complex.

In this paper we evaluate a hybrid web server architecture oriented to the use of secure communication protocol that exploits the best of each one of the discussed server architectures. With this hybrid architecture, an event-driven model is applied to receive the incoming client requests. When a request is received, it is serviced following a multithreaded programming model, with the resulting simplification of the web container development associated to the multithreading paradigm. When the request processing is completed, the event-driven model is applied again to wait for the client to issue new requests. With this, the best of each model is combined and, as it is discussed in following sections, the performance of the server is remarkably increased, especially in presence of secure communication protocols.

The rest of the paper is structured as follows: section 2 describes the HTTP/s protocol, section 3 discusses the characteristics of the multithreaded, event-driven and hybrid architectures; later, in section 4, we present the execution environment where the experimental results presented in this work were obtained and, finally, section 5 presents the experimental results obtained in the evaluation of the hybrid web server and section 6 gives some concluding remarks and discusses some of the future work lines derived from this work.

## 2 HTTP/S and SSL

HTTP/S (HTTP over SSL) is a secure Web protocol developed by Netscape. HTTPS is really just the use of Secure Socket Layer (SSL) as a sublayer under its regular HTTP application layering.

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

The SSL protocol does not introduce a new degree of complexity in web applications structure because it works almost transparently on top of the socket layer. However, SSL increases the computation time necessary to serve a connection remarkably, due to the use of cryptography to achieve their objectives. This increment has a noticeable impact on server performance, which has been evaluated in [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 saturates, it can maintain the throughput if new clients arrive, while if attending SSL connections, the saturation of the server provokes the degradation of the throughput.

The SSL protocol fundamentally has two phases of operation: SSL handshake and SSL record protocol. We will do an overview of the SSL handshake phase, which is the responsible of most of the computation time required when using SSL. The detailed description of the whole protocol can be found in RFC 2246 [10]. The SSL handshake allows the server to authenticate itself to the client using public-key techniques like RSA, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server. Two different SSL handshake types can be distinguished: The full SSL handshake and the resumed SSL handshake. The full SSL handshake is negotiated when a client establishes a new SSL connection with the server, and requires the complete negotiation of the SSL handshake. This negotiation includes parts that spend a lot of computation time to be accomplished. We have measured the computational demand of a full SSL handshake in a 1.4 GHz Xeon machine to be around 175 ms.

The SSL resumed handshake is negotiated when a client establishes a new HTTP connection with the server but using an existing SSL connection. As the SSL

session ID is reused, part of the SSL handshake negotiation can be avoided, reducing considerably the computation time for performing a resumed SSL handshake. We have measured the computational demand of a resumed SSL handshake in a 1.4 GHz Xeon machine to be around 2 ms. Notice the big difference between negotiate a full SSL handshake respect to negotiate a resumed SSL handshake (175 ms versus 2 ms).

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

### 3 Web server architectures

There are multiple architectural options for a web server design, depending on the concurrency programming model chosen for the implementation. The two major alternatives are the multithreaded model and the event-driven model. In both models, the work tasks to be performed by the server are divided into work assignments that are assumed each one by a thread (a worker thread). If a multithreaded model is chosen, the unit of work that can be assigned to a worker thread is a client connection, which is achieved by creating a virtual association between the thread and the client connection that is not broken until the connection is closed. Alternatively, in an event driven model the work assignment unit is a client request, so there is no real association between a server thread and a client.

#### Multithreaded architecture with blocking I/O

The multithreaded programming model leads to a very easy and natural way of programming a web server. The association of each thread with a client connection results in a comprehensive thread life-cycle, started with the arrival of a client connection request and finished with the connection close. This model is especially appropriate for short-lived client connections and with low inactivity periods, which is the scenario created by the use of non persistent HTTP/1.0 connections. A pure multithreaded web server architecture is generally composed by an acceptor thread and a pool of worker threads. The acceptor thread is in charge of accepting new incoming connections, after what each established connection is assigned to one thread of the workers pool, which will be responsible of processing all the requests issued by the corresponding web client. A brief opera-

tion diagram for this architecture can be seen on figure 1(a).

The introduction of connection persistence in the HTTP protocol, already in the 1.0 version of the protocol but mainly with the arrival of HTTP/1.1, resulted in a dramatic performance impact for the existing multithreaded web servers. Persistent connections, which means connections that are kept alive by the client between two successive HTTP requests that in turn can be separated in time by several seconds of inactivity (think times), cause that many server threads can be retained by clients even when no requests are being issued and the thread keeps in idle state. The use of blocking I/O operations on the sockets is the cause of this performance degradation scenario. The situation can be solved increasing the number of threads available (which in turn results in a contention increase in the shared resources of the server that require exclusive access) or introducing an inactivity timeout for the established connections, that can be reduced as the server load is increased. When a server is put under a severe load, the effect of applying a shortened inactivity timeout to the clients lead to a virtual conversion of the HTTP/1.1 protocol into the older HTTP/1.0, with the consequent loss of the performance effects of the connection persistence. If we use HTTP/S, open a new connection for each request can decrease dramatically the throughput of the server, specially when the server are in a overloaded state. In this situation the number of initial SSL handshakes increase quickly and saturates the web server.

In this model, the effect of closing client connections to free worker threads reduces the probability for a client to complete a session to nearly zero. It is especially important when the server is under overload conditions, where the inactivity timeout is dynamically decreased to the minimum possible in order to free worker threads as quickly as possible, which provokes that all the established connections are closed during think times. This causes a higher competition among clients trying to establish a connection with the server. If we extend it to the length of a user session, we obtain that the probability of finishing it successfully under this architecture is still much lower than the probability of establishing each one of the connections it is composed of, driving the server to obtain a really low performance in terms of session completions. This situation can be alleviated increasing the number of worker threads available in the server, but this measure also produces an important increase in the internal web container contention with the corresponding performance slowdown.

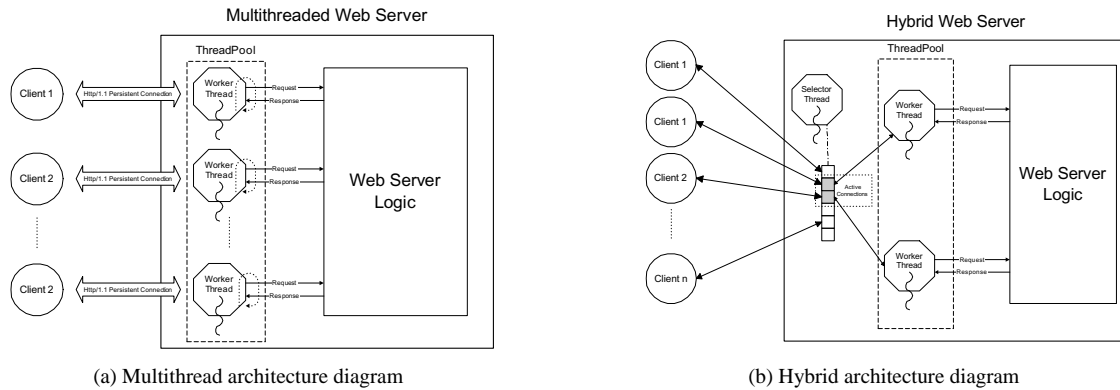


Figure 1.

### Event-driven architecture with non-blocking I/O

The event-driven architecture completely eliminates the use of blocking I/O operations for the worker threads, reducing their idle times to the minimum because no I/O operations are performed for a socket if no data is already available on it to be read. With this model, maintaining a big amount of clients connected to the server does not represent a problem because one thread will never be blocked waiting a client request. With this, the model detaches threads from client connections, and only associates threads to client requests, considering them as an independent work units. An example of web server based on this model is described in [9], and a general evaluation of the architecture can be found in [2].

In an event driven architecture, one thread is in charge of accepting new incoming connections. When the connection is accepted, the corresponding socket channel is registered in a channel selector where another thread (the request dispatcher) will wait for socket activity. Worker threads are only awakened when a client request is already available in the socket. When the request is completely processed and the reply has been successfully issued, the worker thread registers again the socket channel in the selector and gets free to be assigned to new received client requests. This operation model avoids worker threads to keep blocked in socket read operations during client think times and eliminates the need of introducing connection inactivity timeouts and their associated problems. This architecture is specially efficient to deal with HTTP/S protocol because it never close connections unnecessarily. With this approach the server can recover the computation spent to establish the connection serving a number of request for each connection.

A remarkable characteristic of the event-driven archi-

tures is that there is no natural way of controlling the number of connected clients in the system (in a multithreaded server this is limited by the amount of worker threads in the server pool). This implies that in order to limit the maximum number of connected clients on the server (and in turn maintain an acceptable load level in the system to avoid the degradation of the response time and throughput of the server), an admission control[4] policy must be implemented. Additionally, as the number of worker threads can be very low (one should be enough) the contention inside the web container can be reduce to the minimum.

### Hybrid architecture

In this paper we evaluate a hybrid architecture that can take benefit of the strong points of both discussed architectures, the multithreaded and the event driven. In this hybrid architecture, the operation model of the event-driven architecture is used for the assignment of client requests to worker threads (instead of client connections) and the multithreaded model is used for the processing and service of client requests, where the worker threads will perform blocking I/O operations when required. This architecture can be used to decouple the management of active connections from the request processing and servicing activity of the worker threads. With this, the web container logic can be implemented following the multithreaded natural programming model and the management of connections can be done with the highest possible performance, without blocking I/O operations and reaching a maximum overlapping of the client think times with the processing of requests. A brief operation diagram for this architecture can be seen on figure 1(b).

In this architecture the acceptor thread role is maintained as well as the request dispatcher role from the

pure event-driven model and the worker thread pool (performing blocking I/O operations when necessary) from the pure multithreaded design. This makes possible for the hybrid model to avoid the need of closing connections to free worker threads without renouncing to the multithreading paradigm. In consequence, the hybrid architecture makes a better use of the characteristics introduced to the HTTP protocol in the 1.1 version, such as connection persistence, with the corresponding reduction in the number of client re-connections (and the corresponding bandwidth save). When the server is overloaded and the clients use the HTTP/S protocol the fact of keep connections open improve the throughput of the server because the computation spent in each connection established is recovered with the subsequents requests.

Additionally, and as it has been discussed for the event-driven architectures, some kind of admission control policy must be implemented in the server in order to maintain the system in an acceptable load level. More information about the hybrid architecture can be found at [3].

## 4 Testing environment

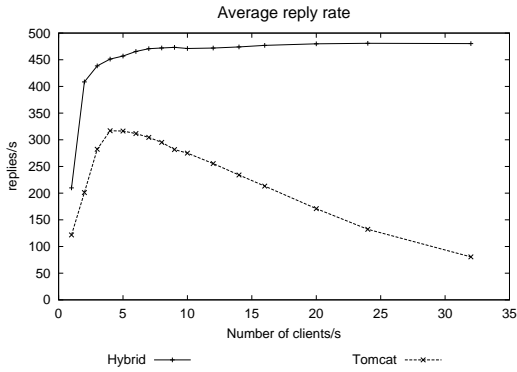
### 4.1 Web server and servlet container

We use the original Tomcat 5.5 [7] and the hybrid server (built upon Tomcat 5.5) as the application server. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and to be a quality production servlet container too. Tomcat can work as a standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). In this paper we use both web servers as a standalone server. Tomcat has a multithreaded architecture and follows a connection service schema where, at a given time, one thread (an `HttpProcessor`) is responsible of accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point, this `HttpProcessor` will be responsible of attending and serving the received requests through the persistent connection established with the client, while another `HttpProcessor` will continue accepting new connections. `HttpProcessors` are commonly chosen from a pool of threads in order to avoid thread creation overheads. We have configured Tomcat setting the maximum number of `HttpProcessors` to 100 and the connection persistence timeout to 3 seconds. In [3] we can found detailed information about the implementation of the hybrid architecture. The hybrid server was configured with one acceptor/selector thread and 10 `HttpProcessors`

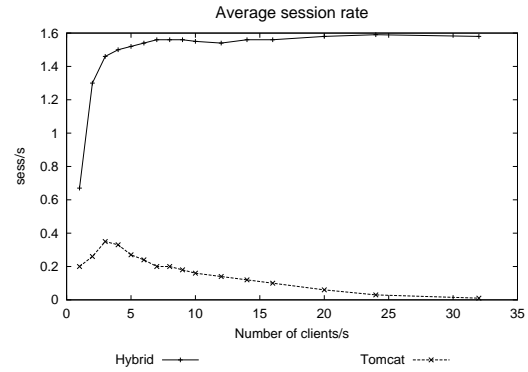
threads. No timeout is needed for the hybrid server.

### 4.2 Benchmark application and workload generator

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [1] benchmark servlets version 1.4 on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS defines 27 interactions. Among the most important ones are browsing items by category or region, bidding, buying or selling items and leaving comments on other users. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB. The client workload for the experiments was generated using a workload generator and web performance measurement tool called `Httpperf` [8]. This tool, which support 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 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 new clients per second initiating an interaction with the server. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP/S connection with the server. Using this connection, the client repeatedly makes a request (the client can also pipeline some requests), parses the server response to the request, and (optionally) follows a link embedded in the response. The workload distribution generated by `Httpperf` was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. This emulates the "thinking" period of a real client who takes a period of time before clicking on the next request. The think time is generated from a negative exponential distribution with a mean of 7 seconds. `Httpperf` allows also configuring a client timeout. If this timeout is elapsed and no reply has been received from the server, the current persistent connection with the server is discarded. We have configured `Httpperf` setting the client timeout value to 10 seconds. RUBiS



**Figure 2. Reply throughput comparison between Tomcat server and hybrid server**



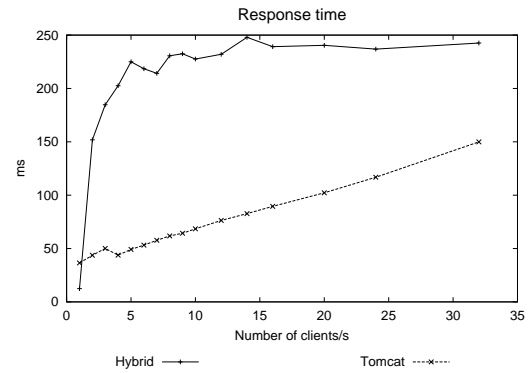
**Figure 3. Session throughput comparison between Tomcat server and hybrid server**

defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions. We use the browsing mix to evaluate the servers.

A secure dynamic content application is characterized by the long length of the user sessions as well as by the high computational cost of the first connection (initial SSL handshake) and subsequent request to be serviced (including embedded requests to external servers, such as databases).

### 4.3 Hardware & software platform

The hardware platform for the experiments presented in this paper is composed of a 4-way Intel Xeon 1.4 GHz with 2GB RAM to run the web servers and a 2-way Intel XEON 2.4 GHz with 2 GB RAM to run the benchmark clients. For the benchmark applications that require the use of a database server, a 2-way Intel XEON 2.4 GHz with 2 GB RAM was used to run MySQL v4.0.18, with the MM.MySQL v3.0.8 JDBC driver. All the machines were running a Linux 2.6 kernel, and were connected through a switched Gbit network. The SDK 1.5 from Sun was used to develop and run the web servers. All the tests are performed 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).



**Figure 4. Response time comparison between Tomcat server and hybrid server**

## 5 Experimental results

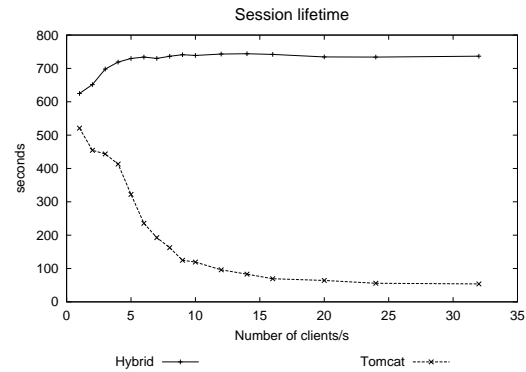
In this section we will evaluate and compare the performance of the proposed hybrid server architecture in front of the out-of-the-box Tomcat architecture, under a secure workload as well as under a plain workload.

The workload generator used for the experiments, Httpperf, was configured using a client timeout value of 10 seconds and according to the configuration described in section 4.2. Each individual benchmark execution had a fixed duration of 30 minutes.

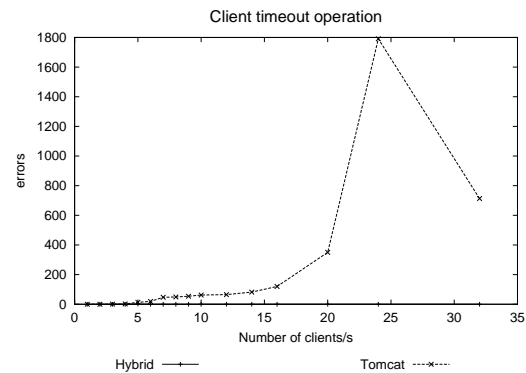
## 5.1 Secure dynamic content

Looking at figure 2, it can be seen that the throughput of the hybrid server is always better than the original Tomcat, moreover the difference between them grows up as the load is increased. When the load is low and the original Tomcat architecture is below saturation, the benefits of the hybrid architecture are already exposed. The use of the non-blocking I/O API (NIO), which offers a higher performance than the standard stream-based Java I/O, leads the hybrid architecture to offer a higher performance than the original Tomcat server. By when the saturation point of the Tomcat server is reached, the hybrid architecture is exposing a throughput that is a 50% higher than the original multithreaded architecture. When the load is increased beyond the saturation point, the performance of the hybrid server keeps nearly constant while the out-of-the-box Tomcat server starts reducing its output level linearly with the load increase. The benefits of the hybrid architecture beyond the saturation point are explained by the higher use of the connection persistence characteristics of the HTTP/1.1 protocol that the hybrid architecture makes in front of the original multithreaded approach. As more TCP connections are reused, more clients can keep their connections established and consequently less connection re-establishments will be required. When the considered workload uses data encryption, reducing the number of connection establishments is synonymous of less SSL handshakes negotiations and in consequence an important reduction of the processing requirements for the server system.

Usually, the workload produced over secure e-business applications is session-based. At the beginning, the client gets connected. After that, the session requests are issued and correspondingly processed by the server. Finally, when the session is finished, the client gets disconnected and the user session is considered completed. The need of client re-connections introduced by the limitations of the multi-threaded server model produces an increasing difficulty to complete user sessions successfully. This effect is shown in figure 5. As it can be seen, when the saturation point is reached, the average length of the user sessions successfully completed for the original Tomcat server starts decreasing. The reason of this phenomena is the higher number of re-connections needed by the longest sessions when the server is saturated under the multi-threaded architecture. In the hybrid architecture, instead, the independence between the number of connected clients and the number of processing threads in the server allows this architecture to avoid the need for re-connections and their associated SSL handshakes, which allows long sessions to



**Figure 5. Lifetime comparison for the sessions completed successfully under a secure dynamic content workload**



**Figure 6. Number of client timeouts under a secure dynamic content workload**

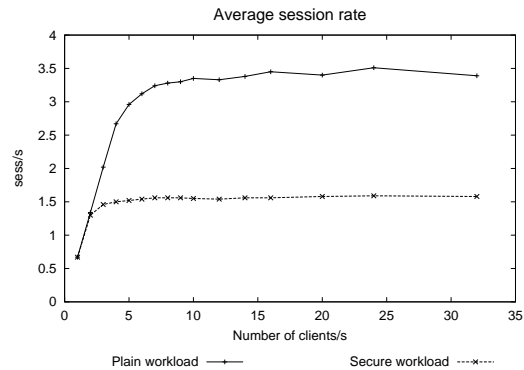
be completed without the difficulties introduced by the multi-threaded architecture.

Additionally to giving a higher chance to the long-lived user sessions to be completed, the hybrid architecture also increases the global average of sessions completed successfully, as it can be seen in figure 3. The original multithreaded Tomcat server starts reducing the number of sessions completed successfully beyond its saturation point. This is caused by the higher number of sessions that are aborted by the clients when numerous re-connections are rejected by the server when it is getting increasingly overloaded. As the load increases in the server, the chance for a connection attempt to be accepted is decreased. As user sessions, for

the multithreaded model server, require a number of re-connections each one, driving the server to higher loads results in a higher probability for the clients to consider the server unavailable if several connection attempts are rejected, which leads to a higher number of clients aborting their navigation sessions under these conditions. On the other hand, the hybrid architecture avoids the need for client re-connections for a user session to be completed, which allows a higher number of them to finish successfully, independently of the system load.

The benefits of the hybrid architecture presented above are not directly translated to the response time offered by it. As it can be seen in figure 4, the average response time obtained for the out-of-the-box Tomcat server is better than the obtained for the hybrid architecture, although it is not unacceptable. A response time bounded to less than 250ms is more than acceptable and, moreover, it remains constant with the load. In our opinion a slightly increase in the average response time obtained is more than tolerable considering the benefits presented for the server throughput obtained for the hybrid architecture.

Finally, another benefit of the hybrid architecture can be seen in figure 6, where it can be observed the number of requests sent to the server that not produce a response after an acceptable period of time. In the benchmark application this time is expressed as a timeout assimilated to the amount of time a human client would expect a reply from its web browser before considering a page request failed. For the experiments, this timeout value was set to 10 seconds. As it can be observed, the hybrid architecture produces no client timeouts while the number of errors generated by the pure multithreaded Tomcat server grows with the system load. This situation is produced, for the hybrid architecture, by the use of an overload control mechanism as well as by the independence between the number of connected clients to the server and the amount of concurrent server threads processing requests, being it possible to keep the latest in a low value and reducing in this way the contention caused by the multithreaded architecture as well as increasing the obtained performance for the server. It can also be seen that the number of timeout operation errors observed for the original Tomcat server decreases when the load level is driven beyond a certain point. This can be explained because under that level of overload, the server starts rejecting connections because no server capacity remains available to process the volume of incoming connection attempts and less clients remain connected concurrently.



**Figure 7. Session throughput for Hybrid server**

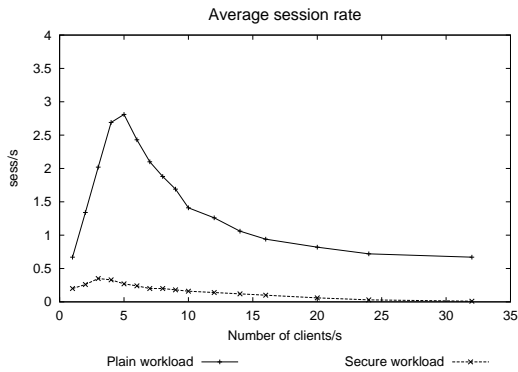
## 5.2 Secure vs plain dynamic content

Once it was proved that for secure content workloads the proposed hybrid architecture provides a higher performance than the original pure multithreaded Tomcat server in several ways, we wanted to measure the performance gap observed for both servers when subject to a plain workload and secure workload. This was achieved by rerunning the benchmark application with the HTTP protocol instead of HTTPS protocol used in the previous experiments. More results with plain workload are discussed in [3].

It can be seen in figures 7 and 8 that another remarkable effect of the hybrid architecture is that it reduces the impact on the performance caused by the introduction of security in the workload from a factor of 7 in the original Tomcat server to only 2 for the hybrid architecture when measuring the throughput in successfully completed user sessions.

## 6 Conclusions and future work

In this paper we have proved that the use of a hybrid web server architecture that takes the best of the two more extended server architectures, the multithreaded and the event-driven, can boost the performance of the server, especially under session-based workloads and even more when the workloads use encryption techniques. As it has been demonstrated, the benefits of this architecture are especially noticeable in the throughput of the server, in terms of individual requests as well as for user sessions, particularly when the server is overloaded. The modified Tomcat server beats the original multithreaded Tomcat in nearly all the



**Figure 8. Session throughput for Tomcat server**

performance parameters studied and minimizes the impact of incorporating secure connections to a web application with respect to the out-of-the-box Tomcat.

This work is a first step toward the creation of an autonomic web container, which will be deployed alone or as a web tier of an application server platform. The hybrid architecture reduces the complexity of the tuning of a web container (the size of the worker threads pool is only related to the server capacity and is not a limiting factor for the amount of connected clients as well as the server timeouts disappear), which is an important step toward the implementation of autonomic servers (see [6] for more details on the autonomic computing topic). Another interesting topic can be the integration and evaluation of this hybrid architecture in load balancers and clustering configurations.

## References

[1] 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.

[2] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the scalability of java event-driven web servers. In *2004 International Conference on Parallel Processing (ICPP'04)*, pages 134–142, 2004.

[3] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. A hybrid web server architecture for e-commerce applications. In *The 11th International Conference on Parallel and Distributed Systems (ICPADS 2005) July 20 - 22, 2005, Fukuoka, Japan, 2005*.

[4] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *INFOCOM*, 2002.

[5] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Characterizing secure dynamic web applications scalability. In *19th International Parallel and Distributed Processing Symposium, Denver, Colorado (USA). April 4-8, 2005, 2005*.

[6] IBM Research. *Autonomic computing*. See <http://www.research.ibm.com/autonomic>.

[7] Jakarta Project. Apache Software Foundation. *Tomcat*. See <http://jakarta.apache.org/tomcat>.

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

[9] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

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

[11] The Apache Software Foundation. *Apache HTTP Server Project*. See <http://httpd.apache.org>.

[12] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.