

Can Thread Priorities be Used to Provide Differentiated Quality of Service for Web Applications?

Javier Alonso, Jordi Guitart and Jordi Torres

Resumen—The e-commerce web sites receive a great and varied number of visitors every day. These visitors share the application server's limited resources and when there are too many clients connecting to the web site, it is possible that they hinder between them, even to overload the application server. These visitors can be divided in different categories, depending on their importance from site viewpoint. Considering the importance that in these web sites some client connections (e.g. buyers' connections) finish successfully before other connections, in this paper we propose a mechanism to provide different quality of service to the different client categories by assigning different priorities to the threads attending the connections. After observing that Java Thread Priorities are only applied within the JVM, and moreover, these priorities do not reach the O.S. threads, we propose to schedule threads using the Linux Real Time priorities. Our results demonstrate that different quality of service classes can be supported using this mechanism.

Palabras clave—SSL, Threads, Priority, QoS, Linux, JVM, Operating System

I. INTRODUCTION

Rapid maturing of web technology has increased considerably the number of users of e-commerce web sites. These users can be classified in three large and non-exclusive groups: curious, prospectors and buyers. Initially, we could think that in an e-commerce application there are only buyers, but that is not correct. In a virtual shop we can find people spending their free time and browsing, but they are not thinking of shopping beforehand. We call them curious customers. This people do not produce revenue, but in future, they can become potential buyers. On the other hand, we can find the prospector customers, that is people visiting the e-shop, finding information about prices and reading about products, but they are not necessarily thinking of buying anything at the moment. However, this target is a potential buyer in a short time. Finally, there are users visiting the web site to buy some product, i.e. the buyers. This group is the most important to the business and deserves special service, but we should not forget the other two groups of users because they are future potential buyers.

The load that the application servers that host e-commerce sites must support is increasing considerably, not only because the large number of visits, but also due to the special features of e-commerce sites:

dynamic web content and confidential information. In an e-commerce web site is essential to guarantee that the users' private data (i.e. bank account, address, etc.) are protected while transmitted over to open Internet.

Traditionally, HTTPS protocol [1] has been used to ensure the secure communication between the server and the clients over Internet. HTTPS, which is based on HTTP [2] over SSL [3], allows to create a secure channel between client and server, supplying authentication and privacy over the data transfer between both. Usually, only the server is authenticated while the privacy is guaranteed in both directions. The authentication is achieved by means of X.509 digital certificates [4].

Both dynamic web content and SSL algorithms require higher computation capacity than simpler sites. For this reason, their use can easily overload the server if the number of received connections temporarily exceeds the service capacity of the server. When an application server is overloaded, the response time can be unacceptable by the users and resources exhaustion can lead the server to work erratically or even crash, causing denial of services.

All the connections inside the application server are competing among themselves for resources. Consequently, in an overload situation, buyers' connections can be held up by other type of connections which could even stop the buyers' from completing successfully: either because the buyer is tired of waiting or because the server cannot attend the request. As one may see in the previous situation, there are connections that are more important for the business than others. The curious' connections should never hinder the prospectors' connections, less still the buyers' connections. Therefore, although the first and second groups are potential buyers, it is actual the buyers that are most important to bring in the revenue for the business.

In this paper, we present a mechanism for being able to prioritize, in a static or dynamic way client connections. This mechanism is especially intended for Multi-Thread Application servers (MTAs). In these servers, every thread is responsible of attending one client connection. We propose to assign different priorities to different threads (or thread groups) in the server, with the aim of guarantying that higher priority threads (i.e. higher priority connections) have enough resources to run and finish successfully in front of lower priority threads. For instance, we could always assign higher priority to the SSL con-

nections (usually, buyers' connections) than to non-SSL connections, or to some parts of the session path, or to the buyers' connections (if you can recognize them), with respect to curious and prospectors' connections.

As we focus our work on J2EE e-business application servers (Tomcat server in our case), our first attempt was to use the Java Thread Priority (JTP) to distinguish the different type of customers. When a thread uses the JTP, this thread will be prioritized by the JVM. However, our experiments in Linux showed that using priorities at this level does not offer a performance or response time improvement. This is because the Java priority is not directly inherited by the native thread. In fact, in Linux systems, all Java threads execute at system level in the same priority class (SCHED_OTHER), competing among themselves for the system resources. This competition is bigger when the server is overloaded.

In order to overcome this problem, we propose to translate Java thread priorities to the operating system level by using Linux Real Time Queues. With our approach, threads will be assigned to a given Real Time queue depending of its priority, guaranteeing in this way that higher priority threads will be scheduled before lower priority threads. Our results demonstrate that this solution allows to create different levels of quality of service according to the type of client.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 details the implementation of our proposal. Section 4 describes the experimental environment used in our evaluation. Section 5 shows the evaluation results obtained and finally, section 6 presents the conclusions of this paper and the future work proposed.

II. RELATED WORK

There has been considerable research about assigning different priorities to connections and try to minimize the response time. In this area, there are different approaches that can be summarized in request scheduling, admission control and service differentiation. Request scheduling refers to how the requests are sorted by the server. Traditionally, the request ordering policy is an operating system task. It is a well-known fact from Queue Theory literature [5], [6], [7], [8] that the Shortest Remaining Processing Time first (SRPT) scheduling reduces the queuing time thus reducing the response time.

There are diversity of techniques based on the SRPT algorithm suggesting policies to prioritize the service of short static content requests in front of long requests [9], [10]. SWIFT [11] presents a policy based on SRPT, but taking into account the distance between client and server and the size of the file requested by the client. With this technique they can obtain an response time improvement between 2.5 and 10%, for long file requests in front of techniques that only take into account the size of the file.

The rescheduling techniques have been proven ef-

fective in providing better response time to high priority requests, but in an overload situation, other techniques are necessary. Anyway, these techniques can be complementary to any other technique to improve the performance and response time in an overloaded server.

Admission control consists of reducing the server workload, limiting the number of accepted requests accepted, so that the server does not overload. Service differentiation is based on distinguishing different type of clients and offering different levels of quality of service to them, therefore, the resources are assigned to higher priority requests, which provokes more priority requests not to acknowledge other requests in the server. Usually, we find these both techniques joined in a lot of works. In this area we can find works as i.e. [12], which presents a proposal focused on admission control for SSL sessions. The policy prioritizes the resumed SSL sessions in front of new SSL sessions, because new SSL sessions require a lot of computation resources, due to the negotiation of full SSL handshakes.

ACES [13] tries to limit the number of accepted requests based on estimated service time, although ACES also carries service differentiation. This approach is only simulated. Other works are focused on dynamic content servers. SEDA [14] decomposes a complex service in multiple stages connected by queues. Admission control based on monitoring the response time in the stage can be performed in these queues. In [15], the authors present an admission control technique for e-commerce sites that observes the execution costs of requests, distinguishing between different request types. [16] proposes client differentiation based on connectivity quality between client and server. For example, if connectivity quality is bad for one client, the server selects a lower quality image to send to the client, and it can even select not to send any image, but only text to improve the response time.

Our solution is complementary and can be combined with any of these admission control or request scheduling techniques.

III. CONNECTION SCHEDULING BASED ON SYSTEM-LEVEL THREAD PRIORITIES

Linux is a multitask Operating System. Therefore, it supports more than one process running on the system. Moreover, Linux takes an unique approach for implementing the process and thread abstractions. In Linux, every thread is understood as a process, and for this reason, it is used the task name, not process or thread. Because of the Linux capacity to support more than one task running concurrently, it is necessary some scheduling mechanism to ensure that tasks can run on the system as if they were running alone on the system.

In our work, we have been carrying our experiments with the Linux kernel-2.6.8.1 scheduler [17]. This scheduler offers three different scheduling policies, one for normal processes and two for real-time

applications. A static priority value `sched_priority` is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible `sched_priority` value, and `sched_priority` can have a value in the range 0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list.

Usually, when a task is created, it is assigned to the default universal time-sharing scheduler policy (`SCHED_OTHER`), while `SCHED_FIFO` and `SCHED_RR` are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution. Processes scheduled with `SCHED_OTHER` must be assigned the static priority 0, processes scheduled under `SCHED_FIFO` or `SCHED_RR` can have a static priority in the range 1 to 99. Only processes with superuser privileges can get a static priority higher than 0 and can therefore be scheduled under `SCHED_FIFO` or `SCHED_RR`. All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned into its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

`SCHED_FIFO` can only be used with static priorities higher than 0, which means that when a `SCHED_FIFO` process becomes runnable, it will always preempt immediately any currently running normal `SCHED_OTHER` process. `SCHED_FIFO` tasks schedule in a first-in-first-out manner without time slicing. `SCHED_RR` is a simple enhancement of `SCHED_FIFO`. Everything described above for `SCHED_FIFO` also applies to `SCHED_RR`, except that each process is only allowed to run for a maximum time quantum. Finally, `SCHED_OTHER` is the standard Linux time-sharing scheduler that can only be used at static priority 0. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level (set by the `nice` or `setpriority` system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all `SCHED_OTHER` processes.

The Java HotSpot [18] virtual machine (JVM) actually associates each Java thread with a unique pthread [21]. The implementation of Pthreads library for Linux associates each pthread with a unique native thread. Therefore, the relationship between the Java thread and the native thread is stable and persists for the lifetime of the Java thread. All the native threads, which are associated with Java threads, are assigned to the default scheduling policy (`SCHED_OTHER`) independently of the Java thread

priority, thus even though the Java threads have a high priority, this priority is not translated into a Linux static priority, thus this Java thread will not preempt lower priority threads.

The only effect of assigning a higher priority to Java threads is that those native threads associated with that Java threads will have a higher dynamic priority, but this is a limited improvement as shown in [19]. In addition, this only occurs from JVM 1.5.0. Summarizing, the JVM cannot change the scheduling policy, so the Java threads (really, the native threads) are always running with the default policy (all of them with the same static priority). According to this, the possibility of increasing the priority of Java threads does not contribute to a noticeable performance improvement of the higher priority threads regarding lower priority threads.

Our solution expects to take advantage of the different scheduling schemes to provide different levels of quality of service (QoS) to clients in an application server as Tomcat [20]. Our proposal exploits the performance characteristics of `SCHED_OTHER`, `SCHED_RR` and `SCHED_FIFO` and the possibility of fine-grain performance adjustment inside every scheduling policy.

As the JVM does not offer the possibility of directly using the different scheduling schemes mentioned before, we must modify the Tomcat server. In our proposal, the application server manages its threads, and configures them with the appropriated scheduling scheme. Changing the scheduling scheme is accomplished using the `pthread_setschedparam` method offered by the Pthread library. In order to call this native method from the server code (written in Java), we have used the Java Native Interface (JNI)[22], which allows to invoke native methods from Java methods. In this way, Tomcat will be able to configure at runtime all its threads to run using any of three scheduling policies, as well as modifying the thread static priority, providing different QoS to the clients being attended by these threads.

To implement our proposal, we have modified the method used by Tomcat to create thread pools. The `HttpProcessors` are joined in pools, and every pool is listening from a different port TCP. Our modification of thread pool creation, allows each pool to be assigned to a scheduling policy, thus every thread in this pool will be running with the same scheduling policy. For our experiments, we use two different SSL thread pools. One pool will be running with `SCHED_OTHER` policy and the other will be running with `SCHED_RR`, demonstrating that the clients attended by the pool of threads that is running with `SCHED_RR` are obtaining better QoS than the clients attended by the `SCHED_OTHER` pool of threads. We used SSL connections because these connections spend more resources than non-SSL connections and it is easier to see the proposal effects. Section 5 presents our experimental results, showing the possibilities of this technique.

IV. EXPERIMENTAL ENVIRONMENT

A. Tomcat servlet container

As commented, we use Tomcat v5.0.29 [20] as the application server. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and to be a quality production servlet container too. Tomcat can work as a standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). In this paper we use Tomcat as a standalone server.

Tomcat follows a connection service schema where, at a given time, one thread (an `HttpProcessor`) is responsible of accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point, this `HttpProcessor` will be responsible of attending and serving the received requests through the persistent connection established with the client, while another `HttpProcessor` will continue accepting new connections. As only one thread is the responsible of attending and serving the requests for a given client, if we increase this thread priority at system level by using a real-time policy, the client will receive a better service. `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 150 and the connection persistence timeout to 10 seconds.

B. Client workload

The client workload for the experiments was generated using a workload generator and web performance measurement tool called `Httpperf` [23]. 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 concurrent clients interacting 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. `Httpperf` generates a simple workload requesting static web content to the application server over secure connections. We consider that this simple workload is enough to demonstrate the benefit of our proposal. Nevertheless, we are currently experimenting with dynamic web content workloads. 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. `Httpperf` allows also configuring a

client timeout. If this timeout is elapsed and no reply has been received from the server, the current persistent connection with the server is discarded, and a new emulated client is initiated. We have configured `Httpperf` setting the client timeout value to 10 seconds.

C. Hardware and software platform

Tomcat runs on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We have also two 2-way Intel XEON 2.4 GHz with 2 GB RAM machines running the workload generators (`Httpperf` 0.8). Each client emulation machine emulates the configured number of clients performing requests to the server during 10 minutes using the browsing mix (read-only interactions). All the machines run the 2.6.8.1 Linux kernel. All the machines are connected through a 1 Gbps Ethernet interface. 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, which we have proven to be enough to avoid memory being a bottleneck for performance. All the tests are performed with the common RSA-3DES-SHA cipher suit.

V. RESULTS

In this section we present the evaluation results of the QoS mechanism on Tomcat server, comparing the results obtained with the original Tomcat when using Java thread priorities and with the original Tomcat when no priorities are used.

A. Maximum achievable performance

In this section, we present the performance results when running the Tomcat server with an unique class of clients. These results will be the reference of the maximum achievable performance. Figure 1 shows the Tomcat throughput as a function of the number of new clients per second initiating a session with the server, comparing the Tomcat throughput when the threads run in `SCHED_OTHER` versus when the threads run in `SCHED_RR` with a maximum priority (99). Notice that achieved throughput in both cases is similar. There is only a little throughput improvement when the server is overloaded (from 22 new clients per second), because the server real-time threads are scheduled before the rest of user threads, even before the system ones, while if the server uses `SCHED_OTHER` threads, these have to compete with the rest of user threads, as well as the system ones. This is also manifested when analyzing the server response time, which is shown in Figure 2. Notice that, when the server is overloaded, average response time is over 90 milliseconds when running with real-time threads, and over 100 milliseconds when running with normal threads.

B. Original Tomcat vs. Tomcat with Java priorities

In this section, we evaluate how Java priorities can help us to offer QoS for two different classes of clients (Gold and Silver) in our web site. To do this, we have

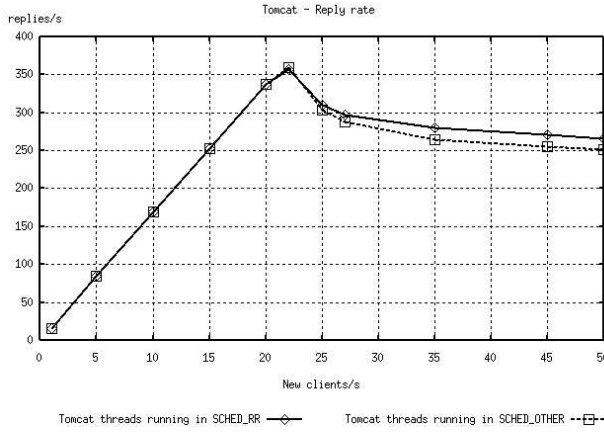


Fig. 1. Tomcat throughput with only 1 client class

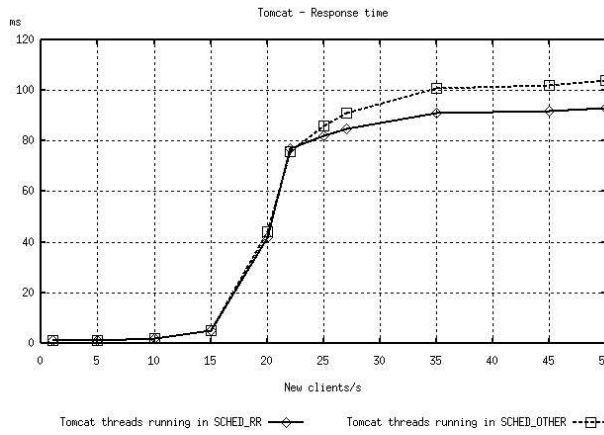


Fig. 2. Tomcat response time with only 1 client class

configured the server with two thread pools attending client SSL requests in two different TCP ports. In the original Tomcat both thread pools have the same priority, but in the Tomcat with Java priorities, the Gold Clients' requests will be attended by a thread pool with high Java priority, while the Silver Clients' requests will be attended by a thread pool with low Java priority. In our experiment, the Gold clients' rate varies from 1 to 50 new clients per second, while Silver clients' rate is fixed. We have made experiments with different fixed rates, in order to analyze how Silver clients affect Gold ones, but in this paper we only present the results with 2 and 22 new clients per second as fixed rate (which represent the situations where the server is not overloaded and overloaded, respectively).

As shown in Figure 3, when the Silver clients' rate is low, i.e. 2 clients per second, we observe that both original Tomcat and Tomcat with Java priorities obtain almost the maximum achievable throughput (see Figure 1) as for both Gold and Silver clients. Only the throughput of Gold clients with original Tomcat is a little lower than with Tomcat with Java Priorities. However, if we analyze the response time, which is shown in Figure 4, we observe that, although the response time of Tomcat with Java priorities is slightly better than with the original Tomcat for both clients' classes, the response time of Silver clients is

better than Gold clients' one, which denotes that Tomcat with Java priorities is not providing higher priority clients with better service.

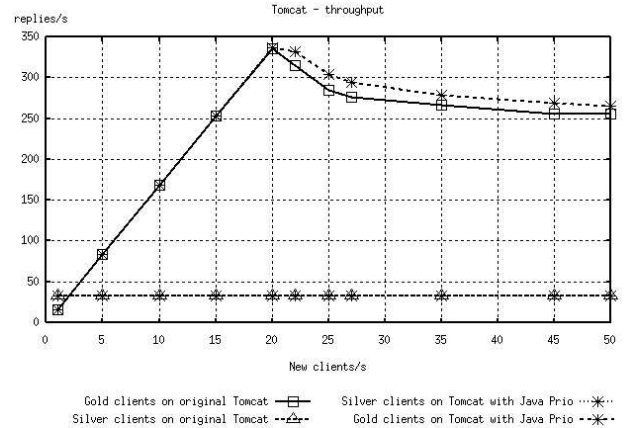


Fig. 3. Throughput with Java priorities vs. original Tomcat with low competition

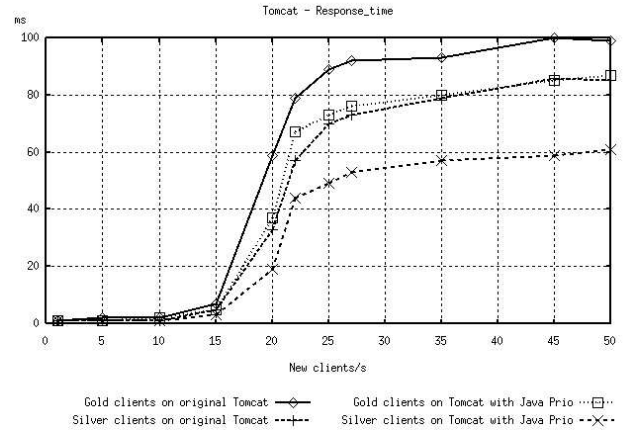


Fig. 4. Response time with Java priorities vs. original Tomcat with low competition

However, as shown in Figure 5, when the Silver clients' rate is high enough to overload the server, i.e. 22 new clients per second, throughput of the original Tomcat and Tomcat with Java priorities is the same, being considerably far from maximum achievable throughput for both clients classes. This occurs because Java priorities cannot ensure more resources to Gold clients, demonstrating that, in overload conditions, Java priorities are not able to offer a real QoS client differentiation. In Figure 6, we observe that, although the response time of both clients' classes with Tomcat with Java priorities is better than the response time over original Tomcat, Gold clients' response time is not better than Silver clients' one. This confirms that Java priorities cannot be used to provide differentiated QoS to clients.

C. Modified Tomcat with QoS vs. Tomcat with Java priorities

In this section, we evaluate how performs our modification of Tomcat server with respect to the original Tomcat using Java priorities in order to provide differentiated QoS for two different classes of

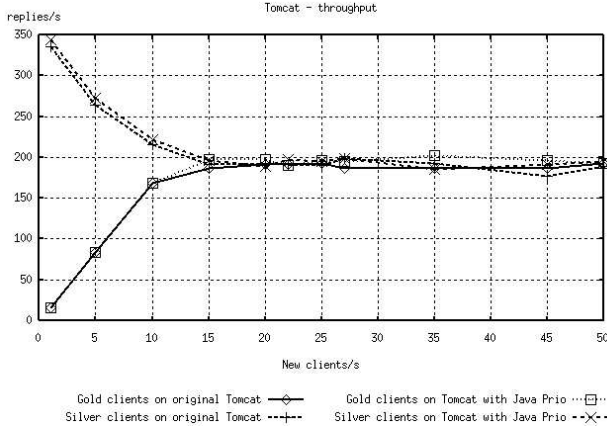


Fig. 5. Throughput with Java priorities vs. original Tomcat with high competition

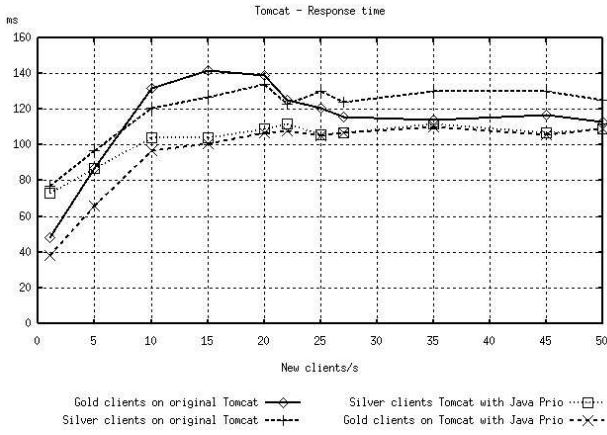


Fig. 6. Response time with Java priorities vs. original Tomcat with high competition

clients (Gold and Silver). We have configured the server with two thread pools attending client SSL requests in two different TCP ports. In the original Tomcat, both pools are running by default in the `SCHED_OTHER` scheme, but with two different Java priorities (10 for Gold clients and 1 for Silver clients). In our Tomcat modification, the thread pool attending requests from the Gold clients runs in the `SCHED_RR` with the maximum priority (99), while the thread pool attending requests from the Silver clients runs in the `SCHED_OTHER` scheme. We have repeated the experiment several times: Gold clients' rate varies from 1 to 50 new clients per second, while Silver clients' rate is fixed. We have made experiments with different fixed rates, in order to analyze how Silver clients affect Gold ones, but in this paper we present the results with 2 and 22 new clients per second as fixed rate (which represent the situations where the server is not overloaded and overloaded, respectively).

As shown in Figure 7, when the Silver clients' rate is low, i.e. 2 new clients per second, we observe that both original Tomcat and Tomcat with QoS obtain almost the maximum achievable throughput (see Figure 1) as for Gold clients as for Silver clients. Only the throughput of Gold clients with original

Tomcat is a little lower than with Tomcat with QoS because this favors Gold clients with more resources. However, when analyzing the response time, which is shown in Figure 8, we see that original Tomcat is able to maintain good response time for both clients classes, but Tomcat with QoS can maintain only response time for Gold clients (response time for Silver clients has increased). This is one side effect of our proposal. It is intended for situations of high competition among clients and it should not be used unnecessarily, because it can be harmful for low priority applications response time.

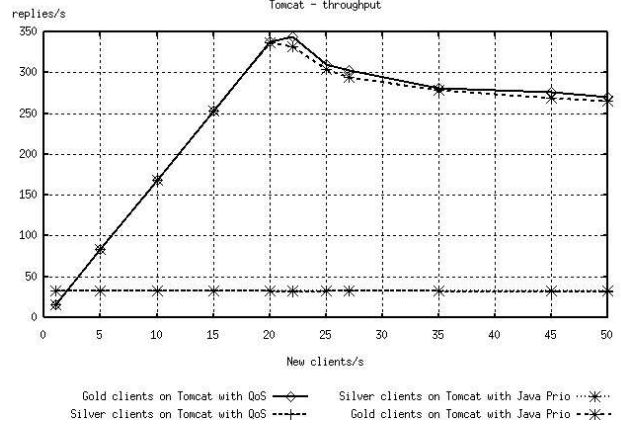


Fig. 7. Throughput with Java priorities vs. QoS mechanism with low competition

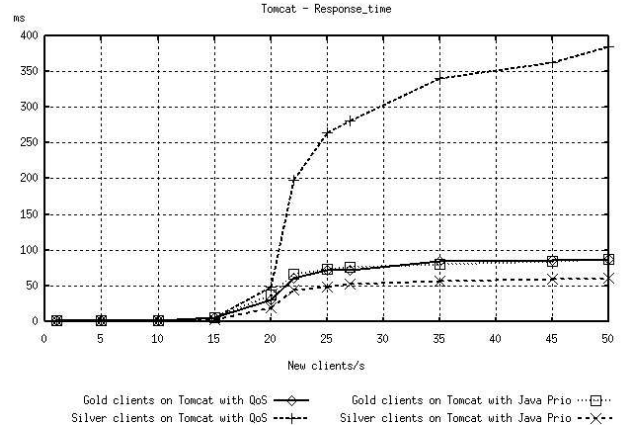


Fig. 8. Response time with Java priorities vs. QoS mechanism with low competition

However, as shown in Figure 9, when the Silver clients' rate is high, i.e. 22 new clients per second, we observe that throughput achieved with original Tomcat is considerably far from maximum achievable throughput for both clients classes (throughput for Gold clients and Silver clients is only 200 replies/s while maximum achievable throughput was over 300 replies/s). This is produced because server resources are limited and must be shared by both clients classes. With original Tomcat with Java priorities, resources are equally distributed among clients. On the other side, Gold clients' throughput obtained with Tomcat with QoS is considerably higher than Silver clients' one. While server has enough re-

sources for both clients classes, their throughput increases linearly, but when resources are exhausted, Gold clients's throughput still increases until being close to the maximum achievable throughput while Silver clients's throughput drops to only around 50 replies/s. This occurs because Tomcat with QoS prioritizes Gold clients in the resource assignation. In the same way, response time for Gold and Silver clients, which is shown in Figure 10, is practically the same (around 100 ms) when running with the original Tomcat, but when running with Tomcat with QoS Gold clients have a response time considerably lower than Silver clients and even lower than Gold and Silver clients when using original Tomcat with Java priorities. In fact, Gold clients' response time is the same as when Gold clients are alone in the server, which is shown in Figure 2. Therefore, we can conclude that original Tomcat with Java priorities is unable to offer differentiated QoS to server clients but this is possible when using our modification of Tomcat server that uses system-level priorities to schedule the connections.

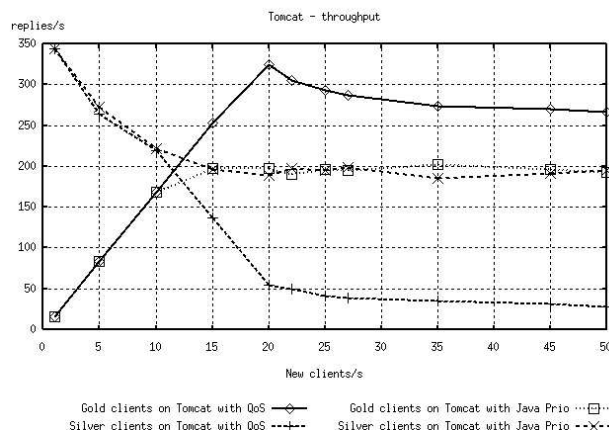


Fig. 9. Throughput with Java priorities vs. QoS mechanism with high competition

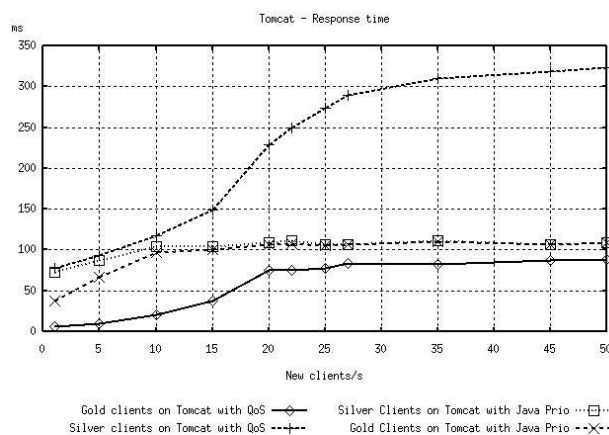


Fig. 10. Response time with Java priorities vs. QoS mechanism with high competition

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a mechanism that provides differentiated QoS to clients in a secure web

environment. Our mechanism assigns different priorities to the threads that attend the clients requests, depending on the type of client. As priorities at the JVM level are not considered for the scheduling of threads at system level, we propose to assign directly system level priorities to Java threads. This is accomplished using the Linux real-time scheduling policies, which ensure that higher priority processes are always executed before than lower priority processes. Our evaluation demonstrates the benefit of our approach offering differentiated QoS to the clients. In addition, our client differentiation technique can be combined with any other admission control or request scheduling technique.

Currently, we are experimenting this mechanism in a dynamic web content environment. Our experimental environment includes a deployment of the RUBiS (Rice University Bidding System) [24] benchmark servlets version 1.4 on Tomcat. Our future work considers extending the mechanism not only to provide different QoS depending on the type of client, but also to dynamically modify client requests priority depending on request importance for the site (for instance, requests can have higher priority when client is approaching to the checkout stage in an on-line store).

REFERENCIAS

- [1] E. Rescorla, *HTTP over TLS*, RFC 2818. May 2000.
- [2] Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and T. Berners-Lee, *Hypertext Transfer Protocol - HTTP/1.1*, RFC 2068. January 1997.
- [3] A. O. Freier, P. Karlton and C. Kocher., *The SSL Protocol*, Version 3.0. November 1996
- [4] R. Housley, W. Polk, W. Ford, D. Solo, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 3280. April 2002.
- [5] L. Schrage and L. W. Miller, *The queue M/G/1 with the shoLLcst remaining processing time disciplin*, Oper. Res., vol. 14, pp. 670-684. 1966.
- [6] L. Schrage, *A proof of the optimality of the shortest remaining processing time disciplin*, Oper. Res., vol. 16, pp. 687-690. 1968.
- [7] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Reading, MA: Addison-Wesley. 1967.
- [8] Goerg, C. *Evaluation of the Optimal SRPT Strategy with Overhead*, Communications, IEEE Transactions on, vol. 34, pp.338-344. Apr 1986
- [9] M. Crovella, R. Frangioso and M. Harchol-Balter, *Connection Scheduling in Web Servers*, 2nd Symposium on Internet Technologies and Systems (USITS99), Boulder, Colorado, USA. October 11-14, 1999.
- [10] M. Harchol-Balter, B. Schroeder, N. Bansal and M. Agrawal. *Size-based Scheduling to Improve Web Performance*, ACM Transactions on Computer Systems (TOCS), Vol. 21 (2), pp. 207-233. May 2003.
- [11] Mayank Rawat and Ajay Kshemkalyani *SWIFT: Scheduling in Web Servers for Fast Response Time*, In Second IEEE International Symposium on Network Computing and Applications. April 2003.
- [12] J. Guitart, V. Beltran, D. Carrera, J. Torres, E. Ayguad. *Session-Based Adaptive Overload Control for Secure Dynamic Web Application*, The 2005 International Conference on Parallel Processing (ICPP-05), Oslo, Norway. June 14-17, 2005.
- [13] X. Chen, H. Chen and P. Mohapatra. *ACES: An Efficient Admission Control Scheme for QoS-Aware Web Servers*, Computer Communications, Vol. 26 (14), pp. 1581-1593. September 2003.
- [14] M. Welsh and D. Culler. *Adaptive Overload Control for Busy Internet Servers*, 4th Symposium on Internet Technologies and Systems (USITS03), Seattle, Washington, USA. March 26-28, 2003.

- [15] S. Elnikety, E. Nahum, J. Tracey and W. Zwaenepoel. *Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites*, 13th International Conference on World Wide Web (WWW04), pp. 276-286, New York, New York, USA. May 17-22, 2004.
- [16] Krishnamurthy, B., Wills, C. E. *Improving Web Performance by Client Characterization Driven Server Adaptation*, In Proc. WWW2002 pp. 305316. Honolulu, HI, May 2002.
- [17] Josh Aas. *Understanding the Linux 2.6.8.1 CPU Scheduler*, 2005 Silicon Graphics, Inc. (SGI). 17th February 2005
- [18] SUN microsystems, Inc. *Thread Priority on the Solaristm Platform*, <http://java.sun.com/j2se/1.5.0/docs/guide/vm/thread-priorities.html>
- [19] Sowmya Manjanatha *Inferring characteristics of the JVM threads implementation*, Boston University, Boston MA. <http://cs-people.bu.edu/sowmya/htdocs/threadsTest.html>
- [20] The Apache Software Foundation *The Apache Jakarta Tomcat 5 Servlet/JSP Container*, <http://tomcat.apache.org/>
- [21] Xavier Leroy. *The LinuxThreads library*, <http://pauillac.inria.fr/~xleroy/linuxthreads/index.html>
- [22] SUN Microsystems, Inc. *Java Native Interface*, <http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html>
- [23] D. Mosberger and T. Jin. *httpperf: A Tool for Measuring Web Server Performance*, Workshop on Internet Server Performance (WISP98) (in conjunction with SIGMETRICS98), pp. 59-67. Madison, Wisconsin, USA. June 23, 1998.
- [24] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. *Specification and Implementation of Dynamic Web Site Benchmarks*, IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, Texas, USA. November 25, 2002.