Differentiated Quality of Service for e-Commerce Applications through Connection Scheduling based on System-Level Thread Priorities

Javier Alonso
alonso@ac.upc.edu

Jordi Guitart
jguitart@ac.upc.edu

Jordi Torres
torres@ac.upc.edu

Barcelona Supercomputing Center
Technical University of Catalonia - Computer Architecture Department
C/ Jordi Girona 1-3 Building C6 Campus Nord, 08034 Barcelona, Spain

Abstract

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.

1 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, curious customers. 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. 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 because they are future potential buyers. All the connections inside the application server are competing among themselves for resources. Consequently, in an overload situation, buyers can be held up by other type of customers 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.

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 connections have enough resources to run and finish successfully in front of lower priority connections. For instance, we could assign higher priority 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, our first attempt was to use the Java Thread Priority to distinguish the different type of customers. However, our experiments 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 order to overcome this problem, we propose to translate Java thread priorities to the operating system level by using Linux Real Time Queues. Our results demonstrate that this solution allows to create different levels of quality of service according to the client’s type.

The rest of the paper is organized as follows: Section 2
2 Related work

There has been considerable research about how to distribute the servers resources between connections and try to minimize their 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 [2] 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 using different metrics to decide the new requests order [5]. 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. These techniques have been proven effective in providing better response time to high priority requests, but in an overload situation, other complementarities are necessary.

Admission control consists of reducing the server workload, limiting the number of accepted requests accepted, so that the server does not overload. In this area we can find works as i.e. [3], which presents a proposal focused on admission control for SSL sessions.

Usually, we find all of these techniques joined in a lot of works [4]. Our solution is complementary and can be combined with any of these admission control or request scheduling techniques.

3 Connection Scheduling based on System-Level Thread Priorities

In our work, we have been carrying our experiments with the Linux kernel-2.6.8.1 scheduler [6]. 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 from 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 list’s head. 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 (only with superuser privileges) scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99. 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 tasks schedule in a first-in-first-out manner without time slicing. SCHED_RR is a simple enhancement of SCHED_FIFO. Every SCHED_RR process is only allowed to run for a maximum time quantum. Finally, SCHED_OTHER 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 [7] virtual machine (JVM) actually associates each Java thread with a unique pthread [10]. 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, the 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 [8]. In addition, this only occurs from JVM 1.5.0. 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 [9]. 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. To implement our proposal, we have modified the method used by Tom-
Figure 1. Throughput with 1 client class

Figure 2. Response time with 1 client class

cat 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 SCHEDRR, demonstrating that the clients attended by the pool of threads that is running with SCHEDRR are obtaining better QoS than the clients attended by the SCHED_OTHER pool of threads. We used SSL connections (HTTPS [1]) because these connections spend more resources than non-SSL connections and it is easier to see the proposal effects.

4 Experimental Environment

We use Tomcat v5.0.29 [9] as the MTA application server. Tomcat is an open-source servlet container developed under the Apache license. In this paper we use Tomcat as a standalone server. We have configured Tomcat setting the maximum number of threads to 150 and the connection persistence timeout to 10 seconds.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [11]. The configuration parameters of the benchmarking tool used for the experiments presented in this paper were set to create a realistic workload. We consider that a simple workload requesting static web content is enough to demonstrate the benefit of our proposal. We have configured Httperf setting the client timeout value to 10 seconds. 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 running the workload generators. All the machines run the 2.6.8.1 Linux kernel with Sun JVM 1.4.2, which are connected through a 1 Gbps Ethernet interface.

5 Results

In this section we present the evaluation comparation of the QoS mechanism on Tomcat with respect to the original Tomcat when using Java thread priorities. By spaces reasons, the evaluation comparison of the original Tomcat when no priorities are used with the original Tomcat when using Java thread priorities can be found in companion report to this paper [12], which shows that both Tomcats obtain almost the same performance results.

5.1 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/sec initiating a session with the server, comparing the Tomcat throughput when the threads run in SCHED_OTHER versus when the server uses SCHEDRR threads, these have to compete with all threads running over the system. This is also manifested when analyzing the server response time, which is shown in Figure 2.

5.2 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 Tom-
cat using Java priorities in order to provide differentiated QoS for two different classes of 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 (GoldC) and 1 for Silver clients (SilverC)). In our Tomcat modification, the thread pool attending requests from the GoldC runs in the SCHED\_RR with the maximum priority (99), while the thread pool attending requests from the SilverC runs in the SCHED\_OTHER scheme. We have repeated the experiment several times: GoldC’ rate varies from 1 to 50 new clients/sec, while SilverC’ rate is fixed. We have made experiments with different fixed rates, in order to analyze how SilverC affect Gold ones, but in this paper we present the results with 2 and 22 new clients/sec as fixed rate (which represent the situations where the server is not overloaded and overloaded, respectively).

As shown in Figure 3, when the SilverC’ rate is low, i.e. 2 new clients/sec, we observe that both original Tomcat and Tomcat with QoS obtain almost the maximum achievable throughput (see Figure 1) as for GoldC as for SilverC. Only the throughput of GoldC with original Tomcat is a little lower than with Tomcat with QoS because this favors GoldC with more resources. However, when analyzing the response time, which is shown in Figure 4, 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 GoldC (response time for SilverC 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. However, as shown in Figure 5, when the SilverC’ rate is high, i.e. 22 new clients/sec, we observe that throughput achieved with original Tomcat is considerably far from maximum achievable throughput for both clients classes. 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, GoldC’ throughput obtained with Tomcat with QoS is considerably higher that SilverC’ one. While server has enough resources for both clients classes, their throughput increases linearly, but when resources are exhausted, GoldC’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 GoldC in the
 resource assignment. In the same way, response time for Gold and Silver clients, which is shown in Figure 6, is practically the same (around 100 ms) when running with the original Tomcat, but when running with Tomcat with QoS GoldC have a response time considerably lower than SilverC and even lower than Gold and Silver clients when using original Tomcat with Java priorities. In fact, GoldC’ response time is the same as when they 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.

6 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. 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 online store).

7 Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01.

References