QoS-aware switching policies for a locally distributed Web system

Mauro Andreolini, Emiliano Casalicchio
University of Roma Tor Vergata
Roma, Italy 00133
{andreolini, ecasalicchio}@ing.uniroma2.it

Michele Colajanni, Marco Mambelli
University of Modena
Modena, Italy 41100
{colajanni, mambelli.marco}@unimo.it

Abstract

We present the implementation and experiments of a Web switch that transforms a cluster-based Web system (*Web cluster*) with best-effort management policies into a system that gives guaranteed performance to different classes of users and applications.

Keywords: Distributed Web systems, Quality of Service, Performance evaluation, Dispatching algorithms.

1 Introduction

The Web is becoming a more mature and business-oriented media, so the need for differentiated users and services is much stronger. This differentiation is desired to accommodate heterogeneous application requirements and user expectations, and to permit differentiated pricing for content hosting or service providing.

In this study, we work on the server side of the Web and implement a prototype satisfying "Quality of Web Services" (*QoWS*) features that are inspired by the well known network QoS principles: service classification, performance isolation, high resource utilization and request admission. In particular, we realize a layer-7 Web switch which may implement several QoWS policies without having to modify the software of other components of the Web cluster, as proposed by other papers in this area [2, 5, 1].

2 Prototype architecture

We consider a multiple node architecture, namely *Web cluster*, as a platform to introduce QoWS mechanisms. A Web cluster refers to a Web site publicized with one hostname that uses two or more server machines housed together in a single location to handle user requests. It has a front-end component, *Web switch*, that acts as the network representative for the Web site, a layer of Web servers for static requests and a layer of back-end servers for dynamic requests. Our prototype is a *two-way* Web cluster, where the QoWS-enhanced Web switch operates at the layer-7 of the OSI protocol stack and implements access control and request dispatching among the server nodes.

Different proposals for introducing QoS and QoWS require modifications in the software of the HTTP server [2, 5], or at the operating system level [1]. We propose a QoWS-enhanced Web switch which may implement several policies for service classification and request admission control, because it works at layer-7. The other two goals of QoS, that is, performance isolation and high resource utilization are achieved by partitioning system resources among the classes of services and users. Since each client request is served by one server (Web and/or back-end), we can consider as resource unit of the Web cluster an entire Web or back-end server. Other research results consider finer grain system resources, such as CPU, disk of a server node.

In this paper we consider three policies that implement QoWS mechanisms (major details can be found in [3].

SwitchAdm works on service classification and resource admission. It does not admit requests when the current sum of the server loads in the Web cluster exceeds a given threshold. Once the request has been admitted to the system, it uses the Weighted Round Robin (WRR) policy to select the target server.

StaticPart adds to SwitchAdm a policy to statically partition the servers in as many sets as the classes of services provided by the Web cluster.

DynamicPart adds to StaticPart a periodic evaluation of the system load. This algorithm can dynamically change the server partitions, if the previous choice does not allow to satisfy the Service Level Agreement (SLA) for the most important service/user classes.

The implementation of the QoWS-enhanced Web cluster is based on off-the-shelf hardware and software components. The nodes of the system are connected through a switched 100Mbps Fast-Ethernet that does not represent the bottleneck for our experiments. The servers run on 6 dual processor PentiumIII-800MHz with 256MB of memory. All nodes of the cluster use a 3Com 3C905B 100bTX network interface, and an IBM Ultra ATA/100 disk with transfer rate of 37MBps and seek time of 8.5 msec. Each node is equipped with Linux operating system (kernel release 2.4.12), and Apache 1.3.12 Web server software.

The layer-7 Web switch is implemented on a dedicated machine having the same characteristics as the server node. It is implemented through the Apache Web server software by selecting the surrogate reverse proxy approach proposed in [4]. The Apache Web server running on the Web switch node is configured as a surrogate (reverse proxy) through the *mod_proxy* and *mod_rewrite* modules. The dispatching algorithms are implemented as C modules that are activated once at startup of the Apache servers. Because the Web switch requires information about the server load, each server uses a *Collector* C module that collects the number of active HTTP connections, by differentiating requests for static and dynamic pages. Every 10 seconds the *Manager* C module running on the Web switch gathers server load information through a socket communication mechanism.

For the experiments described in the following section we consider two classes of users (a *top class* and *normal class*) and two classes of services (*static* and *dynamic* requests). The Service Level Agreement (SLA) for the top class users states that the "*95-percentile of the latency time of static and dynamic requests must be less than T_S seconds and T_D seconds, respectively*". The normal class requests receive best effort services.

The SLA targets that are suitable for the available Web cluster architecture are evaluated through a methodology that cannot be reported here. We have evaluated the server and Web switch capacity for various service classes. From these performance results, we have that the SLAs for the top class users can be set to $T_S = 0.6$ and $T_D = 1$ seconds for static and dynamic requests, respectively.

3 Experimental results

In this section we verify whether the proposed QoWS-aware policies and mechanisms satisfy all SLAs for different workload scenarios, and we compare the performance of the QoWS-enabled Web cluster against a QoWS-blind system. As synthetic workload generator, we use a modified version of the Webstone benchmark tool. The main differences concern the introduction of the HTTP/1.1 protocol, and various modifications on client requests (e.g., user session, user think-time, embedded objects per Web page, file sizes and popularity) that are basically inspired to the SURGE model.

Figures 1 shows the 95-percentile of page latency time of *top* static requests. This figure shows that there is no particular problem to satisfy the SLA for static requests. This is also due to the fact that the most popular static documents are served from the cache of the Web server system rather than from the disk. All QoWS-aware policies are able to achieve the SLA target set to $T_S = 0.6$, whereas a QoWS-blind dispatching policy, such as WRR, is unable to satisfy the SLA when the system load augments. Low percentages of *normal* requests must be rejected to guarantee SLAs (Figure 2): less than 4% for the StaticPart policy, less than 2% and 1% for the DynamicPart and SwitchAdm policies, respectively.

Figures 3 shows the 95-percentile of the page latency time for the *top* dynamic requests. Now, the SLAs are much more critical for the QoWS-enabled Web cluster. From this figure we observe that the QoWS-aware policies are able to satisfy the SLA until the offered load is below 1500 clients per second. After that point, only the DynamicPart algorithm is able to guarantee the SLA until the Web cluster receives 2100 clients per second. These results are achieved at the price of high percentages of dropped requests coming from normal clients, going from 20% for the DynamicPart to 35% for the StaticPart (Figure 4).

If we consider all four figures reporting experimental results, we have that the DynamicPart policy outperforms all others in terms of lowest latency time, lowest percentage of dropped requests and, most important for a QoS study, complete satisfaction of SLAs for all workload conditions.

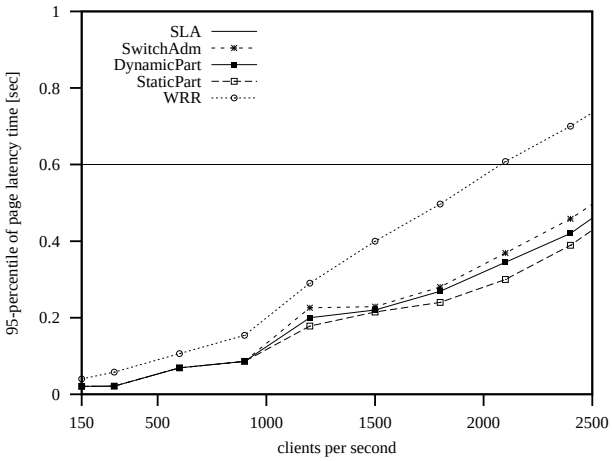

Figure 1. The 95-percentile of latency time for top *static* requests for QoWS-aware and QoWS-blind algorithms.

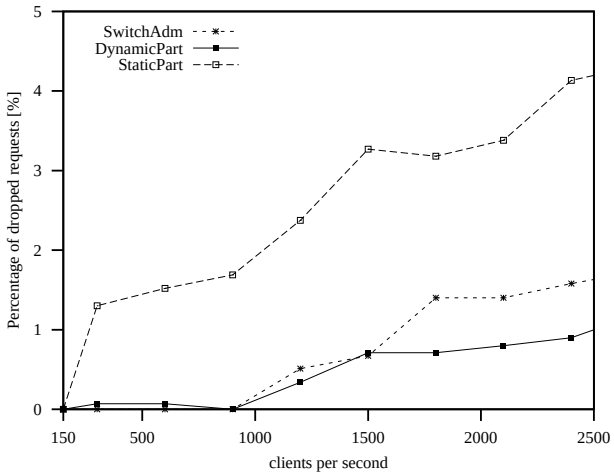

Figure 2. Percentage of rejected *normal static* requests for QoWS-aware algorithms.

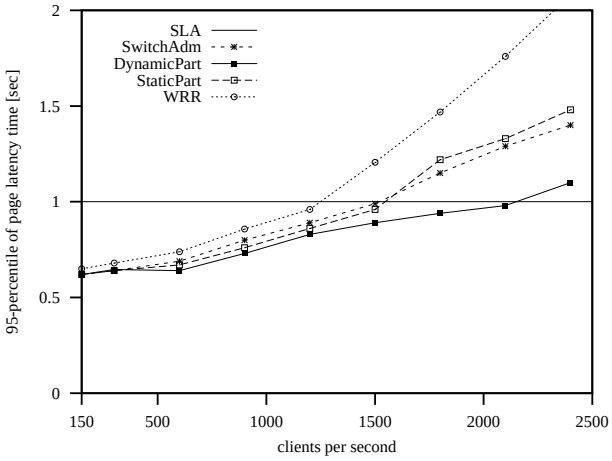

Figure 3. The 95-percentile of latency time for top *dynamic* requests .

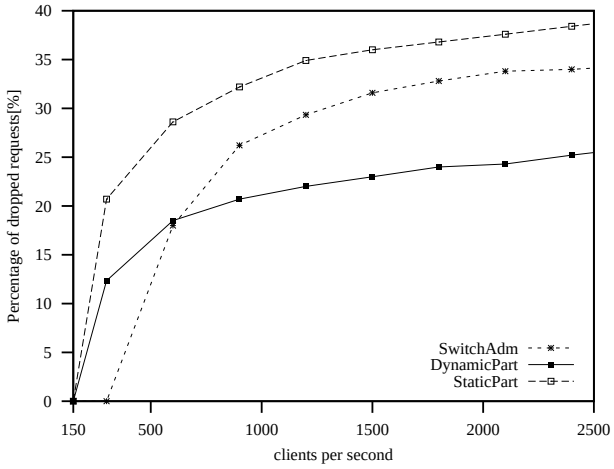

Figure 4. Percentage of dropped *normal dynamic* requests for QoWS-aware algorithms.

References

[1] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proc. of ACM Sigmetrics 2000*, Santa Clara, CA, June 2000.

[2] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proc. of USENIX 1998 Conf.*, Berkeley, CA, June 1998.

[3] V. Cardellini, E. Casalicchio, M. Colajanni, and M. Mambelli. Web switch support for differentiated services. *ACM Performance Evaluation Review*, 29, 2001.

[4] R. Engelschall. Load balancing your web site. *Web Techniques Magazine*, 3, May 1998.

[5] K. Li and S. Jamin. A measurement-based admission-controlled Web server. In *Proc. of IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000.