A cluster-based Web system providing differentiated and guaranteed services

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

In a world where many users rely on the Web for up-to-date personal and business information and transactions, it is fundamental to build Web systems that allow service providers to differentiate user expectations with multi-class Service Level Agreements (SLAs). In this paper we focus on the server components of the Web, by implementing QoS principles in a Web-server cluster that is, an architecture composed by multiple servers and one front-end node called *Web switch*. We first propose a methodology to determine a set of confident SLAs in a real Web cluster for multiple classes of users and services. We then decide to implement at the Web switch level all mechanisms that transform a best-effort Web cluster into a QoS-enhanced system. We also compare three QoS-aware policies through experimental results in a real test-bed system. We show that the policy implementing all QoS principles allows a Web content provider to guarantee the contractual SLA targets also in severe load conditions. Other algorithms lacking some QoS principles cannot be used for respecting SLA constraints although they provide acceptable performance for some load and system conditions.

Keywords: Cluster-based architecture, Quality of Service, Resource management, Performance evaluation, Web server system.

Corresponding author:
Michele Colajanni
Dipartimento di Ingegneria dell'Informazione
Università di Modena
Via Vignolese, 905
41100 Modena, Italy
Phone: +39-059-2056137
Fax: +39-059-2056129
E-mail: colajanni@unimo.it

1 Introduction

The Web is becoming a business-oriented media and the preferred interface for the most recent information systems. Hence, it is always more important to design and implement systems that are able to differentiate the performance offered to users and services. Such systems could accommodate heterogeneous application requirements and user expectations, and permit differentiated pricing for content hosting or service providing. For example, economic transactions are more important than simple browsing, and internal or premium service users may expect a better treatment.

A significant amount of research on Quality of Service (QoS) has focused on the network infrastructure including protocols, routers and Web caching. However, network QoS alone is not sufficient to support end-to-end QoS. To avoid the situation where high priority traffic reaching a server is dropped at the server side, the system hosting the Web site should be enhanced with mechanisms and policies for delivering end-to-end QoS (at least) to some classes of users and services. Although the ultimate challenge for the research in this field is to combine network with server QoS, in this paper we consider the server side of the Web system. Hence, we assume that network congestion problems are solved elsewhere, and client issues (such as insufficient performance of the browser, badly connected client machine) are outside the control of the Web service provider.

We describe the design, implementation and experiments of one of the first cluster-based prototype implementing the so called "Quality of Web-based Services" (*QoWS*) principles that are inspired by the well known network QoS principles: *service classification*, *admission control*, *performance isolation*, and *high resource utilization*. In particular, we apply the QoWS principles to a Web site that is hosted on a system platform consisting of locally distributed Web server nodes, namely Web-server cluster or *Web cluster* in short. With a simple best-effort platform, the only way to guarantee predictable service levels and to respond to congestion peaks is to over-provide system resources. By using QoWS mechanisms, we show that a variety of Web services can be deployed with high confidence and efficiency to selected classes of users.

QoWS-enabled systems for Web-based services have been recently proposed for single and multiple servers platforms. For a Web site hosted on a single server node, the main actions focus on scheduling algorithms and resource management of server components, both at the HTTP server and operating system level [5, 7, 13, 21, 25]. Most results for QoWS-enabled clusters consider *Web content hosting* that is, multiple Web sites co-located on the same platform [4, 22]. Instead, in this paper we investigate mechanisms and policies for a cluster-based system that hosts one popular Web site. Few results exist on this last topic [12, 19, 26] and all of them aim to compare the proposed QoWS-aware policy against a not QoWS-enabled Web cluster, typically through some simulation model instead of a prototype as done in this paper.

Given a Web cluster that provides static and dynamic contents, we first propose a methodology

to determine a set of confident multi-class Service Level Agreements (SLAs). We then describe the implementation (at the Web switch level of the Web cluster) of some QoWS-aware policies that adapt the basic QoS principles to the server side. We demonstrate that in our QoWS-enhanced Web cluster the performance for both static and dynamic services conform to the pre-determined set of SLAs, when all QoWS principles are implemented. Indeed, the policies that do not satisfy some QoS principle can still provide some acceptable performance results; however, they are not able to guarantee SLA requirements for different load and system conditions.

The remainder of this paper is organized as following. Section 2 focuses on the main components of a Web cluster architecture and the modules of the prototype system. Section 3 describes the workload model and main characteristics of the benchmarking tool. Section 4 introduces our methodology for the determination of SLA targets in a given Web cluster, and applies the methodology to the test-bed prototype. Section 5 discusses some policies and mechanisms that enhance the Web cluster with QoWS principles. The experiments in Section 6 aim to verify how the implemented QoWS policies behave when the server components of the Web cluster are subject to load stress testing. Finally, Section 8 presents some concluding remarks.

2 Web system platform

We consider a multiple node architecture, namely *Web cluster*, as a basic platform where to introduce QoWS policies and 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. A Web cluster has typically a front-end component that acts as a network representative for the Web site. In literature, this component is denoted through various definitions. In this paper, we call it *Web switch*. The distributed architecture of the cluster is hidden to the HTTP client through the unique Virtual IP (VIP) address that the Web cluster provides to the external world This address corresponds to the IP address of the Web switch. The primary and secondary DNS servers for the Web site always translates the site name into the IP address of the Web switch, which through this mechanism receives all inbound packets from clients. Besides the Web switch node, the Web cluster consists of HTTP servers connected to the back-end nodes through a high speed LAN. The Web switch includes a dispatching algorithm to select the Web server node best suited to respond, and a dispatching mechanism to route the client request to the chosen node.

The Web switch can operate client request assignment at *layer-4* or *layer-7* of the OSI protocol stack. A layer-4 Web switch is *content information blind*, because it determines the target Web server when the client establishes the TCP connection, before sending out the HTTP request. On the other hand, a layer-7 Web switch is *content information aware* because it first establishes a complete TCP connection with the client and then examines the HTTP request content (URL)

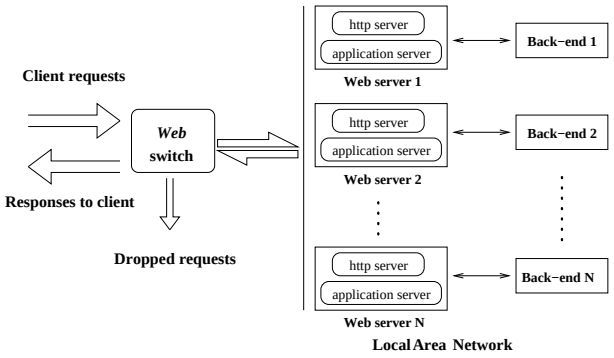

Figure 1: Web cluster architecture with a layer-7 Web switch.

before taking any dispatching or QoWS-related decision. In *one-way* architectures, client requests reach the Web switches, but responses flow through another network connection, while in *two-way* architectures, both requests and responses pass through the Web switch. A complete survey about Web cluster architectures is in [11].

Here we describe the prototype implementation of a two-way Web cluster with layer-7 switch, that we will enrich with the QoWS mechanisms described in Section 5. Figure 1 shows the logical architecture of the Web cluster architecture, where the Web switch implements some access control on requests, the Web server and application server tier replies to static and dynamic requests, respectively. The implementation of the Web cluster is based on off-the-shelf hardware and software components. The prototype cluster is made up of a Web switch, Web servers and back-end servers. They are all equipped with a Linux operating system (kernel release 2.4.12). Apache 1.3.12 is used as the Web server software, while some CGI applications in PHP have been implemented to emulate the dynamic page support and its loads. The layer-7 Web switch is implemented on a dedicated machine. Different mechanisms were proposed to implement a layer-7 Web switch at various system levels. The most efficient solutions are the TCP hand-off [23] and the TCP splicing [14] that are implemented at the kernel level. (Aron et al. clearly show that TCP hand-off outperforms TCP splicing techniques [4].) The application layer solutions are less efficient than kernel level mechanisms, but their implementation is more portable and sufficient for the purposes of this paper, that is focused more on QoWS mechanisms and algorithms than on Web switch performance. Among the application level solutions we select the reverse proxy approach proposed in [17] that is based on the Apache Web server software. This mechanism allows us to implement and test any layer-7 QoWS-aware algorithm without modifications at the kernel level, thus enhancing the portability of the proposed solutions. On the Web switch node, the Apache Web server is configured as a reverse proxy through the *mod_proxy* and *mod_rewrite* modules [3] providing the proxy serving and URL rewrite engine mechanism, respectively. Below, we give some details about the functionality of these two modules.

mod_proxy implements the proxy and reverse proxy functionalities in the Web switch for the protocols FTP, CONNECT(SSL), HTTP/0.9, HTTP/1.0 and HTTP/1.1. The mod_proxy module basically builds a new URL on the basis of the information provided by the mod_rewrite that is, the name of the designed server; it forwards, as a *proxy request*, the incoming request to the appropriate Web server; it modifies the header of the HTTP response provided by the Web server, by changing the Web server name references.

mod_rewrite provides all services related to the URL rewriting mechanism. It uses a configuration directive that permits to intercept an incoming URL the format of which matches a known pattern. This functionality is fundamental to realize request/user classification, content-aware access control and request routing. The rewriting mode may be static (if we use a lookup table) or dynamic (if we use a software module that communicates with the Apache Web server through the stdin/stdout file handle). The dynamic rewriting mode guarantees the highest flexibility to the Web switch. We use this method because it permits to implement the dispatching and QoWS algorithms in a external module (this allows communications with the load monitors running on the Web server nodes, and the possibility to change the content-aware dispatching policy at run time). Finally, it specifies, through a dedicated flag, if the rewrite packet needs to be forwarded directly to the Web server or if it needs to be rewritten again.

The dispatcher/QoWS module communicates with the *rewriting engine*, which is provided by the *mod_rewrite*, over its *stdin* and *stdout* file handlers. For each map-function lookup, the dispatcher/QoWS module receives the key to lookup as a string on *stdin*. After that, it has to return the looked-up value as a string on *stdout*.

For QoWS purposes, the Web switch requires information about the server load. Hence, we implement on each server a *load monitor* that is an application level module dedicated to collect, elaborate and send server load state information to the *collector* module running on the Web switch. The load monitor integrates operating system performance evaluation tools (such as *iostat*) and custom programs that get information from the *proc file system*. It is a flexible module that, depending on the administrator choice, can collect various system load information, such as CPU, disk and memory utilization, number of active connections and combinations of them. For our experiments we used the number of active HTTP connections as the main load metric, by distinguishing those for static and dynamic page requests, respectively.

The load monitor of a Web server periodically sends the load state information to the *collector* module running on the Web switch. The communication between each load monitor and the collector module uses a direct UDP connection on a dedicated channel (one channel for each Web server). The frequency of messages received by the collector over these channels is also used as a

hearth-bit signal. Web servers not sending information are considered unreachable or overloaded, and are excluded from the pool of the possible choices by the dispatcher/QoWS module. Figure 2 shows the architecture of the Web switch software and its interactions with the Web servers.

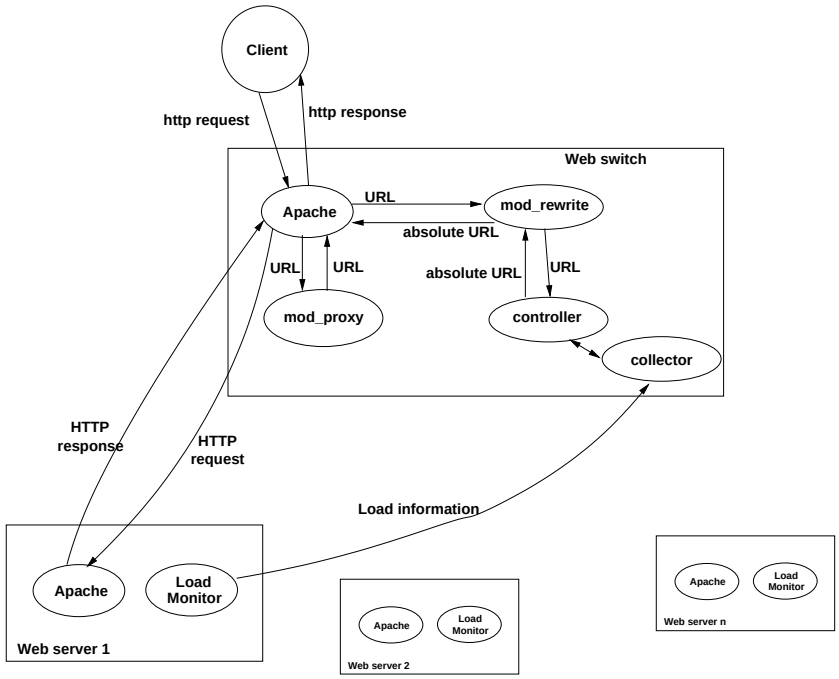

Figure 2: Architecture of the Web switch software and interactions with the servers.

3 Client architecture and workload

For the experiments described in the following sections we use a pool of client nodes running on 5 PC-computers that are interconnected to the Web cluster through a switched 100Mbps Fast-Ethernet that does not represent the bottleneck for our experiments. The Web cluster is composed of 10 server nodes. Each server machine, is a Dual PentiumIII-833Mhz PC with 512MB of memory. All nodes of the cluster use a 3Com 3C905C 100bTX network interface and an IBM Ultra ATA/100 disk (Deskstar60GXP 40GB, Ultra ATA/100, 8.5ms, 7200rpm, buffer 2MB) having a transfer rate of 37MBps and seek time of 8.5 ms. The layer-7 Web switch is implemented on a dedicated machine having the same characteristics as the server platform.

As a synthetic workload generator we use a modified version of the Webstone benchmark tool[1]. The main differences concern the introduction of the HTTP/1.1 protocol, and various modifications on clients behavior because the original design of Webstone was not intend to produce realistic workload, but the heaviest load on the server nodes. Basically, the modifications are inspired to

[1] Webstone Mindcraft Inc., http://www.mindcraft.com/webstone, version 2.5

the SURGE model [6, 15], that is integrated also with user classes and performance metrics oriented to QoS. For example, we use percentiles and distributions instead of average values. We have also introduced the concept of user class, user session, user think-time, embedded objects per Web page, realistic file sizes and popularity. Table 1 summarizes the probability mass function (PMF), the range, and the parameters' value of the workload model.

We have also emulated the impact of dynamic workload and dynamic page composition by creating three CGI services. If not otherwise specified, the basic workload composition consists of 80% of static requests and 20% of dynamic requests. Dynamic requests are further classified as CPU-light (10%), CPU-intensive (6%), and CPU-disk intensive (4%), on the basis of the computational impact they put on the back-end servers. Finally, we have emulated two classes of users that is, *Top class* and *Normal class*, both of them issuing *static* and *dynamic requests* to the Web cluster. The probability of accessing the system for a given service class is defined by a class vector. We denote with ρ_T the average percentage of top user requests.

Category	*Distribution*	*PMF*	*Range*	*Parameters*
Requests per session	Inverse Gaussian	$\sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}$	$x > 0$	$\mu = 3.86$, $\lambda = 9.46$
User think time	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1.4$, $k = 1$
Objects per request	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1.33$, $k = 2$
HTML object size	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$	$x > 0$	$\mu = 7.630$, $\sigma = 1.001$
	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1$, $k = 10240$
Embedded object size	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$	$x > 0$	$\mu = 8.215$, $\sigma = 1.46$

Table 1: Parameters of the workload model.

4 Setting the SLA targets for a Web cluster

The approaches to QoWS studies can have different purposes. For example, a typical question is to find a system platform that allows a service provider to satisfy a set of pre-determined SLA requirements. Another is to find the best choice for SLAs for maximizing profits, for example in [22]. Our approach is slightly different because we assume to have a system platform, and we propose a methodology for the determination of SLA targets that are suitable for the available architecture and the *expected workload* reaching that system. The tradeoff is not to have an over-provisioned Web cluster that remains underutilized for most of the time, and to minimize the risks of not satisfying the SLA contractual parameters that lead to some penalty to the Web services provider. We indicate the main steps of the methodology that we describe in detail by applying it to the previously described Web cluster architecture.

1. First we have to choose some performance measures for the SLAs (Section 4.1).

2. The second step requires the determination of realistic SLAs for the available Web cluster system. To this purpose, we first evaluate the *capacity* of the Web cluster for the main service classes of interest for the considered Web site. Some capacity planning study that consider *future workload* can also be carried out during this phase. From the *basic* performance results we define realistic SLA targets for the considered system under expected or future workload (Section 4.2).

3. After that, we enrich the Web cluster with some mechanisms for achieving QoWS (Section 5): request classification, admission control, request dispatching.

4. We verify whether the QoWS-enabled Web cluster is able to satisfy the SLA requirements for different workload scenarios (Section 6).

4.1 SLA measures

At the base of QoS (and QoWS as well) there are the concepts of *service* and *Service Level Agreement* (SLA). A *service* defines a set of characteristics significant for the Web content provision, specified in quantitative or statistical terms. An SLA is a specified performance contract agreed between the user and the Web service provider. Depending on its strictness, it may be either a *guaranteed service* if the resources are reserved to never violate the agreement, a *predictive service* if rare violation are allowed and the service is expressed in terms of percentile, or a *best-effort service* if the service is provided with no contractual target.

We can consider the Web servers together with the network as parts of a soft real-time system that has to provide information and services to the browsers. The issuer of the request takes into account the time and the request value decreases as time goes on, however there are penalties for the Web service providers but no catastrophic consequences if a deadline is not met. Furthermore, the Web cluster is only part of the client-network-server system and has no control over the rest of it. For these two reasons, we consider most effective to denote the SLA for QoWS in terms of *predictive service*, measured as a 95-percentile of the performance values [20]. In particular, we choose the *latency time of a client request* at the Web cluster as the main measure for SLA. This time measures the completion time of a request at the Web cluster side.

In this paper we consider two SLAs for two classes of users (that is, *top class* and *normal class*), both issuing *static* and *dynamic requests* to the Web cluster (in short, top/normal static and dynamic requests). The *top class SLA* states that the "*95-percentile of the latency time of static and dynamic requests from top class users must be less than a threshold of T_S seconds and T_D seconds, respectively*". On the other hand, the "*requests from normal class users (in short, normal requests) receive best effort service*".

It is worth noting that SLA on top dynamic requests is the most severe contract as they may involve expensive CPU and disk operations. Indeed, we observe that static requests do not represent a real QoWS problem, also because of the high hit rate at the disk caches. Hence, if not otherwise specified, hereafter we focus mainly on SLA for dynamic requests.

4.2 Capacity of the Web cluster and suitable SLA targets

In this section we describe how to determine realistic SLAs for the available Web cluster through a test case. The goal is to evaluate for the considered Web cluster and expected workload a realistic value for the upper bound for the 95-percentile of the latency time for static and dynamic requests. To set the maximum thresholds T_S and T_D, we evaluate the latency time for static and dynamic requests for increasing load at the Web cluster prototype. The goal is to choose the SLAs below the *maximum capacity* of the system that is, the load the system can sustain without being overloaded, also known as the *system breaking point*.

It is important to observe that the SLA for Web requests depends on the capacity of each server node and not on the capacity of the Web cluster, because the granularity of request dispatching is done at the page level and not at the object level (each user request corresponds to a page request, that consist of a request for the base HTML file and multiple requests for embedded objects). Hence, each user request reaching the cluster through the HTTP protocol is served by one Web server (if all embedded hits are for static objects), and its corresponding back-end server (if an embedded hit requires some dynamic evaluation). Hence, when dispatching is done at the page level, the SLAs on page latency time is independent of the number of servers in a Web cluster. This number is important only to evaluate the maximum number of requests that can be concurrently sustained, but it has no influence on the latency time of one page request.

Figures 3 shows the 95-percentile of the page latency time as a function of the client arrival rates to the system. Each client request corresponds to a new connection bringing an entire page request that is dynamic with probability 0.2. (Each point of this figure is obtained as the average of three experimental results in one server.) This figure confirms our supposition that static requests do not represent a performance problem. Indeed, the 95-percentile of latency time for static requests is well below 1 second even when 300 clients per second reach the system. The choice of the SLA parameter for the static requests of the top users depends on the management policy of the service provider. It can choose $T_S = 0.5$ second if he finds convenient to accept a lower number of concurrent requests and guarantees a better service. Otherwise, he can choose a higher SLA threshold, up to $T_S = 1$ second, if he prefers to accept more requests at a price of a slightly worse service. Our choice is for $T_S = 0.5$ that is reached for 210 clients per second.

The 95-percentile of latency time for dynamic requests is about 0.6 seconds when the system is underutilized (less than 50 clients per second), and increases up to about 2 seconds for 200 clients

per second. From this figure, we can also observe that this point is already after the knee of the curve. Hence, it is safer to choose lower values. Again, the final choice depends on the management policy. In our test case, we consider adequate to choose 1.2 second as the upper bound on the 95-percentile of the page latency time, that we consider in a controllable space. In summary, we set the SLA parameters for the top static requests and top dynamic request to $T_S = 0.5$ and $T_D = 1.2$ second, respectively.

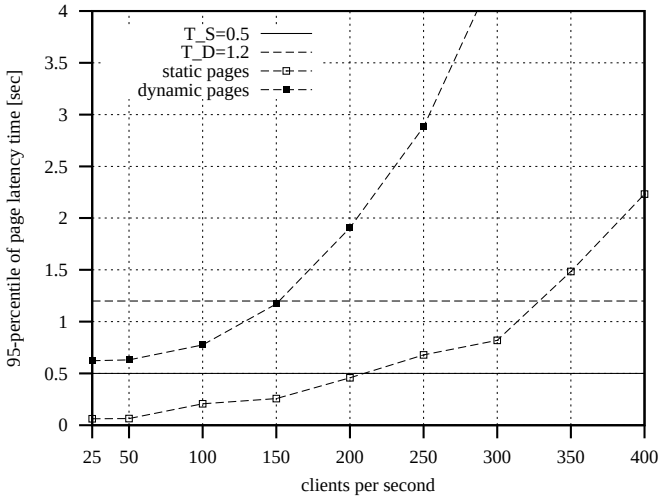

Figure 3: Page latency time as a function of the client arrival rate in one server system.

5 QoWS-aware policies for the Web cluster

5.1 QoWS principles

Web server clusters provide a best-effort service and their structure is designed to increase performance and load sharing between all the Web servers included in the same cluster. The necessity for QoS was born in the computer network and multimedia area, where the QoS topic is highly investigated. We consider four basic principles necessary to provide QoS guarantees to network applications: request classification, admission control, performance isolation and high resource utilization. We investigate how these network QoS principles can be extended to QoWS in cluster-based Web servers with the goal of finding suitable QoWS-aware policies and mechanisms that can be implemented at the Web switch level.

To this purpose, it is important to clarify the definitions of *service class* and *resource*. A *service class* denotes the differentiation of incoming requests and users into classes. The proposed policies must support multiple service classes. To make the presentation lighter, we consider without loss of generality four service classes, obtained by two classes of users denoted as *top* and *normal* classes,

and two main classes of requests that is, *static* and *dynamic*. In a Web system based on one server, a *resource* is typically one of the server components, such as CPU, disk, and network interface [5, 8]. This definition of resource is still applicable to Web clusters, however we prefer to use a coarser grain definition that is, considering an entire (Web or back-end) server as a resource.

Request classification. The first step to differentiate the service is to classify the incoming requests in order to bind each one with the appropriate service class. The classification mechanism of our layer-7 Web switch may use all available information at the application level (e.g., URL content, SSL identifiers, and cookies), thus achieving the most detailed classification including client, user, session and request identification.

Admission control. An admission process is needed to allow a Web cluster to meet the predictive service targets. This process consists of two steps: *declaration* and *access control*. A declaration process with the purpose of estimating the resources needed by a request can be realistically carried out only by a layer-7 Web switch, because it is content information aware. For example, we found useful to differentiate at least static from dynamic requests, because dynamic requests can easily require service times of two or more orders of magnitude higher than static requests.

After the classification of a service, the access control mechanism should check whether there are sufficient resources to satisfy the SLA settled for that service, and decide about allowing or dropping the connection. Admission control is essential to not overload the servers, as overloaded servers experience a significant loss of throughput [13]. Our layer-7 Web switch uses connection dropping as the access control mechanism.

Performance isolation. To force different classes of users and requests not to share the system resources, it is necessary to provide a degree of isolation among the service classes. Performance isolation can be achieved through scheduling policies [16] and/or resource partitioning mechanisms. A scheduling priority mechanism applied to all resources could allow over aggressive classes of services to monopolize the entire system.

The partitioning mechanism can be applied at different levels of granularity, depending on the definition of *resource*. In the Web cluster we consider an entire server as a resource, also to facilitate the applicability of QoWS principles to commodity off-the-shelf software. As a consequence, the most immediate performance isolation policy in a Web cluster is to provide a server partition that follows the classification of services. For example, with two classes of users and two classes of requests it seems convenient to use a subset of (Web and back-end) servers for each class of (top/normal) users and (static/dynamic) requests.

High resource utilization. To use the system resources as efficiently as possible, we have to

make unused resources available to other classes of services. Our solution to satisfy this basic QoS principle is to realize a dynamic server partition. Dynamic partitioning outperforms the static one because it obtains better resource utilization still maintaining good service class isolation. This has been demonstrated in [26] and in our preliminary design studies done through simulation [10]. By referring to the performance isolation considerations, the server partition between two sets would be dynamically determined so that the cardinality of each set could vary, for example depending on the type of offered load during a certain period.

In summary, the layer-7 Web switch of our QoWS-enhanced Web cluster implements some classification policy and admission control mechanism. Moreover, it can use server partition algorithm for performance isolation, and dynamic server partition algorithm for high resource utilization. Placing the admission control and decisions at the entrance point reduces the spurious load at the internal components of the system.

In this section we propose three policies integrated into the Web switch that add some QoWS mechanisms to the Web-server cluster: *Switch Admission*, *Static Partitioning*, *Dynamic Partitioning*. All the considered policies guarantee at least some request/user classification and admission control, because our preliminary experiments confirmed the intuition that it is impossible to guarantee any SLA, if the load offered to the Web cluster is not controlled. The main difference is about the performance isolation of system resources among the user classes: Switch Admission policies do not use any mechanism, Static Partitioning and Dynamic Partitioning policies isolate the resources in a static and dynamic way, respectively.

5.2 Switch Admission algorithms

Switch Admission is an example of centralized admission control without server isolation. If we do not provide server partition, each incoming request can be assigned to any server, independently of the service class it belongs to. Hence, to satisfy the SLA of the most demanding classes also in heavy load condition, the basic idea of the policies without server partition is to deny service to the requests of lower class users when the cluster load exceeds a given threshold, and eventually to reject the requests of top class users only in the case of extremely critical load conditions.

As a representative member of this class, we implement the **SwitchAdm** algorithm that, when necessary, denies service to the requests belonging to the lower class of users. The rejection mechanism is triggered by the Web switch when the current sum of the server load in the Web cluster exceeds a given threshold. Once the request has been admitted to the system, the Web switch uses the Weighted Round Robin (WRR) policy [11] to select the target server in the corresponding set. This can be any server of the cluster, if no server partition is used, or a subset of the servers, otherwise. Therefore, the SwitchAdm policy is equivalent to a QoWS-blind dispatching policy enhanced

with an admission control mechanism to prevent performance degradation [13].

There are several feasible variants, but one of the most important issue to address is the choice of the cluster load measure and of the rejection threshold. When the workload consists of static requests only, the system throughput expressed in MBps is a representative performance measure, and in most cases the throughput expressed in connections per second is acceptable too. On the other hand, none of these measures is representative of the server load when the system receives static and dynamic requests. Indeed, a CPU/disk intensive request can stress the system for orders of magnitude higher than a file transfer that could even be gotten from the disk cache. For this reason, we have chosen two throughput measures: the number of connections bringing requests for (at least one) dynamic document (namely, *dynamic connections*), and those for static documents (namely, *static connections*). Several simulations and experiments carried out by our group have demonstrated that the number of static and dynamic connections handled by each server is a system load measure much more precise than the total number of connections per second and Mbytes transferred during a certain interval.

In the choice of a threshold parameter for deciding about rejection or admission to the cluster, we have also to consider the SLA targets for static and dynamic requests at each server. In particular, from Section 4.2 we write as $MaxConn_S(SLA(T_S))$ and $MaxConn_D(SLA(T_D))$ the maximum number of concurrent static and dynamic connections each server can sustain without performance degradation that is, by guaranteeing the SLA latency times to top class requests.

If we assume that the Web switch dispatcher is able to balance the load among the servers (there are several dispatching policies that guarantee good results in a cluster system, for example WRR), and all N servers have homogeneous capacities, we can write the rejection thresholds as $Thr_S = N \cdot MaxConn_S(SLA(T_S))$ and $Thr_D = N \cdot MaxConn_D(SLA(T_D))$.

We use the notification mechanism outlined in Section 2 to inform the Web switch about the numbers of alive (static and dynamic) connections The Web switch periodically sums all contributions, and determines if the requests have to be dropped or accepted. The rejection phase starts when the sum of the HTTP connections at the servers exceeds the thresholds Thr_S and Thr_D, and ends when the sums return under these threshold values.

Let us describe the choice of the thresholds by referring to the test-bed prototype architecture. The goal is to evaluate $MaxConn_S(SLA(T_S))$ and $MaxConn_D(SLA(T_D))$. From Figure 3 we have obtained $T_S = 0.5$ and $T_D = 1.2$. These page latency times are achieved when the arrival rates to one server are equal to 210 and 150 clients per second, respectively. The next step is to know the server throughput (denoted as static and dynamic connections per second) as a function of the client arrival rate. Figure 4 shows these curves for one server of the prototype. From this figure we can observe that in correspondence of 150 and 210 client arrivals per second we have about 18 dynamic connections and 85 static connections, respectively. This justifies the choice

for $MaxConn_S(SLA(0.5)) = 85$ and $MaxConn_D(SLA(1.2)) = 18$. In the same figure it is worth noticing that the maximum capacity for the server as far of dynamic connections is 26, this is why we set $MaxConn_D = 26$. Beware that this last parameter is only server-dependent and is not related to the SLA.

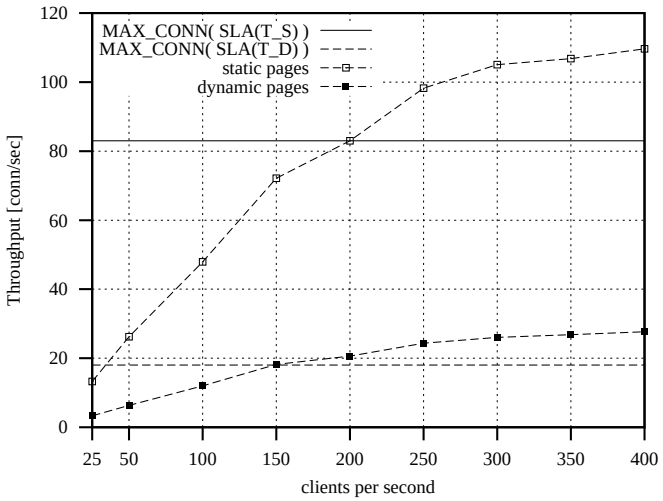

Figure 4: Throughput as a function of the client arrival rate at one server system.

5.3 Static Partitioning algorithms

Performance isolation is one of the main principles that we borrowed from the QoS theory. In Web systems consisting of one server, performance isolation is achieved through priority scheduling algorithm operating at the HTTP server level and/or at the CPU level. On the other hand in a Web cluster a simple way to enforce performance isolation is through a centralized Web switch mechanism that uses same classification and admission policy as SwitchAdm, and partitions the servers in as many sets as the defined service classes. This solution intervenes on the dispatching algorithm and does not require modifications at the application or operating system level of the servers. In the hypothesis of two classes of users (top and normal), we partition the servers into two sets, denoted as *High Set* (HS) and *Low Set* (LS). The cardinality of HS is set to a constant value K. Top and normal user requests are assigned to the servers $HS = \{1, \ldots, K\}$ and $LS = \{K+1, \ldots, N\}$, respectively.

Different algorithms derive from the policy chosen to determine K. For example, a solution is to set $K = \lceil \rho_T N \rceil$, where ρ_T represents the expected percentage of top user requests. This naive solution requires that the Web administrator knows *a priori* the daily distribution of the percentage of top user requests and then allocates the servers proportionally to it: it is fair and does not give any advantage to top users. It is easy to assume a knowledge about ρ_T because the top users have

to be pre-registered. In this paper, we implement the **StaticPart** algorithm that chooses K by considering that the dynamic requests have a much deeper impact on system load than that caused by static requests. Indeed, we have verified that if HS is dimensioned so that SLA(T_D) is satisfied, SLA(T_S) is guaranteed as well. The top users deserves more attention because they have to meet strict SLAs. Even when the daily distribution of the percentage of top class requests is known *a priori*, some variability may occur. From all these premises, it seems convenient to provide some over provision of the partition size of HS servers.

Hence, when we have to satisfy an SLA of T_D seconds, we find appropriate to choose

$$K = \lceil \rho_T N + N * (1 - \rho_T)(1 - \frac{MaxConn_D(SLA(T_D))}{MaxConn_D}) \rceil. \quad (1)$$

where $MaxConn_D$ is the maximum number of dynamic requests that can be accepted without overloading a server, as shown in Section 5.2.

The first term of the equation 1 partitions the servers proportionally to the expected percentage of top user requests. The second term is introduced to increase the cardinality of HS. From the $\lceil N * (1 - \rho_T) \rceil$ remaining servers, we add to HS a number of nodes that is proportional to the ratio $\frac{MaxConn_D(SLA(T_D))}{MaxConn_D}$ that we define as the *safety factor* ranging from $1/MaxConn_D$ to 1. It is a measure of the percentage of server capacity (expressed in connections per second) that is used to satisfy the SLA requirements with major probability. The lower is the ratio, the higher is the possible waste of resources.

The value of K remains constant until the Web administrator realizes that the server partition of the Web cluster based on the previously chosen ρ_T is highly inadequate and changes it. The following numerical example shows the choice of K on the basis of the equation 1. Let us consider a Web cluster with $N = 10$ Web server nodes, a percentage of dynamic requests equal to $\rho_T = 0.2$ and $T_D = 1.2$, as in our test-bed case. From Section 5.2 we have $MaxConn_D = 26$ and $MaxConn_D(SLA(T_D)) = 18$. From equation 1, the cardinality of HS according to StaticPart is equal to $K = 5$.

5.4 Dynamic Partitioning algorithms

Any static resource allocation enforces a strong isolation of the requests but does not guarantee high resource utilization because the percentage of service requests for different classes can be considered periodic, but not static during a day. Because of the high variability of Web traffic, to guarantee SLAs for the classes of top users would require a system dimensioning on peak usage for each combination of offered load: this solution would be unrealistic and expensive.

The goal of Dynamic Partitioning policies is to permit a more efficient use of the resources in different load conditions while still providing isolation among service classes and users. They

determine the initial value for K as in the StaticPart policy, but then they use some mechanism to adapt the size of each partition to the actual workload composition and server load state. In this paper, we consider a representative policy, namely **DynamicPart**[2], even if it is important to observe that many alternative algorithms exist. Some of them have been proposed in literature [26, 10], another example is given in Section 6, others have not been explored yet, so that we can state that there is still room for research in this field.

DynamicPart periodically evaluates whether the servers dedicated to top users (HS) are able to satisfy the SLAs and, if not, changes server partition as following. Let us suppose that at time t the size of the HS partition is $K(t-1)$ and the sum of the servers load in $HS(t-1)$ is $SumLoad_{HS}(t)$. This policy adds the least loaded server of $LS(t)$ to $HS(t)$ that is, $K(t) = K(t-1)+1$, if $SumLoad_{HS}(t) > K(t-1) \cdot MaxConn_D(SLA(T_D))$. From that point, the moved server will receive only new requests from the top users, while it will continue to serve requests of already accepted connections belonging to the normal users. (Stronger actions can be used, such as dropping all pending normal requests, but we do not used them because it is considered an unfair solution.) The server moved to $HS(t)$ will return to $LS(t)$ that is, $K(t) = K(t-1)-1$, when $SumLoad_{HS}(t) < (K(t-1)-1) \cdot MaxConn_D(SLA(T_D))$.

In the basic version of the DynamicPart policy, only requests from normal clients can be rejected when $SumLoad_{LS}(t) > (N - K(t)) \cdot MaxConn_D(SLA(T_D))$.

The DynamicPart algorithm is a feedback controlled system that aims to keep $\frac{SumLoad_{HS}(t)}{K(t)}$ constant: increasing the ratio augments $K(t)$ that decreases the ratio and vice-versa. Hence, we can say that DynamicPart guarantees stability and responsiveness to variations of $SumLoad_{HS}(t)$, at least until some server is available.

The following example motivates that any constant choice of K, as done by StaticPart, may result inadequate. Let us consider again our test-bed case: a Web cluster with $N = 10$ Web server nodes, $T_D = 1.2$ and a $SafetyFactor = 18/26$. The safety factor is a function of the server capacity and SLA requirements, hence it is a constant when the system architecture and the SLA requirements are fixed. The parameter ρ_T is variable and it is a function of the workload composition. Any static choice may work if the workload composition does not vary much from the average during the day. In the reality, the Web workload may be subject to high fluctuations around the average values because of the heavy-tail characteristics of its distributions.

When we vary ρ_T from 0 to 1, we obtain the K plot shown in Figure 5. Our example evidences that also little variations of ρ_T require a change of K. The value of $K = 5$, given by the equation 1 is valid only for an expected percentage of top class requests equal to 20%. If ρ_T augments, K increases to 6 and then to 8 servers when the percentage reaches 60%. An additional important

[2]Unlike the DynamicPart algorithm in [10], the version proposed in this paper considers the number of requests for static and dynamic documents as its system load measure.

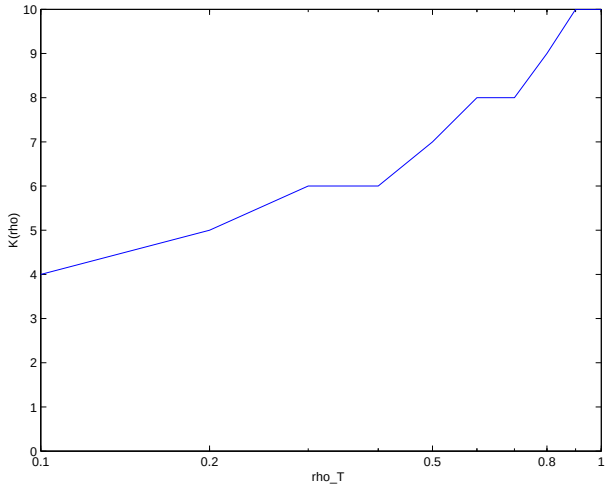

Figure 5: Number of servers dedicated to Top class users as a function of the percentage of top class requests ρ_T

information given by this figure is the maximum percentage of high class requests that the system is able to satisfy for the type of considered workload. Indeed, when $\rho_T = 0.8$, $K = 9$ servers are needed in the HS to satisfy the SLAs with high probability. If ρ augments another server would be required, but this choice would implicate an empty LS. This unfair solution is not acceptable, because all low priority requests would be refused.

6 Experimental results

The experimental results are oriented to verify whether and which of the described QoWS-aware policies satisfy all SLA targets for different scenarios. The workload and system architecture here considered are described in Section3.

6.1 Sensitivity of dynamic partition of servers

We have proposed an example of dynamic partitioning algorithm with the goal of having a QoWS-enhanced Web cluster that should be characterized by *stability* and *responsiveness*. Hence, the first set of experiments aim to check whether DynamicPart satisfies these two characteristics.

A dynamic partitioning algorithm is *stable* if $K(t)$ ranges in a small set of values without frequent oscillations. Figure 6 permits to verify the stability of the dynamic partitioning mechanism adopted by DynamicPart. We set the ρ_T parameter to 0.2 and we observe the value of $K(t)$ every 10 seconds. We find that $K(t)$ ranges from 4 to 5 servers. The initial value is $K = 5$, and it remains stable for most consecutive observations with some acceptable exceptions. The observed peaks ($K = 4$) are due to statistical fluctuations of the value of ρ_T occurring during the experiments. From these

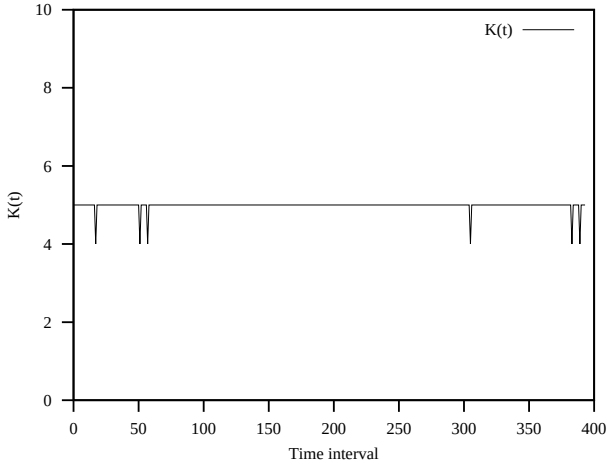

Figure 6: Stability of the mechanism adopted by DynamicPart, when the percentage of top requests remain constant around an average value.

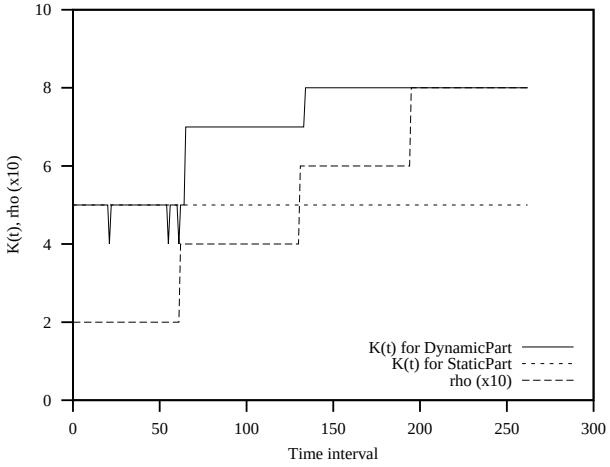

Figure 7: Responsiveness of the mechanism adopted by DynamicPart as a function of the percentage of top requests.

results, we can conclude that even when the ρ_T value is known, any static partitioning algorithm could result in a waste of cluster resource and/or violation of some SLAs.

The *responsiveness* of a dynamic partitioning policy reflects the capacity of the algorithm to respond promptly to the fluctuations of the percentage of Top class users, thus avoiding the risks of violating some SLAs. We test the responsiveness of DynamicPart in a scenario where the percentage of Top class requests ρ_T varies from 0.2 to 0.8 and we compare it with a static K determined by StaticPart. As in the previous experiment, ρ_T is sampled every 10 seconds. Figure 7 shows that the adjustment mechanism provided by DynamicPart is able to change rapidly the value of $K(t)$.. The initial value of $K(t)$ is 5 for both StaticPart and DynamicPart. When ρ_T goes to 0.4, $K(t)$ increases to 6 servers. If the percentage of top requests augments to 0.6, the value of $K(t)$ value jumps to 8 servers and does not change even when $\rho_T = 0.8$. These experimental results clearly show the limits intrinsic to StaticPart: it has to know in advance the ρ_T parameter that has not to change during the day. Otherwise, any small increment of ρ_T would let StaticPart violate some SLAs.

6.2 SLA performance analysis

Unlike a traditional performance analysis where the goal is to evaluate which is the policy that gives best performance results (e.g., minimum response time, maximum throughput), the goal of the following experiments is to verify whether and which of the described QoWS-aware policies satisfy all SLA targets for different scenarios. A secondary goal is to compare the performance of the proposed QoWS-enabled Web cluster against a system using some QoWS-blind dispatching

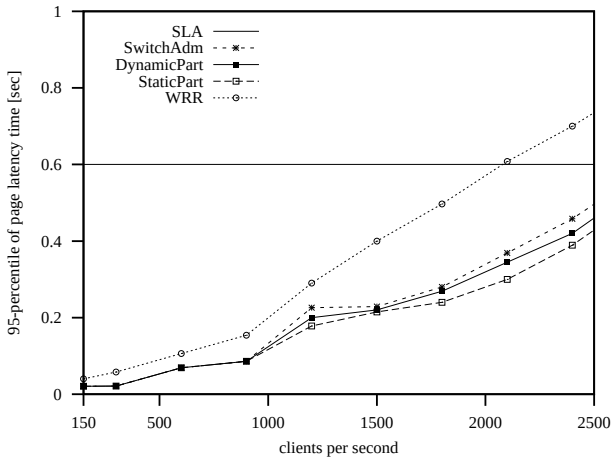

Figure 8: The 95-percentile of latency time for top *static* requests for QoWS-aware and QoWS-blind algorithms.

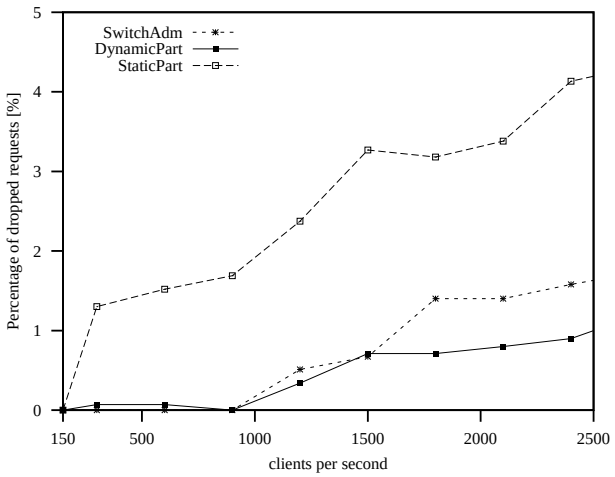

Figure 9: Percentage of rejected *normal* requests for QoWS-aware algorithms.

algorithm, such as WRR. The SLA represents the most important target, so a policy that satisfies SLA requirements for all experiments is preferable to a policy that has a lower latency time in most instances, but it is unable to guarantee SLAs in others.

Figures 8 shows the 95-percentile of the page latency time for the *top* static requests. This figure confirms 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 disk caches of the Web servers rather than from their 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 9): less than 4% for the StaticPart policy, less than 2% and 1% for the DynamicPart and SwitchAdm policies, respectively.

When we pass to consider the *top* dynamic requests, we have to consider SLAs that are much more critical for the QoWS-enabled Web cluster. Figures 10 shows the 95-percentile of the page latency time for QoWS-aware and QoWS-blind policies. From this figure we observe that QoWS-aware policies are able to satisfy the SLA until the offered load is below 1800 clients per second. After that point, just the DynamicPart algorithm is able to guarantee the SLA until the Web cluster receives 2400 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 11). However, this price is worth if the counterpart is the respect of SLA targets.

When we consider all four Figures 8- 11, we have that the DynamicPart policy outperforms

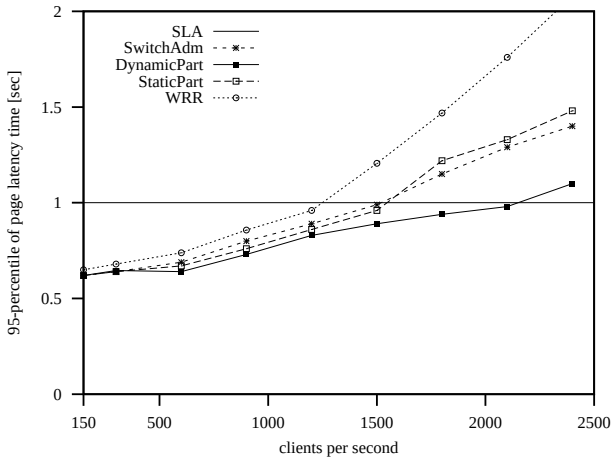

Figure 10: The 95-percentile of latency time for top dynamic requests for QoWS-aware and QoWS-blind algorithms.

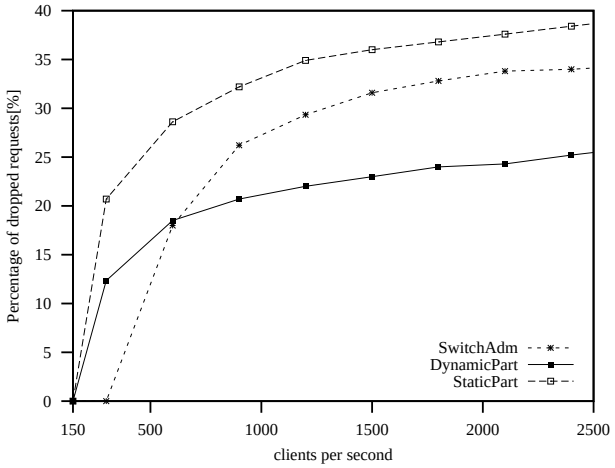

Figure 11: Percentage of rejected (normal) requests for QoWS-aware algorithms.

all others in term of lowest latency time, lowest percentage of dropped requests and, most important, best satisfaction of SLAs for almost all workload conditions. An initially unexpected result is that StaticPart achieves performance worse than that of SwitchAdm both in terms of page latency time and percentage of dropped requests. We recall that StaticPart guarantees differentiated services but not a good resource utilization, while the opposite is true for SwitchAdm. Moreover, experiments are carried out in best conditions for StaticPart that is, with a static percentage of top requests. Nevertheless, the statistical variations intrinsic in the workload seems to penalize any static assignment of Web cluster resources. We can conclude that any QoS property, such as differentiated services, should not be implemented at the price of another important property, such as resource utilization. Indeed, the best performing policy (DynamicPart) is the only one which implements all four QoWS principles described in Section 5.1.

7 Related work

If we exclude research results on network QoS and multimedia servers contributing to the largest part of literature in the QoS field, that is impossible to cite here, we can broadly classify Web server-side efforts for delivering QoS into two categories: those that support QoWS in single-node Web servers at the application level or at the operating system level, and those that provide support for QoWS in cluster-based platforms with multiple server nodes. QoWS in single server systems is achieved primarily by affecting the admission control and scheduling of HTTP requests [5, 7, 8, 13, 18, 21, 25]. The results for QoWS in cluster-based platforms can be further distinguished on the basis of the Web service provider that is, solutions for a single Web site or for Web content hosting,

in which multiple Web sites are co-located on the same cluster-based system. A Web cluster used as a platform for hosting multiple Web sites is considered in [4, 22], where the authors address QoWS issues as an optimization problem. An interesting work from Aron et al. has extended the resource principal abstraction to multi-node Web servers In [4]. the authors have applied to Web clusters the "cluster reserves" concept they used for single Web servers [5].

One of the first study on Web clusters that support a single site is by Chen et al. [12] that have evaluated the impact of the request admission and dispatching mechanisms. Their simulation results clearly show that when the system is highly utilized, differentiated services provide better performance than those achieved by traditional Web clusters. Kanodia et al. have proposed a QoWS-aware policy that uses both admission control and performance isolation mechanisms to guarantee that different classes of service have latencies within pre-specified targets. Unlike our paper that focus on static and dynamic requests, they consider Web sites providing static content only [19]. Two dynamic resource partitioning algorithms for static and dynamic Web requests have been proposed in [10, 26]. Their experiments demonstrate that dynamic server partitioning always outperforms static server partitioning.

Other QoWS supports for Web hosting have been investigated mainly to assign differentiated priorities to the requests according to which site is accessed. Many results show that simple strategies such as controlling the number of processes can improve the response time of high-priority requests while not penalizing the system throughput [2]. To enforce SLA constraints, Pandey et al. [24] examine selective allocation of server resources by assigning different priorities to the page requests. In [1] a control-based approach is proposed for Web service differentiation.

Several companies commercialize as their most recent products content-aware Web switches which can be used for service differentiation in Web clusters (e.g., Nortel Networks' Alteon WebOS, F5's BigIP, Resonate's Central Dispatch, described in [9]). These switches provide only very simple mechanisms. Service differentiation is typically based on either the request source or the requested service, and is provided by statically partitioning server nodes and assigning different classes of requests to different server subsets. Various results in literature [10, 26] and in this paper demonstrate that static partitioning cannot adapt to fluctuating arrival rates and servers load conditions. Moreover, it may lead to waste of resources when some partitions are not fully utilized while others might be overloaded.

8 Conclusions

In this paper we propose a cluster-based Web server enabled with Quality of Service principles at the system level. We first apply the basic QoS principles defined for network routers and protocols to the server side of the Web, and we call them Quality of Web-based Services (QoWS). We consider

various policies that satisfy all or part of the QoWS principles, and we implement them at the Web switch level of the Web cluster. Our experiments show that only the policies that satisfy all QoWS principles are able to meet the SLA targets even for stress load conditions. Indeed, the violation of even one QoWS principle prevents the Web cluster to satisfy the SLAs for the most demanding requests containing dynamic CPU/disk evaluation. The positive results achieved by the DynamicPart policy for all types of stress tests motivate further efforts aiming to combine network and server QoS mechanisms.

References

[1] T. Abdelzaher, K. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Trans. on Parallel and Distributed Systems*, 13(1):80–96, Jan. 2002.

[2] J. Almeida, M. Dabu, A. Manikntty, and P. Cao. Providing differentiated levels of service in Web content hosting. In *Proc. of Workshop on Internet Server Performance*, Madison, WI, June 1998.

[3] Apache Server Foundation. Apache HTTP Server Project. `http://www.apache.org`.

[4] 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.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. *World Wide Web*, 2(1-2), 1999.

[6] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, Mar. 1999.

[7] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, Sept./Oct. 1999.

[8] 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.

[9] V. Cardellini, E. Casalicchio, Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. In *IBM Research Report RC 22209*, Yorktown Heights, NY, Oct. 2001.

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

[11] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server system. *ACM Computing Surveys*, 34(2), June 2002.

[12] H. Chen, X. Chen and P. Mohapatra. An admission control scheme for predictable server response time for Web accesses. In *Proc. of the 10th World Wide Web conference*, Hong Kong, May 2001.

[13] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving performance of commercial Web sites. In *Proc. Int'l Workshop on Quality of Service*, London, June 1999.

[14] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proc. of USENIX Symp. on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[15] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Trans. on Networking*, 5(6):835–846, Dec. 1997.

[16] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of ACM Sigcomm 1989*, Austin, TX, Sept. 1989.

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

[18] HP Labs. WebQoS. `http://www.internetsolutions.enterprise.hp.com/webqos`.

[19] V. Kanodia and E. W. Knightly. Multi-class latency-bounded Web services. In *Proc. of Int'l Workshop on Quality of Service*, Pittsburgh, PA, June 2000.

[20] D. Krishnamurthy and J. Rolia. Predicting the QoS of an electronic commerce server: Those mean percentiles. In *Proc. of Workshop on Internet Server Performance*, Madison, WI, June 1998.

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

[22] Z. Liu, M. Squillante, and J. Wolf. On maximizing service-level-agreement profits. In *Proc. of 3rd ACM Conf. on Electronic Commerce*, Tampa, FL, Oct. 2001.

[23] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and N. E. Locality-aware request distribution in cluster-based network servers. In *Proc. of 8th ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[24] R. Pandey and R. Barnes, J. F. Olsson. Supporting quality of service in HTTP servers. In *Proc. of ACM Symp. on Principles of Distributed Computing*, Puerto Vallarta, Mexico, June 1998.

[25] N. Vasiliou and H. L. Lutfiyya. Providing a differentiated quality of service in a World Wide Web server. *ACM Performance Evaluation Review*, 28(2):22–28, Sept. 2000.

[26] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proc. of IEEE Infocom 2001*, Anchorage, Alaska, Apr. 2001.