Web Server QoS Management by Adaptive Content Delivery

Tarek F. Abdelzaher
Real-Time Computing Laboratory
EECS Department, University of Michigan
Ann Arbor, Michigan 48109–2122
zaher@eecs.umich.edu

Nina Bhatti
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
nina@hpl.hp.com

Abstract

The Internet is undergoing substantial changes from a communication and browsing infrastructure to a medium for conducting business and selling a myriad of emerging services. The World Wide Web provides a uniform and widely-accepted application interface used by these services to reach multitudes of clients. These changes place the web server at the center of a gradually emerging e-service infrastructure with increasing requirements for service quality, reliability, and security guarantees in an unpredictable and highly dynamic environment. Towards that end, we introduce a web server QoS provisioning architecture for performance differentiation among classes of clients, performance isolation among independent services, and capacity planning to provide QoS guarantees on request rate and delivered bandwidth. We present a new approach to web server resource management based on web content adaptation. This approach subsumes traditional admission control based techniques and enhances server performance by selectively adapting content in accordance with both load conditions and QoS requirements. Our QoS management solutions can be implemented either in middleware transparent to the server or by direct modification of the server software. We present experimental data to illustrate the practicality of our approach.

1 Introduction

The Internet is gradually becoming a medium for conducting business and selling services. The web presents a convenient interface for the emerging performance-critical applications, placing more stringent QoS requirements on the web server. A web server today might host several sites on behalf of parties with potentially conflicting interests, and may need to protect each party from possible overload or malicious behavior caused by another. We call this requirement *performance isolation*. In addition, the server may need to give preferential treatment to more important clients, which we call *service differentiation*. Unfortunately, today's web servers offer poor performance under overload, have no means for prioritizing requests, and have no mechanism for preallocating end-system capacity to a particular site or hosted service. Web administrators usually resort to overdesign [1] to achieve overload protection. However, if the aggregate request rate increases beyond total capacity, server response-time and connection error rate deteriorate dramatically, indiscriminately affecting all clients. This paper proposes web content adaptation as a new approach to control server resources, prevent overload, and achieve performance isolation and service differentiation. As a first step towards a more general scheme, we concern ourselves with adapting static web content only. For specificity we offer web hosting as an example of a web service that requires QoS guarantees.

Load balancing [2, 3, 4] and admission control [5] have often been used for overload protection. While admission control improves server performance by preventing overload, it offers no service to rejected connections, and cannot recover the resources wasted in the communication protocol stack on client requests eventually rejected by the server. This wasted kernel overhead may be significant at overload. Our experiences with Apache on an HP-UX platform show that as much as half of the end-system's processing capacity is wasted on eventually rejected requests when the load is only 3 times the server capacity.

As an alternative to rejection by admission control, web server load can be reduced by using multicast to distribute commonly requested pages [6]. A different approach is investigated in [7], where we survey an important category of today's e-commerce sites and present evidence of its suitability for *content adapta-*

tion to reduce overload. GIF and JPG images alone constitute, on average, more than 65% of the total bytes surveyed. In many cases, these images can be significantly compressed without an appreciable decrease in quality. Reducing the number of embedded objects per page on those sites (such as little icons, bullets, bars, separators, and backgrounds) can result in significant additional resource savings. Reducing local links is another way of adapting site content. This reduction will affect user browsing behavior in a way that tends to decrease the load on the server as users access less content. The latter approach is sometimes followed manually by administrators of larger sites such as *www.cnn.com* of the Cable News Network (CNN), e.g., upon overload caused by important breaking news.

When the server is overloaded adapted content must be available at no extra cost. Thus, in this paper we assume that content is pre-processed *a priori* and stored in multiple copies that differ in quality and size. Since a typical web site is usually in the megabyte range, storing multiple copies is cheap in terms of disk space. Multiple content trees, e.g., "/full_content" and "/degraded_content" are populated with the appropriate content off line. A URL, such as, "/my_picture.jpg" is then served from either "/full_content/my_picture.jpg" or "/degraded_content/my_picture.jpg" depending on load conditions. The convention applies to dynamic content as well, e.g., that generated by CGI scripts. Multiple content trees may contain different versions of the named CGI script that vary in resource requirements.

In the rest of this paper, we describe how the "right" content tree is selected. Section 2 describes the main adaptation architecture that allows content to be adapted in accordance with load conditions. Section 3 describes QoS management extensions, such as performance isolation and client prioritization made possible by the mechanism presented in Section 2. Implementation details are discussed in Section 4. The presented architecture is evaluated in Section 5 using a working prototype. The paper concludes in Section 6 with a summary of contributions and suggestions for future work.

2 QoS Adaptation Architecture

We propose to control web server load via content adaptation. In order to do so, we interpose a software layer between the server processes and the communication subsystem. The layer has access to the HTTP requests received by the server and the responses sent. It intercepts each request and prepends the requested URL name by the name of the "right" content tree from which it should be served in accordance with load conditions. To decide on the "right" content tree for each client the interposed content adaptation layer must measure the current degree of server utilization, and decide on extent of adaptation that will prevent underutilization or overload. We call these functions *server load monitoring* and *server utilization control* respectively. These components are described in the following subsections for the simple case when all clients have the same priority. Issues of QoS management in the presence of multiple hosted sites, or clients of different priority are discussed in Section 3.

2.1 Load Monitoring

The objective of load monitoring is to quantify server utilization with a single value that summarizes resource consumption on the scale described above. We noticed that the service time of a request can be decomposed into a fixed overhead component and data-size-dependent overhead component. Thus, if a URL of size x is requested, the request service time can be approximated by $T(x) = a + bx$, where a and b are platform constants. Summing the service times of all requests in a particular observation period and dividing by the length of the period we obtain system utilization, U. Using some algebraic manipulation we can prove that:

$$U = aR + bW. \qquad (1)$$

where R is the observed request rate, and W is the aggregate delivered bandwidth. The load monitor periodically measures R and W online, and returns the corresponding utilization value. The constants a and b in Equation 1 are independent of workload and as such can be determined for the server platform *a priori*.

To compute a and b the server must be profiled. A simple way of profiling the server is to subject it to an increasing request rate and estimate its maximum processing capacity. Let us fix the requested URL size x and increase server request rate gradually until connection errors are observed. The maximum achievable request rate, R_{max}, for which no connection errors occurs[1] is recorded, as well as the corresponding total delivered bandwidth, W_{max}. Let us repeat the experiment with a different requested URL size, and record the new maximum rate and bandwidth. In every case the maximum request rate corresponds (for our purposes) to a fully utilized server, i.e., U = 100%. Thus, each experiment yields a different data point (R_{max}, W_{max}) that satisfies the equation

[1] We take "no errors" to mean an error rate of less that 0.1%

$100 = aR_{max} + bW_{max}$ in the unknowns a and b. Using linear regression coefficients a and b are found. In our implementation, these constants are obtained off-line and written into a configuration file used at run-time by the load monitor to substitute in Equation 1.

2.2 Utilization Control

Ideally, when server utilization is low, all clients receive the best available content. When server utilization approaches saturation a fraction of clients must be degraded (i.e., will be delivered degraded content) in order to avoid server overload and connection failures. The fraction of clients affected and the adaptation action taken on them are determined by a self-regulating utilization control loop. The utilization control loop determines the severity and scope of the adaptation action required, and implements the action on the clients' requests. The following subsections describe its two main components; the *content adaptor* which causes content adaptation, and the *utilization controller* which tells the adaptor how much to adapt by selecting one of a range of content trees.

2.2.1 The Content Adaptor

Let there be M content tress, numbered from 1 to M in increasing order of quality. (For a typical adaptive server we expect that $M = 2$.) We use an abstract parameter G to represent the severity of the adaptation action required from the adaptor. The adaptor accepts input G in the range $[0, M]$. The upper extreme, $G = M$, indicates that all requests are to be served the highest quality content (i.e., served from tree M). This is the nominal operating mode of the server. The lower extreme, $G = 0$, means all requests must be *rejected*. The content adaptor, thus, supersedes and extends admission control by offering adaptation as an additional alternative, hopefully minimizing the need for rejection. The parameter G controls server load, where lowering G reduces the the load on the server. A non-zero integer value of G, say $G = I$, indicates that all requests must be served from tree I. In general, G may be a fractional number. Let I be the integral part of G, and F be the fractional part. The following rules are used by the content adaptor to determine which tree to serve an incoming request from:

- If G is an integer (i.e., $G = I,\ F = 0$), the request is served from tree I.
- If G is not an integer, a pseudo-random value N is computed by a hashing function $H()$ that hashes the client's id (e.g., its IP address) into a number in the range $[0, 1]$ upon the receipt of the request. If $N < F$ the request is served from tree $I + 1$. Otherwise it is served from tree I.

The second rule ensures that when G increases, the likelihood of serving a client from a "better" tree increases, and vice versa. As mentioned above, choosing the non-existing tree 0 means that the request is rejected. Figure 1 illustrates the achieved degradation spectrum (shown as a horizontal slide-bar). It ranges from serving all requests from the highest quality content tree to rejecting all requests, as specified by the continuous tunable parameter, G. The Figure 1 shows how a given value of G determines both the trees from which requests are served and the fraction of requests served from each tree. All requests originating from the same client must be served from the same tree whenever possible. For that purpose, the hashing function, $H()$, maps a given client id to the same number every time. Thus, all requests from the same client will always be served from the same tree for the same load conditions.

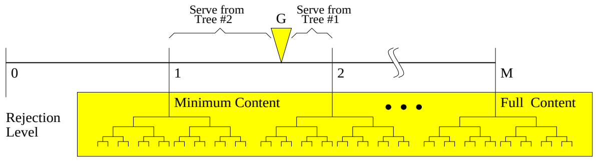


Figure 1: The Degradation Range

2.2.2 The Utilization Controller

This section presents how the parameter G is tuned dynamically in a self-regulating fashion to achieve constant maximum server utilization in the presence of variable load. The maximum server utilization is an arbitrary specified bound that we do not wish server utilization to exceed. A good value would be 85%. Let U^* be the desired "optimal" utilization of the server. Let U be the current utilization computed by the load monitor as described in Section 2.1. Let E be the "utilization error", $E = U^* - U$. The adaptation controller samples the current utilization U, and computes the corresponding error E at fixed time intervals, then produces an output, G, that regulates the extent of adaptation. We use the well-known *integral controller* to produce the control output. The basic integral controller produces an output that is proportional to the sum of the observed input samples since system startup. At each sampling time the controller performs the following computation:

$G = G + kE$; **If** $(G < 0)$ **then** $G = 0$; **If** $(G > M)$ **then** $G = M$; where k is a constant.

Intuitively, if the server is overloaded (i.e., $U > U^*$) the negative error E results in a decrease in G. As a result the fraction of degraded requests increases which decreases server utilization (and vice versa). When the server reaches target utilization (i.e., $U = U^*$), the error E becomes zero, and this G is fixed.

Classic control literature proposes analytic techniques to tune the integral controller, i.e., set the value of k in the above equation for best convergence. More sophisticated versions of that controller include the proportional integral (PI) controller and the proportional integral differential (PID) controller. We use a PI controller in our loop, tuned for quarter amplitude damping, which is a traditional industrial control practice. The controller is modeled by the differential equation characterizing the PI control action [8]. The controlled process is modeled as a dead time element of value equal to half the sampling time of the controller. For space limitations, we omit the details of controller tuning in this paper. An interested reader is referred to any of several classic control theory textbooks [8].

The control loop is depicted in Figure 2. The figure summarizes the elements of our content adaptation architecture and their interaction; The load monitor measures current request rate and delivered bandwidth, and translates them into a single utilization value, U. The utilization controller compares U to the specified desired server utilization, U^*, and computes the required extent of adaptation G. The content adaptor interprets G and degrades or rejects a fraction of requests accordingly by modifying their requested URL names. The server serves the URL names modified by the content adaptor. The load monitor updates its load estimate thereby closing the control loop.

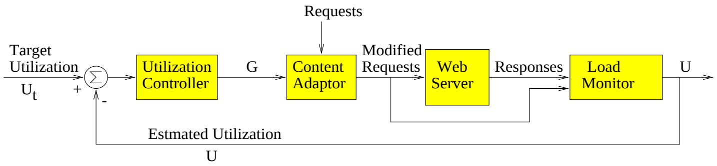


Figure 2: The Utilization Control Loop

Figure 3 depicts the efficacy of the control loop to achieve a desired target server utilization. In this experiment, the request rate on the server was increased suddenly, at $time = 13$, from zero to a rate that offers a load equivalent to 300% of server capacity. Such a sudden load change is much more difficult to deal with than small incremental changes, thereby stress-testing the responsiveness of our control loop. The target utilization, U^*, was chosen to be 85%. As shown in Figure 3, the controller was successful in finding the right degree of degradation such that measured server utilization remains successfully around the target for the duration of the experiment. The experiment demonstrates the responsiveness and efficacy of the utilization control loop.

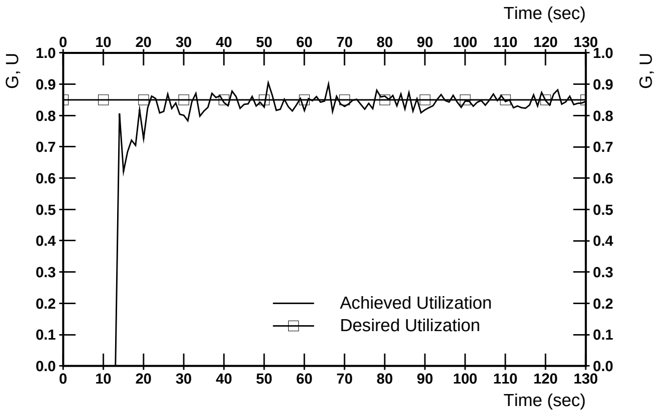


Figure 3: Utilization Control Performance

3 QoS Management

In this section we describe how the general architecture described in Section 2 is extended to support the following important features:

- *Performance isolation and QoS guarantees*: A web server can host multiple independent sites. We associate a *virtual server* with each hosted site. The virtual server guarantees a maximum request rate and maximum delivered bandwidth for the site independently of the load on other sites thereby achieving performance isolation.

- *Service differentiation*: Clients may have different priorities. In addition to achieving performance isolation and QoS guarantees, each virtual server supports request prioritization. Upon overload, lower priority requests are degraded first.

- *Excess capacity sharing*: While each virtual server adapts content under overload to remain within its individual capacity allocation, if some virtual server does not consume all its allotted resources, the excess capacity is made available to other virtual servers allowing them to exceed their capacity allocation if so needed to avoid client degradation.

3.1 Performance Isolation

We export the abstraction of virtual servers. A virtual server is configured for a specified maximum request rate R_{max} and a specified maximum delivered bandwidth W_{max}. The configuration expresses an agreement whereby the server guarantees the ability to deliver an aggregate bandwidth of up to W_{max} as long as the aggregate request rate does not exceed R_{max}.

If the request rate condition is violated (i.e., exceeds R_{max}) the bandwidth guarantee is revoked. The virtual server adapts delivered content to achieve the maximum possible bandwidth delivery for the given request rate without overrunning its capacity allocation. The following provisions in our architecture cooperate to export the virtual server abstraction and achieve performance isolation:

- *Capacity planning:* The maximum maintainable request rate R_{max_i} and the maximum delivered bandwidth W_{max_i} specification of each virtual server i are converted into a corresponding target capacity allocation, $U_i^* = aR_{max_i} + bW_{max_i}$. The target utilization sum $\sum_i U_i^*$ over all virtual servers residing on the same machine should be less than 100% for the guarantees to be realizable. This is checked each time the adaptation software parses its configuration file. If the administrator configures a new virtual server that makes $\sum_i U_i^* > 100\%$, a capacity planning error is returned.

- *Load classification:* A load classifier intercepts input requests and classifies them to identify the virtual server responsible for serving each request. Request classification can be done based on the requested content, addressed site, or other information depending on system administrator's policy. If each virtual server is associated with a hosted site, requests are classified based on the site name embedded in the URL string. Load classification allows proper load bookkeeping for each virtual server independently to achieve performance isolation.

- *Utilization control:* When requests are classified, the request rate R_i and delivered bandwidth W_i can be computed individually for each virtual server i, from which a corresponding utilization value, $U_i = aR_i + bW_i$, is obtained. The utilization U_i of each virtual server is controlled by a separate instance of the utilization control loop described in Section 2.2. Each control loop achieves the degree of content degradation necessary to keep U_i of its virtual server at or below its target value, U_i^*, thereby achieving the server's individual performance guarantees, while preventing overload. The architecture is depicted in Figure 4.

3.2 Service Differentiation

In this section we describe how service differentiation is incorporated into our architecture for adaptive content delivery. The goal is to support client prioritization such that lower priority clients are degraded first. Consider a virtual server that supports client prioritization. Let there be m priority classes defined within that server, such that priority 1 is highest, and priority m is lowest. Collectively, clients of the virtual server are allocated a target utilization U^* derived from a maximum rate and maximum bandwidth specification for that server. This capacity should be made available to clients in priority order. We allocate the entire virtual server capacity to the highest priority class. The unused capacity of each class is measured and allocated to lower priority classes. If this capacity is not enough, these clients will be degraded or rejected accordingly by the utilization control loop. The following rule is used to degrade clients:

- For each priority class j, the target utilization is $U_j^* = U^* - \sum_{i<j} U_i$, where $U_i = aR_i + bW_i$ is the current measured utilization of the (higher priority) class i.

- Given the target utilization of each class j, as well as its measured utilization, $U_j = aR_j + bW_j$, the control technique described in Section 2.2.2 is applied to compute the extent of adaptation required for this class. Let us denote it by G_j.

- Each class in adapted in accordance with the value of its specific G_j as described in Section 2.2.1.

In the presence of low priority traffic, a higher priority class should also account for the overhead it may take to reject lower priority requests under overload. This can be figured in the computation of U_j^* as follows:
$U_j^* = U^* - \sum_{i<j} U_i - \sum_{l>j} U_{reject_l}$.

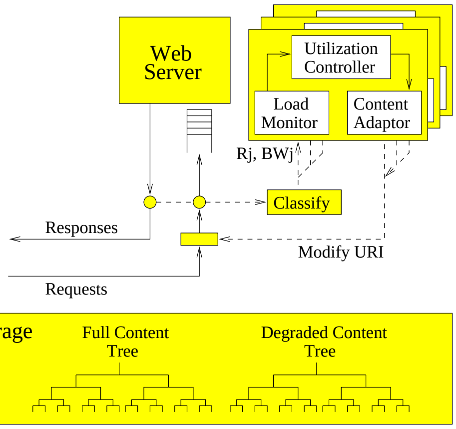


Figure 4: Architecture for Performance Isolation

where $U_{reject_l} = a_{reject} R_l$ is the overhead of rejecting all current requests of a lower priority class l, the overhead of rejecting a single request being a_{reject}.

3.3 Sharing Excess Capacity

An important advantage of grouping several virtual servers on the same machine is the ability to better reuse extra server capacity. Consider two physically separated servers, each of capacity, C. If load on one exceeds capacity while the other is underutilized, there is no way to reroute extra traffic to the idling server (unless a gateway is used in front of the server farm to balance load). Idling resources may be wasted on one server while requests are being rejected on another. A single server of capacity $2C$ does not suffer this problem. We therefore extend the preceding mechanisms to allow virtual servers to exceed their contracted target utilization, U^*, as long as there is extra capacity on the machine. Since the virtual server has no contractual obligation to provide the extra capacity in the first place, extra request traffic for any virtual server is uniformly treated on best-effort basis as non-guaranteed. Non-guaranteed traffic is allowed to occupy the excess capacity on the machine using a mechanism similar to that of service differentiation described in the previous section. Specifically, the degradation level G_n of non-guaranteed traffic is computed from $G_n = G_n + k(100 - U)$, where $U = aR + bW$ is the current aggregate utilization of the computed from the aggregate request rate and bandwidth. We present an evaluation of these techniques in Section 5. Implementation details are discussed next.

4 Implementation

The adaptation software was implemented in C for a UNIX platform. The software was tested on a single processor K460 (HP PA-8200 CPU) workstation running HP-UX 10.20, with 512MB main memory and GSC 100-BaseT network connection. For the purpose of experimentation an Apache 1.3.0 web server was used. In this section we give more details on software implementation, the testing environment and evaluation of adaptation software.

4.1 Web Server Model

In order to handle a large number of clients concurrently, web servers adopt either a multithreaded or a multi-process model. Multithreaded web servers require kernel thread support. Such support is provided in most modern operating systems, e.g., Solaris and Windows NT. A separate kernel thread is assigned by the server to each incoming HTTP request. Threads can share common state in global memory. In a multi-process model, common to UNIX implementations, a separate process is assigned to each incoming request. Since spawning a process is a heavy-weight operation, a pool of processes is usually created at server startup. Created processes listen on a common web server socket, and may communicate via shared memory. A process that accepts a connection handles it until it is closed. Apache 1.3.0, used in our experiments, subscribes to this model.

The adaptation software is designed as a middleware layer between the web server and the underlying operating system. The middleware API may be called directly from the web server if desired, in which case it is not transparent. Alternatively, middleware calls may be made from the socket library used by the server, in which case server code remains unmodified. We begin by describing the API of our adaptation middleware.

4.2 Adaptation Software API

Adaptation mechanisms described in this paper require three entry points. Namely, (i) an initialization point, (ii) a request pre-processing point, and (iii) a request post-processing point. The first point is called once upon server startup. The latter two are called upon the receipt of each request and the sending of each reply respectively. The specific calls are as follows. **adaptsoft_init ()** is called from the main server process before forking workers. The function will initialize some global variables and fork off the *utilization controller* which will implement server utilization control loops. **adaptsoft_adapt (URL, client_IP)** is called by workers each time an HTTP request is received. It classifies the client and returns the actual URL name to be served, or NULL if the request is to be rejected. **adaptsoft_log_size (URL_byte_size)** is called by workers to update transmitted bandwidth measurements by the byte size of a served URL.

4.3 Implementing Load Monitoring

When a request is first dequeued from the server socket's listen queue by some worker process, P_i, the function *adaptsoft_adapt ()* is called in the context of P_i. This function classifies the request as belonging to virtual server j. The function then increments a counter, $r_i[j]$, that accumulates the number of requests for virtual server j seen by worker process P_i. When P_i has finished processing the request, it sends out the response and calls *adaptsoft_log_size()* passing it the number of bytes sent. The function *adaptsoft_log_size()* updates a counter, $b_i[j]$, that accumulates the total bytes sent by process P_i on behalf of virtual server j.

Periodically, a call to *adaptsoft_adapt ()* by process P_i also invokes a utilization measurement function. The function computes on behalf of each virtual server k the request rate $R_i[k] = r_i[k]/t$ that process P_i has seen for the virtual server within the last t time units, and the bandwidth $W_i[k] = b_i[k]/t$ that process P_i has delivered on behalf of the virtual server within that time interval. Finally it computes the utilization $U_i[k] = aR_i[k] + bW_i[k]$ that process P_i consumed on behalf of each virtual server k, and stores the respective values of $U_i[k]$ in shared memory. All counters $r_i[k]$ and $b_i[k]$ are then cleared in preparation for the next period. Note that the utilization measurement function is invoked separately in each worker process P_i to compute its contribution to the utilization of virtual servers.

4.4 Implementing Utilization Control

Utilization control is implemented in a separate process forked off by *adaptsoft_init()* during startup. The process executes a loop that wakes up periodically to compute the extent of degradation for each virtual server then sleeps until the next period. Upon waking up, the controller computes the utilization, U_k of each virtual server k by aggregating the recorded contributions $U_i[k]$ of all worker processes, P_i, towards U_k. Thus, $U_k = \sum_i U_i[k]$. This utilization is then compared to the desired utilization for the virtual server and the degree of degradation G_k is computed accordingly as described in Section 2.2.2. The value of G_k for each virtual server k is stored in shared memory.

Each time **adaptsoft_adapt (URL, IP)** is invoked in the context of a worker process upon the receipt of some new request it will classify it and read from shared memory the current value of G_k for the corresponding virtual server. The function will then execute the algorithm in Section 2.2.1 to determine which content tree to serve the request from, and prepend the requested URL name by the name of that tree. (For simplicity, we omitted in this section the implementation details related to performance differentiation among clients of the same virtual server.)

5 Evaluation

In this section we present a performance evaluation of the developed adaptation software. To emulate a large number of web clients we used httperf [9], a testing tool that can generate concurrently a large number of HTTP requests for specified URLs at a specified rate. In order to overload the web server, httperf was run on 4 workstations collectively emulating the community of clients. The workstations were connected to the server via a 100Mb switched Ethernet.

5.1 Estimating Service Time

In our first experiment, we profiled the Apache server to determine the time, T, it takes to serve a URL of size x. Measuring server response time was found not to be indicative of service time T, because the former includes queuing time, network latency, etc. We therefore measured service time by obtaining the inverse of the maximum throughput. The experiment was repeated for different sizes of the requested URL. We found approximately that $T(x) = a + bx$, where $a = 1.604$ and $b = 0.063$. Table 1 shows the quality of this approximation.

URL Size (KB)	Measured T	$a + bx$	Error
1	1.706	1.677	-1.7%
2	1.73	1.73	0%
4	1.858	1.856	-0.1%
8	2.075	2.108	1.6%
16	2.611	2.612	0%
32	3.322	3.62	8.9%
64	5.917	5.636	-4.7%

Table 1: Approximating Service Time

5.2 Request Rejection Overhead

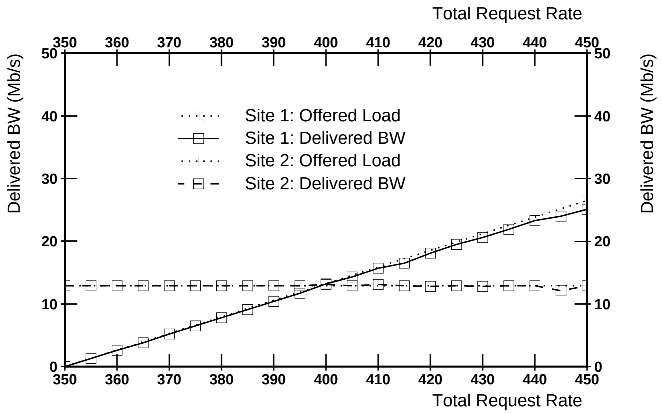
We mentioned in the introduction that rejection of client requests wastes a lot of server resources with no benefit to the rejected client. To quantify the rejection overhead, we instrumented the server to reject all requests by closing the connection as soon as the request is read off the server socket. The request rate on the server was then increased, and the maximum response rate was recorded. The maximum rate was found to be around 900 req/s, which is the maximum rate at which rejection can be processed. The rejection overhead (the inverse of the maximum rejection rate) is thus approximately 1.1 ms/req. This is to be compared with 1.604, the time it takes to serve a "very small" URL (denoted by constant a in Section 5.1). The difference is believed to be due to file system access associated with serving the URL. It appears that this difference is not substantial. Rejecting a set of requests will consume almost 70% of the resources it would take to serve them a short URL. Request rejection, therefore, should be avoided whenever possible, which we believe is achieved in our content adaptation scheme. For better efficiency, request classification and rejection, when necessary, should be done in the kernel at the earliest point possible upon request reception in order to conserve end-system's resources.

5.3 Performance Isolation

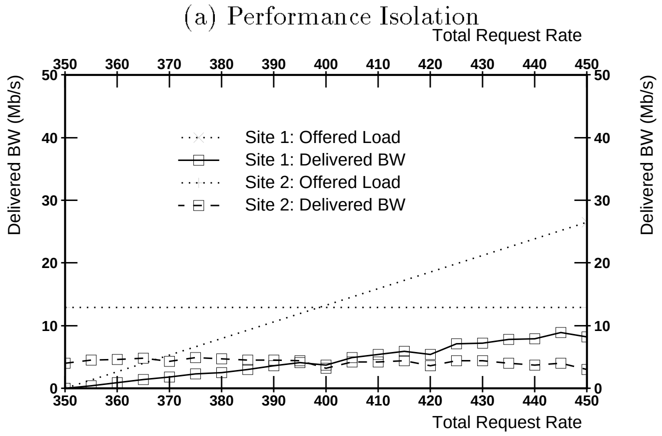
In this paper, we described a performance isolation mechanism that allows creating multiple virtual servers with individual rate and bandwidth guarantees. The mechanism provides protection among individual virtual servers, as well as protection between the virtual servers and the non-guaranteed best-effort traffic. Figure 5-a demonstrates these features. In this experiment a best-effort background load of 300 req/s (for 32KB URLs) was applied to overload the machine. In addition, two virtual servers, V_1 and V_2, were configured. Server V_1 was configured for a maximum guaranteed bandwidth of 13 Mb/s, and maximum guaranteed rate of 50 req/s. Server V_2 was configured for a maximum guaranteed bandwidth of 27 Mb/s and a maximum guaranteed rate of 100 req/s. Each virtual server was associated with a different hosted site. A constant load of 50 req/s (with a total bandwidth requirement of 12.8 Mb/s) was applied to the first site (V_1). The load on the second (V_2) was increased gradually from 0 to its maximum specification. The aggregate load on the machine was well above the overload threshold due to the existence of background best effort traffic (for other non-guaranteed sites).

Figure 5-a depicts the offered load on each of V_1 and V_2 (i.e., the total requested bandwidth), as well as the actual bandwidth delivered. Both are plotted versus the aggregate request rate on the server. For clarity, the best effort load is not shown. It can be seen that the actual bandwidth delivered by each virtual server follows closely the offered load. Thus, despite server overload, due to background best-effort traffic, virtual servers V_1 and V_2 attain the contracted performance guarantees for their respective sites, and suffer virtually no content degradation. Furthermore, variations in load on virtual server V_2 do not affect virtual server V_1. Performance isolation is thus achieved in the sense of maintaining the QoS guarantees independently for each virtual server regardless of other load.

For comparison, we repeated the same experiment using a regular Apache server that does not use our adaptation extensions. Figure 5-b depicts the results obtained. It can be seen that the delivered bandwidth of both sites falls short of the offered load. The difference reflects the fraction of connections that fail and don't get served due to overload. Note also how the increase in delivered bandwidth of one site results in a decreases in delivered bandwidth of another. No performance isolation is observed. The comparison of



(a) Performance Isolation



(b) Regular Apache

Figure 5: Performance Isolation

Figure 5-a and Figure 5-b illustrates the advantage of the developed adaptation software.

5.4 Service Differentiation

Adaptation software allows defining multiple priority classes of requests. In this section we experiment with defining two priority classes, namely a basic class B and a premium class P. Requests of class P are treated as higher priority than those of B. In the experiment, we offered a constant load of 100 premium class requests per second. We then gradually increased the rate of basic class requests. Figure 6 plots the delivered premium and basic bandwidth versus request rate. It also shows the offered load of both premium and basic clients. Note that when the server becomes overloaded, basic clients are degraded before premium clients thus achieving service differentiation.

5.5 Excess Capacity Sharing

As we argued earlier, an important advantage of colocating several adaptive virtual servers on the same machine is the ability to utilize unused capacity of one virtual server by another that is overloaded. The

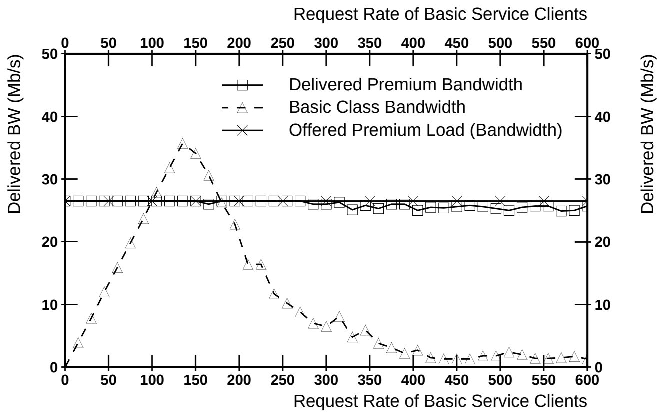


Figure 6: Service Differentiation

overloaded server should be allowed to exceed its individual capacity allocation when extra capacity is available, as long as it does not affect other virtual servers. When the machine is overloaded, however, each virtual server should be policed to its individual capacity allocation in order to achieve performance isolation and overload control. These two features are provided by the excess capacity sharing mechanism described in Section 3.3. To evaluate the efficacy of this mechanism we conducted two experiments. In the experiments a virtual server V_1 is created whose offered load at run-time exceeds its capacity allocation. Low background load is used in the first experiment. As a result, virtual server V_1 is allowed to overrun its capacity allocation utilizing the excess capacity on the machine. In the second experiment, high background load is applied. As a result, the virtual server is policed to its individual capacity limit. Moreover, in both experiments a second virtual server, V_2, is also used. Server V_2, which is loaded within its capacity limit at all times, is shown to exhibit no degradation despite the (controlled) capacity overrun of server V_1, and the background load. Excess capacity sharing is thus shown not to interfere with performance isolation.

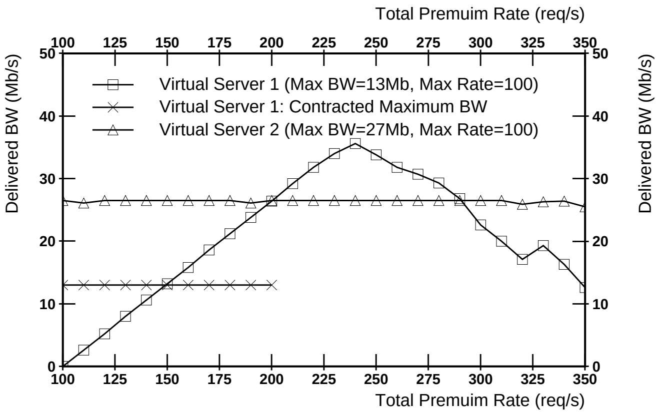
Figure 7-a depicts the results of the first experiment. It shows the contracted as well as the actual bandwidth of servers V_1 and V_2. Server V_1 is configured for maximum bandwidth of 13Mb/s, and maximum request rate of 100 req/s. Server V_2 is configured for maximum bandwidth of 27Mb/s and maximum request rate of 100 req/s. At run time, the request rate of V_2 is held constant at 100, offering a total bandwidth requirement of 25.6Mb/s, i.e., just within its capacity limit. The request rate on server V_1 is increased gradually from 0 to 250 req/s. The aggregate rate of both servers combined is shown on the horizontal axis. It can be seen that server V_2 overruns its capacity allocation delivering a peak of about 35Mb/s at a rate of 140 req/s (at which the aggregate rate is 240 req/s in Figure 7-a). This is to be compared with its guaranteed maximum bandwidth of 27Mb/s and maximum request rate of 100 req/s. Server V_1 remains unaffected, since the excess capacity sharing mechanism ensures performance isolation.

The experiment is repeated after adding a background load of 100 req/s to overload the server. The results are shown in Figure 7-b which depicts the contracted as well as the actual bandwidth of servers V_1 and V_2 in new the case. While the contracted bandwidth is the same as in Figure 7-a, we can see that the actual bandwidth differs. For ease of comparison with Figure 7-a, the horizontal axis, as before, represents the aggregate request rate of V_1 and V_2 combined. It can be seen that V_1 in the second experiment is made to deliver exactly its maximum guaranteed bandwidth, 27Mb/s, when its rate reaches the maximum guaranteed rate, 100req/s (at which the aggregate request rate on the server is 200 in Figure 7-b). That is, in the absence of excess capacity on the machine, the virtual servers are policed to their capacity allocation. Note that request rates higher than 100 on V_1 result in a bandwidth lower than 27Mb/s and vice versa. This is an effect of content adaptation to keep V_1s utilization constant once it reaches V_1s allocated utilization limit.

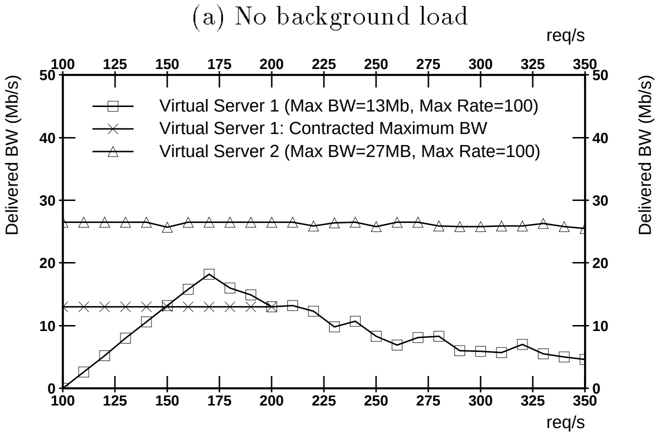
6 Conclusions

In this paper we presented a QoS management architecture that relies on adapting delivered content. Unlike present day non-adaptive servers, and unlike servers that implement binary admission control, content adaptation enables a server to provide a smooth range of client degradation thereby coping with overload in a graceful manner. We described the design and implementation of a utilization control loop that adapts delivered content in a way that respects a specified utilization bound in the presence of variable server load while virtually eliminating connection errors. We demonstrated several extensions to this mechanism that provide performance isolation, service differentiation, sharing excess capacity, and QoS guarantees. The mechanisms described in this paper are largely independent of workload assumptions, and can be easily applied to a different platform by appropriately tuning a small set of parameters using well-founded analytic techniques. The architecture can be implemented in a middleware layer transparently to existing server and browser code thereby facilitating deployment.

There are several outstanding issues and challenges



(a) No background load



(b) High background load

Figure 7: Sharing Excess Capacity

in this context that can be subject of future research. Handling and adapting dynamic content is one interesting issue. The inherent unpredictability of CGI script execution times offers new challenges to load characterization. The experiments reported in this paper used the HTTP 1.0 protocol. It is interesting to see whether the same results will hold for HTTP 1.1. While some aspects of client classification may be simplified, persistent TCP connections may impose less predictable server load characteristics that are more sensitive to client-side bandwidth. The approach of storing multiple copies of content is affordable for the typical web site size. In video servers, however, an important issue to investigate is scalable video encoding schemes that avoid storing multiple copies of the content. Finally, an interesting research area is that of investigating appropriate content authoring and management tools to preprocess web content in a way that preserves enough information, yet consumes a minimal amount of resources.

Acknowledgment

The authors wish to thank Rich Friedrich, Gita Gopal, and Kang Shin for their valuable suggestions and comments on earlier manuscripts of this paper.

References

[1] S. Schechter, M. Krishnan, and M. D. Smith, "Using path profiles to predict http requests," in *7th International World Wide Web Conference*, pp. 457–467, Brisbane, Qld., Australia, April 1998.

[2] M. Colajanni, P. S. Yu, V. Cardellini, M. P. Papazoglou, M. Takizawa, B. Kramer, and S. Chanson, "Dynamic load balancing in geographically distributed heterogeneous web servers," in *Proceedings of 18th International Conference on Distributed Computing Systems*, pp. 295–302, Amsterdam, Netherlands, May 1998.

[3] D. Andersen and T. McCune, "Towards a hierarchical scheduling system for distributed www server clusters," in *Proceedings The Seventh International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[4] A. Vahadat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa, "Webos: operating system services for wide area applications," in *Proceedings The Seventh International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[5] A. Iyengar, E. MacNair, and T. Nguyen, "An analysis of web server performance," in *GLOBECOM*, volume 3, pp. 1943–1947, Phoenix, AZ, Nov 1997.

[6] P. White and J. Crowcroft, "Www multicast delibvery with classes of service," in *HIPPARCH*, University College, London, June 1998.

[7] T. Abdelzaher and N. Bhatti, "Web content adaptation to improve server overload behavior," in *submitted to International World Wide Web Conference*, Toronto, Canada, May 1999.

[8] T. Kaczorek, *Linear control systems*, John Wiley, New York, 1992.

[9] D. Mosberger and T. Jin, "httperf: A tool for measuring web server performance," in *WISP*, pp. 59–67, Madison, WI, June 1998, ACM.