Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach*

Tarek F. Abdelzaher
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Kang G. Shin
Real-Time Computing Laboratory
EECS Department, University of Michigan
Ann Arbor, MI 48109–2122

Nina Bhatti
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

The Internet is undergoing substantial changes from a communication and browsing infrastructure to a medium for conducting business and marketing a myriad of 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 and reliability guarantees in an unpredictable and highly-dynamic environment.

This paper describes performance control of a web server using classical feedback control theory. We use feedback control theory to achieve overload protection, performance guarantees, and service differentiation in the presence of load unpredictability. We show that feedback control theory offers a promising analytic foundation for providing service differentiation and performance guarantees.

We demonstrate how a general web server may be modeled for purposes of performance control, present the equivalents of sensors and actuators, formulate a simple feedback loop, describe how it can leverage on real-time scheduling and feedback-control theories to achieve response-time and throughput guarantees, and evaluate the efficacy of the scheme on an experimental testbed using the most popular web server, Apache. Experimental results indicate that control-theoretic techniques offer a sound way of achieving desired performance in performance-critical Internet applications. The implementation of our QoS (Quality-of-Service) management solutions can be implemented either in middleware that is transparent to the server, or as a library called by server code.

1 Introduction

The Internet has become an important medium for conducting business and selling & buying services. The web presents a convenient interface for the emerging performance-critical applications that require more stringent QoS (Quality-of-Service) from the web server. A web server may 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 party. 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*. With the introduction of applications such as online trading, e-commerce, and real-time databases, failure to meet performance requirements may result in loss of customers, serious financial damages, or legal liabilities.

Despite the increasing need for QoS-aware server design, most of today's web servers offer poor performance under overload, provide no means for prioritizing requests, and have few mechanisms for pre-allocating

*The work reported in this paper was supported in part by the NSF under Grant EIA-9806280.

the end-system capacity to a particular site or hosted service. Web administrators usually resort to overdesign [16] for overload protection as well as for providing an “acceptable” level of performance. As a result, performance guarantees cannot be made for individual clients or client categories. When the server gets overloaded, *all* clients may suffer unacceptably long delays and/or connection failures even if enough resources may exist to serve a *subset* of clients efficiently.

Support for different classes of service on the web has been investigated in recent literature. An important measure of quality is the responsiveness of the server. In the simplest case, it is desired to differentiate between two classes of clients, *premium* and *basic*, such that premium clients receive better service than basic clients in case of overload. For example, the authors of [12] proposed and evaluated an architecture in which restrictions are imposed on the amount of server resources (such as threads) available to basic clients. In [7, 1] admission control and scheduling algorithms are used to provide premium clients with better service. In [9] a server architecture is proposed, maintaining separate service queues for premium and basic clients and thus, facilitating their differential treatment.

While the above architectures offer better service to premium clients, no individual performance guarantees are given in serving requests. This may not be acceptable to critical applications such as online trading. Our work [5, 6, 4, 14] differs from other approaches in that it offer performance guarantees. Server response time, i.e., the time it takes to serve a client's request, is probably the most visible performance metric from the client's perspective. Similarly, server throughput is the performance metric of interest to providers of hosted content.

In this paper, we consider guarantees on server response time and throughput. We show that classical feedback control theory offers a solution to the problem of achieving performance guarantees in unpredictable environments such as the web, and discuss the challenges involved in realizing this approach. We demonstrate that a web server can be approximated by a time-varying linear model for purposes of performance control, describe the needed software actuators and sensors in the software system, and cast server performance control as a classical feedback control problem. Experimental results derived from testing the scheme on a widely-used web server (Apache) illustrate the potential of the approach. Real-time (deadline-monotonic) scheduling theory [8] makes response-time guarantees possible if server utilization is maintained below a pre-computed bound. In the absence of exact knowledge of per-client load, utilization can be maintained around the bound via feedback control to enforce the specified response-time bounds. Feedback control can also be used to guarantee individual hosted sites a given throughput independently of load on other sites, and to provide differentiated services.

In the rest of this paper, we detail our performance control architecture and describe issues in applying control-theoretic approaches to a computing server. Section 2 describes the computing system being controlled. Section 3 presents the control problem, the issues involved with sensors and actuators, their modeling, and closing the feedback control loop around a popular web server. Section 4 describes how the utilization control loop discussed in Section 3 can be used for performance isolation and service differentiation. Section 5 describes the implementation of a working prototype using an Apache web server. Section 6 evaluates the performance of the prototype that implements the feedback control loops on an experimental testbed. Section 7 concludes the paper with a summary of contributions and suggests avenues for future work.

2 The Computing Server

We consider a distributed client-server system in which clients send a succession of requests to the web server across a communication network. Each service request has an implicit soft deadline by which it must be served (perhaps determined from the importance, preferences, or subscription fees of the client). The delay seen by the client includes the time a request spends in the network plus the time it spends on the server. Research in the networking community addressed the problem of bounding network delays, as in diff-serv [10] and int-serv [11] architectures. We address the complementary problem of bounding the *server-side* delay, and the problem of guaranteeing a given throughput to individual hosted sites. With the dramatic increase in the number of Internet clients, servers are becoming potential bottlenecks.

Serving a request consumes multiple server resources, such as memory, disk bandwidth, communication bandwidth, and CPU cycles. The capacity of the server is limited by that of the bottleneck resource. We assume that the ith request has a deadline D_i and consumes an amount C_i of the bottleneck resource. Since each request must be executed by its deadline, the utilization of the bottleneck resource is $U = \sum_i C_i/D_i$, where the summation is carried out over all current requests. A request is said to be *current* if it has arrived but its deadline has not yet expired.

3 The Control Problem

The objective of our performance control loop is to (i) avoid server overload, and (ii) meet the individual response time and throughput guarantees. It has been recently proved [3] that a group of aperiodic tasks with arbitrary arrival times, computation times, and relative deadlines (i.e., maximum response times), scheduled by a fixed-priority (deadline-monotonic) policy, will always meet their deadline constraints as long as $U < 5/8$. This result is a generalization of the famous Liu and Layland's schedulable utilization bound of $ln\ 2$ [13], derived for periodic tasks. The new bound leads to a simple implementation of a server that guarantees aperiodic request deadlines; the server simply needs to ensure that its utilization does not exceed the aforementioned bound. Since it is also desired to maximize server throughput, the utilization should be maintained exactly at the upper bound, if possible. The key difficulty in implementing an algorithm that would observe the utilization bound is the lack of proper estimates of resource load imposed by an individual client.

A classical linear feedback control problem can be formulated to keep server utilization at or below $5/8$. Server load can be approximated by a liquid flow model. Since a typical server can handle thousands of clients concurrently, each client contributes an unknown, but small, amount to this flow, represented by a series of requests. The control loop, depicted in Figure 1, measures server utilization and determines (based on load conditions) a subset of clients that may receive service at the current time. The size of this subset is adjusted to keep the utilization at the desired level. The three main challenges in implementing the control loop are (i) the choice of a proper actuator that can affect server utilization, (ii) the choice of a monitor that can measure current utilization reliably, and (iii) appropriate modeling and control of the server. The above three challenges are elaborated upon below. Section 3.1 describes issues in designing the actuator. Section 3.2 describes the monitor. Section 3.3 discusses system modeling for the purposes of feedback control and presents the procedure we used for controller tuning.

3.1 The Actuator

The actuator is the element responsible for translating abstract controller output into physical action taken by the web server to change its load. In general, *admission control* can be used as an actuator in computing servers. In the simplest case, admission control limits the number of clients who access the server concurrently. Client rejection is undesirable but unavoidable because no scheme can provide QoS guarantees to all web clients, unless it limits the total number of clients on the Internet to the maximum number that can be served by a single server concurrently. The challenge a control loop overcomes is that of providing performance guarantees to the clients whom the "actuator" chooses to admit.

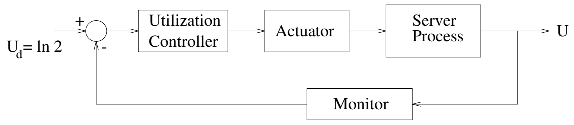


Figure 1. The utilization control loop

An extension of pure admission-control schemes is the *degradation* of clients' QoS. The actuator can offer "degraded" service levels in addition to the nominal service level. Rejection can be thought of as an extreme degradation point at which the client receives no service. Degradation in web servers can be accomplished by content adaptation. The approach is investigated in [5], where we survey an important category of today's e-commerce sites and present evidences of its suitability for degrading content to reduce load. In our study, GIF and JPG images alone were found to constitute, on average, more than 65% of the total bytes surveyed. In many cases, these images can be compressed without an appreciable degradation in QoS. Reducing the number of embedded objects per page (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.

To achieve degraded levels of service, the content must be 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 actuator simply prepends the desired tree name to each requested URL at run-time causing the request to be served from a particular tree. 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. The different content trees correspond to different (discrete) levels of quality that the web service offers to its clients.

In general, let's consider an actuator with M discrete service levels (e.g., content trees). These levels are numbered $1, \ldots, M$ from lowest quality to highest quality. The level 0 is added to denote the special case of request rejection. The actuator accepts as input the control variable m in the range $[0, M]$ and translates it into the fraction of clients to be served at each service level.

In general, m is a fractional number composed of an integral part I and a fraction F, such that $m = I + F$. We let the two nearest integers to m (namely, I and $I+1$, where $I < m < I+1$) determine the two most appropriate service levels at which clients must be served under the given load conditions. The fractional part F determines the fraction of clients served at each of the two levels. In effect, m is interpreted to mean that a fraction $1 - F$ of clients must be served at level I, and a fraction F at level $I + 1$. This specification constrains only the total fraction of clients to be served at each level without dictating *which* clients they should be. The latter is a separate policy that may operate within the confines of the former. For example, if all clients are equal, upon receipt of a request, a pseudo-random value $N \in [0, 1]$ may be computed by hashing the received client's id (e.g., its IP address) into a number in the range [0, 1]. If $N < F$ the request is served from tree $I + 1$. Otherwise, it is served from tree I. A "good" hashing function will map client IDs to the target range in a uniformly distributed fashion. The policy ensures that for a given m the quality level seen by each client is consistent and depends on the client's identity. Figure 2 shows how a given value of m determines both the trees from which requests are served and the fraction of requests served from each tree.

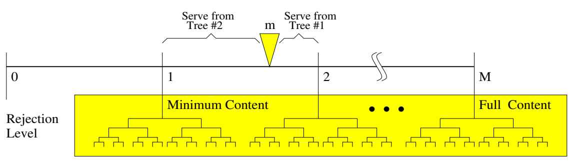


Figure 2. The Degradation range

Server utilization will increase, possibly nonlinearly, when m is increased, and vice versa. At the upper extreme, $m = M$, all requests are given highest quality service. At the lower extreme, $m = 0$, all requests are rejected. The actuator changes the amount of load on a server with discrete service levels, depending on its input m.

3.2 The Monitor

The control loop manipulates the actuator input m based on feedback (from the monitor) on current server resource utilization. Most operating systems provide means for monitoring utilization, such as percentage of consumed CPU cycles, disk and network bandwidth. Unfortunately, utilization measurements in practical systems tend to be extremely noisy. One option would be to use a filter to smooth these measurements. However, the filter will introduce an additional lag that may reduce the tightness of utilization control.

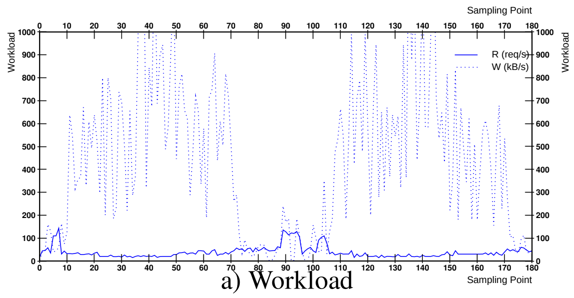
In the case where the web server deals mostly with static files (i.e., if dynamic content is statistically insignificant or served from a separate server), one can express server utilization U as a function of the served request rate R and delivered byte bandwidth W. Both of these variables can be measured very accurately in a computing server. In [2], we showed that bottleneck resource utilization is given by:

$$U = aR + bW \tag{1}$$

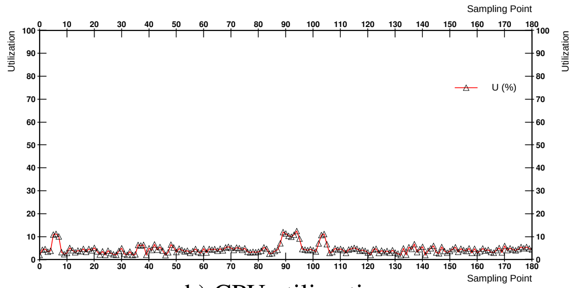
where a and b are constants which can be computed by a linear regression. The intuition behind Eq. (1) comes from the fact that the resource requirements C_i of serving a file are composed from a fixed overhead plus a variable overhead that depends on the length x_i of the file. Thus, $C_i = a + bx_i$, where a and b are constants that depend only on the type of platform used. Aggregating resource consumption over multiple requests and averaging over time we arrive at Eq. (1). The expression gives a noise-free utilization estimate which we use for feedback in the control loop of Figure 1. If only a fraction f of requests are admitted, server utilization is given by:

$$U = aRf + bW + cR(1 - f) \tag{2}$$

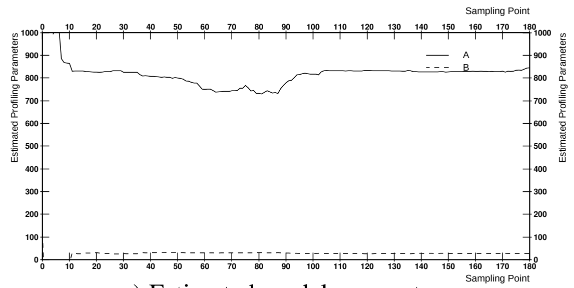
where c is the cost of rejecting one request. In Eq. (2), R is the total request rate received by the server (including requests that will be rejected), f is the fraction of admitted requests, and W is the total byte rate sent by the server. To use Eq. (2), one must compute parameters a, b, and c, and determine their sensitivity to variations in the operating conditions of the computing system. In a preliminary investigation, we compute c experimentally by instrumenting the server to reject all requests and obtaining the reciprocal of the maximum attained rejection rate. This results in $c = 0.55ms$. To estimate a and b, we fit the web server with a recursive least squares estimator that measures R, W, and U and infers the coefficients of Eq. (1). In [4], we present an evaluation of this estimator; a and b are computed using data collected in real-time on an Apache web server subjected to a varying request rate. Figure 3 illustrates



a) Workload



b) CPU utilizartion



c) Estimated model parameters

Figure 3. Online estimation of Apache server model, low load

the conversion of the resulting estimates in a representative experiment. Figure 3-a shows the workload that we applied to the server for parameter estimation purposes. In particular, it depicts the request rate R on the server and the resulting bandwidth W delivered as a function of time during the experiment. Requests for short web pages where interleaved with those for long pages to offer load points with different proportion of R to W. The horizontal axis depicts the sampling count (one sample of R, W, and U was taken every 3 seconds). Figure 3-b depicts the measured CPU utilization, U, during this experiment due to the applied workload. Figure 3-c depicts the output of the least squares estimator. In can be seen that the estimator converges to the values $a = 830\mu s$, and $b = 33\mu s/kB$. Thus, from Eq. (2), $U = 0.83Rf + 0.035W + 0.55R(1-f)$, where R is in $requsts/ms$ and W is in kB/ms. During normal operation the monitor will use the above equation to estimate utilization periodically at some period, T.

3.3 The Controller and System Model

Control theory offers analytic techniques for closing the feedback loop from monitor to actuator in a way that achieves performance specifications. As a first step towards system utilization control, we use a digital approximation of a linear continuous PI controller given by the equation $G(s) = K_p(1 + K_i/s)$. The controlled software system (including the web server, actuator, and monitor) is modeled by a transfer function $P(s)$. In the simplest case, we assume that $P(s)$ is a static gain, p (i.e., process dynamics are negligible). The gain is linearized by obtaining the derivative of the output (utilization) with respect to the input, m. Thus, $p = dU/dm$ around the operating point (which typically is $U = 5/8$). Intuitively, the maximum gain, p_{max}, occurs when the range of m is minimum for the same range in output U, i.e., when the actuator supports only client rejection, but no intermediate service levels. In this case, $m = f$, the fraction of admitted clients, and $p_{max} = dU/df$. Let U_{accept} be the utilization that would result if all clients were admitted. If presently a fraction f of the clients is admitted, the actual utilization is $U = U_{accept}f + c(1-f)R$. Differentiating with respect to f, we obtain $p_{max} = U_{accept} - cR$. It may be more convenient to express the gain in terms of actual current utilization, U. Algebraic manipulation yields:

$$p_{max} = (U - cR)/f \tag{3}$$

Designing the controller for the maximum process gain will guarantee stability for all other gain values. While the server has additional dynamics, they are of the order of milliseconds and thus will be neglected given the range of human perception. In addition to the gain, digital sampling introduces an effective dead-time of half the sampling period, $T_d = T/2$. Under the above general assumptions, the simplified transfer function of the computing system, $P(s)$, is given by the Laplace transform:

$$P(s) = p_{max}.e^{-sT/2} \tag{4}$$

The natural frequency of oscillation of this loop, w, is obtained by setting $s = jw$ in the poles of the closed loop transfer function, where j is the imaginary unit vector. This yields:

$$p_{max}e^{-jwT/2}K_p(1 + K_i/jw) = -1 \tag{5}$$

For stable control, a gain margin G is specified as a design parameter. From Eq. (5), the closed loop gain and phase at the natural frequency of oscillation are:

$$\begin{aligned} p_{max}|e^{jwT/2}|K_p|1 + K_i/jw| &= 1/G && (6)\\ wT/2 + tan^{-1}(K_i/w) &= \pi && (7) \end{aligned}$$

Incidentally, if the damped frequency of oscillation is not far from w, successive peaks of the closed loop oscillations will have a ratio of approximately $1/G^2$. It is a common practice in industrial PI controller tuning to set the controller phase to $-\pi/6$ [17]. In this case:

$$tan^{-1}(K_i/w) = \pi/6, \tag{8}$$

Eq. (6), Eq. (7), and Eq. (8) are three equations in three unknowns, K_p, K_i, and w, two computed model parameters, p_{max} and T, and one design parameter, namely the gain margin G. They can be solved for K_p and K_i for a particular gain margin to achieve a specified transient response. The system is time-varying. Controller settings depend on process gain which changes with the incoming request rate, utilization, and fraction of admitted requests, as seen from Eq. (3). For purposes of controller tuning, utilization in Eq. (3) can be substituted by its set point value.

To evaluate the resulting control loop performance, we consider controlling the utilization of an Apache web server on the Linux operating system. The combination of Apache over Linux is representative of many web server configurations today. The experimental server platform was an AMD-based PC connected via a local area network to client machines. Several machines were used to run client software that tests the server with a synthetic workload. We used a web-load generator, called httperf [15], on the client machines to bombard the server with web requests. The server code was modified to implement the discussed control loop.

Figure 4 depicts the achieved utilization. In this experiment, the request rate on the server was increased suddenly, at $time = 0$, from zero to a rate that overloads the server. Such a sudden load change approximates a step function. It is more difficult to control than small incremental changes, thereby stress-testing the responsiveness of our control loop. The target utilization, U_t, was chosen to be just below $5/8$ (preventing minor fluctuations from exceeding the schedulable utilization bound). The actuator was using admission control. Figure 4 compares the open loop server utilization to the closed loop utilization (with gain margin values of $G = 4$ and $G = 10$). As shown in Figure 4, the controller was successful in reducing server utilization to remain successfully around the target for the duration of the experiment. Utilization control was achieved by admitting the "right" number of clients. One can observe that the exponential decay in tracking error follows a damped profile that converges quickly to the set point. A zero steady state error is achieved.

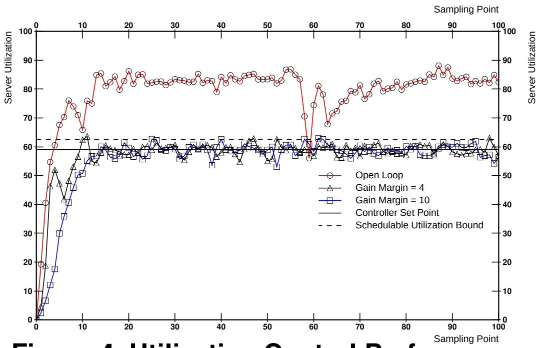


Figure 4. Utilization Control Performance

With server utilization steadily below $5/8$, we could easily verify by instrumenting the client software that individual request deadlines were met. At the expense of rejecting excess clients, all admitted clients received their requested pages within their respective timing constraints.

4 QoS Management

In this section we describe how the general architecture described in the previous section 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.

4.1 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}. Together, these specifications constitute a *throughput* guarantee. Namely, 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 may adapt 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, $U_i^* = aR_{max_i} + bW_{max_i}$. Setting aside an amount U_i^* of the bottleneck resource for virtual server i will allow it to meet its throughput guarantee. To meet service response time guatantees of individual requests, the target utilization sum $\sum_i U_i^*$ over all virtual servers residing on the same machine should be less than $5/8$ [3]. This is checked each time the adaptation software parses its configuration file. If the administrator configures a new virtual server that makes $\sum_i U_i^* > 5/8$, a capacity planning error is returned. Note that if only throughput guarantees are required, we can allow $\sum_i U_i^*$ to be higher than $5/8$ (but less than 1).

- *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 3. 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 5.

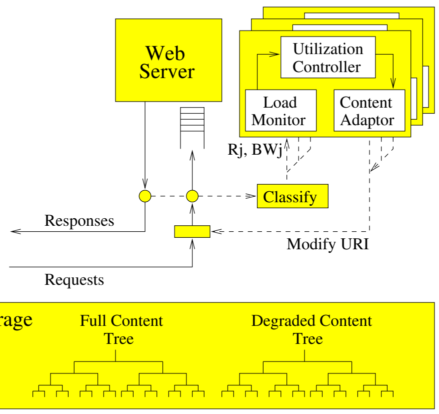


Figure 5. Architecture for Performance Isolation

4.2 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. A separate control loop is used for each class to keep its utilization around the target.
- 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 3 is applied within each control loop to compute the controller output m_j for this class.
- Service (e.g., content tree) for each class in decided by the actuator in accordance with the value of its specific m_j as described in Section 3.

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:
$U_j^* = U^* - \sum_{i<j} U_i - \sum_{l>j} U_{reject_l}$.
where $U_{reject_l} = cR_l$ is the overhead of rejecting all current requests of a lower priority class l, the overhead of rejecting a single request being c. In out current implementation, we support two priority classes, *premium* and *basic*. Premium traffic is that governed by a guaranteed QoS contract. Basic traffic has no guarantees. It is served using the leftover utilization from premium clients. Note that, since basic clients have no guarantees, the utilization restriction $\sum_i U_i^* < ln\ 2$ applies only to premium traffic.

4.3 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. This mechanism requires a simple modification to the actuators of premium traffic. Assume the controller output in the utilization control loop of a premium virtual server i is m_i. Let the controller output of the utilization control loop of best effort traffic be m_b. A request for a given premium virtual server i will be adapted by the actuator of premium traffic if $m_i > m_b$ and with the actuator of best effort traffic otherwise. Thus, the request is handled according to the higher of m_i and m_n. When the individual virtual server is overloaded while the machine as a whole is not, $m_b > m_i$. Consequently, incoming requests are served with quality determined by m_b which is higher than that warranted by m_i thus utilizing excess machine capacity. On the other hand, if the machine is overloaded, $m_b < m_i$. Consequently, the quality of content delivered by virtual server i is determined by m_i. Thus, the individual virtual server is policed not to exceed its capacity allocation. The mechanism allows smooth and informed switching between a mode of operation where an individual virtual server i is allowed to exceed its capacity allocation and a mode of operation where it is policed to capacity. We present an evaluation of these techniques in Section 6. Implementation details are discussed next.

5 Implementation

The discussed software was implemented in C for a UNIX platform. For the purpose of experimentation an Apache web server was used. In this section we give more details on software implementation, the testing environment and evaluation of adaptation software.

5.1 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.

5.2 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 the controllers in 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 implements the actuators that decide on the service levels of individual requests, prepend the right content tree name to the requested URL, and return the new 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.

5.3 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 the utilization monitor. 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.

5.4 Implementing Utilization Control

The utilization controllers are 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 m_k is computed accordingly as described in Section 3. The value of m_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 m_k for the corresponding virtual server. The function will then 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.)

6 Evaluation

In this section we present a performance evaluation of the developed software. This software was tested on multiple testbeds including a Linux platform (some test results of which were presented in Section 3), a solaris platform and an HP-UX platform. In this section we report on our HP-UX tests. In these tests, an Apache 1.3.0 web server was executed 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. To emulate a large number of web clients we used httperf [15], 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.

6.1 Baseline Performance

Figure 6 compares the performance of a server that implements our extensions to that of a regular Apache server. It plots the connection error probability versus request rate. In this experiment we generated requests for 64K images at an increasing rate. An adapted 8K version of the images was available in the degraded content tree. As shown in the figure, the traditional server suffers an increasing error rate when offered load exceeds capacity at about 160 reqs/s. In contrast, the actuator in our adaptive server switches to less resource-intensive content thus exhibiting almost no errors up to about 3 times the above rate. In general, the extent of performance improvement will depend on workload and degree of content degradation available.

6.2 Estimating Service Time

In our first experiment, we profiled the Apache server to determine the time, $T_s(x)$, it takes to serve a URL of size x. Measuring server response time was found not to be indicative of service time $T_s(x)$ because the former includes queuing time, network latency, etc. We therefore measured service time by obtaining the inverse of the maximum throughput. The idea is that if the server can serve no more than n requests per second, then, for all practical purposes, each request takes $1/n$ to serve.

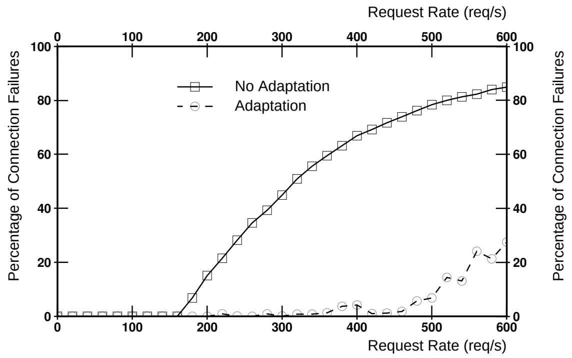


Figure 6. A Baseline Comparison

The experiment was repeated for different sizes of the requested URL. Table 1 shows the maximum throughput and the corresponding service time for each URL size.

Table 2 compares the measured service times to service times computed using the linear approximation $T_s(x) = A + Bx$ where x is URL size, $A = 1.604$, $B = 0.063$. The constant A can be thought of as the time it takes to serve a zero-size URL. The constant B is the additional service time required per KB of URL size. It can be seen that the quality of this approximation is very good for smaller URL sizes, but deteriorates significantly as URL size increases. The reason is that the service time computed from the linear expression approximates the *end-system's* service time. When the retrieved URLs are small the maximum request rate is determined by the end-system's bandwidth (including both CPU and disk access) making the approximation accurate. As URL size increases, the bottleneck shifts from the end-system to the network. Since the end-system is no longer the bottleneck, the estimated service time falls below the observed service time dominated by that of the bottleneck resource.

In order to model service time more accurately we use a composition of two linear approximations, one estimates service time if the end-system is the bottleneck and the other estimates service time if network bandwidth is the bottleneck. While the former is given as before by $T_s(x) = 1.604 + 0.063x$, Table 2 suggests that the latter be given by $T_s(x) = 0.093x$, which is equivalent to stating that the network saturates at a transfer rate of approximately $86Mb/s$. We then take the larger of the two service times to account for the bottleneck resource. Thus, the combined expression for T_s is:

$$T_s(x) = \max\{1.604 + 0.063x, 0.093x\}$$

The accuracy of the above approximation is shown in Table 3. We can see that the approximation is accurate

URL Size (KB)	Max Rate (reqs/s)	T_s ms/req
1	586	1.706
2	578	1.73
4	538	1.858
8	482	2.075
16	383	2.611
32	301	3.322
64	169	5.917
128	85	11.76
256	42	23.81
512	21	47.62

Table 1. Service time vs. request size

URL Size (KB)	Measured T_s	$A + Bx$	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%
128	11.76	9.668	-17.8%
256	23.81	17.73	-25.5%
512	47.62	33.86	-28.9%

Table 2. Approximating service time

over most of the range of URL sizes. The larger error at size 32K is due to particulars of the OS implementation. It appears that HP-UX is optimized for long TCP transfers, making CPU service time increase sublinearly with transfer size thus falling below the linear estimate.

The total service time T_N of N requests is $\sum_{1 \leq i \leq N} T_{s_i}(x_i)$, where x_i is the requested URL size in the ith request, and $T_{s_i}(x_i)$ is the service time of that request. Substituting for $T_{s_i}(x_i)$ we get: $T_N = \max\{1.604N + 0.063 \sum_{1 \leq i \leq N} x_i, 0.093 \sum_{1 \leq i \leq N} x_i\}$ where $\sum_{1 \leq i \leq N} x_i$ is the total bytes requested. Let us denote it by S. Thus, $T_N = \max\{1.604N + 0.063S, 0.093S\}$. If N requests were served by the server within some time interval T, system utilization is $U = T_N/T = \max\{1.604N/T + 0.063S/T, 0.093S/T\}$. Note in this expression that N/T is the observed request rate R, and S/T is the delivered bandwidth BW. Thus:

$$U = \max\{1.604R + 0.063BW, 0.093BW\} \tag{9}$$

In practice, requests for URLs above 64KB will constitute only a small fraction of all requests on the server. Thus, it is probably safe to assume that the first term will usually dominate in the above expression. This reduces it to the linear approximation $U = aR + bBW$ we described earlier, where $a = 1.604$ and $b = 0.063$. Similar results where obtained via least squares estimation.

Note that in Equation (9), U is expressed on a scale from 0 to 1, R is expressed in req/ms and BW is expressed in ms/kB. It is more natural to expressed R in $reqs/s$, and BW in Mb/s. After the appropriate conversion of units we get the more natural expression:

$$U = 0.001604\, R(reqs/s) + 0.007875\, BW(Mb/s) \tag{10}$$

The a and b parameters are robust to changes in workload (e.g., changes in request rate and requested URL size). However, since they represent, in part, the computational overhead of TCP/IP connections, these parameters might change depending on the average number of retransmissions and the number of segments required to send a given amount of bytes. Thus, for example, the a and b parameters might be smaller for clients accessing the server locally via a high bandwidth LAN and larger for clients accessing the server across a congested or lossy wide area network. In the preceding experiments clients were accessing the server via a LAN. We have not experimented with server access over a wide area network to estimate parameter robustness under these conditions. We expect, however, that a and b will remain stable enough in the face of gradual client population changes for the automated profiling to update them in a timely and accurate manner.

6.3 Measuring Response Time

In our experiments, we found that Apache server response time when measured across a fast network (or from a client residing on the same machine with the server) has two important properties. First, it is essentially bi-modal. It remains low until the server becomes overloaded, at which time it increases dramatically. Second, the "knee" in response time seen at overload is roughly equal to the product of service time, T_s, and the maximum length of the listen queue configured for the server. For example, Figure 7 plots server response

URL Size (KB)	Measured T_s	$A + Bx$	Error
1	1.706	1.667	-2.3%
2	1.730	1.730	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	5.952	0.6%
128	11.76	11.90	1.2%
256	23.81	23.81	0%
512	47.62	47.62	0%

Table 3. Approximating service time

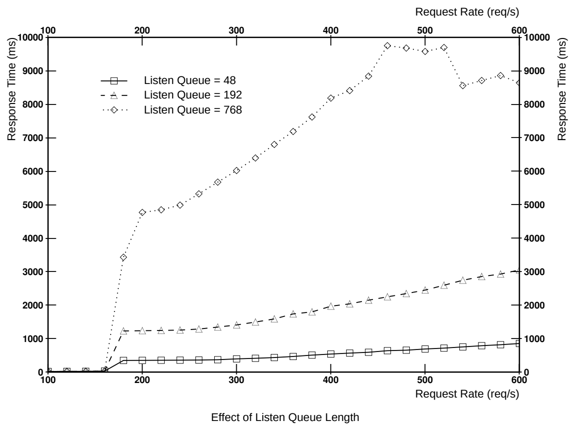


Figure 7. Server response time for different listen queue lengths

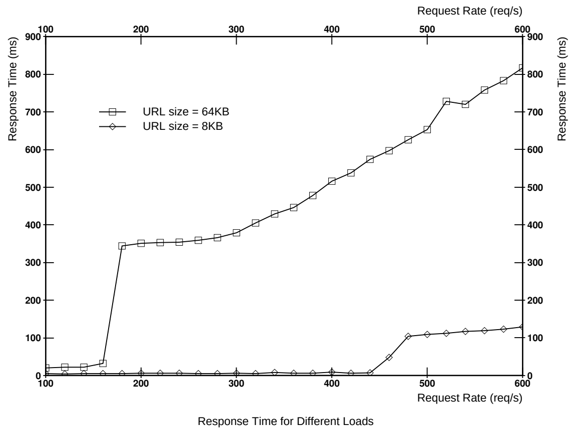


Figure 8. Server response time for different URL sizes

time versus request rate when the listen queue was configured for maximum length of 48, 192, and 768, respectively. In this experiment all requests were for URLs of size 64KB. The sudden increase in server response time when the request rate increases beyond 160 reqs/s makes a clear overload indicator. Figure 8 plots response time versus request rate when the URL size is changed. In this experiment the listen queue was configured for a maximum length of 48. The requested URL size was 8KB in one experiment, and 64KB in another. As before a clear rise in response time was observed when server capacity was exceeded. Note that reducing server utilization below $ln\ 2$ guarantees that the server will operate below the overload threshold.

6.4 Adaptation at Overload

Content adaptation reduces the load on the server thereby avoiding connection failures. As request rate increases on the server a threshold, $R_{degrade}$, is reached where content has to be degraded in order to prevent overload. As request rate continues to increase beyond $R_{degrade}$, more clients must be degraded until, eventually, a point R_{reject} is reached where no further degradation is possible. If request rate increases beyond R_{reject} some clients must be rejected to prevent indiscriminate connection failures. An actuator with no support for degradation exhibits connection failures or client rejection starting at rate $R_{degrade}$, while an actuator with adaptive content will continue to serve all requests up to the higher rate R_{reject}. As shown in Fig 6, we conducted an experiment where the request rate on the Apache server was increased for URLs of size 64KB. An adapted 8K version of the same URL was used for degraded content. In this experiment we found, approximately, that $R_{degrade} = 160$ and $R_{reject} = 460$. The ratio $R_{reject}/R_{degrade}$ is the the maximum sustainable request rate of an adaptive server as compared to the maximum sustainable request rate of a non-adaptive server. The value $R_{reject}/R_{degrade} - 1$ is the net improvement in the maximum sustainable request rate due to adaptation. This improvement depends on the requested URL size. Figure 9 plots the net improvement (in percents) versus the average requested URL size, x. The degraded content, in all cases, was 8 times smaller in size than the full-length content, but the required number of server accesses was the same. The percentage improvement in maximum sustainable rate is illustrated both when the accessed URL is a static file of size x, and when it is a CGI script returning a URL of size x. In the latter case a static memory buffer of the specified size was returned by the script with no initialization and no meaningful content. The CGI scripts were written in C. Results for interpreted Perl scripts were slightly lower (not shown in Figure).

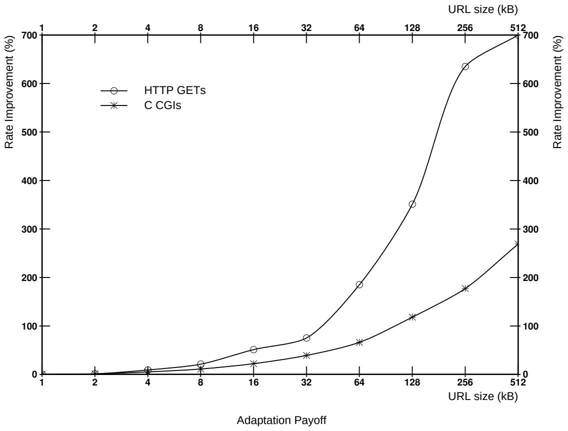


Figure 9. Adaptation payoff: increase in maximum sustainable rate

It can be seen that the percentage improvement in sustainable rate decreases as the requested URL size decreases. This is because the smaller the requested URL the more dominated is service time by the fixed size-independent processing overhead, rather than the size-dependent data transfer cost. The rate improvement achieved by compressing the URLs is relatively insignificant (less than 100%) for URL sizes below 32K. Thus, content dominated by smaller objects should be degraded by reducing the number of embedded objects per page, rather than reducing the bytes per object. Also note that CGI scripts are not amenable to degradation by reducing the size of generated content. The fixed overhead involved in invoking the script is so great that the additional data-size dependent costs are insignificant unless the returned data volume is substantial. We therefore suggest that dynamic content be degraded by converting it to static whenever possible.

6.5 Rejection Overhead

As a last resort, our actuator rejects clients to control utilization when no further degradation is possible. The server can either silently close a client's connection, or return an error message such as "Service not Available". In either case some processing occurs on the end-system before the request is rejected (e.g., protocol processing). To quantify the amount of time spent in processing an eventually rejected request, 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 reqs/s, which is the maximum rate at which rejection can be processed. The time wasted on each rejected request (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 zero-size URL (denoted by constant A in Section 6.1). The difference is believed to be due to file system access associated with serving the URL. It appears that this difference is not substantial. More than one millisecond of processing time is wasted on each request even if it is rejected. Request classification and rejection should thus be done at the earliest point possible upon request reception in order to conserve end-system's resources. One suitable place for this mechanism is at the bottom of the protocol stack in the operating system's communication subsystem. The difficulty in performing classification at the bottom of the protocol stack lies in the necessity to violate protocol boundaries and peek into headers of higher-level protocols such as HTTP.

It is interesting to compare the aforementioned rejection overhead to the overhead wasted on each failed connection in a server that does not support rejection. Let us denote it by T_f. To compute T_f, consider the Figure 10 which depicts the delivered bandwidth in a regular (unmodified) Apache server subjected to an increasing request rate. The maximum delivered bandwidth (of about 84Mb/s) occurs at the overload threshold (at rate 160 reqs/s). Onset of overload indicates that the server is unable to serve successfully more than 160 reqs/s. Substituting in Equation (10), the following equation holds:

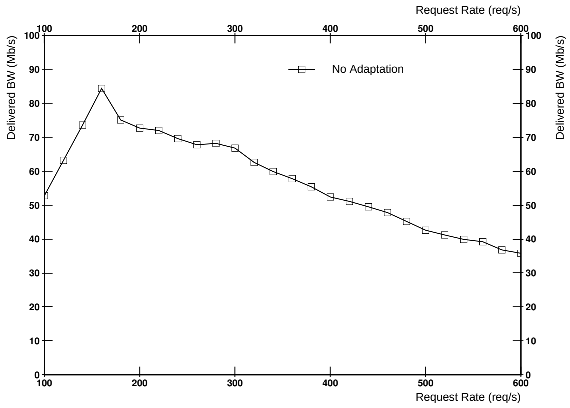


Figure 10. Server Underutilization

$$160a + 84b = U_{max} \quad (11)$$

where U_{max} is the maximum server utilization at overload. As overload continues to increase, the delivered bandwidth declines to only 36 MB/s at rate of 600 reqs/s.

For the worst-case estimate of T_f, assume that the decline in bandwidth is attributed solely to the overhead of handling failed requests. Since the server cannot serve more than 160 requests successfully out of the 600 it receives every second, the number of failed requests is at least 440 reqs/s. The following equation holds:

$$160a + 36b + 440T_f = U_{max} \qquad (12)$$

By subtracting Equation (11) from Equation (12), solving for T_f, then substituting for the value of b (as determined in Section 6.1, $b = 0.007875s/Mb$), we get $T_f = 0.86ms$ in the worst case. Note that this number is less than the 1.1 ms request rejection overhead.

The implications of the above are interesting. User-level admission control mechanisms trivially require that all requests be seen by the actuator (so that an admission control decision can be made for each). This implies that each request, whether it ends up being rejected or not, will have to consume platform resources up to the point where it leaves the kernel and is inspected by the server or middleware. As shown above, each request rejected by the server consumes $1.1ms$, on average.

A best-effort server, on the other hand, will serve requests in a FIFO order. As a result, under overload, its socket's listen queue will overflow in the kernel. Many client connections will time out and fail early in the OS before being seen by the server. As shown, a request failed in the kernel consumes only $0.86ms$. As a result, the resources wasted per failed request are less (about 22% less on our platform). The remaining capacity available to requests that do get through is therefore higher in a best effort server. Thus, while a user-level admission control mechanism will improve the average response time of requests that are not rejected, it will necessarily increase the average rejection rate over the failure rate of a server with no such mechanism. This fact motivates using user-level adaptation instead of rejection as a way to control server overload whenever possible. Content adaptation is especially suited for alleviating light to moderate overload conditions when the server has enough capacity to serve a fraction of, but not all, requests. In cases of severe overload, the server may suffer the receive livelock problem which may preclude serving any requests at all. Methods for resolving the receive livelock problem such as kernel level classification and admission control are beyond the scope of this paper.

6.6 Performance Isolation

We described a performance isolation mechanism that allows creating multiple adaptive 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 11 demonstrates these features. In this experiment all requests were for 32KB URLs.[1] A background best-effort load of 300 reqs/s was applied to overload the machine (see Table 1 for maximum sustainable rate of 32KB requests). In addition, two adaptive virtual servers, V_1 and V_2, were configured. Server V_1 was configured for a maximum guaranteed bandwidth of 13 Mb/s, and a maximum guaranteed rate of 50 reqs/s. During the experiment, a constant load of 50 reqs/s was applied to that server requiring a bandwidth of 12.8 Mb/s, i.e., just within the allocated server capacity (note that bandwidth in Mb/s is 32KB/req times 8 b/B times 50 reqs/s). Server V_2 was configured for a maximum guaranteed bandwidth of 27 Mb/s, and a maximum guaranteed rate of 100 reqs/s. The load on server V_2 was increased gradually from 0 to 100 reqs/sec, giving rise to a bandwidth requirement of up to 25.6 Mb/s, which is also within server capacity. It is important to note that while each virtual server in isolation was loaded within its individual capacity limit, the aggregate load on the machine (including non-guaranteed traffic) was well above the overload threshold because of best-effort load.. Figure 11 depicts the offered load on each virtual server (in terms of bandwidth in Mb/s assuming no content degradation), as well as the actual bandwidth delivered by each server. Both are plotted versus the aggregate request rate. For clarity, the best-effort load is not shown. It can be seen that the actual bandwidth delivered follows closely the offered load on each virtual server. Thus, despite server overload, virtual servers V_1 and V_2 achieve their performance guarantees and suffer no content degradation. Furthermore, variations in load on virtual server V_1 do not affect virtual server V_2. 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 experiment using a regular Apache server that does not use our adaptation extensions. As before, a best-effort load of 300 reqs/s was applied in addition to a 50 reqs/s load on server V_1 and an increasing 0 to 100 reqs/s load on server V_2. Figure 12 depicts the results of this experiment. It can be

[1] In a real-life situation the workload is likely to be less severe.

seen that the delivered bandwidth of both virtual servers 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 server V_1 results in a decrease in delivered bandwidth of server V_2. No performance isolation is observed. The comparison of Figure 11 and Figure 12 illustrates the advantage of the developed adaptation software.

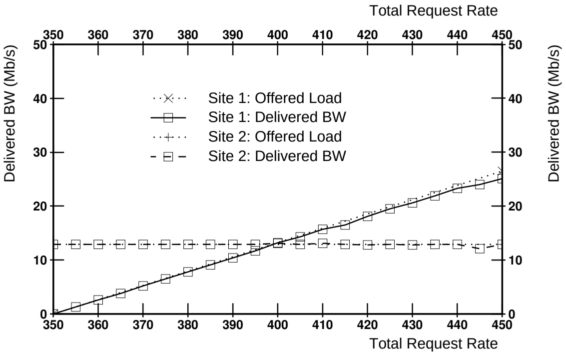


Figure 11. Performance Isolation in Adaptive Server

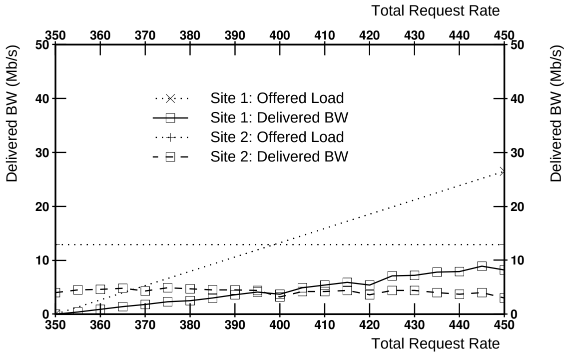


Figure 12. Regular Apache Performance

6.7 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 13 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.

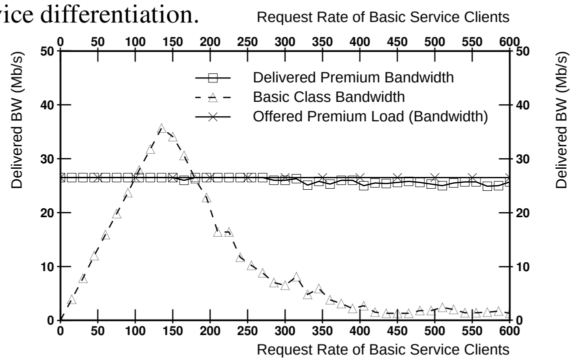


Figure 13. Service Differentiation

6.8 Policing vs. 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 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. 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 overruns 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 operates within its capacity limit at all times, is shown to deliver its offered load without 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 14 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 reqs/s. Server V_2 is configured for maximum bandwidth of 27Mb/s and maximum request rate of 100 reqs/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 reqs/s. The aggregate rate of both servers 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 reqs/s (at which the aggregate rate is 240 reqs/s in Figure 14). This is to be compared with its guaranteed maximum bandwidth of 27Mb/s and maximum request rate of 100 reqs/s. Server V_1 remains unaffected, since the excess capacity sharing mechanism ensures performance isolation.

The experiment is repeated with a background load of 100 reqs/s. It can be seen that V_1 is made to deliver exactly its maximum guaranteed bandwidth (27Mb/s) when its rate reaches the maximum guaranteed rate (100reqs/s). This is equivalent to traffic policing, except that in adaptive virtual servers it is achieved via content degradation. The bandwidth consumed by V_1 drops below its guarantees value when the maximum rate guarantee is violated by the community of clients. This is to ensure that the total system capacity utilization of that virtual server remains constant. Similarly, the server is allowed to deliver more than its maximum guaranteed bandwidth when its rate is below the maximum guaranteed rate. This is an optimization that makes use of the capacity allocated to the server to deliver more bandwidth when the request rate hasn't reached its maximum value. Again, server V_2 is not affected due to correct performance isolation.

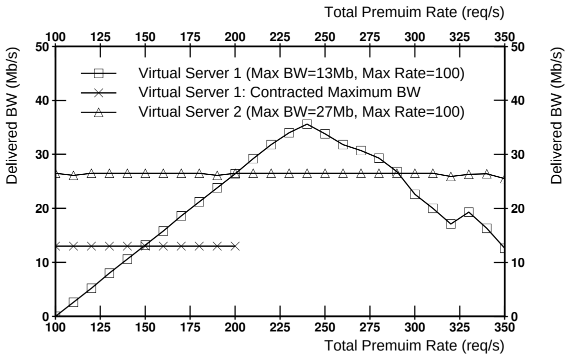


Figure 14. Excess Capacity Sharing (Low Background Load)

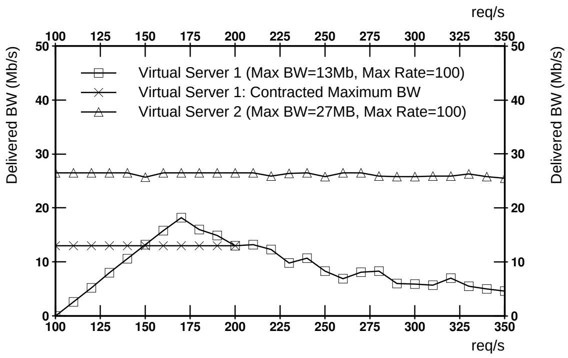


Figure 15. Excess Capacity Sharing (High Background Load)

7 Conclusions and Future Work

In this paper, we demonstrated the application of control theory to Internet server performance control. We presented a QoS-management architecture that relies on adapting the delivered content to control server utilization. Unlike contemporary non-adaptive servers, and unlike servers that implement "binary" — accept or reject — admission control, content adaptation enables a server to provide a smooth range of client degradation, thereby handling overload gracefully. We proposed the design and implementation of a utilization control loop that regulates the extent of degradation (in the service level and number of clients) so as to satisfy a prespecified 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, excess-capacitysharing, and QoS guarantees. The mechanisms described in this paper are largely independent of workload assumptions, and can be easily applied to different platforms 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 its deployment.

We have shown that Internet servers can be modeled by a time-varying linear transfer function for purposes of performance control, and demonstrated how feedback control can be applied to ensure meeting service timing constraints. All measurements in this paper were made on an experimental platform running the popular Apache web server which we have modified to incorporate the proposed control loop. The paper also provides a proof of concept of the utility of control theory to computing systems.

There are several remaining issues and challenges that warrant further research. Handling and adapting dynamic content is an 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 ofcontent is affordable for the typical size of a web site. In video servers, however, an important issue to investigate is scalable video encoding schemes that avoid storing multiple copies of the content. We also need appropriate content authoring and management tools to preprocess web content in a way that preserves enough information, yet consumes a minimal amount of resources.

From the perspective of control theory there are several additional issues that we would like to address. For example, how to determine the set point and parameters of a controller in order to guarantee that utilization is maintained below the schedulable bound a certain percent of the time (given statistical input load characteristics such as standard deviation, or maximum burst), how to model non-linearities peculiar to computing systems? How can these nonlinearities be accounted for in controller tuning? How efficient are adaptive control and robust control techniques in dealing with parameter variations and load uncertainties (e.g., when the statistical model of the load is unknown or non-stationary)? Can automatic identification and estimation techniques be applied to model servers, software sensors, and actuators? How to implement control-theoretical resource management in the operating system? Examples, theoretical foundations, experimental evidence, and practical experience are needed in applying feedback performance control to different computing systems. This is an important focus of our current research.

Acknowledgements

The authors would like to thank Jack Stankovic, Sang Son, and Gang Tao for their useful advice and inspiring discussions.

References

[1] T. Abdelzaher and K. G. Shin. QoS provisioning with qContracts in web and multimedia servers. In *IEEE Real-Time Systems Symposium*, pages 44–53, Phoenix, Arizona, December 1999.

[2] T. F. Abdelzaher. *QoS-Adaptation in Real-Time Systems*. PhD thesis, University of Michigan, Ann Arbor, Michigan, August 1999.

[3] T. F. Abdelzaher. The utilization bound for schedulability of aperiodic tasks. In *Real-Time Systems Symposium*, Orlando, Florida, December 2000.

[4] T. F. Abdelzaher. An automated profiling subsystem for qos-aware services. In *Real-Time Technology and Applications Symposium*, Washington, D.C., June 2000 (in press).

[5] T. F. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. In *International World Wide Web Conference*, Toronto, Canada, May 1999.

[6] T. F. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, London, UK, June 1999.

[7] J. Almedia, M. Dabu, A. Manikntty, and P. Cao. Providing differentiated levels of service in web content hosting. In *First Workshop on Internet Server Performance*, Madison, Wisconsin, June 1998.

[8] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Deadline monotonic scheduling theory. In *Proceedings of 18th IFAC Workshop on Real Time Programming*, pages 55–60, Bruges, Belgium, June 1992.

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

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. In *IETF RFC 2475*, December 1998.

[11] R. Braden, D. Clark, and S. Shenker. Integrated services in the Internet architecture: An overview. *Request for Comments RFC 1633*, July 1994. Xerox PARC.

[12] L. Eggert and J. Heidemann. Application-level differentiated services for web servers. *World Wide Web Journal*, 3(2):133–142, March 1999.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of ACM*, 20(1):46–61, 1973.

[14] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. The design and evaluation of a feedback control edf scheduling algorithm. In *IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

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

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

[17] F. G. Shinskey. *Process control systems: application, design, and tuning*. McGraw-Hil, New York, 4th edition edition, 1996.