Web2K: Bringing QoS to Web Servers

Preeti Bhoj, Srinivas Ramanathan, Sharad Singhal
Internet Systems and Applications Laboratory
HP Laboratories Palo Alto
HPL-2000-61
May, 2000

Web services, quality of service, service management

The recent outages and brown-outs at several large Internet portal sites (trading, auction, etc.) and the associated loss in revenues highlight the central role that web-based services play in the E-commerce era. Unpredictable demands posed by sudden surges of requests from users on the open Internet have made it extremely difficult for web site operators to offer predictable quality of service (QoS). In order to provide stable service to users considered important by the site operator under these conditions, it is imperative that systems handle overloads gracefully.

In this paper, we describe the design and implementation of a QoS-enabled web server called the Web2K server. The Web2K server enables web site operators to optimize the usage of their web sites by prioritizing among requests from different classes of users. Under overload conditions, predictive queue controllers incorporated in the Web2K server enable it to selectively deny service to some percentage of requests based on the associated user classification. Unlike current web servers, the Web2K server also maintains stable response times under overload using a novel, traffic-aware request processing model. A primary concern of operators while deciding to deploy QoS solutions relates to the overall impact of QoS provisioning on the capacity of their servers. To address this issue we present a comprehensive performance study that quantifies the overheads of a Web2K server and demonstrates its significant benefits.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 2000

1 Introduction

1.1 Motivation

The universal availability and easy to use interface provided by web browsers, together with the ever-growing Internet user base is motivating enterprises and service operators to migrate mission-critical commerce services to the Web. On-line banking, reservations, stock trading, product merchandising, to name a few, are services that are being offered via Web front-ends. As this trend continues into the 21st century, quality of service (QoS) considerations are expected to gain tremendous importance. For instance, dropped connections and slow responses from a web site frustrate users and result in lost revenues for the web site operator.

While predicting demand and increasing the capacity of the servers in advance can ensure that a web site meets user expectations of service quality, accurate prediction of demand is difficult, and over-provisioning capacity to handle short periods of overload is usually expensive. An alternative approach is to augment the web servers with QoS capabilities, so that when a server is overloaded, it has the ability to prioritize among incoming requests and to selectively process requests considered important by the site operator. This approach provides a cost-effective way of managing capacity for mission-critical commerce services.

1.2 Related Work

Many efforts [1,2] have been directed at measuring the behavior of web servers under overload. To improve the performance of web servers, Pai et al. [3] have described the design of a Flash web server. Using an asymmetric multi-process event-driven architecture, the Flash server ensures that its threads and processes are never blocked. Effective multiplexing of its resources among incoming requests ensures significant performance benefits. We contrast the flash server design with Web2K server design in section 1.3. Another web server optimization has been proposed by Crovella et al. [4]. In this approach, a web server schedules the order in which requests are processed so that requests for small static pages are provided preferential treatment as compared to other requests. A key limitation of this approach is the expectation that a web server is aware of the processing required by all the requests it receives.

Several efforts have also studied enhancements in server operating systems to enable QoS support [5,6]. Bhatti and Friedrich [7] have highlighted the importance of QoS support in web servers and have proposed an architecture to support tiered web services. Jin and Salehi [8] have described a QOS-aware middleware implementation approach that enables an unmodified web server to transparently deliver guaranteed performance in a shared-hosting environment. Abdelzaher and Bhatti [9] have described how web servers can selectively serve lower quality content under overload conditions to optimize server utilization. They use industrial control techniques to determine the proportion of requests from best effort users that are provided degraded content to ensure that high priority users do not suffer degradation.

Under extreme overloads, a web server has to perform admission control to maintain the quality of service for existing users. Cherkasova and Phaal have proposed that web servers perform admission control on incoming user sessions rather than on each incoming request [10]. A *session* is a series of requests issued by a user within a pre-defined, period of time (e.g., 30 minutes). Session-based admission control is important because many web-based services are transactional in nature, and consist of many requests to the web service. For example, for a banking transaction, a user

may need to access the bank's home page; sign on; check account balances; pay a bill, and finally, sign off. Under overload, session-based admission control schemes allow existing user sessions to continue, while new sessions are either redirected to other web sites, or denied access to the service. Performance results indicate that a web server that implements session-based admission control is able to support a significantly higher number of completed user sessions than a server that uses request-based admission control. Commercial products such as Hewlett-Packard's WebQoS [11] and IBM's Websphere now include some of these QoS capabilities.

1.3 Web2K Server Overview

In this paper, we describe Web2K, a web server that enables web site operators to offer differential performance to their users. Since it is neither always feasible nor practical to expect web site operators to radically modify the operating systems or the web application servers that they currently use to host their web sites (many operating systems and the web application server software use proprietary architectures and can only be modified by their respective vendors), *the Web2K server does not rely on any operating system support to enable differential QoS for users*. Instead, the Web2K server uses a middleware approach similar to that described in [8,11] to enable QoS capabilities without requiring any modifications to the operating system or the web application server software. This middleware is in the form of a dynamically linked library that is transparently added to a web server. Therefore, a Web2K server is a web server augmented with QoS-aware middleware. The middleware functions as an intermediary between the web server and the TCP/IP stack. Based on operator-specified prioritization criteria, the middleware classifies incoming requests as belonging either to a *premium* class or to a *basic* class[1]. The assignment can be based on a number of criteria such as the identity of the user issuing the request, or the destination of the request. For example, users who are active traders can be classified as premium users by an electronic trading web site, or a user can be classified as premium when she is about to buy an item from the a shopping site. Instead of passing incoming requests to the web server in a first-come-first-serve manner, the middleware prioritizes requests so that premium class requests are serviced first by the web server. When it detects overload, the Web2K server performs admission control to avoid over-committing its resources following a similar approach to [10].

The key innovations in the Web2K server design relative to prior work are:

- ***Traffic-aware processing of incoming requests****:* Typically, as a web server reaches overload, the number of requests waiting for service in the TCP listen queue grows to a point when the server's TCP/IP stack begins to drop new connection requests. Dropped connections result in TCP timeouts at the client. Since TCP timeouts last several seconds, users start seeing slow response at overloads. Moreover, the TCP/IP stack does not differentiate between premium and basic class requests when it drops them. To avoid large response times and to ensure prioritization among classes of requests, the Web2K server implements a novel traffic-aware processing scheme. The main benefits of this scheme are:

[1] *To simplify the explanation, the Web2K server design is explained assuming that the server only supports two qualities of service – namely, the premium and the basic QoS class. It is fairly straightforward to extend the Web2K server to handle multiple QoS classes.*

- A Web2K server minimizes the chance of new connection requests being dropped from the TCP listen queue (which is maintained in the server's TCP/IP stack) under overload by rapidly draining the listen queue. This ensures that TCP timeouts are rare and that users of the web site see low response times both for requests that are serviced as well as for requests that are rejected.
- A Web2K server maintains an internal queue of outstanding requests whose length gives an indication of server overload. This provides the Web2K server a mechanism to quickly decide if it should process an incoming request or deny service to it, thereby reducing response times and optimizing server resources.
- Since new requests are no longer dropped from the TCP listen queue, the Web2K server is able to use the priority of a request to determine whether it must be processed even under overload. In this way, the Web2K server is able to ensure that a majority of incoming premium class requests are serviced at all times.

Although web server architectures such as the Flash server design [3] have explored different request scheduling approaches (mainly to ensure overlap of CPU, disk, and network processing), to the best of our knowledge, the prior literature has not explored mechanisms of ensuring QoS protection by ensuring that TCP connections that are dropped by the server's TCP/IP stack without being serviced mainly belong to the lower QoS class(es).

- ***Early prioritization of requests***: Existing approaches [7,11] for QoS-enabled web services classify requests sequentially, offering prioritization only when scheduling requests for processing by the web server. In our experiments we have observed that under certain conditions classification can itself become a bottleneck. The Web2K server therefore uses past history of user sessions to prioritize even the classification of requests. By doing so, the Web2K server ensures that premium users receive service with low and predictable response times even when classification is a bottleneck.
- ***Predictive admission control:*** To ensure that the server resources are never over-allocated, the Web2K server implements admission control, using the number of requests that are pending for service in each class as a metric. To limit the number of pending requests in each class, earlier approaches [7,11] use simple threshold-based admission control policies. Since these approaches are susceptible to sudden changes in traffic patterns or server processing times, the Web2K server uses a predictive approach for admission control. In this approach, the Web2K server uses past history to predict the server's input rates and servicing rates and determines the percentage of incoming requests in each class that can be admitted. In order to exploit the benefits of session-based admission control [10], the Web2K server provides priority to requests belonging to existing sessions over requests for new sessions.
- ***Simple, yet flexible configuration model:*** Ease of configuration and use are critical for any new technology to be widely used. However, typical queue and resource management models require the operator to set parameters such as queue thresholds and configure actions that need to take place when these thresholds are violated. Another attractive feature of a Web2K server is its intuitive and simple configuration and use model. Operators can use the Web2K server's user interface to set preferences for end-user visible behaviors. These preferences are automatically translated into settings for the Web2K server's tunable parameters. Real-time displays of statistics are provided as feedback to the operators, so that they can further adjust the preferences to suit their needs.

Earlier work [6] on QoS enabled web services has focussed on differentiation between classes of users. A primary concern of operators while deciding to deploy QoS solutions relates to the overall impact of QoS provisioning on the capacity of their servers. To address this concern, this paper presents a comprehensive performance study that quantifies the overheads of a Web2K server relative to an unmodified web server, besides demonstrating its benefits. This study uses content from a popular online banking site and workload that is representative of typical user sessions to emulate a realistic environment. To represent a fair comparison the performance study chooses the best configuration settings for the unmodified web server in this environment. The effectiveness of the Web2K server under different operating conditions (with different system bottlenecks) is demonstrated using scenarios that involve accesses to static and dynamic content. We have studied the effectiveness of the Web2K server design for the Apache and Netscape web servers in a HP-UX environment. Experimental results using the Apache web server are reported in this paper. Extension of the Web2K server to handle other environments – e.g., Windows and Linux operating systems, Zeus and IIS web servers – is a subject for future work.

The rest of this paper is organized as follows: Section 2 describes the overall architecture of a Web2K server. Section 3 discusses the details of one of the main components of Web2K server – namely, the *acceptor*. Section 4 presents the performance results and Section 5 outlines the simple usage model we propose for Web2K server. Finally, Section 6 summarizes this work, and outlines opportunities for future work.

2 System Architecture

Figure 1 depicts the architecture of a Web2K server that uses the middleware approach described in [8,11]. A Web2K server comprises of an unmodified web server (referred to as a *vanilla web server* in this paper) that is augmented with a QoS-aware middleware layer. The middleware layer enables the Web2K server to offer differential qualities of service to users, without requiring any changes to the vanilla web server. In order to do so, the middleware layer intercepts a web server's TCP socket calls, interfaces directly with the server's TCP/IP stack, receives and prioritizes requests, and forwards them to the web server for processing.

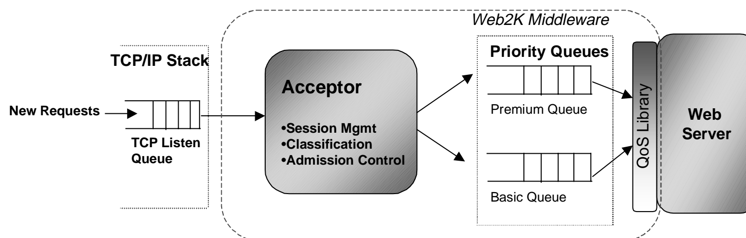


Figure 1: Architecture of a Web2K Server

The main components of the middleware layer are:

- ***Acceptor:*** The acceptor is the component of the middleware layer that interfaces with the server's TCP/IP stack. The acceptor listens on the web server's TCP port (usually 80 or 443) and is responsible for receiving new

connections. For each request that it receives on an incoming connection, the acceptor performs *session management* to determine whether the request is part of an existing session or not. As mentioned earlier, a session is a series of requests issued by a user within a pre-defined time period. To maintain session information, the middleware layer of the Web2K server introduces a special cookie into the header of every HTTP response that it returns to the client. These cookies enable the acceptor to differentiate requests belonging to existing sessions from those belonging to new sessions.

Following session management, the acceptor is responsible for *classification* of requests. A class (i.e., premium or basic) is associated with a session. All requests in a session share the same class. To determine the class of a session, the acceptor uses a number of classification criteria such as the identity of the user issuing the session, the location (IP address) of the user, etc. User sessions may also be upgraded[2] to the premium class depending on the context of the underlying transactions. For example, a session may be upgraded when the user decides to purchase an item from the web site. Back-end applications (e.g., a banking application logic module) can also convey classification information to the Web2K server through a well-defined API. Following classification, the acceptor decides whether or not the request should be admitted for service *(admission control)*. In the event the request has to be denied service, the acceptor is responsible for returning a reject response to the client indicating that the request could not be serviced. All other requests are queued by the acceptor in one of two *priority queues* (premium and basic queues shown in Figure 1).

- ***QoS Library:*** The Web2K middleware layer interfaces with the web server through the QoS library as shown in Figure 1. Through a process called "*qosification*", at the time of installation, the Web2K server maps TCP socket calls made by the web server to corresponding calls to a library of QoS-enabled functions. For each operation that the web server performs on a TCP socket (such as accept, read, send, close, etc.), the QoS library has a corresponding internal function. These functions force the web server to receive new HTTP requests from the priority queues, rather than via the TCP socket library.

3 Acceptor Design

In this section, we discuss the detailed design of the Web2K acceptor.

3.1 Traffic-Aware Request Processing

Many TCP-based applications accept and process incoming connections sequentially – i.e., an application first accepts a connection request from the TCP listen queue, processes the accepted connection, and then proceeds to pick up the next connection. Most web servers (e.g., Apache and Netscape) are no different. Even when a web server has multiple processing threads, each accepting connections from the TCP listen queue, the ratio of connections being processed to those being accepted remains one. Even with approaches such as the Flash server architecture [3], wherein the web server processes multiple requests in parallel, the ratio of connections being processed is equal to the number of connections accepted by the server. Although simple to implement, this model has two key drawbacks:

---

[2] *Once upgraded a session remains at higher priority for the session duration.*

- *Poor response times because of TCP timeouts at the clients:* A web server is typically unaware of the number of requests that are awaiting processing at any instant. While some of the requests waiting for service may arrive over existing TCP connections (HTTP/1.1 clients using persistent connections), a majority of requests arrive over new TCP connections (HTTP/1.0 clients, or HTTP/1.1 clients issuing multiple simultaneous requests over separate connections). When the web server is overloaded, the rate of incoming requests exceeds the rate at which the server is able to process requests. Consequently, as the server becomes a bottleneck, more and more requests (and their associated TCP connections) accumulate in the server's TCP listen queue, waiting for the web server to pick them up. When the number of requests queued in the TCP listen queue exceeds some pre-specified system limit, new requests are dropped by the server's TCP/IP stack itself with the web server application being totally unaware of such drops. Dropped connections usually cause timeouts at the TCP layer of the clients. Since TCP timeouts may last from 3 seconds to 90 seconds, dropped connections usually result in very poor response times for client accesses to the web site.
- *Premium requests may be dropped:* A Web2K server employing the sequential processing model is likely to drop even premium requests at server overload, because the TCP/IP stack does not distinguish between premium and basic requests.

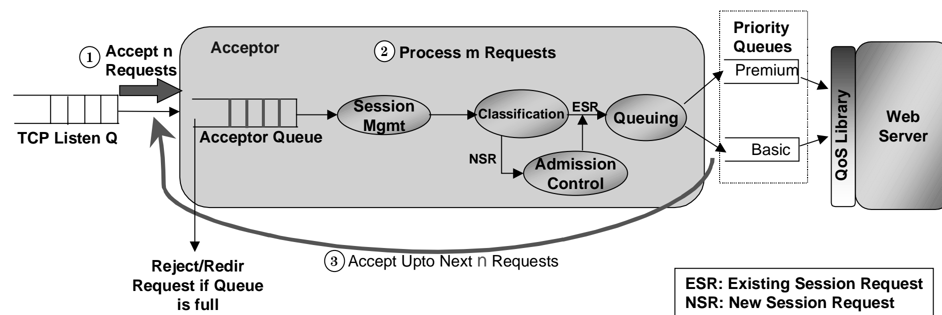


Figure 2: Implementation of the traffic-aware request-processing model in the Web2K acceptor

To avoid both of these drawbacks, the Web2K acceptor implements a traffic-aware request-processing model (see Figure 2). As its name indicates, in this model, the acceptor keeps track of the number of new requests that are pending for service at any instant of time. When it detects that the rate of incoming requests is greater than the rate of processing, the acceptor explicitly rejects/redirects a portion of the incoming requests. By doing so, the acceptor controls when and which requests are rejected/redirected, instead of letting the server's TCP/IP stack decide when requests get dropped.

Figure 2 illustrates the operation of an acceptor that implements the traffic-aware request-processing model. The acceptor operates in repeated cycles, receiving up to n connections[3] from the server's TCP listen queue in each cycle.

[3] *The value of n should be much smaller than the length of the acceptor queue and larger than 1 for traffic-aware processing.*

The acceptor then processes a subset m of the requests it received and proceeds to the next cycle. In order to ensure that it has control over when and which requests are denied service, the acceptor has to keep the number of requests waiting for service in the server's TCP listen queue to a minimum. For this to be true even under extreme overload, the number of requests that processed in each cycle, m has to be much less than the number of connections accepted in each cycle, n. Because m is less than or equal to n, the acceptor has to maintain an internal queue of requests that it has accepted but not yet processed. This internal queue is referred to as the *acceptor queue* in Figure 2. With this design, because requests are being held in the acceptor queue rather than in the TCP listen queue, the size of the internal acceptor queue should be at least as much as the size of the TCP listen queue. This allows the acceptor to handle bursts of incoming requests effectively. When the number of pending connections exceeds the length of the acceptor queue, the acceptor explicitly denies service to subsequent incoming requests until it can make room in the acceptor queue (by processing one or more of the requests that are in the acceptor queue).

There are interesting tradeoffs in the choice of the value of m used in the traffic-aware request processing approach. The smaller the value of m relative to n, the higher the chance that the acceptor is able to drain the listen queue without entailing queue overflows (more requests will be queued in the acceptor queue in this case). At the same time, however, the smaller the value of m, the larger the delay[4] before a request is processed by the server. Hence, while a smaller value of m is preferred when the server is overloaded, a larger value of m (approaching the value of n) is preferred when the server is lightly loaded. To achieve optimum performance, the acceptor dynamically adjusts the value of m based on the input load. In our prototype implementation, for each cycle in which no new requests are received, the acceptor doubles the value of m, subject to the constraint that the value of m is bounded by n. Likewise, for each cycle in which any new requests are received, the acceptor halves the value of m, subject to the constraint that m should have a value of at least 1. Using this scheme, at low loads, m remains equal to n and at extreme overloads, m reduces to 1.

The traffic-aware request-processing model has the following advantages:

- Since it avoids connection drops from the TCP/IP stack, it averts TCP timeouts, thereby significantly improving the response times seen by users. This is true both for requests processed by the server as well as for requests that are denied service.
- This model ensures that the acceptor, rather than the server's TCP/IP stack, decides which requests must be denied service. Such a decision can be based on the classification of requests (e.g., only basic class requests may be denied service).
- The number of requests received by the acceptor during each cycle provides an operating system independent method of determining the occupancy of the listen queue. While many vendors provide tools to query the instantaneous occupancy of the listen queue [12], these tools are not universally applicable.

3.2 Early Prioritization in the Acceptor

[4] *Note, however, that the queuing delay in the acceptor queue is much smaller than the delay that requests are likely to see as a result of TCP timeouts.*

While the traffic-aware request processing model dictates how many requests (*m*) should be processed by the acceptor in each cycle, this model does not dictate *which* requests must be processed. Figure 3 depicts how the acceptor decides which requests to process in each cycle. When it picks up a request for processing from the acceptor queue, the acceptor has to perform session management in order to determine the class of the request and whether the request pertains to an existing session. In order to ensure that premium requests receive higher priority starting at the earliest point in the Web2K server, the acceptor holds basic requests that it receives in one of two pending request queues. (Two queues are necessary for the acceptor to ensure that requests from existing sessions receive priority over requests from new sessions.) However, when the acceptor detects a premium request, it immediately processes the request, adding the request to the premium queue (as shown in Figure 3) from which the web server receives new requests. Since basic requests cannot be held in the pending queues forever, the acceptor uses a parameter called the *look-ahead* (*k*) to determine when it should process basic requests. The look-ahead represents the maximum number of requests that the acceptor will scan from the acceptor queue looking for premium requests before it decides to process a basic request. When the look-ahead is set to one, the acceptor treats both premium and basic requests in the same manner. The larger the look-ahead value, greater the prioritization provided for premium requests as compared to basic requests.

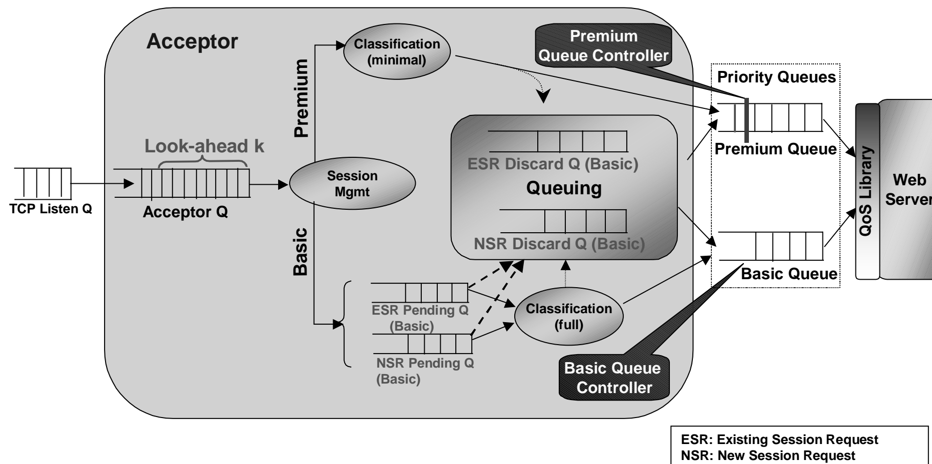


Figure 3: Details of how the acceptor enables prioritization between premium and basic class requests. To simplify the illustration, the discard and pending queues for the premium class are not shown in the figure.

The scheduling model depicted in Figure 3 has the following advantages:

- First, depending on the look-ahead value (derived from operator-specified preferences), premium requests are serviced at higher priority than basic requests. To ensure further prioritization, the QoS library (which forces the web server to receive HTTP requests from the Web2K priority queues, rather than the TCP listen queue) implements a weighted fair-share scheduling policy. The weights used in the QoS library for scheduling requests are based on the look-ahead specification. Larger the look-ahead value, larger the weight assigned for servicing the premium priority queue relative to the basic priority queue.

- At peak loads, this model also optimizes the utilization of the acceptor. This is because premium requests do not need to be re-classified unless their associated sessions expire. Consequently, classification overheads are minimal for premium requests as compared to basic requests.

3.3 Predictive Queue Control

To ensure that the response time seen by users of a web site does not increase exponentially with server load, the acceptor has to control the number of requests that are admitted for service. The acceptor uses the occupancies of the premium and basic priority queues (from which the QoS library picks requests for the web server's processing) as a guide for enforcing admission control. Treating the premium and basic priority queues independently, the acceptor uses industrial proportional integral (PI) control techniques [13] for determining when it must deny service to an incoming request. While these techniques have been widely used in manufacturing systems to respond to changes, such as an increase in temperature or pressure in a chemical plant, their application to the control of incoming requests to a web server is one of the novel features of the Web2K server design. Figure 3 shows the premium and basic priority queue controllers as part of the acceptor.

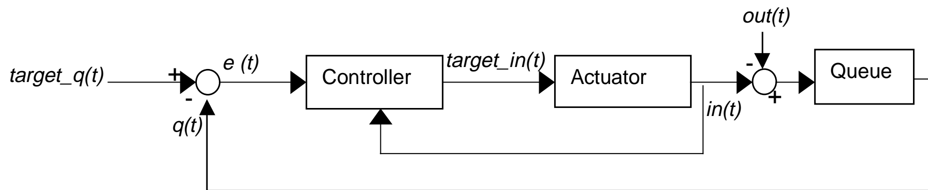


Figure 4: Design of an predictive queue controller used by the Web2K acceptor

Each predictive queue controller has two main components, namely the *controller* and the *actuator* (see Figure 4) that together combine to ensure that the occupancy of the priority queue stays at or below a pre-specified target value. To understand the operation of the controller and the actuator, consider the operation of the Web2K server during a time period *[t-T, t]*. Suppose *in(t)* represents the number of requests that the acceptor receives during this period for a given class. Also, suppose *out(t)* is the number of requests for that class that the web server application processes in this period. If *q(t-T)* represents the length of the basic priority queue at time *t-T*, the queue length at time *t* is given by $q(t) = q(t-T) + in(t) - out(t)$. As indicated in Figure 4, the input to the controller is the difference *e(t)* between the target queue length (*target_q(t)*) and the current queue length, *q(t)*, i.e., $e(t) = target_q(t) - q(t)$ in this example. Based on this difference, the controller determines that the number of requests that can be added to the priority queue during the next period *[t,t+T]* is: $target_in(t) = in(t) + q0*e(t) + q1*e(t-T)$ where *q0* and *q1* are constants that are determined based on the transient response desired from the controller to sudden changes in the input or output rates [13]. Note that in determining *target_in(t)*, the controller has made no assumptions about the rate of processing requests by the web server. In fact, the controller adapts its output based on the web server's processing rates from the previous interval.

The actuator is responsible for ensuring that no more than *target_in(t)* requests are added to the priority queue in the period *[t, t+T]*. Based on its estimate of the incoming request rate, *in_rate_estimate(t)* the actuator computes the probability *p* with which an incoming request can be admitted for service to be $p = (target_in(t) / in_rate_estimate(t))$. Whenever possible, the actuator attempts to admit requests belonging to existing sessions, at the cost of request for new

sessions. Since input load to a web server can be bursty, requests that are denied admission by the actuator are held back in a set of discard queues (one per request class) that are internal to the acceptor (see Figure 3). During the next period, the requests that are held in these discard queues are given preference for admission over new requests. When it is unable to admit the requests held in the discard queues over multiple processing intervals, the acceptor denies access to these requests and terminates the corresponding TCP connections. This avoids rejection of requests for short spikes in server load or transient changes in processing rate of the server.

3.4 Multi-Stage Admission Control

Implementing admission control based on the occupancies of the priority queues alone may not be sufficient to ensure that the acceptor is able to prioritize premium requests over basic requests. For example, the Web2K server only has a finite number of file descriptors that it can use[5] at any instant of time. Each incoming request takes up a file descriptor. So, at extreme overload, the occupancy of any of the several queues that the Web2K server maintains could grow to such an extent that all of the available file descriptors are in use. When such a condition is reached, all incoming TCP connection requests (including premium requests) to the server are denied service by the server's TCP/IP stack. To avoid such a condition from ever occurring, the Web2K server keeps track of the number of file descriptors that are in use simultaneously. As this value reaches close to the operating system limit, the acceptor decides to explicitly deny service to a portion of the incoming requests. As before, the acceptor strives to give priority to requests of existing sessions over requests from new sessions.

Frequently, when a server is overloaded, users may decide to terminate requests they have issued to a web site. Most often, the web server is unaware of such events and ends up wasting critical resources in servicing the terminated request. The Web2K server can avoid such a condition with adequate support from the operating system. For example, the HP-UX 10.20 operating system provides a way in which an application can query the kernel to determine the state of each TCP connection. If a connection is in the CLOSE_WAIT TCP state, the application can determine that the request has been stopped by the user. Since not all operating systems provide such an interface, the acceptor can also timeout requests based on an operator-specified timeout value. In our implementation, the acceptor keeps track of the time when it receives each request and when it determines that a request has spent too much time waiting for service in the acceptor, it explicitly rejects the request (since most probably, the user has stopped the request anyway).

4 Performance Evaluation

In this section, we present a comprehensive analysis comparing the performance of a Web2K server with a vanilla Apache web server. The performance results demonstrate the effectiveness of the Web2K server design. Moreover, these results serve to quantify the overheads associated with the Web2K server.

4.1 Test Methodology

For the performance study, Apache version 1.3.6 was used as the vanilla web server. The Web2K server used in this study was derived from the vanilla Apache 1.3.6 web server. To evaluate the effectiveness and the overheads of the

---

[5] *There is an operating system-specified limit on the number of file descriptors that can be in use simultaneously by each process.*

Web2K server under different conditions, two types of workloads were used. While the first workload involved retrieval of static pages by clients from the web servers, the second workload involved access to a mix of static and dynamic content. To compare the relative performance of the Web2K server and the vanilla web server, the incoming load to the server was varied (from low loads to overload). The mix of premium and basic sessions that the server handled was also varied. The key metrics used to compare the performance of the Web2K server and the vanilla web server were:

- Percentage of incoming premium sessions that were successfully handled by the server
- Average time for the successful completion of a premium session
- Total number of sessions successfully handled by the server, for a given workload

The following sections go into the details of the testbed configuration, the workload used for the study, and the performance results.

Server	**Configuration parameters and their values**[6]
Vanilla Apache web server	• Listen queue length: 1024 • Initial number of server processes: 4 • Min number of spare server processes: 4 • Max number of server processes: 256
Web2K server	• Listen queue length: 1024 • Initial number of server processes: 4 • Min number of spare server processes: 4 • Max number of server processes: 256 • Controller period: 250ms • Acceptor queue length: 2048 • Pending queue lengths: 1500 • Default target queue lengths: 500 • Default look-ahead: 12 • Maximum priority queue lengths: 1000

Table 1: Configuration of the web servers used for the performance evaluation

4.2 Testbed Configuration

The Web2K and vanilla Apache web servers were executed on an R-class HP-UX 11.0 system. Client workload to the server was emulated using three workstations connected to the server via an isolated 100Base-T local area network. Of the three clients, two emulated basic user sessions and one client emulated premium user sessions. To generate

[6] *While the Web2K server has a number of parameters that determine its behavior and overheads, many of these parameters can be automatically set based on the web server's configuration settings and certain operating system limits. Section 5 illustrates a simple configuration model that requires just two configuration settings from the human operator which are automatically translated to configuration parameters for the Web2K server.*

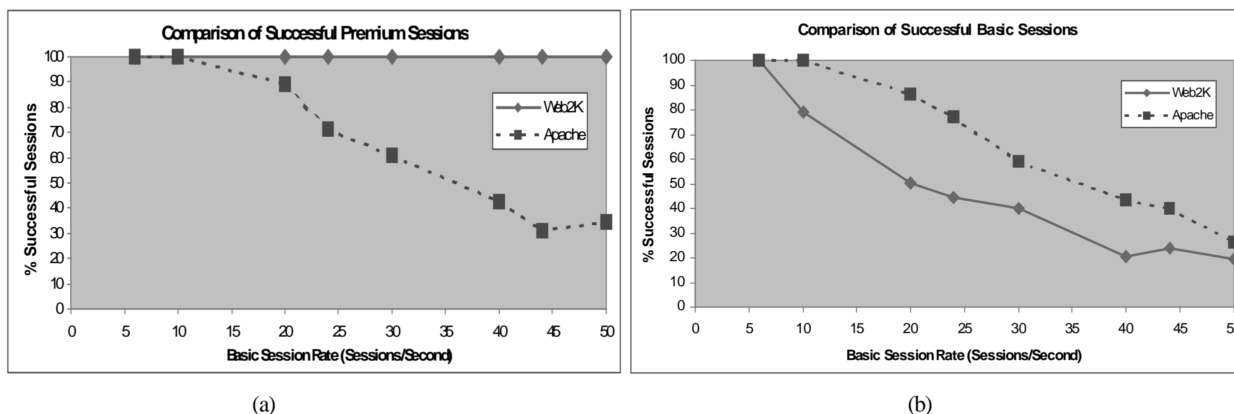
workload for the performance study, the public domain *httperf* [14] tool was used. *Httperf* was chosen because of its ability to accurately emulate user sessions, including the ability to incorporate user think times between successive requests. Moreover, *httperf* also emulates the behavior of web browsers by opening multiple simultaneous connections to fetch the images embedded on a page in parallel.

Table 1 indicates the configuration settings used for the performance evaluation. A listen queue length of 1024 was used because the Apache web server performed best at this setting in our testbed.

4.3 Workload

The content used to generate the synthetic workload mirrored a subset of a popular on-line banking web site. Each user session emulated a user logging in to the web site, checking the balance in his/her accounts, browsing through the last transactions posted to one of his/her accounts, transferring money between accounts, and finally logging out. Each session included 40 unique URLs. A session lasted a few minutes, with the total user think time during the session being set to 75 seconds. The timeout period for each request was set to 20 seconds.

To study the effectiveness of Web2K server under different conditions, two different workloads were used. The first workload (referred to henceforth as the *static workload*) involves accesses to static URLs only. In this case, a session involved access to 10 HTML pages. These HTML pages together had 30 embedded gif and jpeg images. The size of each page accessed ranged from 200 bytes to 20 Kbytes. For the second workload (referred to as the *dynamic workload*), two of the HTML accesses were replaced by accesses to CGI programs. These CGI programs, which were intended to emulate application processing for retrieval of account information from a backend database, were designed to take up 25ms of CPU processing time before completion.



(a) (b)

Figure 5: Comparison of the percentage of (a) premium and (b) basic sessions that completed for the static workload scenario.

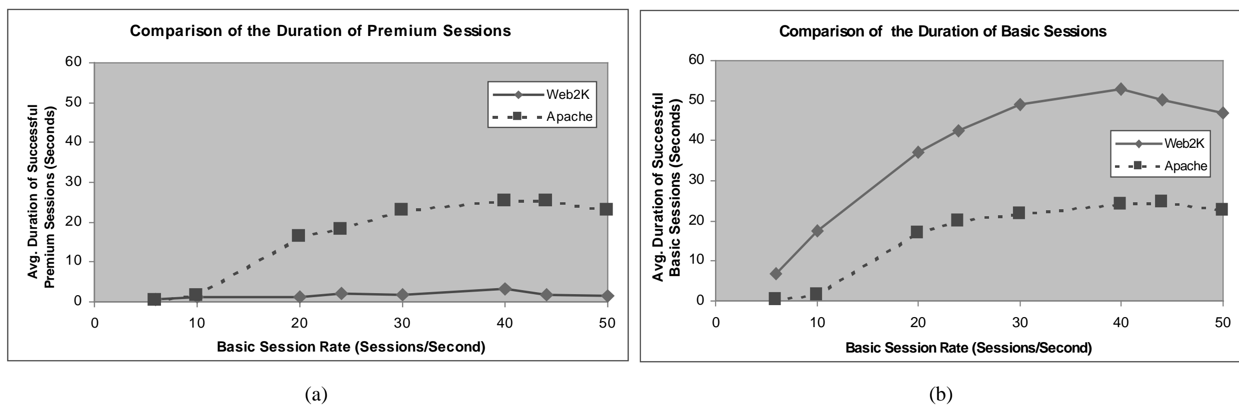
4.4 Performance Results

4.4.1 Static Workload

For the static workload, Figure 5(a) and (b) compare the performance of the Web2K server with the vanilla Apache server in terms of the percentage of premium and basic sessions that completed successfully[7]. In this experiment, the premium class workload was held constant at 10 sessions/second, while the basic class workload was increased from 6 sessions/second to 50 sessions/second.

It is evident from Figure 5 that as the workload increases, both premium and basic sessions experienced similar performance when accessing a web site that was deployed using the vanilla Apache web server. A detailed analysis of the results (not shown in the figure) revealed that all of the sessions that failed were terminated as a result of connection failures seen by the clients. These connection failures can be attributed to TCP listen queue overflows at the web server resulting from the sequential request processing model that is used by the Apache server. Since session failures begin to occur when the total session rate increases beyond 20 sessions/second, this value represents the total capacity of the web server.

As can be seen from Figure 5(a), *the Web2K server was able to handle all of the premium user sessions even when the total session rate was three times the server capacity* (i.e., 60 sessions/second – 10 sessions/second of premium and 50 sessions/second of basic sessions). This is attributable to the traffic-aware request processing model that avoids TCP listen queue overflows and to the prioritization of premium class requests by the acceptor. Of course, the preferential handling of premium requests is at the expense of basic requests. As Figure 5(b) indicates, the Web2K server was only able to support a lower percentage of successful basic user sessions than the vanilla Apache web server.



(a) (b)

Figure 6: Comparison of the duration of (a) successful premium sessions and (b) successful basic sessions for the static workload scenario. The slight decrease in session duration at peak load is attributable to the decrease in successful basic sessions at peak load (see Figure 5).

Figure 6(a) and (b) compare the average duration of premium and basic sessions that completed successfully (user think times of 75 seconds, which are the same for premium and basic users, have been factored out from the session duration values). As in Figure 5, the Web2K server was able to offer significantly better performance for premium users

---

[7] *A session is considered to have completed successfully if all its requests completed successfully.*

compared to basic users. At peak loads, while basic sessions took about 50 seconds to complete, premium sessions took only a couple of seconds.

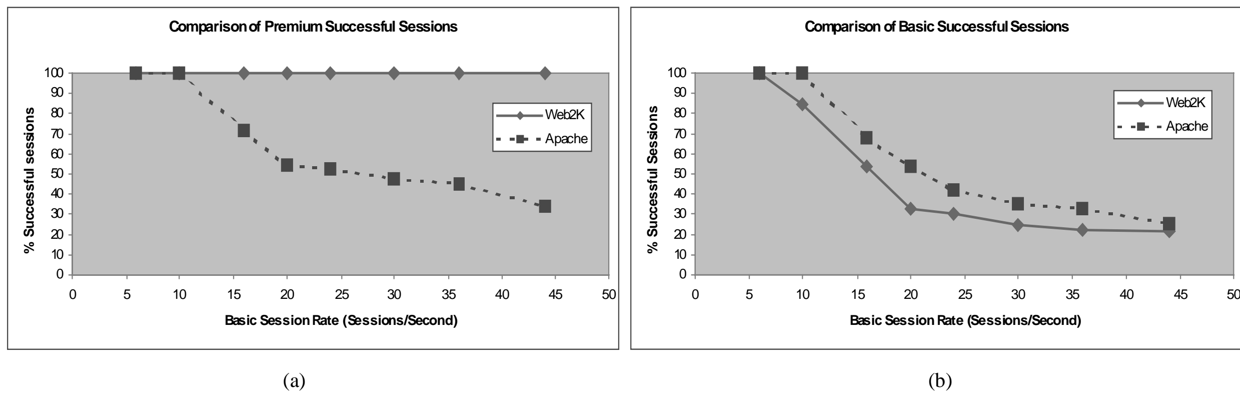


Figure 7: Comparison of the percentage of (a) successful premium sessions and (b) successful basic sessions for the dynamic workload scenario.

4.4.2 Dynamic Workload

The comparison between the Web2K server and the vanilla Apache web server for the dynamic workload is similar to the one for the static workload in Section 4.4. Figure 7(a) and (b) compare the percentage of premium and basic sessions that succeed as the load on the server was increased. In this scenario, the premium workload was held constant at 4 sessions/second, while the basic workload was increased from 6 sessions/second to 44 sessions/second. Since the vanilla Apache server started to drop sessions as the total load of server increased beyond 14 sessions/second (4 sessions/second of premium sessions and 10 sessions/second of basic sessions), this rate represents the capacity of the web server for the dynamic workload. Figure 7(a) indicates that the Web2K server ensured that all premium user sessions succeed even when the incoming load to the server was over three times the capacity of the server.

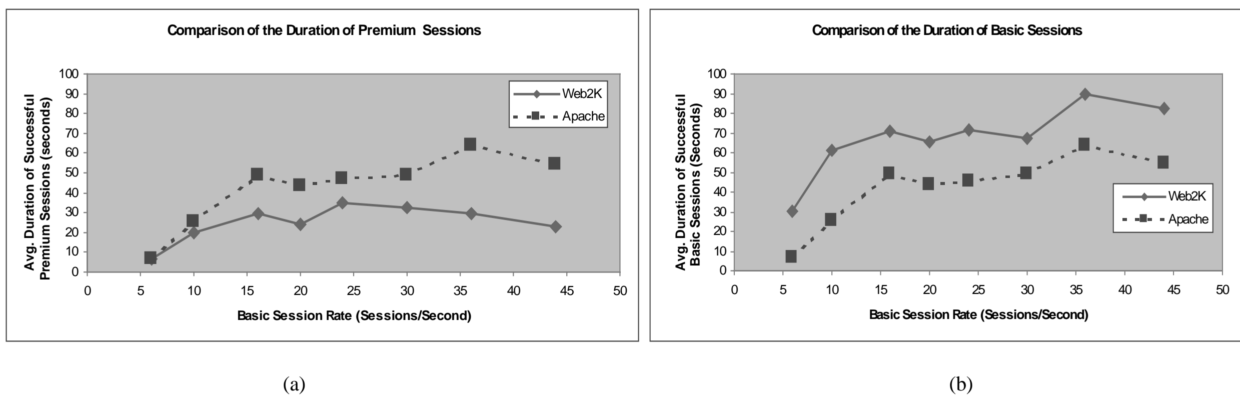


Figure 8: Comparison of the duration of successful sessions for (a) premium users and (b) basic users for the dynamic workload scenario.

For completeness, the duration of successful premium and basic sessions for the dynamic workload is shown in Figure 8(a) and (b) respectively.

4.4.3 Overheads of the Web2K Server

Having illustrated the effectiveness of the Web2K server under different workloads, next we proceed to quantify the overheads that the Web2K middleware layer introduces relative to the vanilla web server. For the static workload and the dynamic workload scenarios, respectively, Figure 9(a) and (b) represent the total number of sessions (includes premium and basic) that completed successfully. The difference between the total number of successful sessions when the vanilla Apache server was used and when the Web2K server was used represents the overhead of the Web2K middleware layer. For the static workload, the Web2K server overhead varied from 10-30% depending on the load on the server. At extreme overloads, since it offers higher priority to requests from existing sessions compared to requests from new sessions, the Web2K server was even able to handle more sessions than the vanilla Apache server. For the dynamic workload, the Web2K server overhead remained around 10% at most loads.

However, note that the overheads quantified based on Figure 9 hold for the configuration of the Web2K server used thus far in the performance study – where the look-ahead was 12 and the target priority queue occupancies were set to 500. In the next section, we consider how tuning these two parameters changes the performance overhead of a Web2K server.

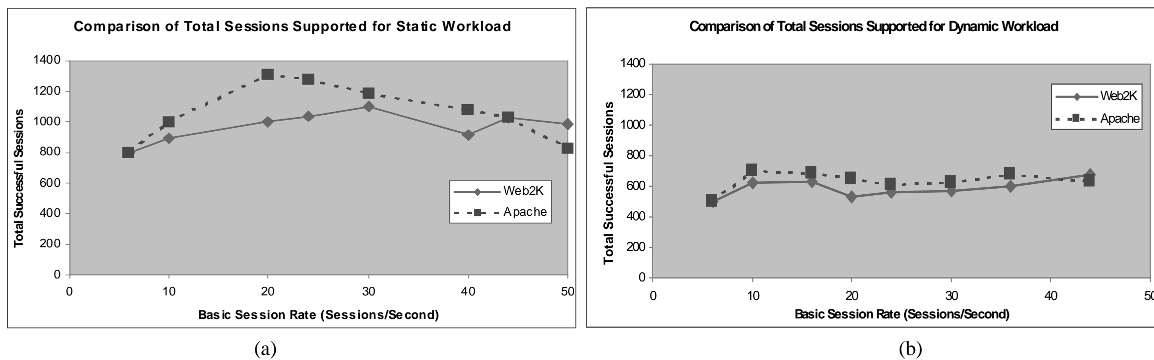


Figure 9: Comparison of total number of successful sessions for (a) the static workload scenario and for (b) the dynamic workload scenario.

4.4.4 Tuning the Web2K Server

Many of the parameters that Web2K server requires can be statically assigned values depending upon the web server's configuration. For instance, the maximum lengths of all the Web2K acceptor internal queues can be set based on the maximum number of file descriptors allowed for a process (typical values range from 1024-4096). The burst of requests received by the Web2K acceptor from the TCP stack can be set to the length of the server's TCP listen queue (typical values range from 32 to 1024).

There are two key parameters, namely the look-ahead and priority queue occupancy targets that cannot be statically assigned because they have performance and capacity implications. As discussed earlier, the look-ahead value determines the level of prioritization offered to premium user requests in the Web2K acceptor, while the priority queue occupancy targets are used by the predictive queue controllers in the Web2K acceptor to decide whether to admit or deny service to a request. As we will see, these parameters can have significant impact on the effectiveness of Web2K server and its associated overheads. Hence, these parameters must be set based on operator preferences or computed

dynamically based on some optimization criteria. A configuration and use model that can be used by operators to specify their Web2K server preferences is described in Section 5.

4.4.4.1 Choosing the Look-Ahead

Considering the static workload scenario, Figure 10 compares the total number of sessions that Web2K server was able to support for different choices of the look-ahead parameter. Notice that the optimum value of the look-ahead varies with input load. When the basic class session rate was less than 30 sessions/sec, using a look-ahead of 1 enabled the Web2K server to support the maximum number of total sessions. Above this value of basic session rate, a higher look-ahead (48, in the example shown) performed best.

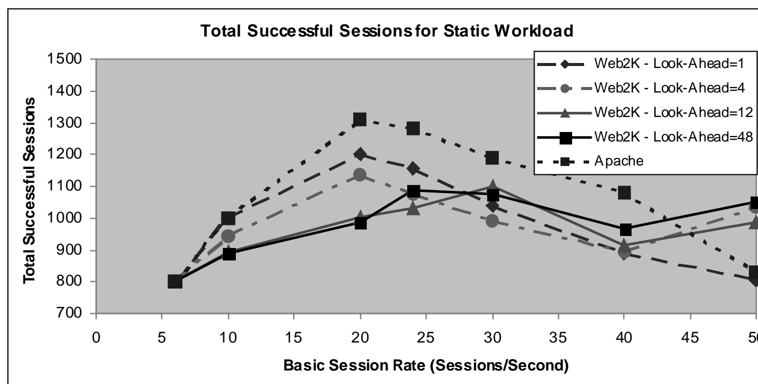


Figure 10: Variation in total successful sessions with change in the Web2K look-ahead value.

To understand the tradeoffs involved in choosing the look-ahead value, consider Figure 11, which illustrates how the percentage of successful premium and basic sessions varied with change in the look-ahead value. Figure 11(a) indicates that although a look-ahead of one enabled the Web2K server to support a greater number of sessions at lower loads, the Web2K server ceased to offer any differentiation between premium and basic sessions at this look-ahead setting. In fact, the Web2K server even dropped premium sessions as the server reached overload. For instance, when the basic session rate was 20 sessions/sec, the Web2K server dropped 20% of the premium sessions. As the basic session rate increased further, more premium sessions were denied service. Because of the lack of differentiation between premium and basic sessions, at this look-ahead setting, premium users also do not see any difference in response times compared to basic users (this is not shown in the figure).

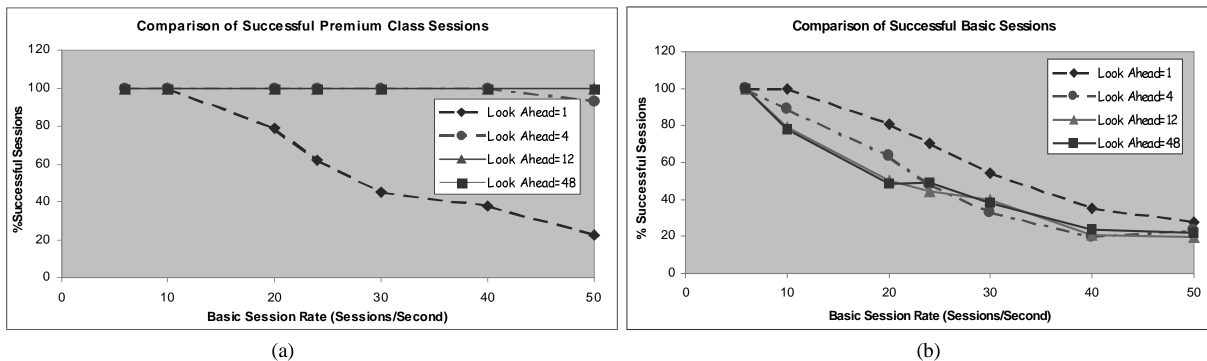


Figure 11: Comparison of successful premium and basic sessions for different look-ahead values

Even when the look-ahead was increased to 4, the Web2K server denied service to about 5% of the premium sessions at extreme overloads (see Figure 11(a)). Increasing the look-ahead further ensured that all of the premium sessions succeeded. In general, we observed that to ensure that the Web2K server processes all incoming premium sessions, the look-ahead value must be greater than or equal to the inverse of the ratio of premium requests to total number of requests received by the web server. To clarify this further, consider Figure 11(a). At a load of 40 sessions/second (30 of which are basic and 10 are premium), the premium session requests accounted for $1/4^{th}$ of the total traffic to the server. A look-ahead of at least 4 is necessary to ensure that all of the premium requests are handled at high priority by the Web2K server. Similarly, at the peak load of 60 sessions/second, the premium requests accounted for $1/6^{th}$ of the total traffic to the server (the input load is 10 premium sessions and 50 basic sessions per second at the last point in Figure 11(a)). Hence, a look-ahead of at least 6 is necessary for the Web2K server to be able to successfully service all of the premium sessions.

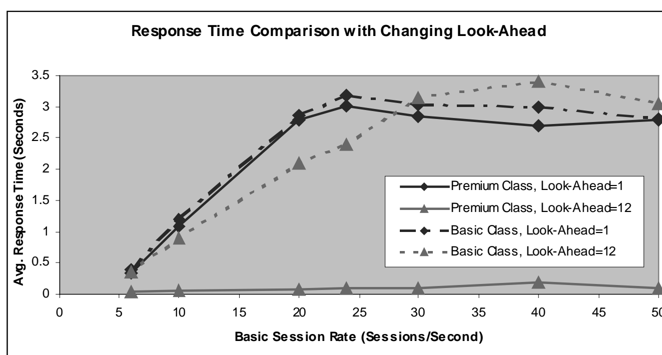


Figure 12: Effect of changing the look-ahead value on the response times seen by premium and basic class requests.

Figure 12 compares the response time seen by premium and basic users as the look-ahead value changes. Like Figure 11, Figure 12 also indicates that a look-ahead of 1 offers almost no differentiation between the premium and basic classes. Both classes see similar response times when accessing the Web2K server. On the other hand, when the look-ahead value is increased (to 12 in the example shown in Figure 12), premium users see significantly smaller response times relative to basic users.

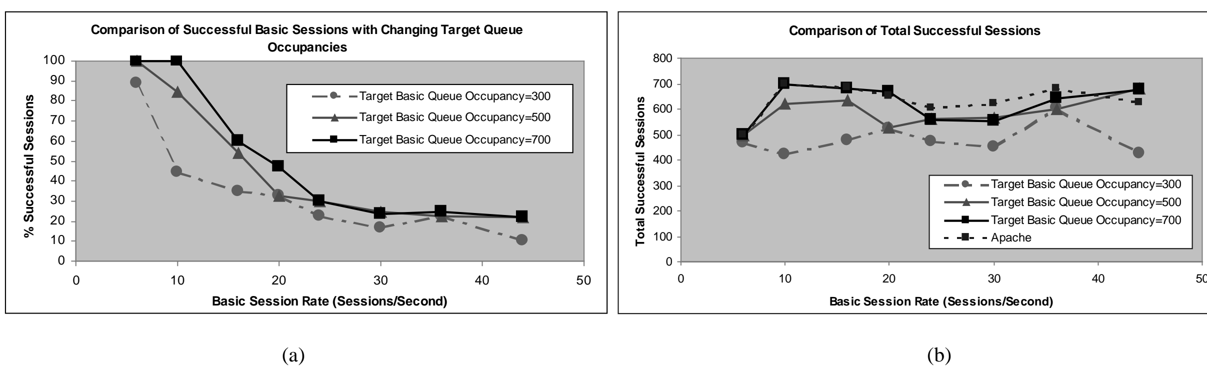


Figure 13: (a) Variation in successful basic sessions with change in target basic queue occupancy; (b) Variation in total number of successful sessions with change in target basic queue occupancy.

4.4.4.2 Choosing Priority Queue Occupancy Targets

Next, we consider how the target occupancies of the premium and basic priority queues provided to the predictive queue controllers (see Figure 3) impacts the behavior of the Web2K server. As the target occupancy of a priority queue increases, the number of successful sessions of the corresponding class increases, since the Web2K server has more room to hold requests for that class. For the dynamic workload scenario, Figure 13 indicates that as the target queue occupancy for the basic priority queue was increased, the number of successful basic sessions increased, thereby increasing the total number of sessions handled by the server. Because premium requests have priority, changing the target point for the basic priority queue had no effect on the number of successful premium sessions in this example. Since it increases the number of pending requests for a class, increasing the target occupancy for a priority queue increases the response time seen by requests for that class.

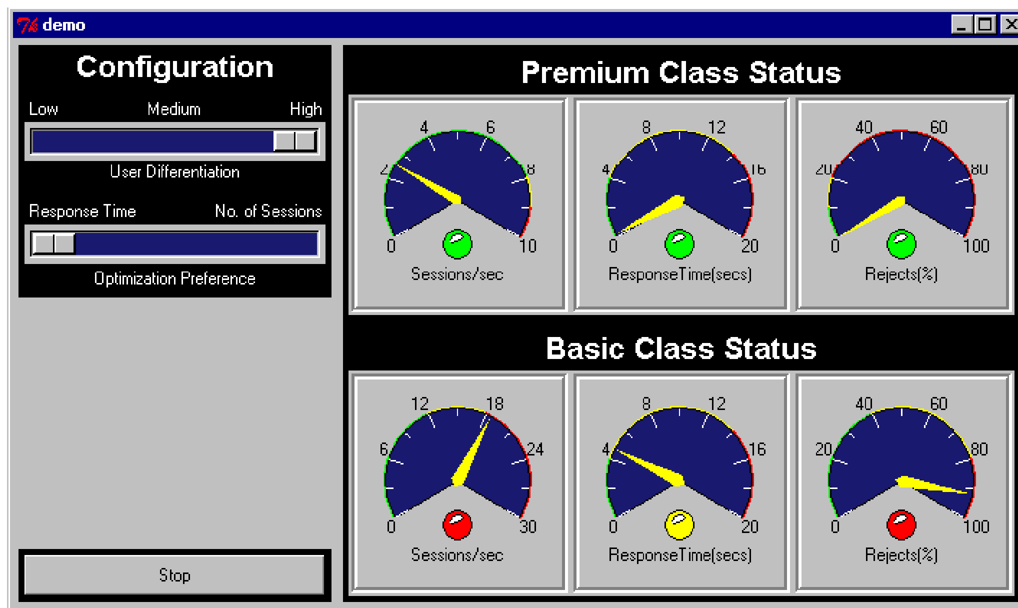


Figure 14: User interface for configuring and monitoring a Web2K server.

5 Configuration Model

Typically, QoS-aware processing depends on the operator setting QoS targets (response time, throughput etc.) for the system as well as policies that specify actions that need to be taken when those targets are violated. Figure 6(a) and Figure 8(a) show the average session duration for premium sessions for static and dynamic workloads respectively. Note that while the average session duration of premium sessions remains around 2 seconds for the static workload, the corresponding value for the dynamic workload is around 30 seconds. These results demonstrate the stark difference in response times that can result from minor changes[8] in the content being accessed. More importantly, these results highlight the difficulty in supporting strict response time guarantees for web accesses. To offer such guarantees, the

[8] *While the static workload involved accesses to static pages only, in the dynamic workload, two of the static pages being accessed were replaced by CGI invocations.*

Web2K server has to be aware of the semantics of the request, which is extremely difficult because of the diverse nature of web content and the frequency with which the content is updated. In this section, we discuss how the Web2K server uses operator-specified *preferences* (rather than strict guarantee specifications) to modify its operation to meet the operator's performance goals. Using the results from Section 4.4.4, we see that a web site operator using a Web2K server has to decide:

- What is the degree of differentiation (between premium and basic classes) that she/he wants the web site to offer?
- What are her/his optimization preferences – should the Web2K server be optimized to ensure minimum response time for all users, or should it be optimized to support the maximum number of sessions?

The operator's choice of the above decision metrics can be directly translated into parameter settings for the Web2K server. For instance, greater the degree of differentiation desired, greater the value of the look-ahead. Likewise, greater the operator's preference for maximizing the server capacity, greater the value of the target queue occupancy settings for the Web2K server priority queues.

Figure 14 depicts the proposed configuration interface for a Web2K server. Through the sliders on the configuration panel, an operator can specify his/her preferences of how the Web2K server should operate. Real-time statistics displayed using the meters on the performance panel provide indications about the Web2K server's performance, which can be used by the operator to further tune his/her preferences.

6 Summary

In this paper, we have described the design and implementation of a QoS enabled web server, called the Web2K server that offers differentiated service to different classes of users without requiring any modifications to the operating system or the web application server software. One of the key innovations in the design of the Web2K server is the traffic-aware request processing model. Using this model, the Web2K server ensures that it is able to handle the majority of incoming requests, even at extreme overloads. Predictive queue controllers enable the Web2K server to decide whether to admit all of the incoming requests, or deny service to some percentage of requests. The Web2K server uses operator specified classification criteria to maximize the number of high priority requests that it services during overload.

Performance results presented in the paper demonstrate the effectiveness of the Web2K server in ensuring that high priority users accessing a web site see low response times even at extreme overloads (over 3x capacity). A comparison of total sessions supported indicates that the Web2K server is able to offer differential service quality without significantly impacting the server's capacity.

References

[1] J.M. Almeida, V. Almeida, and D.J. Yates, Measuring the behavior of a world-wide web server, *In Proceedings of the Seventh Conference on High Performance Networking*, pp. 57-72, White Plains, NY, 1997.

[2] J.C. Mogul, Network behavior of a busy web server and its clients, *Compaq Western Research Laboratory Research Report 95/5*, October 1995, http://www.research.digital.com/wrl/techreports/abstracts/95.5.html

[3] V.S. Pai, P. Druschel, and W. Zwaenepoel, Flash: An efficient and portable web server, *In Proceedings of the 1999 Annual Usenix Technical Conference,* Monterey, CA, June 1999.

[4] M. Crovella, R. Frangioso, and M. Harchol-Balter, Connection scheduling in web servers, Boston University Technical Report, 1999-003, http://lite.ncstrl.org:3803/Dienst/UI/2.0/Describe/ncstrl.bu_cs%2f1999-003, March 1999.

[5] J.C. Mogul, Operating system support for busy Internet servers, *Compaq Western Research Laboratory Technical Note TN-49*, May 1995, http://www.research.digital.com/wrl/techreports/abstracts/TN-49.html. *Also in Proceeedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[6] A. Vahadat, T. Anderson, M. Dahlin, B. Belani, D. Culler, P. Eastham, and C. Yoshikawa, Webos: operating system services for wide area applications, *In Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1995, http://now.cs.berkeley.edu/WebOS/papers/hpdc98.ps.

[7] N. Bhatti and R. Friedrich, Web Server Support for Tiered Services, *IEEE Network*, Vol. 13, No. 5, pp. 64-71, Sept/Oct 1999.

[8] T. Jin and J.D. Salehi, Capacity guarantees for web servers, *Hewlett-Packard Laboratories Technical Report HPL-98-155*, 1998.

[9] T.F. Abdelzaher and N. Bhatti, Web content adaptation to improve server overload behavior, *In Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, May 1999, http://www8.org/w8-papers/4c-server/web/web.pdf

[10] L. Cherkasova and P. Phaal, Session-based admission control – a mechanism for improving performance of commercial web sites, *In Proceedings of IEEE/IFIP IWQoS'99*, London, UK, June 1999.

[11] HP WebQoS Technology Overview, Hewlett-Packard Company White Paper, http://www.internetsolutions.enterprise.hp.com/webqos/products/infocenter/qps2_whitepaper.pdf

[12] B.W. Curtis, listenq for Solaris, http://playground.sun.com/pub/brutus

[13] R. Isermann, *Digital Control Systems*, ISBN 3-540-10728-2 Springer-Verlag, New York, 1981.

[14] D. Mosberger and T. Jin, Httperf – A Tool for Measuring Web Server Performance, *Performance Evaluation Review,* Vol. 26, No. 3, pp. 31-37, December 1998.