Web Server Support for Tiered Services

Nina Bhatti, Rich Friedrich
Internet Systems and Applications Laboratory
HP Laboratories Palo Alto
HPL-1999-160
December, 1999

web server, QoS

The evolving needs of conducting commerce using the Internet requires more than just network quality of service (QoS) mechanisms for differentiated services. Empirical evidence suggests that overloaded servers can have significant impact on user perceived response times. Furthermore, FIFO scheduling done by servers can eliminate any QoS improvements made by network differentiated services. Consequently, Server QoS is a key component in delivering end to end predictable, stable, and tiered services to end users. This paper describes our research and results for WebQoS an architecture for supporting server QoS. We demonstrate that through classification, admission control and scheduling we can support distinct performance levels for different classes of users and maintain predictable performance even when the server is subjected to a client request rate that is several times greater than the server's maximum processing rate.

Web Server Support for Tiered Services

Nina Bhatti
Rich Friedrich
Hewlett-Packard Research Labs
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

The evolving needs of conducting commerce using the Internet requires more than just network quality of service (QoS) mechanisms for differentiated services. Empirical evidence suggests that overloaded servers can have significant impact on user perceived response times. Furthermore, FIFO scheduling done by servers can eliminate any QoS improvements made by network differentiated services. Consequently, Server QoS is a key component in delivering end to end predictable, stable, and tiered services to end users. This paper describes our research and results for WebQoS an architecture for supporting server QoS. We demonstrate that through classification, admission control and scheduling we can support distinct performance levels for different classes of users and maintain predictable performance even when the server is subjected to a client request rate that is several times greater than the server's maximum processing rate.

1 Introduction

Much of the current research and engineering in the field of quality of service (QoS) has focused on network performance attributes [3]. This has been principally motivated by the application requirements for isochronous streams which have strict constraints on bandwidth, delay and jitter. The IETF [6, 5] has codified proposals for network QoS with the goal of supporting differentiated services.

As Internet usage grows it has become apparent that non-isochronous applications such as delivering static and dynamically generated web pages can also benefit from QoS. As demonstrated in North America during the holiday season in late 1998, the market for electronic commerce increased dramatically. This unexpected and unpredictable increase in client demand resulted in increased congestion and queuing in the servers. Consequently, server response times grew and consumer patience shrunk. Additionally, some applications desire preferential treatment for users or services which is foreign to the egalitarian philosophy of TCP/IP and HTTP. For example, during the US stock market panic of October 1998 large and small investors alike were greeted with "server busy" errors from on-line trading companies. For these applications significant revenues were lost by not being able to process large trades. As these examples

illustrate network QoS by itself is not sufficient to support end to end QoS. The servers must also have mechanisms and policies for establishing and supporting QoS.

We hypothesized that servers would become an important element in delivering QoS. In 1997 our research set out to answer the following questions related to making Internet based applications predictable and stable so that they could support the impending electronic economy. These Internet applications have a potential client population in the millions of users and it is important to understand what requirements they place on servers and networks.

- What is the impact of Internet workloads on servers?
- What is the impact of server latency on end to end latency?
- What server mechanisms are necessary to improve quality of service?
- How can servers be protected from overload?
- How can servers support tiered (differentiated) user service levels with unique performance attributes?

In this paper we will examine these questions and provide results based on empirical observations of our WebQoS research prototype. We will show that the servers play an increasing role in providing end to end QoS and that tiered services can enable new application capabilities for existing Internet based applications such as the web and e-commerce.

2 Servers and QoS

In the summer of 1997 HP Labs researchers had the opportunity to instrument and monitor one of the largest ISPs in North America. This research allowed us to quantify the delay components for web, news and mail services. A simplified topology of the ISP's network and data center is shown in figure 1.

The data center contained hundreds of servers supporting web, news, mail and other IP based services. Clients accessed these systems through the 200 points of presence (POP) located in the United States.

A mixture of active and passive measurement techniques were used to collect performance metrics [7]. The active techniques simulated actual users by periodically issuing predefined requests. For example, the news servers were tested with a client based script that always returned a known 40 Kbyte responses. The passive measurements derived data from performance counters and log files in applications and system software.

A representative sample of measurements for this well engineered network are shown in the following two figures. In figure 2 the nntp server response time for requests of 40 Kbytes sized articles are plotted during a 24-hour period. Most of the time the smoothed response time was on the order of 1-2 seconds with a peak near 7 seconds. Figure 3 plots the coast to coast network response for a 40 Kbytes transfer for 8 of the busiest POPs. The median value ranges from 300 to 700 milliseconds. From the data collected at this large ISP site we concluded that servers were currently a significant component in end to end delay.

E-commerce applications are more complex than this simple news workload. They consist of client, network, web server and application components each of which contributes to the end to end performance.

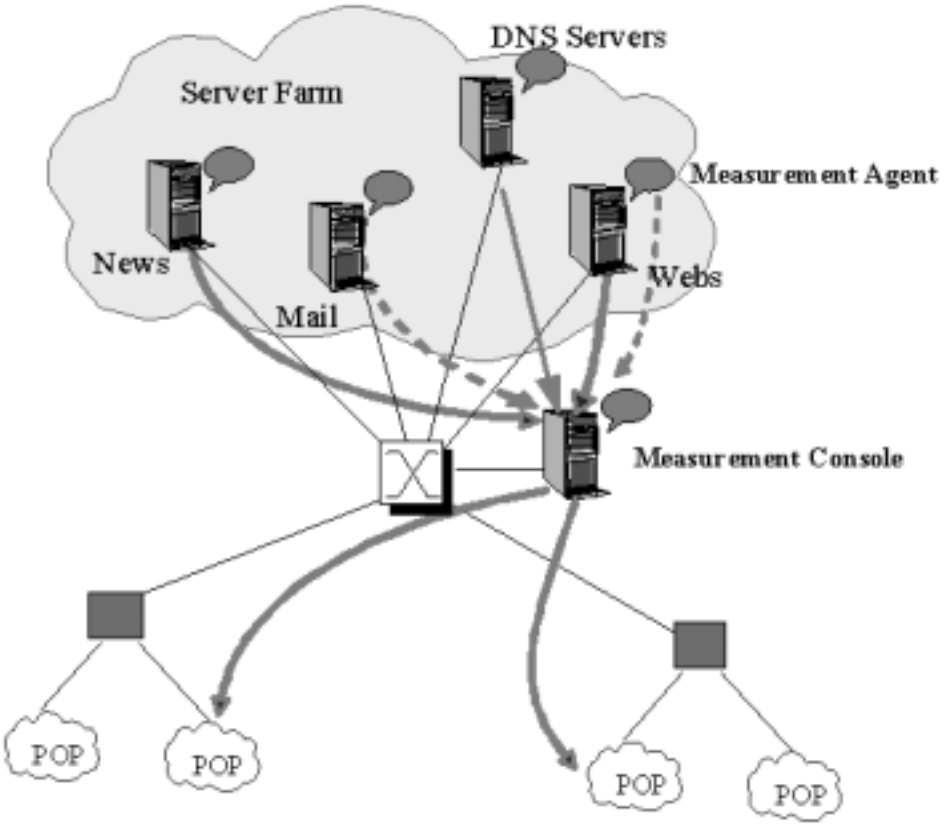

Figure 1: ISP Server and Network Topology

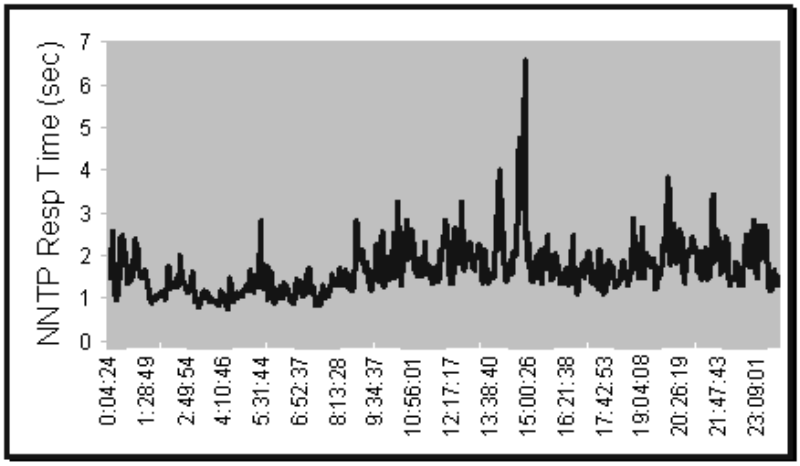

Figure 2: ISP server response time

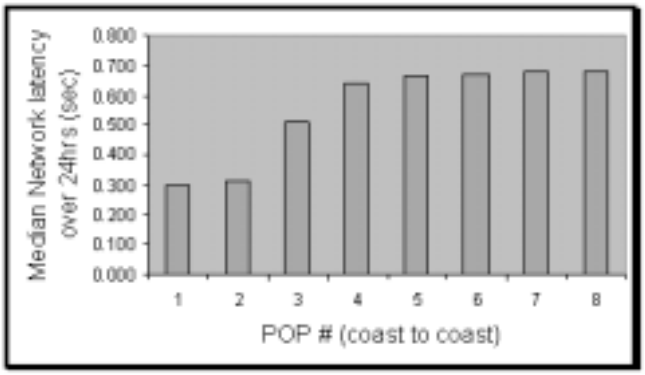

Figure 3: ISP network response time

Several trends are affecting network and server performance for these complex applications. First, network latency is decreasing as the backbone moves towards 622 Mbps and 1 Gbps bandwidth. Second, some ISPs provide guaranteed network latency and in the future private peering agreements between backbone providers will further improve the performance of IP traffic. As caches become more pervasive static content response time will be improved. Furthermore, client access network latency is improving two to ten times with the rapid adoption of ISDN, DSL and cable modems.

At the same time, there are several trends that are increasing server latency time for sophisticated Internet applications. First, flash crowds can overload a popular server leading to sluggish response times or, worse yet, an inability for clients to connect at all [10]. Furthermore, servers can become so busy that they end up wasting resources by processing requests only to find too late that the user is no longer interested and has terminated the TCP/IP connection. Second, new application technologies also increase the processing demands on the server. Some of these new technologies include JAVA, Secure Socket Layer (SSL) for security, dynamic data, database transactions, and sophisticated application middleware components. Third, the media also has become much richer with larger and more images being used as sites try to woo consumers by distinguishing their site from another. Audio, voice, and video are also becoming more prevalent and estimates are that they will become a significant fraction of all IP traffic in the next few years.

We also discovered that overloaded servers are a major force in poor end to end QoS. Our measurements of busy web sites found the following problem. In figure 4 the response rate versus the HTTP GET rate is plotted for a typical web server. As expected, when measured from an HTTP request perspective, the response rate grows linearly until the server nears maximum capacity. At this point an asymptote is reached but the maximum throughput of the server is unchanged over a range of increasing request rates. In figure 5 the same data is presented but now analyzed from a client's perspective. This *session view* represents the sequence of HTTP requests that occur between a client browser and a web server in order to download all required pages and logically complete a user transaction. For example, a session may include browsing for a book title, adding items to a shopping cart, and paying with a credit card. From this session perspective the session throughput collapses rapidly as the server becomes busy due to queuing and congestion on the server. This queuing results in timeouts and clients that give up and abort the request.

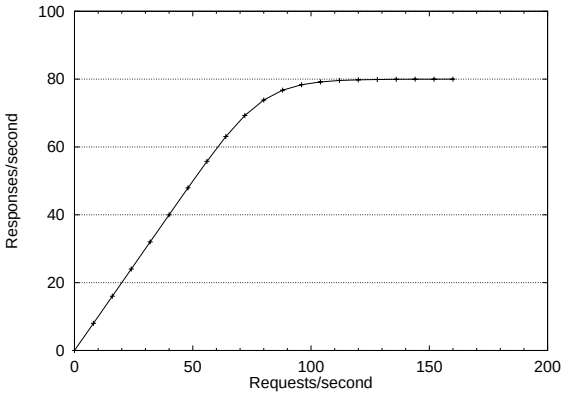

Figure 4: Server HTTP GET throughput

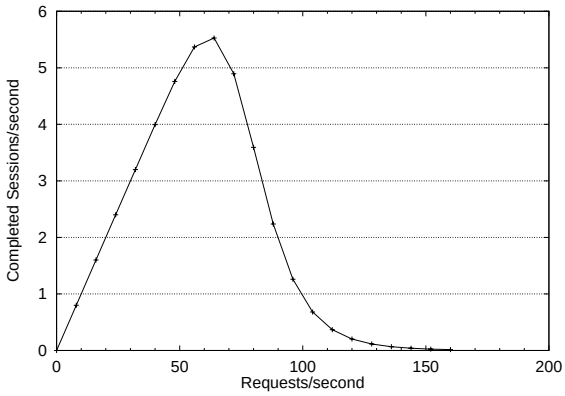

Figure 5: Server session throughput

Furthermore, longer length sessions, which have more commercial value since they are more likely to result in a financial transaction, have a dramatically lower throughput compared to shorter sessions since the probability that an HTTP request in this sequence is lost grows exponentially as the session length increases. Note that the network performance is not a factor in this scenario.

But why not over-provision the server and not have to deal with the issue? Unfortunately, if one examines the evolution of web applications one finds a steep growth in the client demand curve that makes provisioning problematic and not conducive to static resource allocation approaches. Historically, applications executing on a mainframe supported a few thousand clients. These applications evolved to a client-server world of tens of thousands of clients. Now Internet application environments are faced with a client population that appears essentially infinite. Consequently, brute force server resource provisioning is not fiscally prudent since no reasonable amount of hardware can guarantee predictable performance for flash crowds. Furthermore, over-provisioning of servers cannot provide tiered services for users or applications that require preferential treatment.

Furthermore, while Network QoS provided by differentiated services is a critical component for the predictability and stability of an Internet-based application it is only part of the picture. Intelligent network bandwidth management and congestion avoidance features cannot resolve scheduling or bottleneck problems at the server. Networking devices that perform classification operations and bandwidth reservation to speed along packets are undermined by the prevalent First-In-First-Out (FIFO) scheduling policies used in most Unix kernels since a busy server indiscriminately drops high priority network packets [8]. Network QoS and its associated packet priorities are ineffective when the server drops such a packet. Supporting tiered services on a user or application basis requires server QoS mechanisms. Clearly the server must play an increasingly important role in delivering end to end QoS both to provide overload protection and to also enable tiered service support.

3 Architecture for Server Based QoS

Our WebQoS server architecture has focused on an application environment that consists of multiple nodes supporting web server, application server and database servers. Our philosophy was to create a low

overhead, scalable infrastructure that was transparent to applications and web servers.

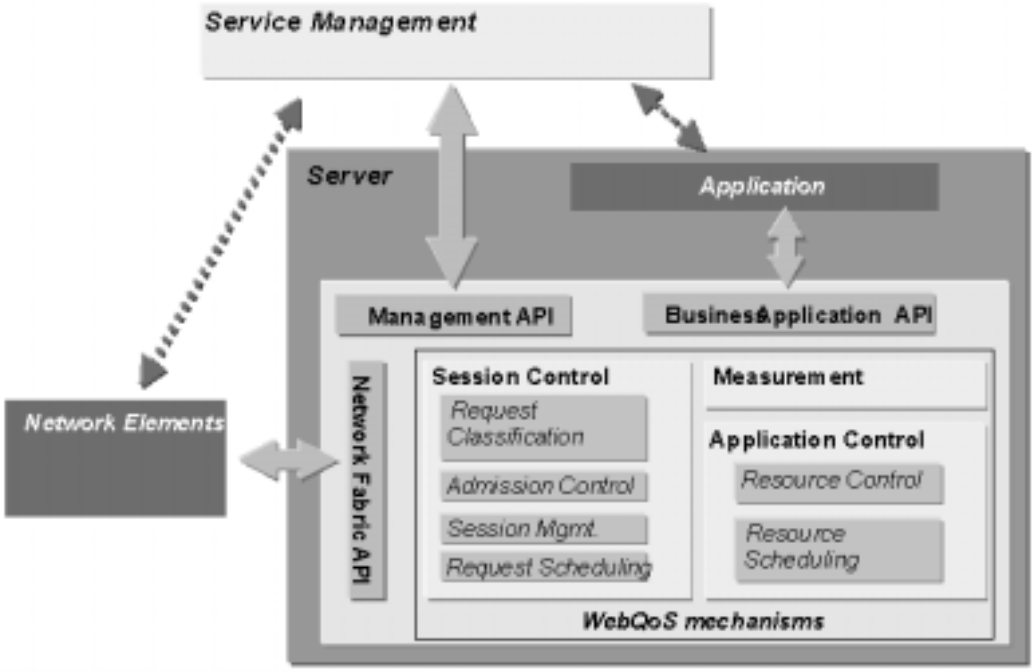

Figure 6: WebQoS architecture

Our goal was to support two key capabilities. First, the architecture must effectively manage peaks in client HTTP request rates. Second, it must support tiered service levels that enable preferential treatment of users or services. In order to support tiered service levels we observed that without classes best effort traffic competes equally with premium traffic for scarce resources. This results in poor performance for all users. Our solution uses scheduling and admission control to improve performance for premium tiers. The architecture is shown in figure 6. This architecture accepts incoming requests, reorders them based on scheduling policies, and transparently submits them to the web server for normal processing.

In order to process requests according to their importance the request's class must be determined. The *request classification* sets QoS attributes based on the filters and policies defined by the system administrator. Requests from *premium users* are classified as high priority and given preferential processing treatment. After a request is classified the request can be rejected or executed according to the scheduling policies appropriate to each class. The *admission control* component determines if this request should be processed. If so, then the classification attributes are used by *request scheduling* to determine how to enqueue and dispatch this request. *Session management* is used to provide session semantics and maintain session state for the stateless HTTP protocol. Additional mechanisms can be used to control resource allocation on the server. *Resource scheduling* ensures that high priority tasks are allocated and executed as high priority processes by the host operating system.

The architecture also supports integration with network QoS mechanisms such as reading and marking IP Type of Service (TOS) or Differentiated Services fields which can give drop preference behavior for

TCP connections. We also support integration with management systems so that WebQoS configuration parameters can be remotely set and internal measurements can be exorted for monitoring purposes.

We realized this architecture in a prototype that intercepts, classifies, queues and schedules the TCP/IP requests transparently to an Apache web server.

3.1 Related Work

Other efforts have been made to implement differentiated services on web servers. In [4], *Resource Containers* are used to account for and control consumption of operating system resources by different classes of requests [4]. This operating system control mechanism has been shown to ensure class-based performance in web servers. Another approach is the scheduling of web server worker processes with the same priorty as the request [2]. This ensures that the higher priority requests are executed first once they are assigned to a worker process.

External devices such as intelligent switches or routers can be placed in front of a web server to intercept web traffic. Alteon, Packeteer, and Local Director switch products can be configured perform request aware traffic shaping. Typically, these devices are used as load balancers for a cluster of web servers. Other switches can differentiate performance based on application type using admission control and bandwidth reservation [12].

4 Prototype

To achieve preferential treatment of different classes we modified the FIFO servicing model of a popular web server – Apache, version 1.2.4 [9]. This typical web server is composed of a collection of identical worker processes that listen on a UNIX socket for HTTP connections and serve requests as they are received.

In this section we discuss the major components of the prototype. These include a connection manager, request classifier, admission controller, request scheduler, and resource scheduler.

4.1 Connection Manager

We modified Apache by creating a new unique acceptor process, the *connection manager*, that intercepts all requests. This process classifies the request and places the request on the appropriate tier queue. The Apache worker processes receive requests from these queues instead of directly from the HTTP socket. A worker process selects the next request based on the scheduling policy. For example, work from the top tier will be processes first for priority scheduling. This bit of indirection allows our scheduling policies to decide which requests are processed first and which requests can be rejected. Figure 7 illustrates the design of the prototype.

Since all requests must be accepted by the connection manager it is essential that the connection manager runs frequently enough to keep request queues full. If the connection manager does not run frequently enough worker processes may execute requests from lower tiers because all the requests from higher tiers have been processed even though there may be top tier requests waiting to be accepted. This

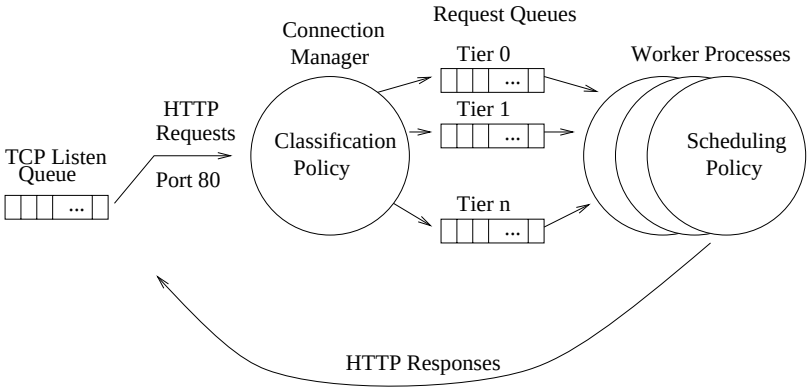

Figure 7: Tiered Services Design

results in a server that is more "fair" but may allocate server processes to lower priority work and thus make it difficult to quickly respond to newly arriving premium requests thereby violating the preferential treatment policy. In addition, if the connection manager does not run frequently enough requests may fill the TCP listen queue until its limit is reached and no other TCP connections will be allowed to the HTTP port; consequently, premium requests are prohibited from establishing a TCP connection and thus dropped.

4.2 Request Classification

A key requirement to support tiered services is the ability to identify and classify the incoming requests of each class. There are several ways to classify requests. These classification mechanisms can be divided into two categories, user-class or target-class based. User-class based classification characterizes requests by the source of the request, and target-based classification classifies by the content or destination of the request. Specifically, we support the following classification filters.

- User-class based classification
  - Client IP address is used to distinguish an individual client from another. This method is the simplest to implement. However, the client IP address can be masked due to proxies or fire walls, so this method has limited application.
  - HTTP cookies, a unique identifier sent to the browser, can be embedded in the request to indicate which class the client belongs to. For example, a subscription to a particular service is implemented as a persistent cookie. A cookie can also be used to identify a session that has been established for session based classification.
  - Browser plug-ins can also embed special client identifiers in the body of each HTTP request. Such a plug-in could be downloaded by clients who have paid for a subscription to premium service.
- Target-class based classification

- URL request type or filename path can be used to classify the relative importance of the request. In this case the sender of the request is irrelevant. Content can be classified as mission critical, delay sensitive or best-effort. This would allow e-commerce purchase activities, for example, to have higher priority than browsing activites.
- Destination IP addresses can be used by a server when the server supports co-hosting of multiple destinations (web sites) on the same node.

4.3 Admission Control

When the server processing rate falls behind the client demand rate the server becomes unresponsive to both premium and basic classes. To protect the server from high client loads some requests must be rejected. Naturally, basic requests rather than premium requests should be rejected first and existing sessions should be maintained. Admission control of basic requests is triggered when the server starts to be loaded.

We have two admission control trigger parameters one based on the ability of total capacity the server and the second based only on its ability to keep up with the performance demand of premium tiers. During our experiments we found that when too many total requests entered the server premium tier performance suffered so lower tier requests are rejected to shed load. The second triggers when there are too many premium requests waiting, since premium tier responsivenes is essential.

- *Total requests queued:* when the WebQoS environment reaches a programmable trigger point, *j*, the server will reject basic requests until the total number of queued requests falls below *j*. Note that this limit is based on all requests queued not on requests for any particular class.

- *Number of premium requests queued:* When the WebQoS environment reaches a programmable trigger point, *k*, the server will reject all basic requests until the number of queued premium requests falls below *k*. This rejection policy is sensitive to the waiting time of premium requests. As long as premium requests are not waiting basic requests can be accepted.

Rejection in the prototype is done by simply closing the connection which results in a "connection reset by peer error". A more client-friendly server could instead return a customized "sorry server is too busy, please try later" HTML response (although this will consume additional CPU and network resources that might be scarce during periods of overload).

4.4 Request Scheduling

Once requests are classified according to one of the above classification schemes and admitted by admission control then the server must actually realize different service levels for each class of requests. This is done by selecting the order of request execution. Workers are autonomous processes that select requests to process based on the scheduling policy. The scheduling policy may depend on queue lengths. Worker processes may be able to execute requests from any class, or to reserve capacity for higher class processes may be restricted to executing premium class traffic. Below we outline several potential policies. To preserve Apache worker process design, once a request has been selected it will run until completion. Only then is the worker available to process another request.

- *Strict Priority* schedules all higher class requests before lower class requests even when low priority requests are waiting.
- *Weighted Priority* schedules a class based on its weighted importance. For example, one class will get twice as many requests scheduled if its class weight is twice another's.
- *Shared Capacity* schedules each class to a set capacity and any non-used capacity can be given to another class. The class may also have a minimum reserve capacity that cannot be assigned to another class.
- *Fixed Capacity* schedules each class to a fixed capacity that cannot be shared with another class.
- *Earliest Deadline First* schedules based on the deadline for completion of each request. This can be used to give predicted response time guarantees.

4.5 Resource Scheduling

To provide even more control, the execution of requests several methods can be used to favor the execution of workers executing premium requests and retard the execution of basic requests. The server can set priorities on worker threads or processes to match the priorities of the request. The "nice" UNIX system call can give more CPU to higher tiers. Finally, the HP-UX kernel facility Process Resource Manager which is built upon a fair share scheduler can control each workers share of system resources. Each of these schemes can be either statically set up when a worker process is created or dynamically changed depending on the request class being executed.

4.6 Apache Source Modifications

In keeping with our philosophy of application transparency the number of Apache code changes is minimal. The `http_main.c` file has small modifications to start the connection manager process, setup queues (socket pairs), and change the child Apache process to accept requests from the connection manger process instead of directly from the HTTP socket. One additional file containing about 900 lines of C, `connection_mgr.c`, is linked when building the Apache server. This new file contains the classification policy, enqueue mechanisms, dequeue policies, and connection manager process code.

Additional shared memory and semaphores are required for our prototype. Shared memory is used for the state of the queues and contains each class queue length, number of requests executing in each class, last class to have a request dequeued, and the total count of waiting requests on all classes. Access to shared memory is synchronized through the use of a semaphore, and one additional semaphore is used to signal waiting workers when requests are available.

5 Results

In this section we evaluate the performance of the Apache based prototype. We give comparisons of response time, throughput, and error rates for premium and basic clients running with priority scheduling.

We compare the performance of premium and basic clients where the premium request rate is fixed and also with the premium request rate identical to the basic clients. We will demonstrate in both cases that premium clients receive much better service quality than basic clients. To further illustrate the point we compare a more real world metric, session throughput, of premium clients with basic clients.

Test results are based on four client machines simulating many clients issuing uniform size HTTP GETs. The experimental setup is as follows:

- **Server**: Single-processor K460 (PA-8200 CPU) running HP-UX 10.20, 512 MB main memory, 100-BaseT network connection. The size of the listen queue is set to 32.[1] Apache was configured to run with 32 worker processes.
- **Clients**: Four HP 9000 workstations (two 735/99 and two 755/99) running HP-UX 10.20, 100-BaseT network connection. `httperf` is used on each client to generate the workload [11].
- **Network**: 100-BaseT Ethernet through an HP AdvanceStack Switch 2000.

The four clients communicate with the server over the 100-BaseT Ethernet. On each client, the `httperf` application is configured to issue requests at a given fixed rate (e.g., 50 req/s) and for a given static web page of 8 KBytes for a 5 minute period. Each request has a 5 second timeout, and if no response arrives from the server within that time period, the client aborts the connection and logs an error. This effectively sets an upper limit (excluding connection setup time) on response delay as measured at the client. This will be evident in the graphs. At the end of the 5 minute period, statistics are collected at each client (number of successfully completed requests, average response time for these requests, number of requests which did not complete successfully) and at the server (CPU utilization). The client request rates are then increased and the process is repeated, yielding a graph which shows performance as a function of increasing offered load.

Our results are from the Apache prototype using premium tier strict priority scheduling i.e., lower tiers are only executed if no work exists in any higher tiers. We also configured admission control to trigger rejection when there are j or more total number of queued requests, or when there are k or more premium requests waiting. We evaluated the effect of these parameters with file requests for a variety of sizes and rates. Through experimentation we determined optimal j and k values for our server test configuration.

Three graphs illustrate experimental results for each test case. The first figure 8 plots the percent of client requests that encounter an error as a function of the total offered client demand rate (which is equal to the sum of premium and basic clients). The second graph, figure 9, plots the data throughput in Mb/sec as a function of the total offered client demand rate. Finally, the third graph, figure 10, plots the average response time in milliseconds for all completed client requests as a function of the total offered client demand rate.

In the first set of tests we analyze the errors, byte throughput and response time of a server handling one premium client at a fixed rate of 50 req/s and three basic clients with request rates that monotonically increase from 25-325 req/s. This results in a maximum combined client demand rate of 1025 req/s. All client requests are for a fixed object size of 8 Kbytes. This experiment models a situation where there are

[1] The listen queue is set with the listen() system call and when set with a value the actual queue length is 1.5 times that value. In this case we are setting a value of 48.

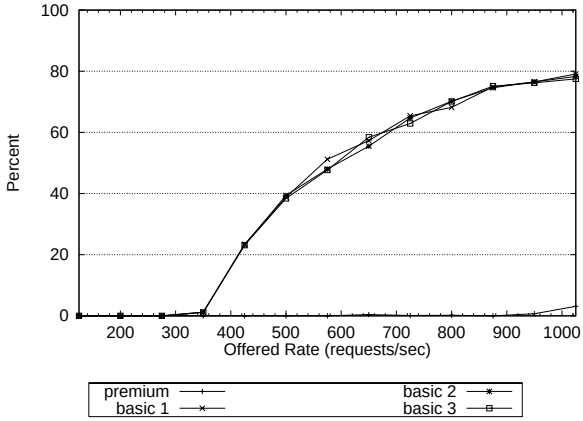

Figure 8: Errors, premium at 50 req/s

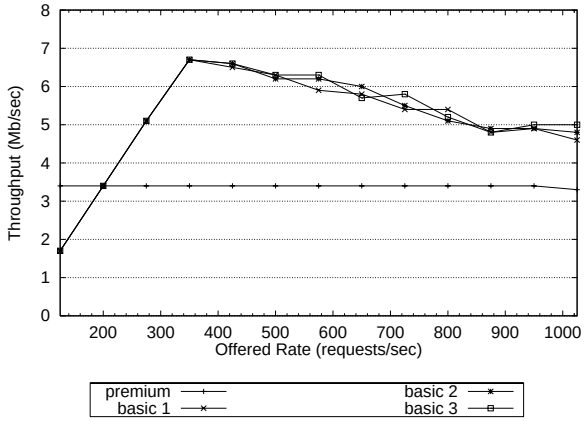

Figure 9: Throughput, premium at 50 req/s

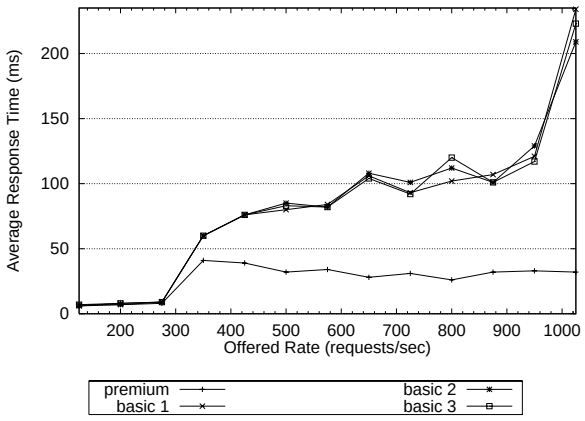

Figure 10: Response, premium at 50 req/s

a fixed number of clients with a subscription for premium service. We hoped to verify if a premium load can be protected while more and more basic traffic is added. As shown in figure 8 at low request rates the server can easily satisfy all client requests and premium and basic clients receive excellent service. As the offered load increases the basic clients consume more and more bandwidth until the server's capacity is exhausted at around 350 req/s. At this point the basic client requests start to be rejected to shorten the premium queue length and thus waiting time. Notice that the errors start to climb only for requests from basic service tiers as shown in figure 8. The premium client does not experience any errors until the very end of the test at 1025 req/s (which is nearly three times the maximum capacity of the server). At this point the premium client service experiences a small error rate of only 3% (note that the premium error rate may be difficult to read since it is at zero for most of this test). The premium throughput is very well protected as it appears as a straight line at 3.4 Mbps while the basic request throughput reaches a maximum at 350 req/s and then continues to drop despite an increasing request rate as shown in figure 9. In figure 10 the average response time for premium clients also is protected while the basic response time starts to climb sharply above 300 req/s.

The second performance measurements are for premium and basic clients that monotonically increase

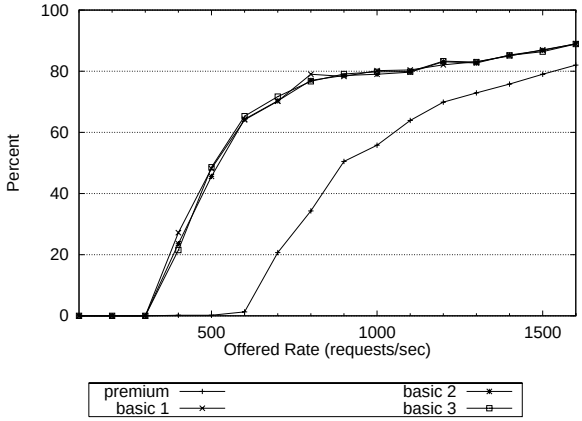

Figure 11: Errors

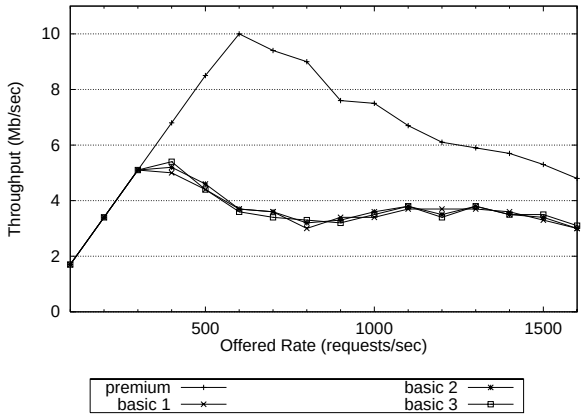

Figure 12: Throughput

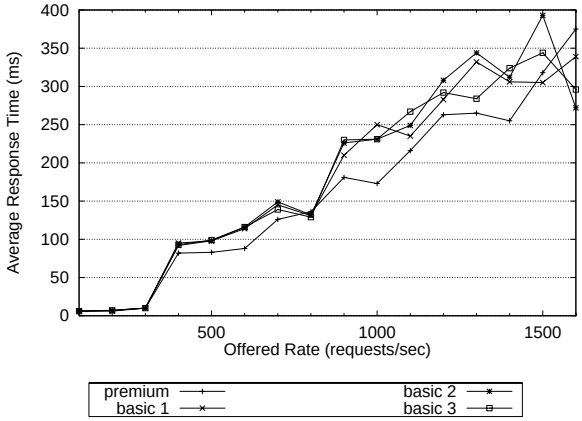

Figure 13: Response

with each client issuing an equal number of requests. These results are shown in figures 11, 12, 13. All four clients have the same request rates for a given point on each of these graphs. Thus, the proportion of basic to premium requests is 3:1. For this experiment resource allocation at the server is given preferentially to premium users and they always have a larger fraction even when the load is well beyond the maximum capacity of the server as shown in figure 12. The premium clients have better throughput and somewhat better error rates. The response time is somewhat misleading since basic requests are only processed when the server can process them quickly, otherwise basic requests are rejected and the response time is not counted. Thus the response time plotted here is optimistic since it does not include requests that are rejected or encounter an error. The error rates are not as different as we would like. All the errors for premium clients are due to timeouts which we set to 5 seconds in the client load generator.

Finally, the session completion times are shown in figure 14. This analysis builds upon the above results to evaluate a more complex interaction such as a page download or credit card payment. For this graph the session length is defined to be a sequence of N consecutive HTTP requests with minimal inter-arrival time. N ranges from 2 to 16. Completion time (i.e. total session duration) is plotted in milliseconds as a function of the total client request rate. The premium session completion times are the lower four

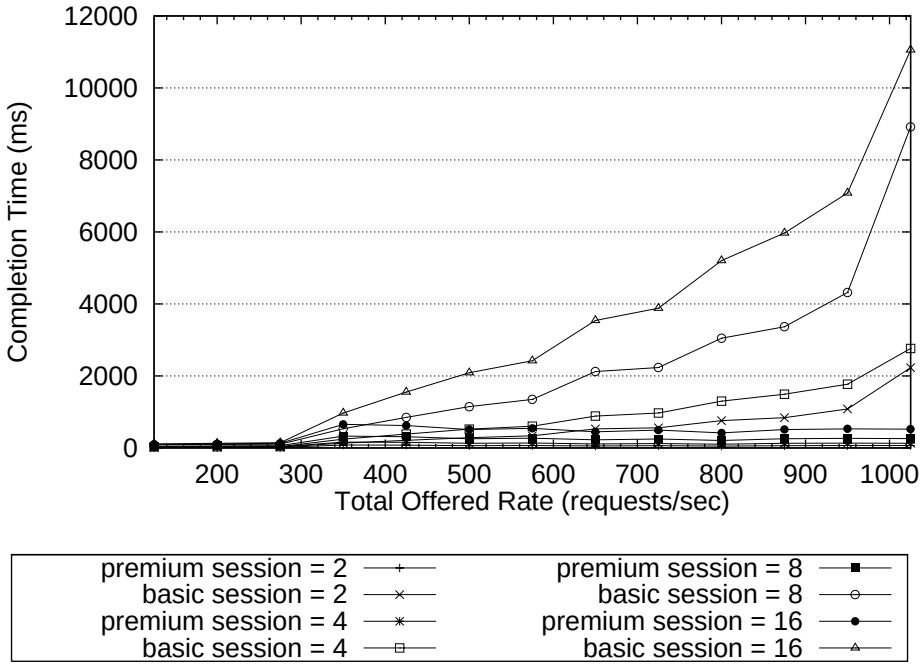

Figure 14: Session Completion Time

lines and the steeply climbing top four lines are the session completion times of basic client sessions. For a session length of 2 the premium and basic response times start to diverge above 350 req/s. At a request rate of 650 the premium session completion time is 60 milliseconds while the basic session completion time has grown to 530 milliseconds. Similarly, for a session length of 16 the premium and basic response times start to diverge above 350 req/s. At a request rate of 650 the premium session completion time is 450 milliseconds while the basic session completion time has grown to 3550 milliseconds. With more complex Web pages the differentiation between premium and basic session completion times will grow. Workloads measurements that we have conducted of major web sites indicate that a single page download may require as many as 50 HTTP requests so the service quality differentiation results in figure 14 are less than what a typical user is likely to encounter on the World Wide Web.

6 Summary

In this paper we have motivated the need for Server QoS mechanisms to support tiered user service levels and protect servers from client demand overload. We developed an architecture, WebQoS, that can support these QoS benefits transparently to a TCP/IP based application. Finally we discussed a prototype that realized the architecture and illustrated its benefits through experimental results. Specifically we demonstrated that it can provide preferential treatment to premium users for throughput, response time and error rates.

There are still many unsolved problems in providing tiered service levels end to end. The first is a tighter integration of server and network QoS and the ability to communicate QoS attributes across

network domains. The applicability of QoS routing to this problem also needs evaluation. Second, more flexible admission control mechanisms might employ a continuum of content adaptation instead of request rejection. This would result in the benefit of a server returning degraded content instead of rejecting a request [1]. Third, lightweight signalling mechanisms are needed to ensure that high priority traffic is given preferential treatment along the path between client and server. This is especially important for e-commerce workloads where TCP session times are very short. Finally, the implications of end to end QoS on devices at the edge of the network should be explored to determine what benefits can be derived from them.

Acknowledgments

The authors would like to thank the WebQoS Team – Martin Arlitt, John Dilley, Tai Jin, David Mosberger, Jim Salehi and Anna Zara – for their efforts in refining the notion of tiered web services and for creating the experimental testbed we used to evaluate our ideas. We are also indebted to Ed Perry, Srinivas Ramanathan and Preeti Bhoj for their ISP measurement techniques and results.

References

[1] Tarek Abdelzaher and Nina Bhatti. Web server qos management by adaptive content delivery. In *Seventh International Workshop on Quality of Service*, May 1999.

[2] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing different levels of service in web content hosting. In *Proceedings of the Internet Server Performance Workshop*, March 1998.

[3] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A survey of QoS architectures. *ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151, May 1998.

[4] Gaurav Banga and Peter Druschel. Resource containers: A new facility for resource management in server systems. In *Proceedings of 3nd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[5] Y. Bernet, R. Yavatkar, P. Ford, F. Baker, and L. Zhang. *A Framework for End-to-End QoS Combining RSVP/Intserv and Differentiated Services*. Internet Engineering Task Force, March 1998.

[6] S. Blake. *An Architecture for Differerentiated Services*. Internet Engineering Task Force, October 1998.

[7] C. Darst and S. Ramanathan. Measurement and management of internet services. In *Sixth IFIP/IEEE International Symposium on Integrated Network Management*, pages 125–140, Boston, MA, May 1999.

[8] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1997.

[9] Apache Group. http://www.apache.org.

[10] Jeffery Mogul and K. K. Ramakrishnan. Eliminating receive likelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, pages 217–252, August 1997.

[11] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *Proceedings of the Internet Server Performance Workshop*, pages 59–67, Madison, WI, June 1998. Association of Computing Machinery.

[12] V. Srinivasan, G. Vargehese, S.Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of SIGCOMM '98 Symposium*, Vancouver, Canada, September 1998. Association of Computing Machinery.