Supporting Quality Of Service in HTTP Servers

Raju Pandey J. Fritz Barnes Ronald Olsson
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{pandey, barnes, olsson}@cs.ucdavis.edu

Abstract

Most implementations of HTTP servers do not distinguish among requests to different pages. This has the implication that requests for popular pages have the tendency to overwhelm the requests for other pages. In addition, HTTP servers do not allow a site to specify policies for server resource allocation. This paper presents a notion of *quality of service* that enables a site to customize how an HTTP server should respond to external requests by setting priorities among page requests and allocating server resources. It also describes a design and an implementation of a distributed HTTP server, QoS Web Server, that enforces the quality of service constraints. The performance analysis of the prototype server indicates that the server provides the desired quality of service with minimal overhead.

1 Introduction

With the advent of the WWW [13], there has been a fundamental shift in the way information is exchanged among systems connected to the Internet. Three elements [26] of the WWW make this possible: a uniform naming mechanism (URL) for identifying resources, a protocol (HTTP) [2] for transferring information, and the client-server based architecture [17]. A client such as a browser uses the URL of a resource to locate an HTTP server that provides the resource. It then requests for services associated with the resource. The HTTP server performs the requested services (such as fetching a file or executing a program) and returns the results back to the client.

The architecture of HTTP servers has been studied in great detail and different variations of HTTP servers have been proposed. Much of the work has focussed on addressing the performance limiting behaviors [22] of HTTP servers. The research has, thus, focussed on developing techniques (such as information caching [7, 20, 9, 23] and distribution, partitioning [16] of server load across clients and servers, and parallelization [15, 4, 14, 18] of HTTP servers over SMPs and workstations) for eliminating performance bottlenecks arising due to the lack of sufficient CPU, disk, and network bandwidths as well as inherent limitations in the implementation techniques of HTTP servers.

While this has led to a deeper understanding of how HTTP servers operate when there are sufficient resources for various requests, not much work has been done in cases when HTTP servers are overwhelmed by the sheer volume of requests. The behavior of HTTP servers is quite unpredictable in such cases: they either completely bog down with pending requests resulting in unacceptable response times or start to drop requests indiscriminately. In addition, requests for popular pages have the tendency to overwhelm the requests for other, possibly more important, pages. Addition of new resources (such as new machines) may not solve the problem as requests for the popular page may continue to overwhelm other requests. Further, most implementations of HTTP servers treat all requests uniformly. A site, thus, cannot assign priorities to different pages or control how its server resources should be used. For instance, a site may wish to state that a set of specific pages (such as its main page or product page) be always available irrespective of the demands for other pages or that only 20% of its resources be allocated to anonymous `ftp` requests.

One possible mechanism for ensuring that requests for a collection of pages are guaranteed some server resources is to physically separate pages from each other by hosting them on separate servers. The problem with this approach is that it is difficult to map allocation of resources to various requests statically. First, such allocation may not be precise. Second, it may lead to inefficient utilization of server resources. Third, the granularity of such partitioning can be applied only to large groups of pages.

What is needed is a notion of *quality of service* (QoS) that characterizes the behavior of an HTTP server given a set of requests. This paper presents such a notion for HTTP servers and describes a design and an implementation of an HTTP server, QoS Web Server, that enforces the proposed quality of service model. Specifically, this paper addresses the following:

- *What is an appropriate quality of service model for HTTP servers?* The quality of service model presented in this paper is aimed at enabling a site to customize how an HTTP server should respond to external requests. This includes setting priorities among page requests, allocating different kinds (absolute and relative) server resources to different requests, and specifying constraints such as "always" which indicate that a specific page (or groups of pages) should always be available.

PODC 98 Puerto Vallarta Mexico
Copyright ACM 1998 0-89791-977-7/98/ 6...$5.00

- *How can such HTTP servers be implemented?* An implementation requires creation of a resource model for determining various resources that exist at any given moment. The paper describes an algorithm for scheduling various requests given a resource model such that the QoS constraints are satisfied.
- *What is the performance behavior of such servers?* We are interested in characterizing the execution behavior and responsiveness of HTTP servers. The results from the prototype server indicate that the implementation provides the desired quality of service with little overhead.

This paper is organized as follows: Section 2 presents a quality of service model for HTTP servers. Section 3 describes an abstract model of an HTTP server that implements this QoS model. It also includes the description of a distributed HTTP server that we have implemented. Section 4 presents an analysis of the performance behavior of the server. Section 5 contains a comparison of our work with related work. Finally, Section 6 summarizes the results and discusses future work.

2 A Quality of Service Model for HTTP Servers

The notion of quality of service has been addressed in great detail in the network and multi-media community [24]. Within a client-server framework, we can think of quality of service as a quantification of level of services that a server can guarantee its clients. Examples of typical parameters that servers have used to guarantee services are transmission delay, network transfer rate, image resolution, video frame rate, and audio or video sequence skew, among others. Clearly, the quality of service parameters depend on the kind of services that a server provides. In this section, we develop a model of quality of service for HTTP servers.

In traditional quality of service models, the emphasis has been on developing notions of service guarantees that a server can provide to its clients. For HTTP servers, we develop two views of the quality of service: client-based and server-based. In the client-based view, the HTTP server guarantees specific services to its clients. Examples of such quality of service are a server's guarantees on lower bounds on its throughput (for instance, number of bytes/second) or upper bounds on response times for specific requests. In the server-based view, the quality of service pertains to implementing a site's view of how it should provide certain services. This includes setting priorities among various requests and limits on server resource usages by various requests. We develop the QoS model by first constructing a model of client requests.

We model web pages as objects and requests to access pages as method invocations on pages. For instance, an invocation `<page>.read`$(p_1, p_2, \ldots p_n)$ denotes a request to read `<page>`. $p_1, p_2, \ldots$ and p_n denotes parameters of the request. An HTTP server, therefore, can be thought of as a runtime system that manages executions of various method invocations. Traditional HTTP servers do not distinguish among different method invocations. Each method invocation is serviced in the order it is received (unless it is dropped due to resource contentions [6]). The QoS model here allows one to specify priority relationships among method invocations. Further, a site may specify a set of resource usage constraints for controlling the amount of server resources allocated to requests.

Note that the constraints over different requests can be classified into two types: server-centric and client-centric. Server-centric constraints depend on the attributes of servers only. Such constraints do not distinguish among different requests to the same page. Hence, priority is established among requests for different pages. Client-centric constraints depend on the attributes of clients as well. Here, requests for the same page are distinguished and may be provided different quality of service. Our focus in this paper is on server-centric constraints only.

As part of the QoS model, we have devised a notation, which we call WebQoSL. WebQoSL supports specifications of the following:

- Allocation of specific and relative amount of server resources to specific page requests
- Availability of groups of pages at all time
- Time-based and link-relation-based allocation of resources
- Scalable allocation of resources
- Specification of guarantees about byte transfer and page request rates

Below, we provide a brief overview of the notation informally. We emphasize that WebQoSL is still evolving as we are still experimenting with the notation by implementing different kinds of quality of service models.

2.1 Specification of server resources

WebQoSL allows one to model server resources explicitly:

- Percentage of server resources

  Notation: Let the term `<page>.server_resource` denote percent of total server resources associated with requests to `<page>`.
- Requests/second

  Notation: Let the term `<page>.num_requests` denote number of requests per second associated with `<page>`.
- Number of bytes/second

  Notation: Let the term `<page>.num_bytes` denote number of kilobytes of `<page>` transmitted per second.

2.2 Specification of QoS constraints

A site specifies how its server resources should be allocated by defining a number of resource constraints of the form:

```
<condition> => <QoSConstraint>
```

The constraint specifies that if `<condition>` is true, the `<QoSConstraint>` must hold. Boolean expression `<condition>` can include specific attributes (such as time, size, owner, client, time of last access and time of last modification) of pages in constraint specifications.

QoS constraints for various requests can be defined as absolute, relative, scalable and time-bound. Absolute constraint are used to specify specific resources that are allocated to various requests. Relative constraints, on the other hand, allow one to assign various priorities among different requests. Scalable constraints allow QoS specifications to scale as new server resources (such as new machines) are added. Time-bound constraints allow one to specify constraints that have temporal characteristics (e.g., if page p is accessed at time t, page q should be available until time $t + \Delta t$.) Due to lack of space, we will only describe absolute QoS constraints here.

The absolute constraints are specified by allocating a specific amount of resources to various requests or putting a lower or upper bound on resources. For instance, the constraint

```
<page>.server_resource = r
```

specifies that <page> be allocated r units of resources. The constraint

```
<page>.server_resource < r
```

specifies that <page> be allocated at most r units of resources. The constraint

```
<page>.server_resource > r
```

specifies that <page> be allocated at least r units of resources. Another way of specifying a lower bound on resource allocations is to assert that a page should be available at all times.

```
<page>.available = always
```

The language also supports specification of allocation of default, equal and other scalable allocation of server resources.

Example 2.1. *(QoS Specification).* The following constraints

```
<www.commerce.com/free>.server_resource < 0.1
<www.commerce.com/paid/full>.server_resource > 0.5
```

are used to divide the server resources at www.commerce.com into two: free that can be given up to 10% of the server resources, and full that should be given at least 50% of the resources.

The next example specifies that a particular group of pages should always be available:

```
<www.commerce.com/index>.available = always
```

■

3 QoS Web Server

In this section, we describe an abstract model for the QoS Web Server. A distributed QoS Web Server is implemented in terms of a set of HTTP servers, each executing on a different host.

In figure 1, we show the architecture of a distributed QoS Web Server which is implemented in terms of five HTTP servers ($s_1, \cdots, s_5$) executing on different hosts. Each server responds to user's requests by accessing files from either the local disk or remote disk through the network file system and transmitting them to the client. We assume that a client can send a request to any of the HTTP servers directly by using one of the routing mechanisms (such as the Domain Name Server's redirection [8], ONE-IP mechanism [10] and router-based redirection [11]).

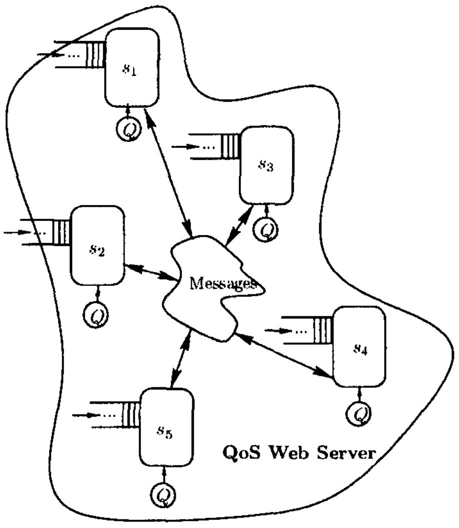


Figure 1: Architecture of a QoS Web Server

The primary goal of a QoS Web Server is to serve a file request only if servicing the requests does not violate the quality of service constraint that a site imposes. Each server, upon start, reads a file containing the quality of service specifications. It then constructs a *priority model* and a *resource requirement* model. The priority model defines a partial order among various requests to different pages and specifies the order in which requests should be handled. The resource requirements model, on the other hand, specifies the amount of resources that must be allocated to specific groups of requests. The servers then start to run and accept requests from clients.

Unlike the traditional HTTP servers where servers do not discriminate between various requests, a QoS Web Server must ensure that QoS constraints are met when requests are accepted. This is achieved by constructing a global model of resource availability and a global queue of all outstanding requests. The global resource model predicts the total amount of resources available at the hosts. The global request queue contains the pending requests. The priority model, global resource model, and global request queue are used to determine (i) the requests that will be granted service at this moment and (ii) the location of the server where a request will be executed.

We have implemented a version of a distributed HTTP server in which the global request queue and the resource model are centralized. Further, the algorithm for allocating resources is centralized as well. We describe the resource model and the HTTP server algorithm in Sections 3.1 and 3.2.

3.1 Resource model

This section briefly describes how we construct a resource model of the underlying system. The resource model specifies the capacity (in terms of bytes/second) of each HTTP server at a given moment. This provides an abstraction of resources (CPU, memory, network bandwidth) that the HTTP server can provide.

The resource model is evaluated by requiring that

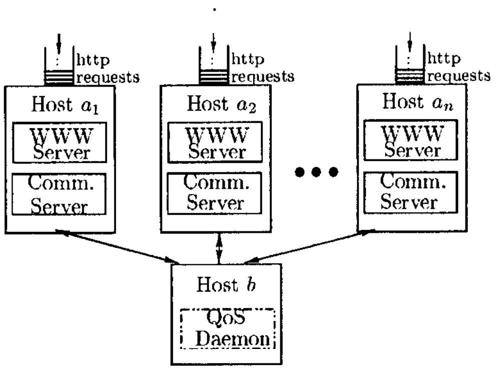


Figure 2: Architecture of the QoS Web Server

each HTTP server periodically determine the number of bytes/second it can deliver. Note that a machine's ability to serve a specific bandwidth depends on a number of factors: CPU speed, local CPU load factor, file server's capacity, file server's load factor, and local area network characteristics. In [19], an analytical model is created for evaluating the cost associated with accessing remote files, whereas in [25], the experimental technique used in the NFS benchmark (LADDIS) for evaluating the performance behavior of NSF servers is described. Both of these techniques can be extended to construct a resource model for the QoS Web Server.

Currently, we are using a simple experimental technique for constructing the resource model. We measure the length of time to send a request and use it to extrapolate the amount of bytes the HTTP servers can handle. This is performed as follows: Each HTTP server keeps a table of various local load factors and its capacity to access its local and remote files. In addition, the servers keep track of the average number of concurrent HTTP requests being served. Every time a job finishes, the table is updated and revised by calculating the transmission time. The total bandwidth is then calculated (approximately) by utilizing the average number of concurrent HTTP requests made during the interval. The average of the total bandwidth calculated by the recent jobs is then used to determine the total bandwidth for the server at a given CPU load.

3.2 An HTTP server

In this section, we describe the implementation of the QoS Web Server.

3.2.1 Architecture

We have implemented the QoS Web Server by modifying the NCSA's `httpd` web server. In figure 2, we show the architecture of the QoS Web Server. The QoS Web Server is implemented in terms of a set of components: a WWW server, a communications server and a centralized quality of service daemon (`qosd`). The WWW server is a modified version of the stand alone NCSA `httpd` WWW server [1]. It is used to handle individual HTTP requests. The modification in the NCSA server involves adding a check to ensure that a request is served only if the quality of service constraints are not violated. The modified server, therefore, sends a query to the `qosd` if the HTTP request should be handled. The `qosd` returns one of three values: handle the HTTP request, deny the HTTP request (because of QoS constraints), or redirect the HTTP request to a WWW server at a different host.

The *communication server* at a host performs two tasks: forwarding messages between the WWW Server at the host and the `qosd`, and implementing the resource model (Section 3.1). The communication server periodically transmits the WWW capacity to the `qosd` so that the `qosd` can update the global resource model. We have separated the communication server from the WWW server in order to avoid the overhead of initiating a new connection to the `qosd` every time an HTTP request is made. Also, the separation allows us to add new functionalities to the NCSA server without requiring extensive modifications in the NCSA server source code.

The *quality of service daemon* maintains global information for the distributed server and schedules HTTP requests. It maintains a quality of service model for various pages indicating priorities and resources associated with different requests, a global queue of outstanding HTTP requests, and a global resource model indicating the capacities of the WWW servers. We now describe how we use this set of information for implementing the `qosd`.

3.2.2 Implementation of the QoS daemon

The `qosd` first reads the QoS specification and constructs a QoS model. The QoS model defines categories or subsets of the document space and is used to associate an absolute or relative resource allocation with documents within the subset.

The `qosd` models each WWW server as a pipe capable of supporting a dynamic byte stream. It determines the capacity of the pipe in terms of number of bytes transmitted per second. Each WWW server periodically sends its projected capacity over the next allocation time unit to the `qosd`. Each pipe is further subdivided into smaller units, called *channels* (figure 3). A channel forms a connection between a server and a single HTTP client. It is the unit of allocation and resource control in the QoS Web Server.

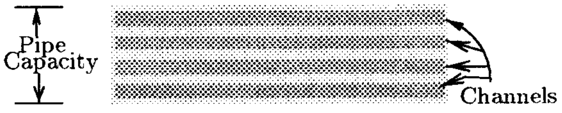


Figure 3: Pipes and channels

The size of each channel (in terms of bandwidth) is dependent on how many times we subdivide a pipe. For example, if a server indicates that it can serve 20 MB/second, the pipe size is 20 MB/second. Further, this pipe can be subdivided into 10 2 MB/second channels or 40 .5 MB/second channels. A channel with 2 Mb/second capacity is different from a channel with 0.5 MB/second capacity in that it can serve a request 4 times faster than the latter channel. The channel capacity has, thus, implications on response time. Our implementation allows a site administrator to specify the server response time for a given file of certain size[1]. The administrator can specify that a WWW page of size x

[1] The response time does not consider the latency and transmission costs across a wide area network.

should be served in time t. This can be handled by defining the channel size to be x/t.

The scheduling of HTTP requests is achieved by keeping track of two sets of requests: requests waiting to be serviced and currently being serviced. We first schedule all jobs in categories which should *always* be served. We then determine the number of remaining channels that can be allocated to requests with bounds on resource usage.

For each such category, we determine the number of channels available. We subtract from the number of available channels for this category the number of channels currently in use by requests in this category. This tells us how many channels we can allocate for new jobs in this category. We start jobs if we can start them on the server at which they arrived. After applying the algorithm, some categories may not have used all of their slots because the server at which the request arrived does not have any open channels. At this time the qosd redirects the request to a server with a free channel.

We assign all requests in the bounded quality of service category a lifetime. When a request surpasses a set age, QoS Web Server send a message to the HTTP client denying their HTTP request. Such a denial allows the QoS Web Server to put a limit on the implicit resources it allocates to various requests. For instance, each request occupies a space on request queue, holds a socket connection, and may even have a process assigned to it. By dropping connections, the server indicates that the request is not going to be assigned any resources in the near future as it is still trying to serve more important jobs.

4 Performance analysis

In this section, we present an evaluation of the QoS Web Server. The objectives of the evaluation are to address the following:

- How does the QoS Web Server perform for different kinds of resource constraints?
- What is the overhead of adding the notion of quality of service to HTTP servers?

4.1 Performance analysis environment

Our test environment consists of ten Sun workstations, consisting of a combination of Sparc 2, Sparc 5, Sparc 10, and Sparc 20 workstations. These workstations are connected on a local area network.

For the purpose of comparing results, we created a benchmark program based on ptester, a HTTP retrieval benchmark program included in the phttpd package [12]. The benchmark program takes as input a trace of requests and times, and uses the trace to send requests to the QoS Web Server. We generate traces that reflect specific or random mixes of various requests for different pages. All of our experiments, thus, were conducted on synthetic page requests.

The benchmark program is also responsible for calculating response times and storing the results for each request as to whether the request was accepted, was denied or failed. It allows reply of a trace of requests so that we can compare the behavior of the QoS Web Server under different configurations. The benchmark program is multi-threaded and distributed across multiple processes. This distribution is utilized in order to avoid limits due to the number of open sockets per process.

The tests were conducted on a local area network. As a result, the measurements obtained by these experiments provide a look at how to optimize the sending of pages from the Web Server's standpoint. They do not address issues related to the bandwidth of the network between the server and the clients.

4.2 Resource usage constraints

In this section, we present the set of experiments that characterize the behavior of the QoS Web Server with respect to different resource usage constraints. Specifically, our concern here is addressing the following issue: Does the QoS Web Server implement specified constraints on resource allocation to various requests? The experiments show that achieving the desired service specification depends on several facts:

- Our scheduling algorithm tries to satisfy resource constraints and, at the same time, utilize server resources effectively. Hence, if the QoS Web Server in not in contention, allocation of resources to various requests reflect the mix of the input requests. However, when the QoS Web Server is in contention, resources are allocated according to the constraints.
- In a given request mix, the QoS Web Server allocates a categories entire portion of resources only if there are enough requests in that category. For instance, the QoS Web Server can allocate 60% of its resources to requests for page A only if the requests are greater than 60% of the total QoS Web Server bandwidth.
- Channel size and request queue lifetime both affect how precisely the QoS Web Server can allocate various resources. Increasing channel size and lengthening the request queue lifetime increase accuracy but decrease response time.

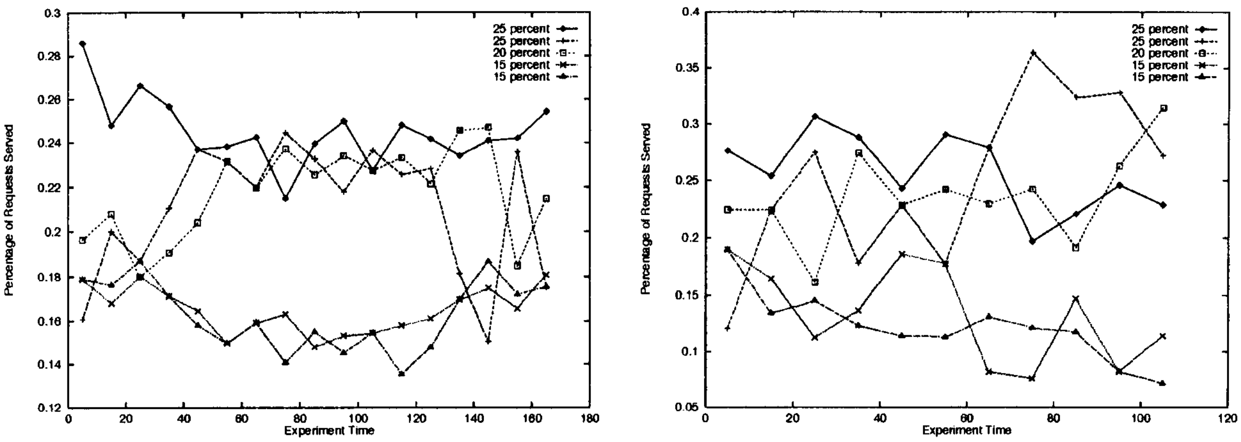
In the resource usage constraint experiments, we specify fixed percentages for jobs in a given category. We then randomly requests jobs from the different categories. Also, we utilize two to five categories of pages. We carried out the the various experiments by changing the following parameters: page size, resource usage constraints, queue lifetime and channel size.

4.2.1 Percentage requests handled

This experiment measures the number of pages served in each of the five categories over a ten second interval. We then calculate the percentage of pages served from each of the five categories.

In the first set of experiments, the benchmark program sends 18 requests per second for 16K files and 8 requests per second for 128K files. The life time for each request on the request queue was set to be 1/2 second. Figure 4(a) displays the results for files of size 16K; figure 4(b) displays the results for files of size 128K.

The graph shows the experiment time and plots the percentage of the server responses for the five different categories. The legend shows the resource constraints for various pages. As we can see, the server enforces the constraints on amount of resources that can be allocated to various pages.



(a) 16K Files

(b) 128K Files

Figure 4: Percentages of requests served for pages with different resource usage constraints

Note that there are some fluctuations in the percentage of pages served. The fluctuations arise primarily due to the randomness in the number of various category requests that arrive at the server.

4.2.2 Guaranteed service

In this experiment, we determine if the QoS Web Server can enforce resource constraints that specify that a set of pages should always be available. We request pages in five categories (A, B, C, D and E). We specify the constraint that A should always be available and that B, C, D and E receive 30%, 30%, 20% and 20% of the remaining server resources respectively.

We ran two sets of experiments: one for 16K pages and another for 128K pages. The results of the two experiments show that the QoS Web Server accepts 100% of A requests. In table 1, we show the percentages and numbers of requests accepted by the server for the two experiments.

Pages (constraints)	Experiment 1 (16K files)		Experiment 2 (128k files)	
	% served	# served	% served	# served
A (always)	100.0	704	100.0	298
B (30%)	90.0	538	59.7	172
C (30%)	84.1	530	62.2	173
D (20%)	52.6	339	42.4	123
E (20%)	54.5	354	45.5	122

Table 1: Performance behavior of server with *always* constraint

Note that the server accepts all requests for the guaranteed category. It denies about 750 requests in the 16K experiment and 500 requests in the 128K experiment for the remaining categories.

4.2.3 Different file sizes

We ran another set of experiments in order to analyze the behavior of the server when clients request files of different sizes. In this experiments, requests for files of sizes 16K, 32K, and 64K are respectively allocated 10%, 35%, and 55% of server resources.

The results of the percentages of requests handled in each of these categories are shown in figure 5(a). Instead, if we scale the results to measure the number of bytes served in each of these categories, the results appear as shown in figure 5(b).

Note that the percentage of bytes seems to match the QoS specification best. This matches our resource model that considers the resources of the server to be the bandwidth. Although, this fits better we also note that the larger file receives a disproportionate amount of the server resources. This is due to the diminishing effect of the constant overhead of making a connection to the server.

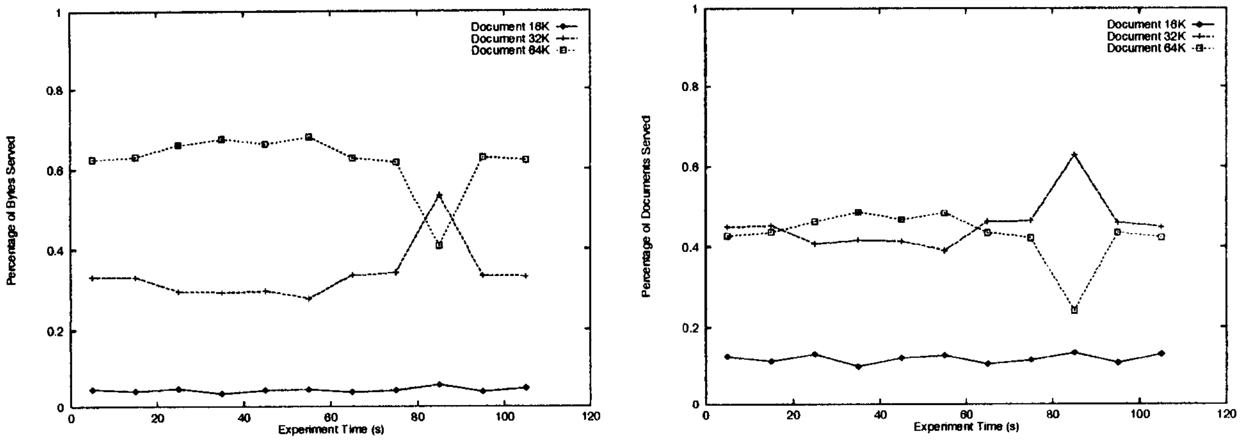
4.2.4 Flash crowds

In this experiment, we observe the behavior of the server when there is a drastic change in the number of requests for a specific page. This experiment aims to simulate the situation when there is high demand for a temporarily popular page. All file sizes are 15K and we create five categories each of which has a resource usage constraint of 20%. In this experiment, an equal number of requests arrive at the server at first. However, after 50 seconds, a large number of requests for page A arrives for the next 20 seconds. In figure 6(a), we show the request pattern for various requests.

In figure 6(b), we show the percentages of requests accepted by the server. Note that the percentages of requests served for A do not change.

4.2.5 Contention and non-contention behavior

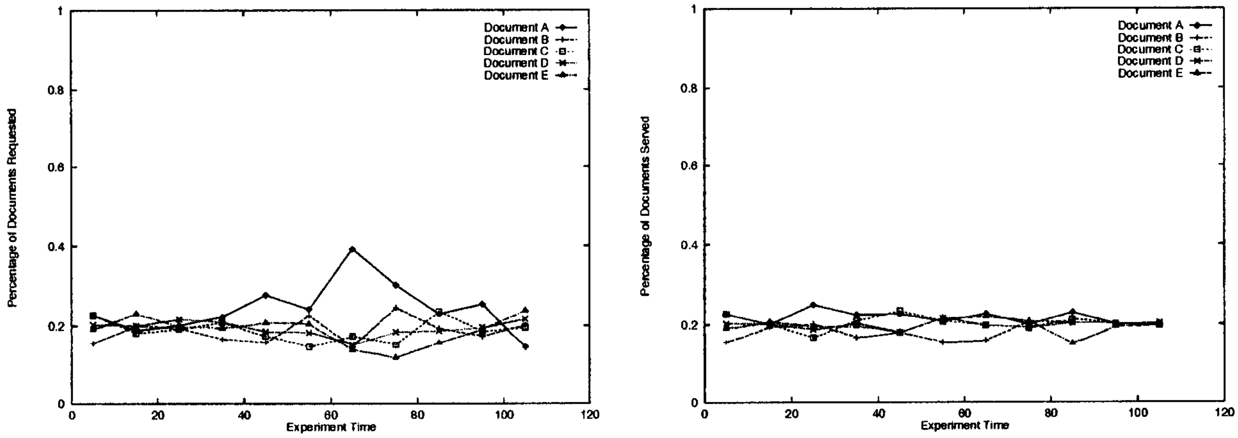
As we stated earlier, the scheduling algorithm in the QoS Web Server operates in two modes: if there is no contention, the server tries to optimally utilize resources by serving all requests. However, if there is contention, it enforces the



(a) Percentage of requests served

(b) Percentage of bytes served

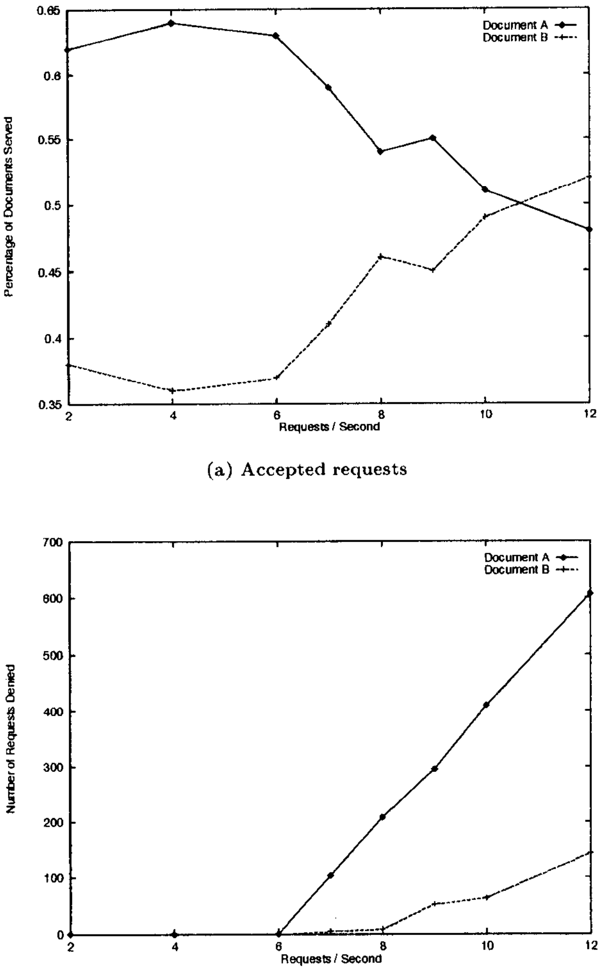
Figure 5: Behavior of server for requests of different sizes



(a) Request pattern

(b) Service pattern

Figure 6: Behavior of server with flash crowds



(a) Accepted requests

(b) Denied requests

Figure 7: Behavior of server with differing number of concurrent requests

resource constraints. This set of experiments shows how the behavior of the server changes when contention arises in the server.

In this experiment, clients request two files, denoted A and B. The size of each file is 128K. The incoming requests are a mix of 65% page A and 35% page B. The QoS specification assigns equal resources to both A and B. The request lifetime for each file is 1 second. In figure 7, we show the behavior of the server.

In figure 7(a), we show the percentages of requests of A and B accepted. Note that contention begins to occur at about 8 requests/second. At about 12 requests/second, the server is in full contention. Note that as long as there is no contention, the percentages of server's acceptances of A and B match those of the requests. However, as we reach contention, the percentages of server's acceptances start to match those of the resource specifications.

In figure 7(b), we show the number of requests denied to meet the resource constraints. As long as we are not under contention, no requests are dropped. However, when in contention the server begins to deny requests in a manner that attempts to satisfy the resource constraints.

4.3 Performance comparisons

In this set of experiments, we compare the performance behavior of the QoS Web Server with respect to the NCSA HTTP server which we modified. We have compared two characteristics of the servers: throughput and average response time. In the experiments here, the tester program requests 8 files every second. The size of the files is 128K.

In figure 8, we show the throughput of the two servers. For the NCSA server, it is about 0.78 M bytes/second. The throughput for the QoS Web Server ranges from 0.42 M bytes/second to about 0.7 M bytes/second. The graph illustrates two points: First, the throughput of the QoS Web Server is only marginally less than that of the NCSA server. Hence, the overhead of adding the notion of quality of service to an HTTP server does not cause the performance of the HTTP server to degrade significantly. Second, increasing the life time of requests on the request queue increases the throughput of the QoS Web Server up to some point. When the request life time is low, QoS Web Server rejects many requests which would have been granted resources. However, by rejecting these requests, the QoS Web Server wastes all resources (such as queue space, socket overhead, process creation and deletion overhead) it devoted to the requests. However, as the requests stay on the queue longer and longer, the probability that they will be served increases more, thereby leading to better utilization of server resources.

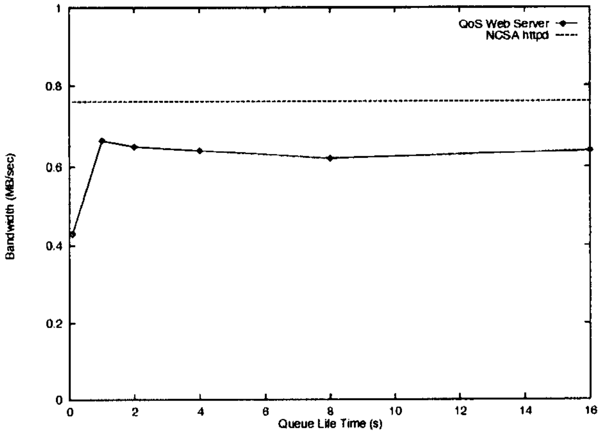


Figure 8: Comparison of throughput of NCSA and QoS Web servers

In figure 9, we show the average response times for the two servers. The lifetime for requests on the request queue is about 4 seconds. The graph highlights the fact that the average response time for the QoS Web Server remains fairly constant, whereas the response time for NCSA server is increasing. This is because the QoS Web Server drops all requests that it cannot serve after they stayed in the queue, whereas the NCSA server continues to accept requests even if it cannot handle them promptly.

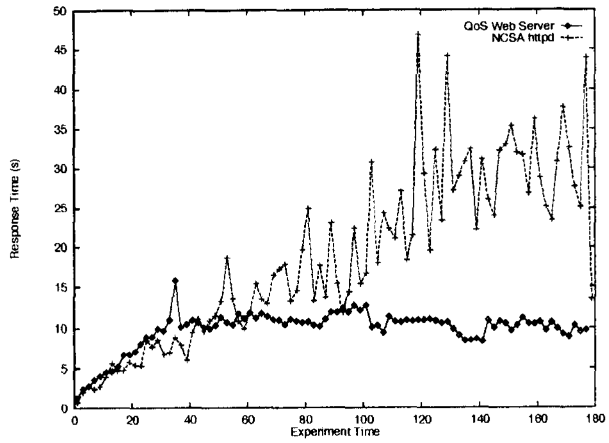


Figure 9: Comparison of average response times of NCSA and QoS Web servers

5 Related Work

There are two bodies of research with which our work overlaps: research on HTTP servers and research on quality of service in distributed systems. The focus in the first is on the design of HTTP servers, whereas the focus in the second is on developing various quality of service models and scheduling algorithms for supporting specific quality of service guarantees.

5.1 HTTP servers

As we described earlier, the primary goal of an HTTP server is to service requests for web pages. Much of the HTTP server work has focussed on developing variations of HTTP server architectures that reduce the CPU, network, and disk bottleneck. We will focus only on the distributed HTTP server work [15, 14, 4] because of the similarity in the issues addressed by these approaches and our approach. The focus in the distributed server research has been on using the resources of distributed hosts to increase the throughput of HTTP servers. Most of the research here has been aimed at addressing the notion of load balancing and scalability: given a request, how should the server schedule this request so that resources on the distributed hosts are optimally utilized. Our work, on the other hand, addresses additional issues in the design of HTTP servers:

- Should the server accept a request?
- If so, how much resources should be allocated to the request?

There has been some work that looks at the notion of quality of service for HTTP servers. [3] proposes a notion of quality of service by associating priorities with requests from different sites. The HTTP server schedules requests according to priorities, thereby ensuring that preferred sites (with higher priority) are allocated resources before other sites. Our work differs in many ways: first, the focus in [3] is on proposing techniques for structuring single host HTTP servers in order to improve the response times of high priority requests. Our work primarily involves distributed HTTP servers. Second, our notion of quality of service is more general in that we not only allow a site to specify priorities but also allow it to specify resource usage constraints on a group of requests. In [5], a notion of quality of service is proposed with respect to the content. However, there is not support for any notion of quality of service with respect to resource usage, throughput or response time.

5.2 Quality of service in Distributed Systems

The notion of quality of service [21] has been studied in great detail within the context of networking [21] and multimedia [24]. The focus of work here has been on developing varying level of services (including low-level notions such as number of bytes/second to high-level notions such as jitter-free play of images etc.) and on developing algorithms for scheduling CPU, memory and networking resources such that the quality of service guarantees are met. In [27] mechanisms for specifying service guarantees with method invocations of CORBA objects is presented. Our work is similar to these works in that we also associate quality of service with resources in order to schedule resources. However, our work differs from them in the nature of resources (web pages), in terms of constraints on usage of resource and how they should be scheduled.

6 Summary

We have presented the design and implementation of a distributed HTTP server that implements a quality of service model. In this model, a site can determine how requests for various pages should be served. This includes setting priorities among the requests as well as associating constraints on resource usages. Resource usage constraints provide a useful tool for providing services on the WWW. They support the ability to guarantee documents and set desired performance characteristics by denying requests rather than serving all requests at the same time.

We have also analyzed the performance characteristics of the QoS Web Server. The analysis shows that the server enforces user specifiable constraints on resource usages. Further, the performance behavior of the server is comparable to that of the standard NCSA HTTP server.

Our future work involves formalizing WebQoSL, refining the resource model, and implementing a distributed version of the qosd.

References

[1] NCSA Web Server Source. ftp://ftp.ncsa.uiuc.edu/ Web/httpd/Unix/ ncsa_httpd/httpd_1.5.2a/httpd_1.5.2a-export_source.tar.Z.

[2] Hypertext Transfer Protocol (HTTP): A protocol for networked information. http://www.w3.org/hypertext/WWW/Protocols, 1995.

[3] Almeida, J., Dabu, M., Manikutty, A., and Cao, P. Providing Differentiated Levels of Service in Web Content Hosting. Tech. rep., University of Wisconsin-Madison, 1998.

[4] Andersen, D., Yang, T., Egecioglu, O., Ibarra, O., and Smith, T. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. In *Proceedings of*

the Tenth IEEE International Symposium on Parallel Processing (1996), IEEE Computer Society, pp. 139–148.

[5] Banatre, M., Issamy, V., Leleu, F., and Charpiot, B. Providing Quality of Service over the Web: A Newspaper-based Approach. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[6] Banga, G., and Druschel, P. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems,Monterey, California, USA* (December 1997).

[7] Bestavros, A. WWW Traffic Reduction and Load Balancing Through Server-Based Caching. *IEEE Concurrency* (1997), 56–67.

[8] Brisco, T. DNS Support for Load Balancing. *Network Working Group, RFC 1794* `http://andrew2.andrew.cmu.edu/rc/rfc1794.html`.

[9] Caughey, S., Ingham, D. B., and Little, M. C. Flexible Open Caching for the Web. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[10] Damani, O. P., Chung, P. E., Huang, Y., Kintala, C., and Wang, Y. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[11] Dias, D., Kish, W., Mukherjee, R., and Tewari, R. A Scalable and Highly Available Server. In *COMPCON* (1996), pp. 85–92.

[12] Eriksson, P. The PHTTPD World Wide Web Server. `http://www.signum.se/phttpd`.

[13] et al., T. B.-L. The World-Wide Web. *CACM 37*, 8 (August 1994), 76–82.

[14] Garland, M., Grassia, S., Monroe, R., and Puri, S. Implementing Distributed Server Groups for the World Wide Web. Tech. Rep. CMU-CS-95-114, Carnegie Mellon University, 1995.

[15] Katz, E., Butler, M., and McGrath, R. A Scalable HTTP Server: The NCSA Prototype. In *Proceedings of the First International Conference on the World-Wide Web* (May 1994).

[16] Kong, K., and Ghosal, D. Pseudo-serving: A User-Responsible Paradigm for Internet Access. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[17] Kwan, T. T., McGrath, R. E., and Reed, D. A. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer* (November 1995), 68–74.

[18] Liu, Y., Dantzig, P., Wu, C. E., Challenger, J., and Ni, L. M. A Distributed Web Server and Its Performance Analysis on Multiple Platforms. In *Proceedings of the The Sixteenth International Conference on Distributed Computing Systems* (1996), pp. 665–672.

[19] Melamed, A. S. Performance Analysis of Unix-based Network File Systems. *IEEE Micro* (February 1987), 25–38.

[20] Nabeshima, M. The Japan Cache Project: An Experiment on Domain Cache. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[21] Nahrestedt, K., and Smith, J. M. The QoS Broker. *IEEE MultiMedia Magazine* (Spring 1995), 53–67.

[22] Prefect, F., Doan, L., Gold, S., Wicki, T., and Wilcke, W. Performance Limiting Factors in HTTP (Web) Server Operations. In *COMPCON* (1996), IEEE, pp. 267–272.

[23] Scheuermann, P., Shim, J., and Vingralek, R. A Case for Delay-Conscious Caching of Web Documents. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[24] Vogel, A., Kerhervé, B., von Bochmann, G., and Gecsei, J. Distributed Multimedia and QoS: A Survey. *IEEE MultiMedia 2*, 2 (1995), 10–19.

[25] Wittle, M., and Keith, B. E. LADDIS: The Next Generation of NFS File Server Benchmarking. In *Proceedings of the USENIX Summer 1993 Technical Conference* (June 1993), Usenix.

[26] Yeager, N. J., and McGrath, R. *Web Server Technology: The Advanced Guide for World Wide Web Information.* Morgan Kaufmann, 1996.

[27] Zinky, J. A., Bakken, D. E., and Schantz, R. D. Architectural support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems 3*, 1 (1997), 1–20.