# Application-Level Differentiated Services for Web Servers

Lars Eggert and John Heidemann

USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6695 USA
larse@isi.edu, johnh@isi.edu

September 22, 1999

USC Technical Report 99-695
In *World Wide Web Journal*, Volume 3 (1999), Issue 2, pp. 133-142

## Abstract*

*The current World-Wide Web service model treats all requests equivalently, both while being processed by servers and while being transmitted over the network. For some uses, such as web prefetching or multiple priority schemes, different levels of service are desirable. This paper presents three simple, server-side, application-level mechanisms (limiting process pool size, lowering process priorities, limiting transmission rate) to provide two different levels of web service (regular and low priority). We evaluated the performance of these mechanisms under combinations of two foreground workloads (light and heavy) and two levels of available network bandwidth (10Mb/s and 100Mb/s). Our experiments show that even with background traffic sufficient to saturate the network, foreground performance is reduced by at most 4-17%. Thus, our user-level mechanisms can effectively provide different service classes even in the absence of operating system and network support.*

## 1. Introduction

The World-Wide Web is a typical example of a client/server system: in a web *transaction*, clients send *requests* to servers, servers process them and send corresponding *responses* back to the clients. Concurrent transactions with a server compete for resources in the network and server and client end systems. Inside the network, messages contest for network bandwidth and with other messages flowing between the same end system pair and with other traffic present at the time. Inside the end systems, transactions compete for local resources while being processed. Servers implementing the process-per-request (or thread-per-request) model will allocate one process (or thread) to an incoming request.

The current web service model treats all transactions equivalently according to the Internet best-effort service [Clark 1988]. Neither the network nor the end systems typically prioritize traffic. However, there are cases where having multiple levels of service would be desirable. Not all transactions are equally important to the clients or to the server, and some applications need to treat them differently. One example is prefetching requests for web pages by proxies; such speculative requests should receive lower priority than user-initiated, non-speculative ones. Another simple example is a web site that wishes to offer better service to paying subscribers. We explore these and other examples in Section 2.

Ongoing efforts attempt to provide multiple levels of service, both in the server operating system (OS) and in the network (see Section 6). Although promising in the long run, replacing the OS of end systems or upgrading all routers in the network is often impractical. Instead, we will show that substantial benefit can be achieved with server-side, application-level-only mechanisms.

We have designed and implemented three simple server-side, application-level mechanisms that approximate a service model with two levels of service, in which high-priority responses preempt low-priority ones. The key characteristic of such ideal *background* responses is that their presence in the system never decreases the performance of concurrent *foreground* transactions. This is approximated by slowing down the serving of background responses to make more resource capacity available to the average foreground response. Our results show that our most effective mechanism has an over-

* This research is supported by the Defense Advanced Research Projects Agency (DARPA) through FBI contract #J-FBI-95-185 entitled "Large Scale Active Middleware". The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of the Army, DARPA, or the U.S. Government. The authors can be contacted at 4676 Admiralty Way, Marina del Rey, CA 90292-6695, or by electronic mail at `larse@isi.edu` or `johnh@isi.edu`.

head on foreground performance of only 4-17%. This indicates that it is possible to provide effective background data traffic service even without network-level or operating-system-level support.

## 2. Three cases for differentiated services

This section describes three cases where multiple levels of service for web transactions are needed. The first example is a web server offering *less-effort* serving of background requests. The second example is a web server that assigns different priorities to responses based on the requested object. In the third example, response priorities are assigned based on an external policy.

### 2.1. Background requests and responses

Background transactions are low-priority transactions that are preemptable. The key characteristic of a background transaction is that its presence in the system never decreases the performance of concurrent foreground transactions. This may be achieved by only transmitting or processing it if enough idle resource capacities are available. If not, a background transaction may be indefinitely delayed or dropped. Thus, background transactions receive *less-effort* service.

One application that would greatly benefit from the availability of background transactions is anticipatory caching (for example, [Touch 1998]). Currently, speculative transactions and pushes can only be sent as regular (foreground) traffic, and may thus interfere with non-speculative traffic. Caches using speculative transactions (prefetching) and servers using speculative pushes need to balance the amount of speculative traffic sent against possible future traffic reduction due to cache hits. If such transactions could be serviced in the background, interference with non-speculative traffic could be eliminated. This would lead to a better overall system performance, as well as a simplified cache system, because the penalty of sending too much speculative traffic would be greatly reduced.

One example of a cache using speculative pushes is the LSAM Proxy Cache [Touch and Hughes 1998]. It uses background multicasts of related web pages, based on automatically-selected interest groups, to load caches at natural network aggregation points. The proxy is designed to reduce server and network load, and increase client performance. Other applications that would benefit from the availability of background processing include data-driven push [Touch 1995], subscription push [Pointcast 1998], web prefetching [Padmanabhan and Mogul 1996] and TCP pacing [Visweswaraiah and Heidemann 1997; Padmanabhan and Katz 1998].

### 2.2. Content-derived priorities

Having different levels of service may improve user-perceived rendering time of web pages by sending HTML responses at a higher priority than all others. The second example is a web server assigning different priorities to responses based on the requested objects.

A typical web page consists of both HTML parts (one or more frames) and inline images. For each of those parts, one request will be issued by the client more or less concurrently. These requests may compete for resources inside the network [Balakrishnan *et al.* 1998] and at the end systems. If the transaction uses HTTP 1.0, the responses will typically be sent as an ensemble of TCP connections, which will compete for bandwidth along the path back to the client. If HTTP 1.1 is used, the responses will be sent over a single shared connection, but since responses cannot be interleaved, there will still be competition for the order in which they will be sent. Thus, image responses may interfere with HTML responses. However, HTML responses are more important to a browser, because they drive the rendering of the whole page. The server could reflect this by giving priority to delivering HTML over images.

In this example, the requested content controls the priority of a transaction. Even though transactions have different priorities, none are expendable; all of them must be processed.

### 2.3. Policy-derived priorities

In the previous case, transaction priorities were derived from the type of the requested object. Different levels of service are also useful when priorities are assigned according to an external policy.

Consider the example of a web site offering information both to paying subscribers and the public. Transactions by paying customers should be favored over those of nonpaying ones by serving the former at a higher priority. Here, transaction priorities are assigned depending on the requester. A second example, where a different policy is enforced, is a web hosting service managing multiple sites on the same end system. Here, the hosting service might want to guarantee its clients' sites receive outgoing bandwidth proportional to the amount of money payed. Thus, transaction priorities would be assigned based on the requested object.

In these two simple examples, external (management) policies control priority assignments. Depending on the nature of the policy, it may or may not be acceptable to delay or drop transactions.

## 3. Finding the server bottleneck resource

In the previous section, we have described several cases in which different levels of service for web transactions are useful. The first step in designing an effective background processing (*backgrounding*) mechanism is to locate the bottleneck resource of the system. Control of the bottleneck resource has primary influence on overall system behavior by granting or not granting the resource to processes. For example, in a CPU-bound system, a process that is not being granted the CPU cannot use other resources; thus, CPU-scheduling controls system performance. In the same scenario, network scheduling would have little effect on performance. A successful backgrounding mechanism will control the scheduling decisions of the bottleneck resource to optimize performance.

Any resource of a web server (CPU, physical memory, disk, network) may become the bottleneck, depending on the kind of workload it is experiencing. We evaluated the bottleneck resource in two web serving scenarios: a web server connected to its clients by private, non-switched 10Mb/s and 100Mb/s Ethernet links. We conducted experiments to determine which server resources became saturated first. The server was monitored under a growing request load generated by an increasing number of clients, each of which made requests at a fixed rate of (at most) ten requests per second. The aggregate request load exceeded 1200 requests per second, which was more than enough to fully load the server.

The server machine was a 300Mhz Pentium-II PC with 128MB of physical memory running FreeBSD 2.2.6. The kernel had been optimized for web serving [Apache HTTP Server Project 1998a] by increasing the socket listen queue to 256 connections and increasing the MAXUSERS kernel parameter to 256. We modified the Apache version 1.3 beta 1 web server [Apache HTTP Server Project 1998b] to collect CPU, physical memory, page fault and physical disk I/O statistics. The server load was generated by a version of Webstone-1.1 [Trent and Sage 1995] that we modified to gather more extensive per-request statistics. Each point in the graphs below is based on data gathered during a five minute period in which several thousand requests were processed. No other traffic was present during the experiment. Network utilization could therefore simply be measured by the amount of data transferred in a test period.

During both experiments, requests were made over the standard Webstone file set, which is about 2MB in size and is modeled after a small, static web server. The entire file sets easily fit into the disk buffer cache of our server. Thus, repeated requests for the same file were always served from the cache. Consequently, the disk subsystem was mostly idle. Furthermore, all pages were static, i.e. no additional server-side processing (CGI scripts, database queries, etc.) was done. Characterizing dynamic web workloads is still an area of study. We consider how this affects our conclusions in Section 5.3.

### 3.1. Results for 10Mb/s Ethernet

The results for the 10Mb/s Ethernet case show that the server was network-bound during this experiment. In the left graph of Figure 1, HTTP transaction throughput is plotted over the number of clients. Throughput quickly reached 7Mb/s and then settled around that number. A single bulk TCP connection can achieve around 7.6Mb/s over the same link (measured with `netperf` [Netperf Project 1998]).

All other monitored resources were mostly idle: The server CPU utilization (right graph of Figure 1) was never higher than 25%. Server memory was never fully utilized; we observed no page faults during the experiment. The disk subsystem was also idle; there were no physical (not served from the buffer cache) disk inputs. The disk output rate peaked at around 10 physical disk writes per five minute test period, all of which were due to logging. The local file system can sustain several thousand physical disk writes per second at less than 25% CPU utilization, so the measured rate is not significant.

### 3.2. Results for 100Mb/s Ethernet

For 100Mb/s Ethernet, the server was CPU-bound. The right graph of Figure 1 shows that the server CPU utilization rose rapidly to around 95%. Network throughput stagnated at around 30Mb/s, (left graph of Figure 1) which is well below the 72.1Mb/s (measured with `netperf` [Netperf Project 1998]) that a single bulk TCP connection can achieve over the same link. The server was clearly not network-bound. We believe the relatively low network throughput to be an artifact of the Webstone benchmark, which only supports HTTP 1.0 and will thus open a new TCP connection for each transaction, causing significant CPU overhead.

As in the 10Mb/s case before, we did not observe any page faults or disk input operations. The measured physical disk output rate never exceeded 50 writes per five minute test run; as explained in Section 3.1, this rate is not significant.

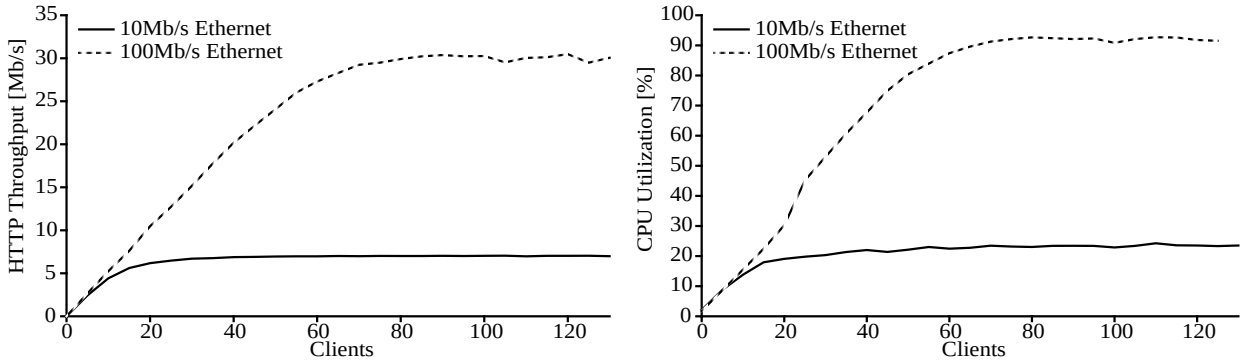


**Figure 1.** HTTP throughput and server CPU utilization over both 10Mb/s and 100Mb/s Ethernet.

## 4. Designing application-level background processing

As mentioned above, transactions compete for resources inside the network and at the end systems. Thus, full support for different levels of service for web transactions would require both network and end system software (OS and applications) to be extended. These extensions are still under development; and even when finished, deployment will take time, because many routers in the network must be updated for the system to be effective. In the meantime, application-level mechanisms promise most of the benefits of a OS/network solution with the additional advantage of being easy to deploy. Only the application software of the server needs to be modified to offer different service levels.

We have designed and implemented three server-side, application-level background processing mechanisms that approximate a service model with two classes: Regular *foreground* transactions, and preemptable, lower-priority *background* transactions. We assume a process-per-request model, with pools of *foreground processes* and *background processes*. (Our results also apply to thread-based servers, and our third and most effective mechanism can be implemented in an event-driven server.) All processes in one such class form the *foreground pool* and *background pool* of server processes, respectively. Since we implemented server-side-only mechanisms, requests are always being sent in the foreground; our mechanisms can only control processing and sending of the responses. The idea of background processing can also be applied to clients (see Section 5.3).

The key idea behind all our application-level backgrounding mechanisms is to slow down the background pool, thus making more resource capacity available to the average foreground process. Our three mechanisms differ in how they slow down background processing. We assume that the request stream is demultiplexed by the OS before reaching the server; the server application has two queues from which to accept foreground and background requests.

Our first mechanism limits resource usage of background processes by limiting concurrency. This is achieved by imposing an upper bound on the number of processes in the background pool. If all background processes are busy, additional incoming background transactions are delayed (in the OS) until a background process becomes available. No such bound is enforced for the foreground pool, and consequently the average foreground transactions will experience less delay than background ones under an increasing background load. The size of the background pool is a parameter tunable by the administrator of the web server, based on the allowable overhead on foreground traffic. We picked a value of five background servers. Fewer background servers would result in less background traffic, which would make it difficult to compare the overhead of the backgrounding mechanisms. Using many more than five would diminish the differences between foreground and background traffic classes.

This first backgrounding mechanism could even be implemented without changing the server code, simply by running two web servers configured with different pool sizes on the same machine. These servers would need to serve the same documents, but accept connections on different ports.

Our second backgrounding mechanism also limits the size of the background pool, but in addition also lowers the process priority of the background processes to the minimum. For CPU-bound servers, this approach should produce better control than the first.

The two prior mechanisms directly reduce CPU usage only. Usage of network I/O and other resources is only indirectly controlled. Our third mechanism limits the aggregate network transmission rate of background processes by coordinating and scheduling their send opera-

tions. Background processes intentionally slow their transmission, monitoring and explicitly pacing their sending rate by pausing while sending. Multiple background processes collaborate to split the limit fairly. The rate limit is a parameter tunable by the administrator of the web server, based on the permissible overhead on foreground traffic. We picked a rate limit of 1Mb/s. As with the first mechanism, a significantly lower value would make comparisons of the backgrounding mechanisms more difficult, and a much greater value would diminish the differences between the two traffic classes.

Our third mechanism also limits the size of the background pool to five processes running at the lowest process priority. Note that limiting the background pool in this scenario is not necessary to enforce service differentiation; that is established through the send rate limit. Here, limiting the background pool will simply control the send rate for each response: With only one background process, background responses will be sent at full rate limit (but only one at a time); with more than one, multiple background responses will be sent, each at a fraction of the rate limit. Lowering the process priority is also not strictly necessary, but since it is an extremely simple addition, we included it in the mechanism.

One problem with the third approach is that even if the network is underutilized, the background processes can never exceed the rate limit, because they have no means of detecting idle network capacity. However, background transactions are not important by definition, so serving them at less-than-peak performance is appropriate. More elaborate rate-limiting algorithms (see Section 7) may solve this limitation.

None of our three background processing mechanisms rely on OS-level or network-level support for QoS. However, if such support was available, they could all be easily modified to take advantage of such mechanisms.

## 5. Background processing evaluation

We implemented the three background processing mechanisms described above in Apache version 1.3 beta 1 [Apache HTTP Server Project 1998b]. The server ran on the same machine as during the bottleneck resource experiments (see Section 3). Foreground and background transactions were generated by two synchronized Webstone [Trent and Sage 1995] benchmarks, each with several clients. Foreground load was kept at a fixed level during an experiment while increasing background load over time. We expect that increasing the background load will reduce foreground performance in a basic system. By introducing specific background processing mechanisms, we attempt to reduce foreground performance degradation.

To quantify the effect of background traffic on foreground load, we measured the response time and size of each transaction. Since different size replies have different response times, we normalize these times by dividing them by the best observed time for the respective size for each network configuration. Normalized times are thus dimensionless. The best possible normalized response time is 1 (all responses took the minimum time). Because we aggregate traffic from a number of clients, typical normalized times are 1-2 for light loads, or 3-5 for heavier loads where foreground traffic has more self-interference.

To characterize the variability in measured traffic, we report median and quartiles of normalized foreground response times for all transactions measured during a five minute test run (typically several thousand transactions). As background load rises, we would expect the median to rise and the quartiles to spread, indicating more interference and variability. The ideal background processing mechanism will minimize these effects, resulting in a flat, low foreground performance curve and a low interquartile gap.

Figures 2 and 3 summarize the results of our experiments. To explore the design space, we varied:

- *Backgrounding Algorithm*:
  unmodified server (no distinction between request priorities), and each of our three background processing mechanisms
- *Network*:
  10Mb/s and 100Mb/s private, non-switched Ethernets with no other traffic present
- *Foreground Load*:
  light load (causing 20% bottleneck resource utilization) and heavy load (causing 80% utilization)

For 10Mb/s Ethernet, the bottleneck was the network, and high foreground request loads were generated by 3 and 15 Webstone clients, respectively. For 100Mb/s Ethernet and a CPU-bound system, we used 15 and 52 clients to generate the loads (numbers chosen according to Figure 1.)

### 5.1. Results for 10Mb/s Ethernet

The first two graphs show foreground response times in the basic case with no backgrounding being performed. With one service class, median performance grew up to 40 times worse (from 1.05 without background load to about 40) under light load (Figure 2: light/basic). Under heavy load (Figure 2: heavy/basic), it grew about 15

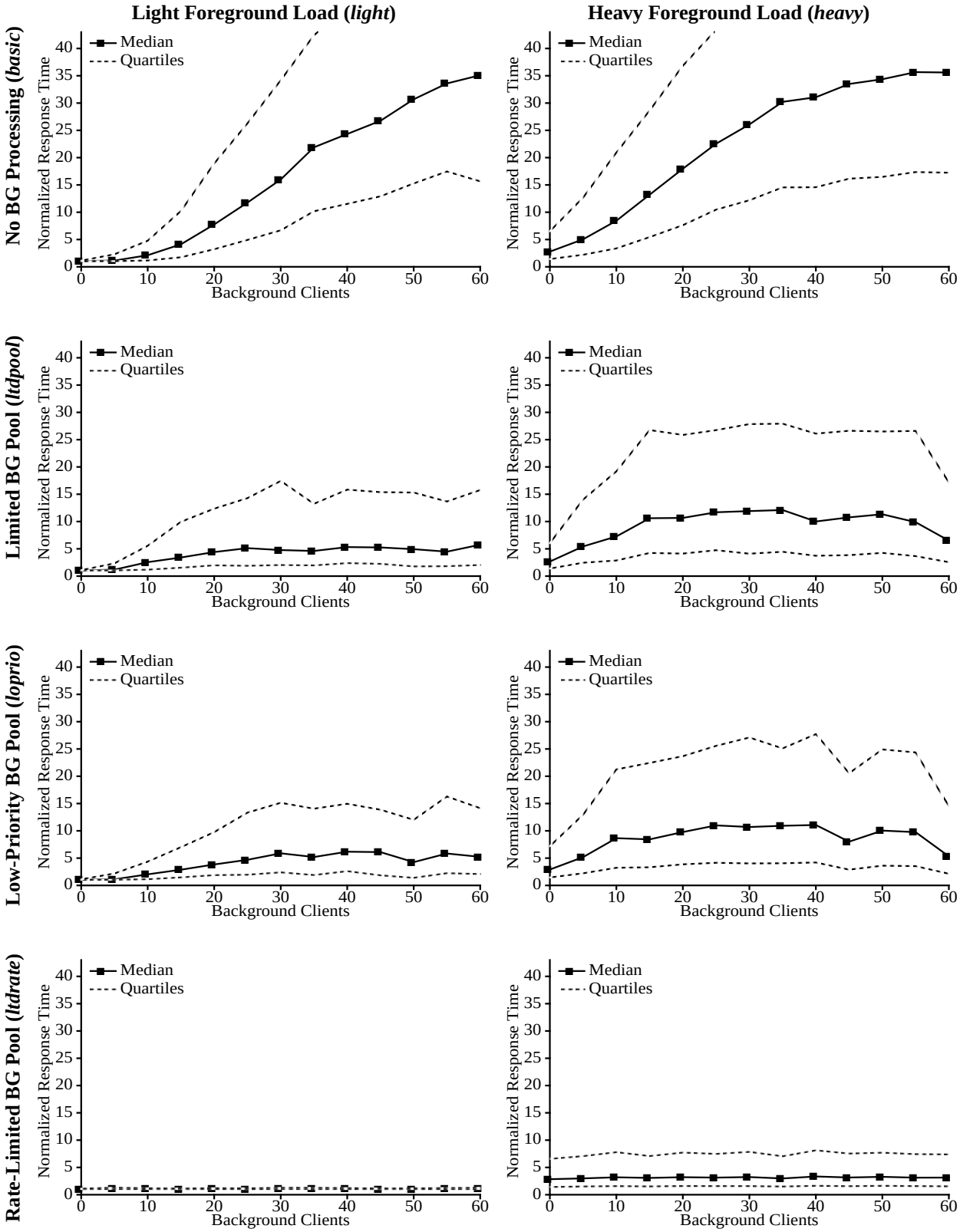


**Figure 2.** Normalized median foreground response times (with first and third quartiles) for the baseline case and the three different backgrounding mechanisms over 10Mb/s Ethernet; both under light and heavy foreground load.

times worse (from 2.8 to 42). We also saw a substantial increase in response time variation, as illustrated by the wide inter-quartile gap. Under heavy foreground load there was substantial interference within the group of foreground connections: With no background traffic present, we observed a median response time that was two to three times slower than under light load. From this, we conclude that background requests can substantially reduce median performance in an unmodified system.

Next, we will look at the result from our first backgrounding algorithm, where the server limited its background pool size to five. For both light and heavy (Figure 2: light/ltdpool, heavy/ltdpool) foreground load, median performance only grew 5-6 times worse. The simple idea of limiting the background pool resulted in a considerable improvement compared to the basic case. However, median performance was degraded noticeably, and the variance in observed median performance was substantial, although smaller than in the basic case. This simple mechanism keeps median performance under 10 times normal for half of all requests.

Our second algorithm also lowers the process priority of the background processes to the minimum in addition to keeping the pool size limited to five servers. Median performance under light (Figure 2: light/loprio) load was unchanged from the previous case, while median performance under heavy load (Figure 2: heavy/loprio) was marginally better than during the previous experiment (four times worse compared to five times before). Performance variance was also virtually identical to the previous experiment. We have shown above that CPU is not the bottleneck for 10Mb/s Ethernet. Thus, even low-priority processes received enough CPU time to generate a substantial amount of network traffic. Process priorities are therefore not an adequate mechanism to establish different levels of service in this scenario. This result emphasizes the point that knowledge of the bottleneck resource is essential.

The third backgrounding mechanism we evaluated was rate-limiting background sends. It performed best, with very low overhead and variance, under both foreground loads: With light load (Figure 2: light/ltdrate), median performance grew by only 4% and variance was also extremely low. Under heavy foreground load (Figure 2: heavy/ltdrate) median performance degraded by less than 18%.

### 5.2. Results for 100Mb/s Ethernet

We expected different results for 100Mb/s Ethernet, because of the different bottleneck resource. As before, performance (both median and variance) degraded in the basic case with increasing background load: For light foreground load (Figure 3: light/basic), it grew almost ten times worse (from 1.3 with no background load to about 11.6) For heavy load (Figure 3: heavy/basic) it grew from 2.8 to almost 16; over five times worse. Variance in both cases was extremely high. Again, we see substantial interference within the group of foreground connections alone; with no background load, median performance for heavy load is more than twice as bad than for light load. Comparing this case against the 10Mb/s case, note that the normalized response times here are about 50% smaller than before. This is because in the network-bound 10Mb/s case, delays in response time are mostly due to packet losses and the incurred retransmission. In the 100Mb/s case there is plenty of idle network capacity. Thus, delays in response time are mostly due to queueing inside the kernel.

By limiting the background pool, both median performance and its variance was improved under both sets of foreground load. As for the 10Mb/s case, limiting the size of the background pool is an effective first step to establish different levels of service. Under light foreground load (Figure 3: light/ltdpool) median performance only grows worse twofold, while under heavy load (Figure 3: heavy/ltdpool) it only increases by 40%. Again, this very simple mechanism can limit the excesses of backgrounding.

Our second backgrounding mechanism also lowered the priority of the background processes. We had designed this mechanism specifically for a CPU-bound system to evaluate if process priorities would help in this scenario. Our results indicate that this is not the case. Both under light and heavy (Figure 3: light/loprio, heavy/loprio) background loads, median performance is only marginally better than in the previous case (Figure 3: light/ltdpool, heavy/ltdpool), where the background servers ran at the same priority as the foreground ones. One possible explanation for this lies in the nature of the 4.4BSD CPU scheduler [McKusick *et al.* 1996]. It lowers the priority of processes that have accumulated more CPU time than others, and it raises the priority of process that are blocked. These two features of the scheduler counteract our intention to use priorities to further slow down background processes.

Rate-limiting the background pool works best again in this scenario. Under light foreground load (Figure 3: light/ltdrate), median performance only degrades by about 6%, and the performance variance is extremely small. Under heavy foreground load (Figure 3: heavy/ltdrate), median performance decreases by 11%, which is a moderately better than the first two algorithms, but variance is significantly reduced, as shown by the quartiles.

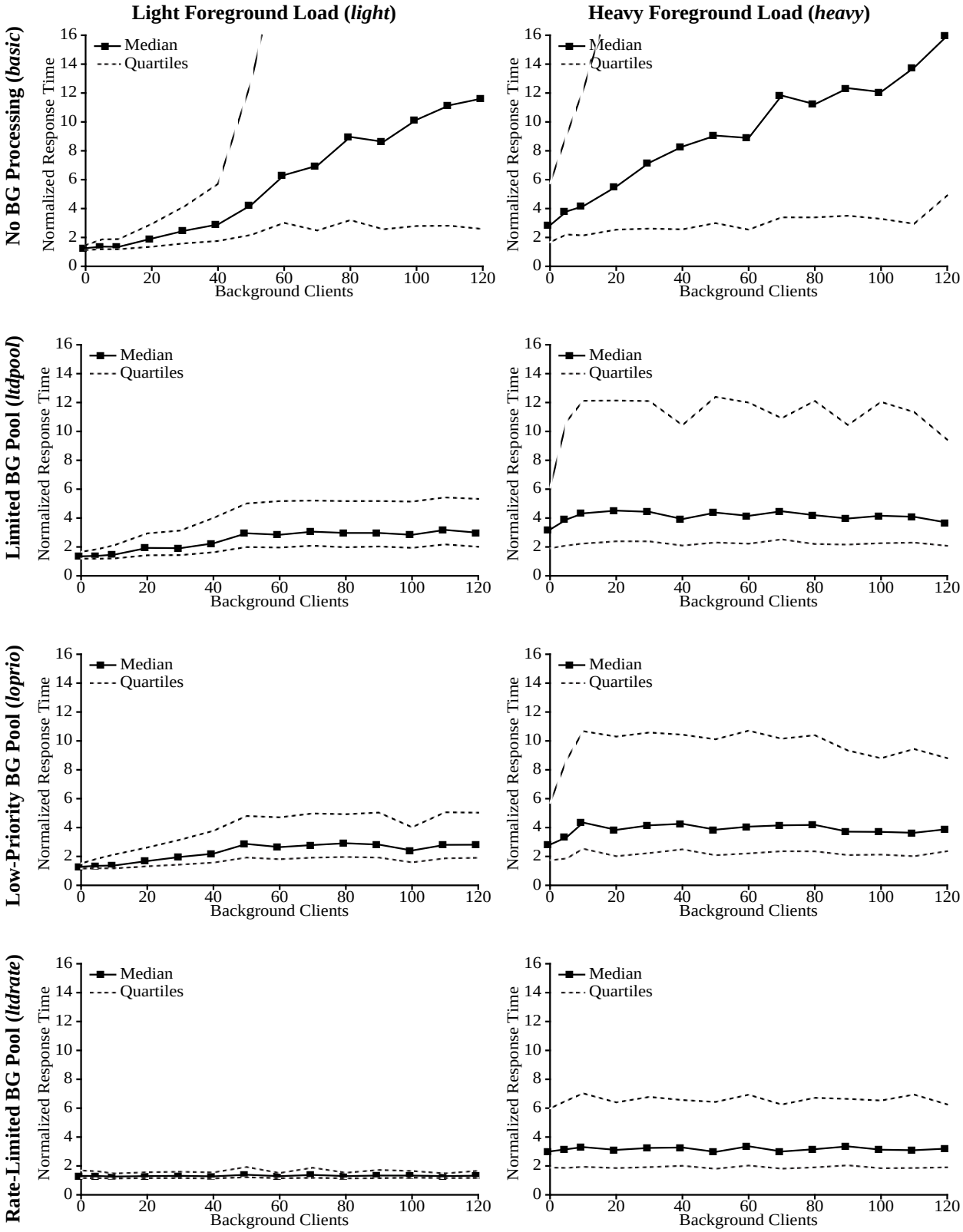


**Figure 3.** Normalized median foreground response times (with first and third quartiles) for the baseline case and three different backgrounding mechanisms over 100Mb/s Ethernet; both under light and heavy foreground load.

### 5.3. Discussion of results

In this section, we will summarize the experimental results of our three background traffic mechanisms, and then discuss how our mechanisms can be applied to scenarios where the server is not CPU- or network-bound, or to scenarios where request messages need to be sent in the background.

An important first result of our experiments is that substantial benefits can be provided with user-level changes. Even the very simple approach of limiting the background server pool works well in both scenarios: The median foreground response time is kept around five and ten times the minimum for the 10Mb/s and 100Mb/s cases. A surprising outcome is that our second mechanism (lowering the process priority of the background pool) did not result in the expected improvement over the first one (just limiting the pool size) - especially in the CPU-bound case, where process priorities should be most useful. As described above, the BSD CPU scheduler diminishes the difference between high-priority and low-priority processes by rewarding I/O. On other systems, especially non-Unix systems, this may be different. However, since there are minor median performance improvements in some cases (and no penalties in the other ones), we consider lowering the priority of the background pool useful in addition to other measures.

Of the three simple backgrounding mechanisms we have designed, limiting the network sending rate of background processes performs best. In all cases, median foreground performance decreased slowly (and only by about 4-17%) as background load increased substantially. This is the primary requirement for a good backgrounding mechanism (see the beginning of Section 5). Another improvement of rate-limiting (compared to simply limiting the background pool size) is that rate limits offer a much finer granularity of control. Even a single server process can put a considerable load on a system, if presented with enough requests. Thus, an increase of one in the background pool size can translate into a large change in bottleneck resource utilization due to background requests. For our third mechanism to be effective, it is important to set the rate limit to a fraction of the available uplink bandwidth to the Internet. Even then, background traffic may interfere with other traffic after the first hop, if a bandwidth bottleneck exists further up the path. To minimize these interferences, the rate limit should be kept low both relative to the uplink bandwidth and in absolute terms. An additional minor benefit of this mechanism is that it may generate less bursty background traffic by spreading out the transmission of the response message over an interval of time.

Our experiments were conducted using a small, static set of web pages. A server offering dynamic content will usually have higher local resource utilization (CPU, disk and physical memory) due to the extra processing involved with each request. Our experiments show that application-level backgrounding mechanisms are effective in the CPU-bound case. (As CPU requirements per request increase, our second mechanism may provide better service discrimination than the first.) For a disk- or memory-bound server, we believe our current mechanisms would be effective, since slowing down the background pool will result in fewer resource requests from those processes, so a larger share of the critical resource is available to foreground processes. Knowledge of the system bottleneck (see Section 3) would allow generalization of our approaches to further address this situation, such as rate-limiting the disk I/O of background processes.

We have limited ourselves to implementing server-side backgrounding mechanisms. Thus, all request messages are sent in the foreground. Since most requests are small [Mah 1997], requests will not typically lower performance. If backgrounding of request messages is of prime concern, our mechanisms can also be applied on the client-side, to allow sending requests in the background.

## 6. Related work

Extensions for differentiated services have been proposed at the application-, kernel- and network layer.

Almeida et al. [Almeida *et al.* 1998] have designed several application-level and kernel approaches to web QoS. Their first application-level mechanism limits the server pool sizes allocated to requests of different classes. It is similar to out first mechanism (limiting the background pool) except that they demultiplex and queue requests inside the application. The second mechanism they have implemented is a kernel level-scheduler that allows preemption of low-level requests and assigns process priorities based on the request class, which is similar to our lowered-priority approach. While they confirm our result that simple application-level mechanisms (such as a limited pool of servers) are effective, they claim that under heavy load, kernel-level preemption mechanisms are needed to improve performance. We examined application-level mechanisms in more depth, evaluating three different mechanisms. We demonstrated that a carefully designed application-level method will perform well even under heavy load. Thus, additional kernel mechanisms may not be required.

Several soft-realtime kernel extensions to give applications more control about scheduling and resource allo-

cation have been proposed. AQUA [Lakshman *et al.* 1998] is a kernel-level framework that allows cooperating multimedia applications to dynamically negotiate their CPU and network I/O requirements with the kernel. If a resource becomes congested, applications are notified by AQUA and may adapt to the new service environment. This approach allows background processes to use allocated resources, addressing the first problem we identify in Section 7. Unfortunately, it requires kernel changes and does not address non-allocated bottlenecks. OMEGA [Nahrstedt and Smith 1996] is an end-system kernel framework that supports soft-realtime scheduling of CPU, memory and network resource allocation to provide end-to-end QoS. OMEGA is similar to AQUA; applications dynamically negotiate their resource requirements with a QoS broker.

Waldspurger and Weihl have successfully applied their proportional-share resource schedulers [Waldspurger and Weihl 1994, 1995] to CPU and network interface scheduling for a modified Linux kernel. Experiments show that they are successful in allocating different shares of the managed resource to different applications. As with AQUA before, these schedulers can improve application-level backgrounding, but require kernel changes.

Application-level mechanisms cannot directly control what happens to their traffic inside the network. Network-level mechanisms could be used to improve application-level backgrounding mechanisms. At the network-level, several proposals have been made to accommodate different levels of service. One such proposal is to extend IP for integrated services [Wroclawski 1997]. In this scheme, receivers initiate a resource reservation request to receive a guaranteed service commitment with the Resource Reservation Protocol (RSVP) [Zhang *et al.* 1993]. A second proposal is to extend IP to support differentiated services [Blake *et al.* 1998]. This approach allows high priority traffic to take precedence over existing traffic on a per-packet basis. Compliant routers will respect priorities in their queueing and forwarding decisions.

Ultimately the network and end system OS are the best places to provide differentiated services. A router can react to traffic requirements directly, and the end system OS has better means of enforcing QoS than non-privileged applications. Deployment of these mechanisms is difficult since many routers must support these protocols for the system to become effective. Our work suggests that much of the benefits of background service is possible through application-level mechanisms. For best results, however, the administrator must tune the background transfer rate proportional to the bottleneck bandwidth. If this bottleneck is not known, network support is important, but if the bottleneck is well understood (such as at the server's Internet connection) this tuning is straightforward.

## 7. Future work

We have shown that rate-limiting background sends is an effective server-side, application-level backgrounding mechanism. The major problem of that approach is that the rate limit can never be exceeded; even if the network could sustain the additional traffic without a decrease in foreground performance. If foreground load could be quantified, this limitation could be overcome. We plan on experimenting with more elaborate background processing schemes to that purpose. One such scheme (requiring OS support) would be to have background processes send only if the foreground socket buffers are empty. Another mechanism might be to have the (foreground) server pool aggregate throughput statistics over time to estimate the available network bandwidth.

At this time, our modified web server demultiplexes the request stream into service classes at the OS-level by using different sockets for background and foreground requests. We would like to investigate a server that demultiplexes its request stream at the application level. This gives the server more control over how and when to process each request, but raises the issue of head-of-line blocking (background request at the head of the socket queue delays foreground requests queued behind it). To overcome this problem, application-level queueing needs to be implemented.

This paper has concentrated on backgrounding of unicast traffic. However, multicast traffic may also benefit from the availability of background service. One example are multicast content-push applications such as video-conferencing: the audio channel could be transmitted in the foreground, since humans are more sensitive to interruptions of the audio stream, while the video channel could be transmitted in the background. We have applied the idea of application-level backgrounding to multicast distribution in the LSAM system [Touch and Hughes 1998].

One limitation of the Webstone benchmark we used to generate the request load during our experiments is the inability to generate a load that completely overloads the server [Banga and Druschel 1997]. Future experiments should use a more realistic model to simulate client behavior.

## 8. Conclusion

We have described several scenarios in which having different levels of service for web requests would result in a better overall service model. An ideal system requires extensions to most network routers and the end system OS and applications. These extensions are under development, but will take time to standardize and deploy.

Application-level mechanisms can achieve several of the key benefits of a complete solution while being extremely easy to set up. Knowing the bottleneck resource of the system is essential in designing an effective mechanism. A web server has been monitored in two different experiments to detect its bottleneck resource. Using that information, we have designed and implemented three simple, server-side, application-level mechanisms to support different levels of service. These mechanisms have been compared against the basic system in four different sets of experiments. Analyzing the results showed that while any of our mechanism performs better than the basic case, limiting the send rate of background responses is particularly effective in establishing different levels of service: The performance impact of this mechanism on foreground traffic was less than 4-17% in all cases.

## Acknowledgments

We would like to thank Joe Touch for his detailed discussions of background processing alternatives and for his valuable comments on an earlier draft of this paper. Ted Faber, Steve Hotz and Joe Bannister have also provided helpful feedback for the paper.

## References

Almeida, J., M. Dabu, A. Manikutty and P. Cao (1998), "Providing Differentiated Levels of Service in Web Content Hosting," In *Proceedings of the 1988 SIGMETRICS Workshop on Internet Server Performance*, Madison, WI, USA, June 1998, pp. 91-102.

Apache HTTP Server Project (1998), "Running a High-Performance Web Server for BSD."
web page http://www.apache.org/docs/misc/perf-bsd44.html

Apache HTTP Server Project (1998).
web page http://www.apache.org/

Balakrishnan, H., V. Padmanabhan, S. Seshan, M. Stemm and R. Katz (1998), "TCP Behavior of a Busy Internet Server: Analysis and Improvements," In *Proceedings of the IEEE INFOCOM '98*, 1, pp. 152-162.

Banga, G. and P. Druschel (1997), "Measuring the Capacity of a Web Server," In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, USENIX Association, Berkeley, CA, pp. 61-71.

Blake, S., D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss (1998), "An Architecture for Differentiated Services," RFC 2475, Internet Request For Comments.

Clark, D. (1988), "The Design Philosophy of the DARPA Internet Protocols," *Computer Communication Review 18*, 4, pp. 106-114.

Netperf Project (1998).
web page http://www.netperf.org/

Lakshman, K., R. Yavatkar and R. Finkel (1998), "Integrated CPU and network-I/O QoS management in an endsystem," *Computer Communications 21*, 4, pp. 325-333.

Mah, B. (1997), "An Empirical Model of HTTP Network Traffic," In *Proceedings of the IEEE INFOCOM '97*, IEEE Computer Society Press, Los Alamitos, CA, pp. 592-600.

McKusick, M., K. Bostic, M. Karels and J. Quarterman (1996), *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, Reading, MA, pp. 92-97.

Nahrstedt, K. and J. Smith.(1996), "Design, Implementation and Experiences with the OMEGA End-point Architecture," *IEEE Journal on Selected Areas in Communications*, pp. 1263-1279.

Padmanabhan, V. and J. Mogul (1996), "Using Predictive Prefetching to Improve World Wide Web Latency," *ACM Computer Communication Review 26*, 3, pp. 22-36

Padmanabhan, V. and R. Katz (1998), "TCP Fast Start: A Technique for Speeding Up Web Transfers," In *Proceedings of the IEEE GLOBECOM Internet Mini-Conference*, pp. 41-46.

Pointcast, Inc. (1998), "How Pointcast Works."
web page http://www.pointcast.com/products/pcn/hwork.html

Touch, J. and A. Hughes (1998), "The LSAM Proxy Cache - a Multicast Distributed Virtual Cache," *Computer Networks and ISDN Systems 30*, 22-23, pp. 2245-2252.

Touch, J. (1998), "LowLat 'Containment' Issues," In Preparation, Technical Report, USC Information Sciences Institute.

Touch, J. (1995), "Defining 'High Speed' Protocols: Five Challenges & an Example That Survives the Challenges," *IEEE Journal on Selected Areas in Communications 13*, 5, pp. 828-835.

Trent, G. and M. Sage (1995), "WebSTONE: The First Generation in HTTP Server Benchmarking," Technical Report, MTS, Silicon Graphics, Inc., Mountain View, CA, now maintained by Mindcraft, Inc.
web page http://www.mindcraft.com/webstone/

Visweswaraiah, V. and J. Heidemann (1997), "Improving Restart of Idle TCP Connections," Technical Report 97-661, Computer Science Department, University of Southern California, Los Angeles, CA.

Waldspurger, C. and W. Weihl (1994), "Lottery Scheduling: Flexible Proportional-Share Resource Management," In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, USENIX Association, Berkeley, CA, pp. 1-11.

Waldspurger, C. and W. Weihl (1995), "Stride Scheduling: Deterministic Proportional-Share Resource Management," Technical Memoran-

dum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Wroclawski, J. (1997), “The Use of RSVP with IETF Integrated Services,” RFC 2210, Internet Request For Comments.

Zhang, L., S. Deering, D. Estrin, S. Shenker and D. Zappala (1993), “RSVP: A New Resource ReSerVation Protocol,” *IEEE Network 7*, 5, pp. 8-18.