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
E-mail: larse@isi.edu

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 (10 Mb/s and 100 Mb/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 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 overhead 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 [Padmanabhan and Katz 1998; Visweswaraiia and Heidemann 1997].

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 paid. 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 10 Mb/s and 100 Mb/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 300 Mhz Pentium-II PC with 128 MB of physical memory running FreeBSD 2.2.6. The