Efficient Support for P-HTTP in Cluster-Based Web Servers *

Mohit Aron Peter Druschel Willy Zwaenepoel
Department of Computer Science
Rice University

Abstract

This paper studies mechanisms and policies for supporting HTTP/1.1 persistent connections in cluster-based Web servers that employ content-based request distribution. We present two mechanisms for the efficient, content-based distribution of HTTP/1.1 requests among the back-end nodes of a cluster server. A trace-driven simulation shows that these mechanisms, combined with an extension of the locality-aware request distribution (LARD) policy, are effective in yielding scalable performance for HTTP/1.1 requests. We implemented the simpler of these two mechanisms, *back-end forwarding*. Measurements of this mechanism in connection with extended LARD on a prototype cluster, driven with traces from actual Web servers, confirm the simulation results. The throughput of the prototype is up to four times better than that achieved by conventional weighted round-robin request distribution. In addition, throughput with persistent connections is up to 26% better than without.

1 Introduction

Clusters of commodity workstations are becoming an increasingly popular hardware platform for cost-effective high performance network servers. Achieving scalable server performance on these platforms is critical to delivering high performance to users in a cost-effective manner.

State-of-the-art cluster-based Web servers employ a front-end node that is responsible for distributing incoming requests to the back-end nodes in a manner that is transparent to clients. Typically, the front-end distributes the requests such that the load among the back-end nodes is balanced. With *content-based request distribution*, the front-end additionally takes into account the content or type of service requested when deciding to which back-end node a client request should be assigned.

Content-based request distribution allows the integration of server nodes that are specialized for certain types of content or services (e.g., audio/video), it permits the partitioning of the server's database for scalability, and it enables clever request distribution policies that improve performance. In previous work, we proposed *locality-aware request distribution* (LARD), a content-based policy that achieves good cache hit rates in addition to load balance by dynamically partitioning the server's working set among the back-end nodes [23].

In this paper, we investigate mechanisms and policies for content-based request distribution in the presence of HTTP/1.1 [11] persistent (keep-alive) client connections (P-HTTP). Persistent connections allow HTTP clients to submit multiple requests to a given server using a single TCP connection, thereby reducing client latency and server overhead [19]. Unfortunately, persistent connections pose problems for clusters that use content-based request distribution, since requests in a single connection may have to be assigned to different back-end nodes to satisfy the distribution policy.

This paper describes efficient mechanisms for content-based request distribution and an extension of the LARD policy in the presence of HTTP/1.1 connections. It presents a simulation study of these mechanisms, and it reports experimental results from a prototype cluster implementation. The results show that persistent connections can be supported efficiently on cluster-based Web servers with content-based request distribution. In particular, we demonstrate that using *back-end forwarding*, an extended LARD policy achieves up to 26% better performance with persistent connections than without.

The rest of the paper is organized as follows. Section 2 provides some background information on HTTP/1.1 and LARD, and states the problems posed by HTTP/1.1 for clusters with content-based request distribution. Section 3 considers mechanisms for achieving content-based request distribution in the presence of HTTP/1.1 persistent connections. The extended LARD policy is presented in Section 4. Section 5 presents a performance analysis of our request distribution mechanisms. A simulation study of the various mechanisms and the extended LARD policy is described in Section 6. Section 7 discusses a prototype implementation, and Section 8 reports measurement results obtained using that prototype. We discuss related work in Section 9, and conclude in Section 10.

*To appear in Proc. of the 1999 Annual Usenix Technical Conference, Monterey, CA, June 1999.

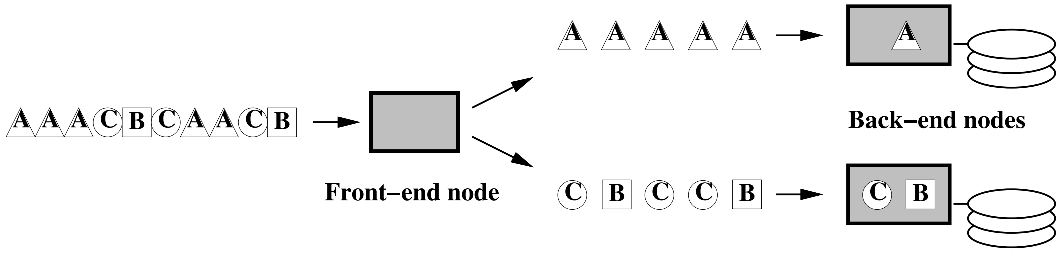

Figure 1: Locality-Aware Request Distribution

2 Background

This section provides background information on persistent connections in HTTP/1.1, content-based request distribution, and the LARD strategy. Finally, we state the problem that persistent connections pose to content-based request distribution.

2.1 HTTP/1.1 persistent connections

Obtaining an HTML document typically involves several HTTP requests to the Web server, to fetch embedded images, etc. Browsers using HTTP/1.0 [5] send each request on a separate TCP connection. This increases the latency perceived by the client, the number of network packets, and the resource requirements on the server [19, 22].

HTTP/1.1 enables browsers to send several HTTP requests to the server on a single TCP connection. In anticipation of receiving further requests, the server keeps the connection open for a configurable interval (typically 15 seconds) after receiving a request . This method amortizes the overhead of establishing a TCP connection (CPU, network packets) over multiple HTTP requests, and it allows for pipelining of requests [19]. Moreover, sending multiple server responses on a single TCP connection in short succession avoids multiple TCP slow-starts [29], thus increasing network utilization and effective bandwidth perceived by the client.

RFC 2068 [11] specifies that for the purpose of backward compatibility, clients and servers using HTTP/1.0 can use persistent connections through an explicit HTTP header. However, for the rest of this paper, HTTP/1.0 connections are assumed not to support persistence. Moreover, this paper does not consider any new features in HTTP/1.1 over HTTP/1.0 other than support for persistent connections and request pipelining.

2.2 Content-based Request Distribution

Content-based request distribution is a technique employed in cluster-based network servers, where the front-end takes into account the service/content requested when deciding which back-end node should serve a given request. In contrast, the purely load-based schemes like *weighted round-robin* (WRR) used in commercial high performance cluster servers [15, 8] distribute incoming requests in a round-robin fashion, weighted by some measure of load on the different back-end nodes.

The potential advantages of content-based request distribution are: (1) increased performance due to improved hit rates in the back-end's main memory caches, (2) increased secondary storage scalability due to the ability to partition the server's database over the different back-end nodes, and (3) the ability to employ back-end nodes that are specialized for certain types of requests (e.g., audio and video).

With content-based request distribution, the front-end must establish the TCP connection with the client *prior* to assigning the connection to a back-end node, since the nature and the target[1] of the client's request influences the assignment. Thus, a mechanism is required that allows a chosen back-end node to serve a request on the TCP connection established by the front-end. For reasons of performance, security, and interoperability, it is desirable that this mechanism be transparent to the client. We will discuss mechanisms for this purpose in Section 3.

2.3 Locality-aware request distribution

Locality-aware request distribution (LARD) is a specific strategy for content-based request distribution that focuses on the first of the advantages cited above, namely improved cache hit rates in the back-ends [23]. LARD strives to improve cluster performance by *simultaneously* achieving load balancing and high cache hit rates at the back-ends.

Figure 1 illustrates the principle of LARD in a cluster with two back-ends and a working set of three targets (A, B, and C) in the incoming request stream. The front-end directs all requests for A to back-end 1, and all requests for B and C to back-end 2. By doing so, there is an increased likelihood that the request finds the requested target in the cache at the back-end.

In contrast, with a round-robin distribution of incoming requests, requests for all three targets will arrive at

[1] In the following discussion, the term *target* is used to refer to a Web document, specified by a URL and any applicable arguments to the HTTP GET command.

both back-ends. This increases the likelihood of a cache miss, if the sum of the sizes of the three targets, or, more generally, if the size of the working set exceeds the size of the main memory cache at an individual back-end node. Thus, with a *round-robin* distribution, the cluster does not scale well to larger working sets, as *each* node's main memory cache has to fit the entire working set. With LARD, the effective cache size approaches the *sum* of the individual node cache sizes. Thus, adding nodes to a cluster can accommodate both increased traffic (due to additional CPU power) and larger working sets (due to the increased effective cache size).

2.4 The problem with HTTP/1.1

HTTP/1.1 persistent connections pose a problem for clusters that employ content-based request distribution, including LARD. The problem is that existing, scalable mechanisms for content-based distribution operate at the granularity of TCP connections. With HTTP/1.1, multiple HTTP requests may arrive on a single TCP connection. Therefore, a mechanism that distributes load at the granularity of a TCP connection constrains the feasible distribution policies, because all requests arriving on a given connection must be served by a single back-end node.

This constraint is most serious in clusters where certain requests can only be served by a subset of the back-end nodes. Here, the problem is one of correctness, since a back-end node may receive requests that it cannot serve.

In clusters where each node is capable of serving any valid request, but the LARD policy is used to partition the working set, performance loss may result since a back-end node may receive requests not in its current share of the working set. As we will show in Section 6, this performance loss can more than offset the performance advantages of using persistent connections in cluster servers.

3 Mechanisms for content-based request distribution

A front-end that performs content-based request distribution must establish a client HTTP connection before it can decide which back-end node should serve the request. Therefore, it needs a mechanism that allows it to have the chosen back-end node serve request(s) on the established client connection. In this section, we discuss such mechanisms.

The simplest mechanisms work by having the front-end "redirect" the client browser to the chosen back-end node, by sending an HTTP redirect response, or by returning a Java applet that contacts the appropriate back-end when executed in the browser [1].

These mechanisms work also for persistent connections, but they have serious drawbacks. The redirection introduces additional delay; the address of individual back-end nodes is exposed to clients, which increases security risks; and, simple or outdated browsers may not support redirection. For these reasons, we only consider client-transparent mechanisms in the remainder of this paper.

3.1 Relaying front-end

A simple client-transparent mechanism is a *relaying front-end*. Figure 2 depicts this mechanism and the other mechanisms discussed in the rest of this section. Here, the front-end maintains persistent connections (back-end connections) with all of the back-end nodes. When a request arrives on a client connection, the front-end assigns the request, and forwards the client's HTTP request message on the appropriate back-end connection. When the response arrives from the back-end node, the front-end forwards the data on the client connection, buffering the data if necessary.

The principal advantage of this approach is its simplicity, its transparency to both clients and back-end nodes, and the fact that it allows content-based distribution at the granularity of individual requests, even in the presence of HTTP/1.1 persistent connections.

A serious disadvantage, however, is the fact that all response data must be forwarded by the front-end. This may render the front-end a bottleneck, unless the front-end uses substantially more powerful hardware than the back-ends. It is conceivable that small clusters could be built using as a front-end a specialized layer 4 switch with the ability to relay transport connections. We are, however, not aware of any actual implementations of this approach. Furthermore, results presented in Section 6.1 indicate that, even when the front-end is not a bottleneck, a relaying front-end does not offer significant performance advantages over more scalable mechanisms.

3.2 Multiple TCP connection handoff

A more complex mechanism involves the use of a *TCP handoff protocol* among front-end and back-end nodes. The handoff protocol allows the front-end to transfer its end of an established client connection to a back-end node. Once the state is transferred, the back-end transmits response data directly to the client, bypassing the front-end. Data from the client (primarily TCP ACK packets) are forwarded by the front-end to the appropriate back-end node in an efficient manner.

In previous work, we have designed, implemented, and evaluated a handoff protocol for HTTP/1.0 [23]. This *single handoff* protocol can support persistent connections, but all requests must be served by the back-end node to which the connection was originally handed off.

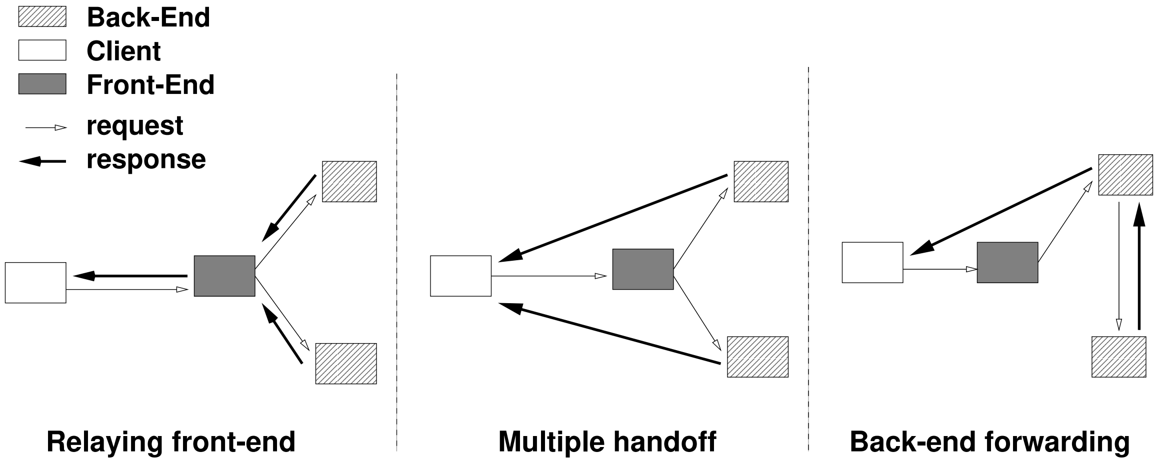

Figure 2: Mechanisms for request distribution

The design of this handoff protocol can be extended to support HTTP/1.1 by allowing the front-end to migrate a connection between back-end nodes. The advantage of this *multiple handoff* protocol is that it allows content-based request distribution at the granularity of individual requests in the presence of persistent connections. Unlike front-end relaying, the handoff approach is efficient and scalable since response network traffic bypasses the front-end.

The handoff approach requires the operating systems on front-end and back-end nodes to be customized with a vendor-specific loadable kernel module. The design of such a module is relatively complex, especially if multiple handoff is to be supported. To preserve the advantages of persistent connections – reduced server overhead and reduced client latency – the overhead of migrating connections between back-end nodes must be kept low, and the TCP pipeline must be kept from draining during migration.

3.3 Back-end request forwarding

A third mechanism, *back-end request forwarding*, combines the TCP single handoff protocol with forwarding of requests and responses among back-end nodes. In this approach, the front-end hands off client connections to an appropriate back-end node using the TCP single handoff protocol. When a request arrives on a persistent connection that cannot (or should not) be served by the back-end node that is currently handling the connection, the connection is *not* handed off to another back-end node.

Instead, the front-end informs the connection handling back-end node A which other back-end node B should serve the offending request. Back-end node A then requests the content or service in question directly from node B, and forwards the response to the client on its client connection. Depending on the implementation, these "lateral" requests are forwarded through persistent HTTP connections among the back-end nodes, or through a network file system.

The advantages of back-end request forwarding lie in the fact that the complexity and overhead of multiple TCP handoff can be avoided. The disadvantage is the overhead of forwarding responses on the connection handling back-end node. This observation suggests that the back-end request forwarding mechanism is appropriate for requests that result in relatively small amounts of response data. Results presented in Section 6 show that due to the relatively small average content size in today's Web traffic [19, 3], the back-end request forwarding approach is very competitive.

4 Policies

This section presents an extension of the LARD policy that works efficiently in the presence of HTTP/1.1 persistent connections, when used with the back-end request forwarding mechanisms presented in the previous section.

Both the front-end relaying mechanism and the TCP multiple handoff mechanism allow requests to be distributed at the granularity of individual requests. As such, they do not place any restriction on the request distribution policies that can be used. In particular, the LARD policy can be used in combination with these mechanisms without loss of locality.

The back-end forwarding mechanism, on the other hand, does place restrictions on the distribution policy, as it mandates that a connection can be handed off to a back-end node only once. If requests arrive on a persistent connection that cannot or should not be served by that back-end node, the policy must instruct that back-end node to forward the request to another back-end node.

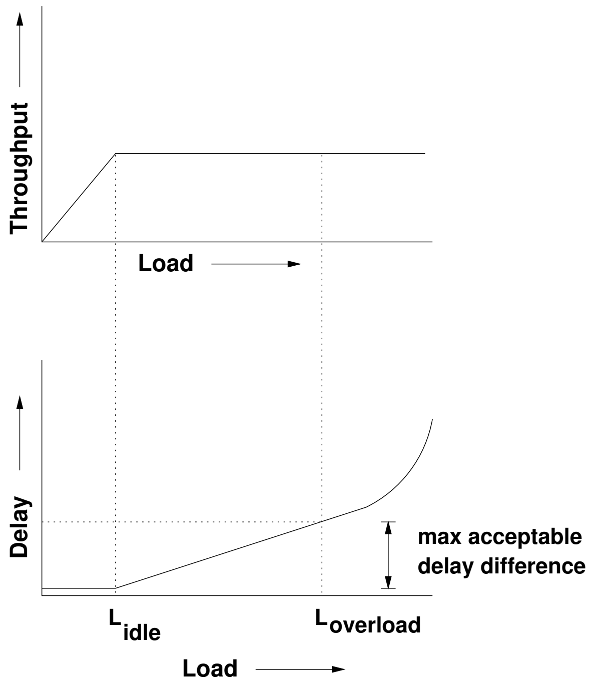

Figure 3: Server Throughput and Delay

We have developed an extension of the LARD policy that can efficiently distribute HTTP/1.1 requests in a cluster that uses the back-end forwarding mechanism. The following subsection briefly presents the standard LARD strategy. Then, we proceed to present our extension.

4.1 The LARD strategy

The LARD strategy yields scalable performance by achieving both load balancing and cache locality at the back-end servers. For the purpose of achieving cache locality, LARD maintains mappings between targets and back-end nodes, such that a target is considered to be cached on its associated back-end nodes. To achieve a balance between load distribution and locality, LARD uses three cost metrics: *cost_balancing*, *cost_locality* and *cost_replacement*. The intuition for the definition of these metrics can be explained using Figure 3, which shows the throughput and delay characteristics of a typical back-end server as a function of load (measured in number of active connections).

The load point L_{idle} defines a value below which a back-end node is potentially underutilized. $L_{overload}$ is defined such that the difference in delay between a back-end node operating at or above this load, compared to a back-end node operating at the point L_{idle}, becomes unacceptable.

The metric $cost\_balancing$ captures the delay in the servicing of a request because of other queued requests. $Cost\_locality$, on the other hand, reflects the delay arising due to the presence or absence of the target in the cache. $Cost\_replacement$ is a cost that reflects the potential future overhead caused by the replacement of a target in the cache. The three cost metrics are then defined as shown in Figure 4.

The unit of cost (and also of load) is defined to be the delay experienced by a request for a cached target at an otherwise unloaded server. The aggregate cost for sending the request to a particular server is defined as the sum of the values returned by the above three cost metrics. When a request arrives at the front-end, the LARD policy assigns the request to the back-end node that yields the minimum aggregate cost among all nodes, and updates the mappings to reflect that the requested target will be cached at that back-end node[2].

Our experimental results with the Apache 1.3.3 webserver running on FreeBSD-2.2.6 indicate settings of $L_{overload}$ to 130, L_{idle} to 30 and $Miss\ Cost$ to 50. We have used these settings both for our simulator as well as for our prototype results in this paper.

4.2 The extended HTTP/1.1 LARD strategy

The basic LARD strategy bases its choice of a back-end node to serve a given request only on the current load and the current assignment of content to back-end nodes (i.e., the current partitioning of the working set.) An extended policy that works for HTTP/1.1 connections with the back-end forwarding mechanisms has to consider additional factors, because the choice of a back-end node to serve a request arriving on a persistent connection may already be constrained by the choice of the back-end node to which the connection was handed off. In particular, the policy must make the following considerations:

1. The best choice of a back-end node to handle a persistent connection depends on all the requests expected on the connection.

2. Assigning a request to a back-end node other than the connection handling node causes additional forwarding overhead. This overhead must be weighed against the cost of reading the requested content from the connection handling node's local disk.

3. Given that a requested content has to be fetched from the local disk or requested from another back-end node, should that content be cached on the connection handling node? Caching the content reduces the cost of future requests for the content on the node handling the connection, but it also causes potential replication of the content on

[2] Although we present LARD differently than in Pai et al. [23], it can be proven that the strategies are equivalent when $L_{idle} \equiv T_{low}$ and $Miss\ Cost \equiv T_{high} - T_{low}$.

$$cost\_balancing(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ Infinity & Load(server) > L_{overload} \\ Load(server) - L_{idle} & otherwise \end{cases}$$

$$cost\_locality(target, server) = \begin{cases} 1 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

$$cost\_replacement(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ 0 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

Figure 4: LARD Cost Metrics

multiple back-end nodes, thus reducing the aggregate size of the server cache.

The intuition behind the extended LARD policy is as follows. Regarding (1), due to the structure of typical Web documents, additional requests on a persistent connection normally do not arrive until after the response to the first request is delivered to the client. For this reason, the front-end has to base its choice of a back-end node to handle the connection on knowledge of only the first request.

With respect to (2), our extended LARD policy adds two additional considerations when choosing a node to handle a request arriving on an already handed off persistent connection. First, as long as the utilization on the connection handling node's local disk is low, the content is read from that disk, avoiding the forwarding overhead. Second, in choosing a back-end to forward the request to, the policy only considers those nodes as candidates that currently cache the requested target.

Regarding (3), the extended LARD policy uses a simple heuristic to decide whether content should be cached on the connection handling node. When the disk utilization on the connection handling node is high, it is assumed that the node's main memory cache is already thrashing. Therefore, the requested content is not cached locally. If the disk utilization is low, then the requested content is added to the node's cache.

We now present the extended LARD policy. When the first request arrives on a persistent connection, the connection handling node is chosen using the basic LARD policy described in Section 4.1. For each subsequent request on the persistent connection:

- If the target is cached at the connection handling node or if the disk utilization on the connection handling node is low (less than 5 queued disk events), then the request is assigned to the same.

- Else, the three cost metrics presented in Section 4.1 are computed over the connection handling node and any other back-end nodes that have the target cached. The request is then assigned to the node that yields the minimum aggregate cost.

For the purpose of computing the LARD cost metrics, a single load unit is assigned to the connection handling node for each active connection that it handles. When the back-end forwarding mechanism is used to fetch documents from other nodes, every such node is additionally assigned a load of 1/N units—where N is the number of outstanding requests in a batch of pipelined HTTP/1.1 requests—for the duration of the request handling of all N requests.

Ideally, the front-end should assign a load of 1 to a remote node during the service time of a request. However, the front-end cannot determine when exactly a HTTP/1.1 request is being served; it can, however, estimate the service time for a batch of N pipelined HTTP/1.1 requests. Therefore, it assigns a load of 1/N to each remote node for the entire batch service time.

The front-end estimates N as the number of requests in the last batch of closely spaced requests that arrived on the connection and it estimates the batch service time as the time it takes until the next batch arrives or the connection goes idle[3]. That is, the front-end assumes that all previous requests have finished once a new batch of requests arrives on the same connection.

As in LARD, mappings between targets and back-end nodes are updated each time a target is fetched from a back-end node. It is to be noted that the extended LARD policy is equivalent to LARD for HTTP/1.0 requests.

5 Performance Analysis of Distribution Mechanisms

This section presents a simple analysis of the fundamental performance tradeoff in the use of the multiple handoff mechanism versus the back-end forward-

[3] An idle connection can be detected at the front-end by the absence of ACKs from the client.

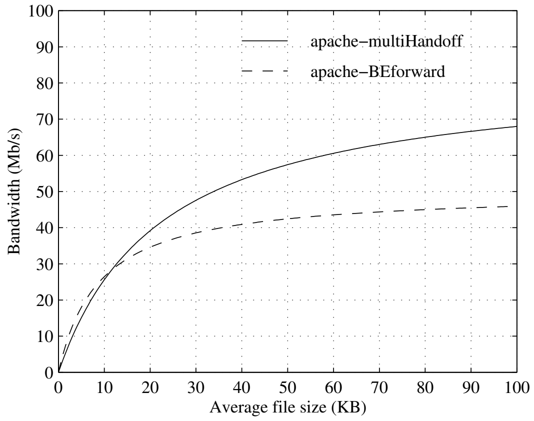

Figure 5: Apache

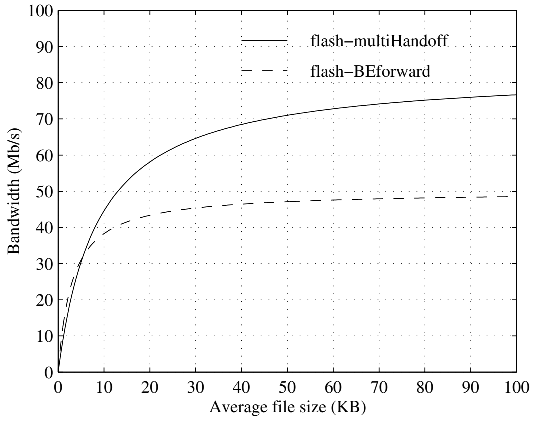

Figure 6: Flash

ing mechanism for request distribution in the presence of persistent connections.

When compared to the multiple handoff mechanism, the back-end forwarding mechanism trades off a per-byte response forwarding cost for a per-request handoff overhead. This would suggest that back-end request forwarding might be most appropriate for requests that result in small amounts of response data, while the multiple handoff approach should win in case of large responses, assuming that all other factors that affect performance are equal.

Figures 5 and 6 show the results of a simple analysis that confirms and quantifies this intuition. The analysis predicts the server bandwidth, as a function of average response size, that can be obtained from a cluster with four nodes, using either the multiple handoff or the back-end forwarding mechanism. The analysis is based on the values for handoff overhead, per-request overhead, and per-byte forwarding overhead reported above for the Apache and Flash Web servers, respectively.

To expose the full impact of the mechanisms, pessimal assumptions are made with respect to the request distribution policy. It is assumed that all requests after the first one arriving on a persistent connection have to be served by a back-end node other than the connection handling node. Since most practical policies can do better than this, the results indicate an upper bound on the impact of the choice of the request distribution mechanism on the actual cluster performance.

The results confirm that for small response sizes, the back-end forwarding mechanism yields higher performance, while the multiple handoff mechanism is superior for large responses. The crossover point depends on the relative cost of handoff versus data forwarding, and lies at 12 KB for Apache and 6 KB for Flash. These results are nearly independent of the average number of requests received on a persistent connection. Since the average response size in today's HTTP/1.0 Web traffic is less than 13 KB [19, 3], these results indicate that the back-end forwarding mechanism is indeed competitive with the TCP multiple handoff mechanism on Web workloads.

6 Simulation

To study various request distribution policies for a range of cluster sizes using different request distribution mechanisms and policies, we extended the configurable Web server cluster simulator used in Pai et al. [23] to deal with HTTP/1.1 requests. This section gives an overview of the simulator. A more detailed description of the simulator can be found in Pai et al. [23].

The costs for the basic request processing steps used in our simulations were derived by performing measurements on a 300 MHz Pentium II machine running FreeBSD 2.2.6 and either the widely used Apache 1.3.3 Web server, or an aggressively optimized research Web server called Flash [24, 25]. Connection establishment and teardown costs are set at 278/129 μs of CPU time each, per-request overheads at 527/159 μs, and transmit processing incurs 24/24 μs per 512 bytes to simulate Apache/Flash, respectively.

Using these numbers, an 8 KByte document can be served from the main memory cache at a rate of approximately 682/1248 requests/sec with Apache/Flash, respectively, using HTTP/1.0 connections. The rate is higher for HTTP/1.1 connections and depends upon the average number of requests per connection. The back-end machines used in our prototype implementation have a main memory size of 128 MB. However, the main memory is shared between the OS kernel, server applications and file cache. To account for this, we set the back-end cache size in our simulations to 85 MB.

The simulator does not model TCP behavior for the data transmission. For example, the data transmission is assumed to be continuous rather than limited by the TCP slow-start [29]. This does not affect the throughput results as networks are assumed to be infinitely fast and thus throughput is limited only by the disk and CPU overheads.

The workload used by the simulator was derived from logs of actual Web servers. The logs contain the name and the size of requested targets as well as the client host and the timestamp of the access. Unfortunately, most Web servers do not record whether two requests arrived on the same connection. To construct a simulator working with HTTP/1.1 requests, we used the following heuristic. Any set of requests sent by the same client with a period of less than 15s (the default time used by Web servers to close idle HTTP/1.1 connections) between any two successive requests were considered to have arrived on a single HTTP/1.1 connection. To model HTTP pipelining, all requests other than the first that are in the same HTTP/1.1 connection and are within 5s of each other are considered a batch of pipelined requests. Clients can pipeline all requests in a batch but have to wait for data from the server before requests in the next batch can be sent. To the best of our knowledge, synthetic workload generators like SURGE [4] and SPECweb96 [28] do not generate workloads representative of HTTP/1.1 connections.

The workload was generated by combining logs from multiple departmental Web servers at Rice University. This trace spans a two-month period. The same logs were used for generating the workload used in Pai et al. [23]. The data set for our trace consists of 31,000 targets covering 1.015 GB of space. Our results show that this trace needs 526/619/745 MB of memory to cover 97/98/99% of all requests, respectively.

The simulator calculates overall throughput, cache hit rate, average CPU and disk idle times at the back-end nodes, and other statistics. Throughput is the number of requests in the trace that were served per second by the entire cluster, calculated as the number of requests in the trace divided by the simulated time it took to finish serving all the requests in the trace. The request arrival rate was matched to the aggregate throughput of the server.

6.1 Simulation Results

In this section we present simulation results comparing the following mechanisms/policy combinations.

1. TCP single handoff with LARD on HTTP/1.0 workload [simple-LARD]

2. TCP single handoff with LARD on HTTP/1.1 workload [simple-LARD-PHTTP]

3. TCP multiple handoff with extended LARD on HTTP/1.1 workload [multiHandoff-extLARD-PHTTP]

4. Back-end forwarding with extended LARD on HTTP/1.1 workload [BEforward-extLARD-PHTTP]

5. Ideal handoff with extended LARD on HTTP/1.1 workload [zeroCost-extLARD-PHTTP]

Most of these mechanisms have already been described in Section 3. The "ideal handoff" is an idealized mechanism that incurs no overhead for reassigning a persistent connection to another back-end node. It is useful as a benchmark, as performance results with this mechanism provide a ceiling for results that can be obtained with any practical request distribution mechanism.

Figures 7 and 8 show the throughput results with the Apache and Flash Web servers, respectively, running on the back-end nodes. For comparison, results for the widely used Weighted Round-Robin (WRR) policy are also included, on HTTP/1.0 and HTTP/1.1 workloads.

When driving simple LARD with a HTTP/1.1 workload (simple-LARD-PHTTP), results show that the throughput suffers considerably (up to 39% with Apache and up to 54% with Flash), particularly at small to medium cluster sizes. The loss of locality more than offsets the reduced server overhead of persistent connections.

The key result, however, is that the extended LARD policy both with the multiple handoff mechanism and the back-end forwarding mechanism (multiHandoff-extLARD-PHTTP and BEforward-extLARD-PHTTP) are within 8% of the ideal mechanism and afford throughput gains of up to 20% when compared to simple-LARD. Moreover, the throughput achieved with each mechanism is within 6%, confirming that both mechanisms are competitive on today's Web workloads.

The performance of LARD with HTTP/1.1 (simple-LARD-PHTTP) catches up with that of the extended LARD schemes for larger clusters. The reason is as follows. With a sufficient number of back-end nodes, the aggregate cache size of the cluster becomes much larger than the working set, allowing each back-end to cache not only the targets assigned to it by the LARD policy, but also additional targets requested in HTTP/1.1 connections. Eventually, enough targets are cached in each back-end node to yield high cache hit rates not only for the first request in a HTTP/1.1 connection, but also for subsequent requests. As a result, the performance approaches (but cannot exceed) that of the extended LARD strategies for large cluster sizes.

WRR cannot obtain throughput advantages from the use of persistent connections on our workload, as it remains disk bound for all cluster sizes and is therefore unable to capitalize on the reduced CPU overhead of persistent connections. As previously reported [23], simple-LARD outperforms WRR by a large margin as the cluster

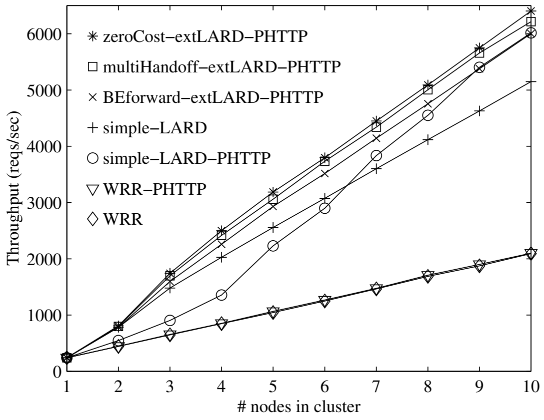

Figure 7: Apache Throughput

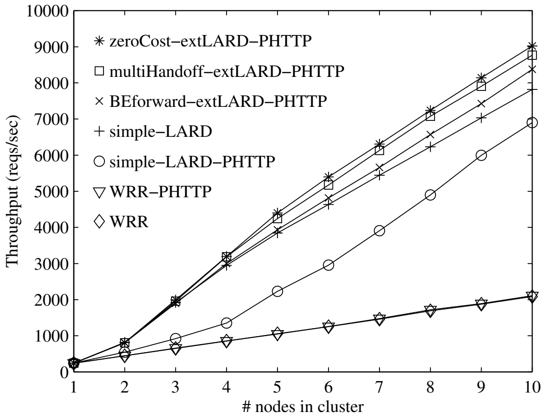

Figure 8: Flash Throughput

size increases, because it can aggregate the node caches. With one server node, the performance with HTTP/1.1 is identical to HTTP/1.0, because the back-end servers are disk bound with all policies.

The results obtained with the Flash Web server, which are likely to predict future trends in Web server software performance, differ mainly in that the performance loss of simple-LARD-PHTTP is more significant than with Apache. This underscores the importance of an efficient mechanism for handling persistent connections in cluster servers with content-based request distribution.

The throughput gains afforded by the hypothetical ideal handoff mechanism might also be achievable by a powerful relaying front-end (see Section 3.1) *as long as* it is not a bottleneck. However, as shown in Figures 7 and 8, such a front-end achieves only 8% better throughput than the back-end forwarding mechanism used with the extended LARD policy.

7 Prototype Cluster Design

This section describes the design of our prototype cluster. Given the complexity of the TCP multiple handoff mechanism, and the fact that simulation results indicate no substantial performance advantages of multiple handoff over back-end request forwarding, we decided to implement the back-end forwarding mechanism in the prototype.

Section 7.1 gives an overview of the various components of the cluster. Section 7.2 describes the TCP single handoff protocol. Section 7.3 describes *tagging*, a technique by which the front-end instructs the connection handling node to forward a given request to another back-end node. In Section 7.4, we describe how the back-end nodes fetch requests remotely from other nodes in a manner that keeps the server applications unchanged.

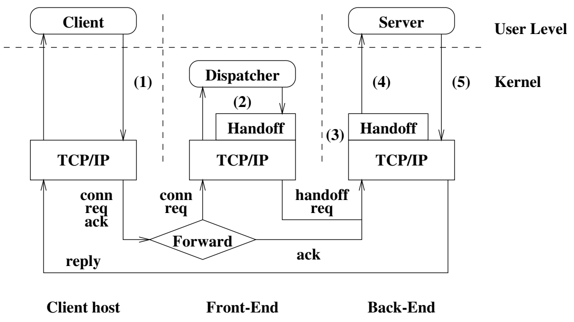

Figure 9: TCP connection handoff

7.1 Overview

The cluster consists of a front-end node connected to the back-end nodes with a high-speed LAN. HTTP clients are not aware of the existence of the back-end nodes, and the cluster effectively provides the illusion of a single Web server machine to the clients.

Figure 9 shows the user-level processes and protocol stacks at the client, the front-end and the back-ends. The client application (e.g., Web browser) is unchanged and runs on an unmodified standard operating system. The server process at the back-end machines is also unchanged, and can be any off-the-shelf Web server application (e.g., Apache [2], Zeus [31]). The front-end and back-end protocol stacks, however, employ some additional components, which are added via a loadable kernel module.

The front-end and back-end nodes use the TCP single handoff protocol, which runs over the standard TCP/IP to provide a control session between the front-end and the back-end machine. The LARD and extended LARD policies are implemented in a *dispatcher* module at the front-end. In addition, the front-end also contains a *forwarding module*, which will be described in Section 7.2. The front-end and back-end nodes also have a user-level startup process (not shown in Figure 9) that is used to initialize the dispatcher and setup the control sessions between the front-end and the back-end handoff protocols. After initializing the cluster, these processes remain kernel resident and provide a process context for the dispatcher and the handoff protocols. Disk queue lengths at the back-end nodes are conveyed to the front-end using the control sessions mentioned above.

7.2 TCP Connection Handoff

Figure 9 illustrates a typical handoff: (1) the client process (e.g., Netscape) uses the TCP/IP protocol to connect to the front-end, (2) the *dispatcher* module at the front-end accepts the connection, and hands it off to a back-end using the TCP handoff protocol, (3) the back-end takes over the connection using its handoff protocol, (4) the server at the back-end accepts the created connection, and (5) the server at the back-end sends replies directly to the client.

The handoff remains transparent to the client in that all packets from the connection handling node appear to be coming from the front-end. All TCP packets from the client are forwarded by the front-end's forwarding module to the connection handling back-end. A copy of any packets containing requests from the client is sent up to the dispatcher to enable it to assign the requests to back-end nodes. HTTP/1.1 request pipelining [19, 21] is fully supported by the handoff protocol, and allows the clients to send multiple requests without waiting for responses from previous requests.

The TCP multiple handoff mechanism discussed in Section 3.2 can be implemented by extending the above design in the following manner. As soon as the back-end server at the connection-handling node indicates that it has sent all requisite data to the client, the handoff protocol at the back-end can *hand-back* the connection to the front-end that can further hand it to another back-end. Alternatively, the connection can be handed directly to another back-end after informing the front-end to forward future packets from the client appropriately. One of the main challenges in this design is to prevent the TCP pipeline from draining during the process of a handoff.

7.3 Tagging requests

As mentioned in the previous subsection, the forwarding module sends a copy of all request packets to the dispatcher once the connection has been handed off. Assignment of subsequent requests on the connection to back-end nodes other than the connection handling node is accomplished by *tagging* the request content. The dispatcher sends these requests reliably to the connection handling back-end using the control session between the handoff protocol modules. The handoff protocol at the back-end receives the requests, and places them directly into the Web server's socket buffer. The tags enable the Web server to fetch the target using back-end forwarding

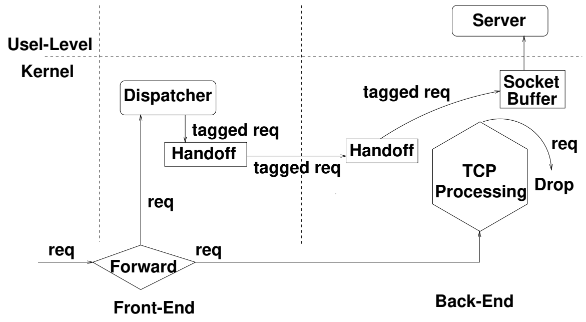

Figure 10: Tagging P-HTTP requests

(see Section 7.4). It remains, however, unaware of the presence of the handoff protocol.

After the handoff, all packets from the client are sent by the forwarding module to the connection handling node where they undergo TCP processing. Thus, after the handoff, data packets from the client are acknowledged by the connection handling node. The contents of these request packets, once received, are however discarded by the connection handling node (see Figure 10). Instead, the tagged requests received from the front-end via the control connection are delivered to the server process.

7.4 Fetching remote requests

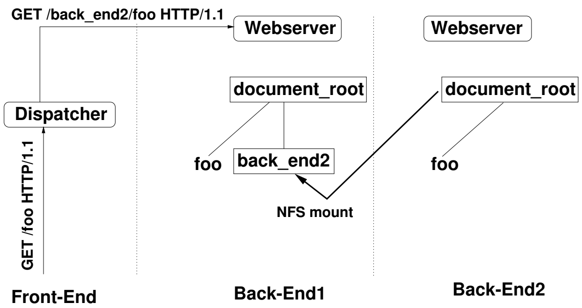

Figure 11: Transparent remote request fetching

Web server applications typically serve documents from a user-configurable directory which we will refer to as *document_root*. To implement remote fetching transparently to the Web server application, each back-end node NFS mounts the document_root from other back-end nodes on a subdirectory in its own document_root directory. Tagging is accomplished by the front-end dispatcher changing the URL in the client requests by prepending the name of the directory corresponding to the remote back-end node. Figure 11 depicts the situation where the dispatcher tags an HTTP GET request by prepending *back-end2* to the URL in order to make *back-end1* fetch file *foo* using NFS.

An issue concerning the fetching of remote files is NFS client caching, which would result in caching of targets at multiple back-end nodes and interfere with LARD's ability to control cache replication. To avoid this problem, we made a small modification in FreeBSD to disable client side caching of NFS files.

8 Prototype Cluster Performance

In this section, we present performance results obtained with a prototype cluster.

8.1 Experimental Environment

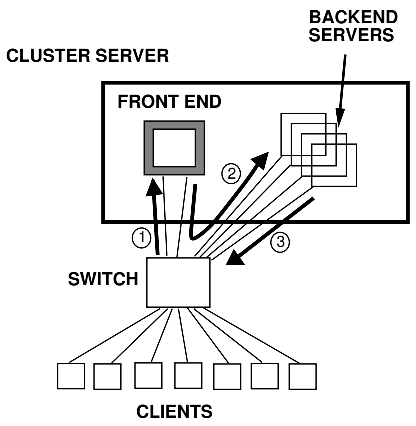

Figure 12: Experimental Testbed

Our testbed consists of a number of client machines connected to a cluster server. The configuration is shown in Figure 12. Traffic from the clients flows to the front-end (1) and is forwarded to the back-ends (2). Data packets transmitted from the back-ends to the clients bypass the front-end (3).

The front-end of the server cluster is a 300MHz Intel Pentium II based PC with 128MB of memory. The cluster back-end consists of six PCs of the same type and configuration as the front-end. All machines run FreeBSD 2.2.6. A loadable kernel module was added to the OS of the front-end and back-end nodes that implements the TCP single handoff protocol, and, in the case of the front-end, the forwarding module. The clients are seven 166MHz Intel Pentium Pro PCs, each with 64MB of memory.

The clients and back-end nodes in the cluster are connected using switched Fast Ethernet (100Mbps). The front-end and the back-end nodes are equipped with two network interfaces, one for communication with the clients, one for internal communication. Clients, front-end, and back-ends are connected through a single 24-port switch. All network interfaces are Intel EtherExpress Pro/100B running in full-duplex mode.

The Apache-1.3.3 [2] server was used on the back-end nodes. Our client software is an event-driven program that simulates multiple HTTP clients. Each simulated HTTP client makes HTTP requests as fast as the server cluster can handle them.

8.2 Cluster Performance Results

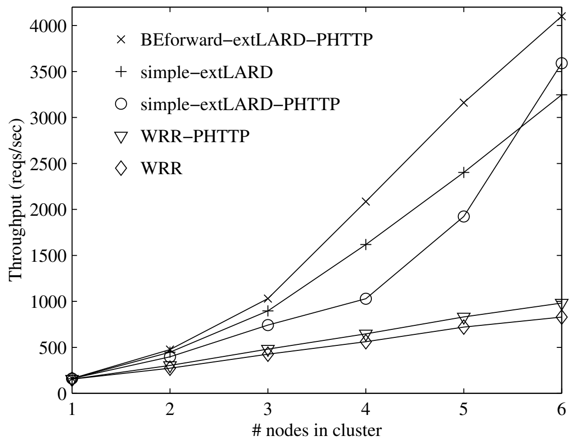

Figure 13: HTTP Throughput (Apache)

We used a segment of the Rice University trace alluded to in Section 6 to drive our prototype cluster. A single back-end node running Apache 1.3.3 can deliver about 151 req/s on this trace.

The Apache Web server relies on the file caching services of the underlying operating system. FreeBSD uses a unified buffer cache, where cached files are competing with user processes for physical memory pages. All page replacement is controlled by FreeBSD's page-out daemon, which implements a variant of the clock algorithm [18]. The cache size is variable, and depends on main memory pressure from user applications. In our 128MB back-ends, memory demands from kernel and Apache server processes leave about 100MB of free memory. In practice, we observed file cache sizes between 70 and 95 MB.

The mechanism used for the WRR policy is similar to the simple TCP handoff in that the data from the back-end servers is sent directly to the clients. However, the assignment of connections to back-end nodes is purely load-based.

Several observations can be made from the results presented in Figure 13. The measurements largely confirm the simulation results presented in Section 6.1. Contrary to the simulation results, WRR realizes modest performance improvements on HTTP/1.1 on this disk-bound workload. We believe that HTTP/1.1 reduces the memory demands of the Apache server application, and therefore leaves more room for the file system cache, causing better hit rates. This effect is not modeled by our simulator.

The extended LARD policy with the back-end forwarding mechanism affords four times as much throughput as WRR both with or without persistent connections and up to 26% better throughput with persistent connections than without. Without a mechanism for distributing HTTP/1.1 requests among back-end nodes, the LARD policies perform up to 35% worse in the presence of persistent connections.

Running extended LARD with the back-end forwarding mechanism and with six back-end nodes results in a CPU utilization of about 60% at the front-end. This indicates that the front-end can support 10 back-ends of equal CPU speed. Scalability to larger cluster sizes can be achieved by employing an SMP based front-end machine.

9 Related Work

Padmanabhan and Mogul [22] have shown that HTTP/1.0 connections can increase server resource requirements, the number of network packets per request, and effective latency perceived by the client. They proposed persistent connections and pipelining of HTTP requests, which have been adopted by the HTTP/1.1 standard [11]. The work in [19, 21] shows that these techniques dramatically improve HTTP/1.0 inefficiencies. Our work provides efficient support for HTTP/1.1 on cluster based Web servers with content-based request distribution.

Heidemann [13] describes performance problems arising from the interactions between P-HTTP and TCP in certain situations. The work also proposes some fixes that improve performance. The proposed solutions are complimentary to our work and can be applied in our cluster environment. In fact, most of the proposed fixes are already incorporated in Apache 1.3.3 [2].

Much current research addresses the scalability problems posed by the Web. The work includes cooperative caching proxies inside the network, push-based document distribution, and other innovative techniques [20, 7, 10, 16, 17, 27]. Our proposal addresses the complementary issue of providing support for HTTP/1.1 in cost-effective, scalable network servers.

Network servers based on clusters of workstations are starting to be widely used [12]. Several products are available or have been announced for use as front-end nodes in such cluster servers [8, 15]. To the best of our knowledge, the request distribution strategies used in the cluster front-ends are all variations of weighted round-robin, and do not take into account a request's target content. An exception is the Dispatch product by Resonate, Inc., which supports content-based request distribution [26]. The product does not appear to use any dynamic distribution policies based on content and no attempt is made to achieve cache aggregation via content-based request distribution.

Hunt et al. proposed a TCP option designed to enable content-based load distribution in a cluster server [14]. The design is roughly comparable in functionality to our TCP single handoff protocol, but has not been implemented.

Fox et al. [12] report on the cluster server technology used in the Inktomi search engine. The work focuses on the reliability and scalability aspects of the system and is complementary to our work. The request distribution policy used in their systems is based on weighted round-robin.

Loosely-coupled distributed servers are widely deployed on the Internet. Such servers use various techniques for load balancing including DNS round-robin [6], HTTP client re-direction [1], Smart clients [30], source-based forwarding [9] and hardware translation of network addresses [8]. Some of these schemes have problems related to the quality of the load balance achieved and the increased request latency. A detailed discussion of these issues is made in the work by Goldszmidt and Hunt [15] and Damani et al. [9]. None of these schemes support content-based request distribution.

10 Conclusions

Persistent connections pose problems for cluster based Web servers that use content-based request distribution, because requests that appear in a single connection may have to be served by different back-end nodes. We describe two efficient mechanisms for distributing requests arriving on persistent connections, TCP multiple handoff and back-end request forwarding.

A simulation study shows that both mechanisms can efficiently handle Web workloads on persistent connections. Moreover, we extend the *locality aware request distribution* (LARD) strategy to work with back-end request forwarding and show that it yields performance that is within 8% of results obtained with a simulated idealized mechanism. The proposed policies and mechanisms are fully transparent to the HTTP clients.

Finally, we have implemented the extended LARD policy and the back-end request forwarding mechanism in a prototype cluster. Performance results indicate that the extended LARD strategy affords up to 26% improvement in throughput with persistent connections over HTTP/1.0. Our results also indicate that a single front-end CPU can support up to 10 back-end nodes of equal speed.

In this paper, we have focused on studying HTTP servers that serve static content. Further research is needed for supporting request distribution mechanisms and policies for dynamic content.

11 Acknowledgments

We would like to thank Erich Nahum and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, MIP-9521386, by Texas TATP Grant 003604, and by an IBM Partnership Award.

References

[1] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proccedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[2] Apache. http://www.apache.org/.

[3] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[4] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS Conference*, Madison, WI, July 1998.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext transfer protocol – HTTP/1.0, May 1996.

[6] T. Brisco. DNS Support for Load Balancing. RFC 1794, Apr. 1995.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, Jan. 1996.

[8] Cisco Systems Inc. LocalDirector. http://www.cisco.com.

[9] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[10] P. Danzig, R. Hall, and M. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of the ACM SIGCOMM '93 Conference*, Sept. 1993.

[11] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. RFC 2068: Hypertext transfer protocol – HTTP/1.1, Jan. 1997.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[13] J. Heidemann. Performance interactions between P-HTTP and TCP implementations. *ACM Computer Communication Review*, 27(2):65–73, April 1997.

[14] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[15] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[16] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[17] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push-based distribution substrate for Internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[19] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Symposium*, 1995.

[20] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM '97 Symposium*, Cannes, France, Sept. 1997.

[21] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM '97 Symposium*, Cannes, France, Sept. 1997.

[22] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[23] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

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

[25] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[26] Resonate Inc. Resonate dispatch. http://www.resonateinc.com.

[27] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[28] SPECWeb96. http://www.specbench.org/osg/web96/.

[29] W. Stevens. *TCP/IP Illustrated Volume 1 : The Protocols*. Addison-Wesley, Reading, MA, 1994.

[30] B. Yoshikawa et al. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, Jan. 1997.

[31] Zeus. http://www.zeus.co.uk/.