Scalable Content-aware Request Distribution in Cluster-based Network Servers *

Mohit Aron Darren Sanders Peter Druschel
Willy Zwaenepoel
Department of Computer Science, Rice University, Houston, TX 77005

Abstract

We present a scalable architecture for content-aware request distribution in Web server clusters. In this architecture, a level-4 switch acts as the point of contact for the server on the Internet and distributes the incoming requests to a number of back-end nodes. The switch does *not* perform any content-based distribution. This function is performed by *each* of the back-end nodes, which may forward the incoming request to another back-end based on the requested content.

In terms of scalability, this architecture compares favorably to existing approaches where a front-end node performs content-based distribution. In our architecture, the expensive operations of TCP connection establishment and handoff are distributed among the back-ends, rather than being centralized in the front-end node. Only a minimal additional latency penalty is paid for much improved scalability.

We have implemented this new architecture, and we demonstrate its superior scalability by comparing it to a system that performs content-aware distribution in the front-end, both under synthetic and trace-driven workloads.

1 Introduction

Servers based on clusters of workstations or PCs are the most popular hardware platform used to meet the ever growing traffic demands placed on the World Wide Web. This hardware platform combines a cutting edge price—performance ratio in the individual server nodes with the promise of perfect scalability as additional server nodes are added to meet increasing demands. However, in order to actually attain scalable performance, it is essential that scalable mechanisms and policies for request distribution and load balancing be employed. In this paper, we present a novel, scalable mechanism for content-aware request distribution in cluster based Web servers.

State-of-the-art cluster-based servers employ a specialized front-end node, which acts as a single point of contact for clients and distributes requests to back-end nodes in the cluster. Typically, the front-end distributes requests such that the load among the back-end nodes remains balanced. With *content-aware request distribution*, the front-end additionally takes into account the content or type of service requested when deciding which back-end node should handle a given request.

Content-aware request distribution can improve scalability and flexibility by enabling the use of a partitioned server database and specialized server nodes. Moreover, previous work [6, 26, 31] has shown that by distributing requests based on cache affinity, content-aware request distribution can afford significant performance improvements compared to strategies that only take load into account.

While the use of a centralized request distribution strategy on the front-end affords simplicity, it can limit the scalability of the cluster. Very fast layer-4 switches are available that can act as front-ends for clusters where the request distribution strategies do not consider the requested content [11, 21]. The hardware based switch fabric of these layer-4 switches can scale to very large clusters. Unfortunately, layer-4 switches cannot efficiently support content-aware request distribution because the latter requires layer-7 (HTTP) processing on the front-end to determine the requested content. Conventional, PC/workstation based front-ends, on the other hand, can only scale to a relatively small num-

*Appears in Proceedings of the 2000 Annual Usenix Technical Conference, San Diego, CA, June 2000.

ber of server nodes (less than ten on typical Web workloads [26].)

In this paper, we investigate scalable mechanisms for content-aware request distribution. We propose a cluster architecture that decouples the request distribution strategy from the *distributor*, which is the front-end component that interfaces with the client and that distributes requests to back-ends. This has several advantages: (1) it dramatically improves the scalability of the system by enabling multiple distributor components to co-exist in the cluster, (2) it improves cluster utilization by enabling the distributors to reside in the back-end nodes, and (3) it retains the simplicity afforded by a centralized request distribution strategy.

The rest of the paper is organized as follows. Section 2 presents some background information for the rest of the paper. In Section 3, we discuss the cluster configurations currently used for supporting content-aware request distribution and we provide experimental results that quantify limitations in their scalability. Section 4 describes our scalable cluster design for supporting content-aware request distribution. We discuss our prototype implementation in Section 5 and Section 6 presents experimental results obtained with the prototype. Related work is covered in Section 7 and Section 8 presents conclusions.

2 Background

This section provides background information on content-aware request distribution and the existing mechanisms that support it.

2.1 Content-aware Request Distribution

Content-aware request distribution is a technique employed in cluster-based network servers, where the request distribution strategy 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 most commercial high performance cluster servers [11, 21] 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-aware 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). Locality-aware request distribution (LARD) is a specific strategy for content-aware request distribution that focuses on the first of the advantages cited above, namely improved cache hit rates in the back-ends [6, 26]. LARD improves cluster performance by simultaneously achieving load balancing and high cache hit rates at the back-ends.

In order to inspect the content of the requests, a TCP connection must be established with the client *prior* to assigning the request to a back-end node. This is because, with content-aware request distribution, the nature and target[1] of the client's request influences the assignment. Therefore, a mechanism is required that allows a chosen back-end node to serve a request on a TCP connection that was established elsewhere in the cluster. For reasons of performance, security and interoperability, it is desirable that this mechanism be transparent to the client. We discuss such mechanisms in the next subsection.

2.2 Mechanisms

A simple client-transparent mechanism for supporting content-aware request distribution is a *relaying front-end* depicted in Figure 1. An HTTP proxy running on the front-end accepts client connections, and maintains persistent connections with all the back-end nodes. When a request arrives on a client connection, the connection is assigned according to a request distribution strategy (e.g., LARD), and the proxy forwards the client's request message on the appropriate back-end connection. When the response arrives from the back-end node, the front-end proxy forwards the data on the client connection, buffering the data if necessary. The principal advantage of this approach is its simplicity and the

[1] The term *target* refers to a Web document, specified by a URL and any applicable arguments to the HTTP GET command.

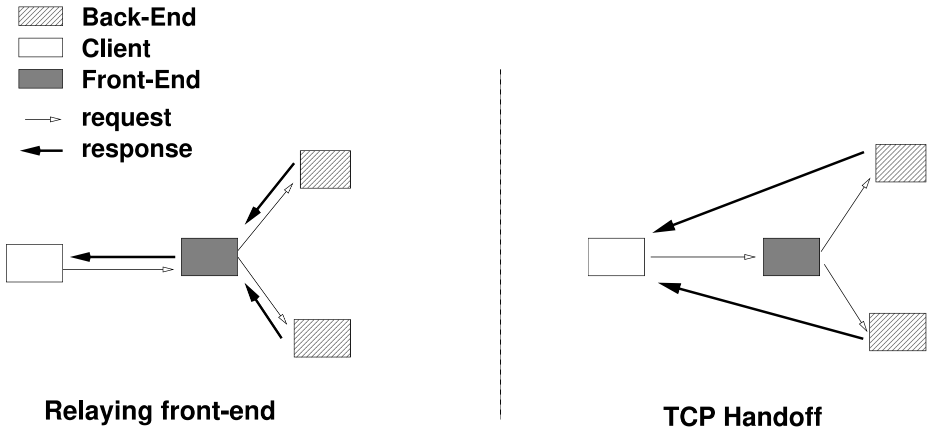

Figure 1: **Mechanisms for request distribution**

ability to be deployed without requiring any modification of the operating system kernel on any cluster node. The primary disadvantage, however, is the overhead incurred in forwarding all the response data from the back-ends to the clients, via the proxy application.

TCP splicing [15, 12] is an optimization of the front-end relaying approach where the data forwarding at the front-end is done directly in the operating system kernel. This eliminates the expensive copying and context switching operations that result from the use of a user-level application. While TCP splicing has lower overhead than front-end relaying, the approach still incurs high overhead because all HTTP response data must be forwarded via the front-end node. TCP splicing requires modifications to the OS kernel of the front-end node.

The TCP handoff [26] mechanism was introduced to enable the forwarding of back-end responses directly to the clients without passing through the front-end as an intermediary. This is achieved by handing off the TCP connection established with the client at the front-end to the back-end where the request was assigned. TCP handoff effectively serializes the state of the existing client TCP connection at the front-end, and instantiates a TCP connection with that given starting state at the chosen back-end node. The mechanism remains transparent to the clients in that the data sent by the back-ends appears to be coming from the front-end and any TCP acknowledgments sent by the client to the front-end are forwarded to the appropriate back-end. TCP handoff requires modifications to both the front-end and back-end nodes, typically via a loadable kernel module that plugs into a general-purpose UNIX system.

While TCP handoff affords substantially higher scalability than TCP splicing by virtue of eliminating the forwarding overhead of response data, our experiments with a variety of trace-based workloads indicate that its scalability is in practice still limited to cluster sizes of up to ten nodes. The reason is that the front-end must establish a TCP connection with each client, parse the HTTP request header, make a strategy decision to assign the connection, and then serialize and handoff the connection to a back-end node. The overhead of these operations is substantial enough to cause a bottleneck with more than approximately ten back-end nodes on typical Web workloads.

3 Scalability of a single Front-end

Despite the use of splicing or handoff, a single front-end node limits the scalability of clusters that employ content-based request distribution. This section presents experimental results that quantify the scalability limits imposed by a conventional, single front-end node.

To measure the scalability of the splicing and TCP handoff mechanisms, we conducted experiments with the configurations depicted in Figure 1. Our testbed consists of a number of client machines, connected to a cluster server. The cluster nodes are 300MHz Intel Pentium II based PCs with 128MB of memory. All machines are configured with the FreeBSD-2.2.6 operating system.

The requests were generated by a HTTP client program designed for Web server benchmarking [8]. The program generates HTTP requests as fast as the Web server can handle them. Seven 166 MHz Pentium Pro machines configured with 64MB of memory were used as client machines. The client machines and all cluster nodes are connected via switched 100Mbps Ethernet. The Apache-1.3.3 [3] Web server was used at the server nodes.

For experiments with TCP handoff, a loadable kernel module was added to the OS of the front-end and back-end nodes that implements the TCP handoff protocol. The implementation of the TCP handoff protocol is described in our past work [6, 26]. For splicing, a loadable module was added to the front-end node. Persistent connections were established between the front-end node and the back-end webservers for use by the splicing front-end.

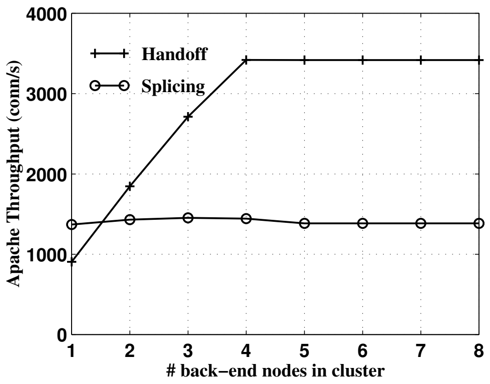

Figure 2: **Throughput, 6 KB requests**

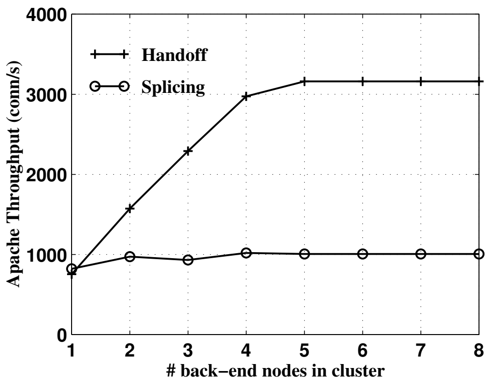

Figure 3: **Throughput, 13 KB requests**

In the first experiment, the clients repeatedly request the same Web page of a given size. Under this artificial workload, the LARD policy behaves like WRR: it distributes the incoming requests among all back-end nodes in order to balance the load.

Figures 2 and 3 compare the cluster throughput with the handoff versus the splicing mechanism as a function of the number of back-end nodes in the cluster, and for requested Web page sizes of 6 KB and 13 KB, respectively. These two page sizes correspond to the extrema of the range of average HTTP transfer sizes reported in the literature [24, 4]. Since the requested page remains cached in the servers' main memory with this synthetic workload, the server displays very high throughput, thus fully exposing scalability limitations in the front-end request distribution mechanism.

The results show that the TCP handoff mechanism scales to four back-end nodes, while splicing is already operating at front-end capacity with only one server node. In either case, the scalability is limited because the front-end CPU reaches saturation.

For the 6 KB files, the performance of splicing exceeds that of handoff at a cluster size of one node. The reason is that splicing uses persistent connections to communicate with the back-end servers, thus shifting the per-request connection establishment overhead to the front-end, which results in a performance advantage in this case. For larger cluster sizes and large files size, this effect is more than compensated by the greater efficiency of handoff, and by the fact that splicing saturates the front-end.

Additionally, it should be noted that with the larger page size (13 KB), the throughput with splicing degrades more than that with handoff (27% versus 7%, respectively). This is intuitive because with splicing, the higher volume of response data has to pass through the front-end, while with handoff, the front-end only incurs the additional cost of forwarding more TCP acknowledgments from the client to the back-ends.

In general, the maximal throughput achieved by the cluster with a single front-end node is fixed for any given workload. For example, the throughput with the handoff mechanism is determined by (a) the rate at which connections can be established and handed off to back-end nodes and (b) the rate at which TCP acknowledgments can be forwarded to back-end nodes. With slow back-end webservers and/or with workloads that cause high per-request overhead (e.g., frequent disk accesses) at the back-end nodes, the maximum throughput afforded by a

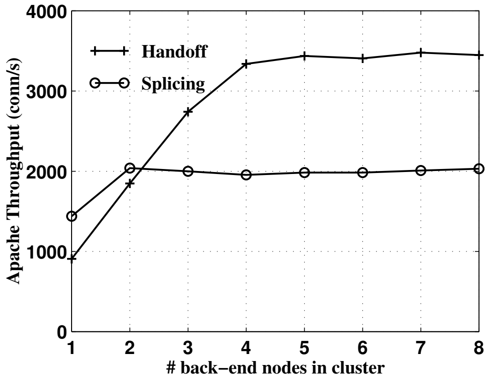

Figure 4: **Throughput, IBM trace**

back-end is lower and the front-end is able to support more back-end nodes before becoming a bottleneck. With efficient back-end webservers and/or workloads with low per-request overheads, a smaller number of back-ends can be supported.

The results shown above were obtained with a synthetic workload designed to expose the limits of scalability. Our next experiment uses a realistic, trace-based workload and explores the scalability of handoff and splicing under real workload conditions.

Figure 4 shows the achieved throughput using handoff versus splicing on a trace based workload derived from www.ibm.com's server logs (details about this trace are given in Section 6). As in the previous experiments, the handoff scales linearly to a cluster size of four nodes. The splicing mechanism scales to two back-end nodes on this workload. Thus, splicing scales better on this workload than on the previous, synthetic workload. The reason is that the IBM trace has an average page size of less than 3 KB[2]. Since splicing overhead is more sensitive to the average page size for the reasons cited above, it benefits from the low page size more than handoff.

The main result is that both splicing and handoff only scale to a small number of back-end nodes on realistic workloads. Despite the higher performance afforded by TCP handoff, its peak throughput is limited to about 3500 conn/s on the IBM trace and it does not scale well beyond four cluster nodes. In Section 6 we show that a software based layer-4 switch implemented using the same hardware can afford a throughput of up to 20,000 conn/s. Therefore, the additional overhead imposed by content-aware request distribution reduces the scalability of the system by an order of magnitude.

[2]This is substantially less than the average Web page size reported by several studies. The likely reason is that the content designers of this high-volume site have minimized the size of the most popular pages for performance reasons.

The limited scalability of content-aware request distribution cannot be easily overcome through the use of multiple front-end nodes. Firstly, employing multiple front-end nodes introduces a secondary load balancing problem among the front-end nodes. Mechanisms like round-robin DNS are known to result in poor load balance. Secondly, many content-aware request distribution strategies (for instance, LARD) require centralized control and cannot easily be distributed over multiple front-end nodes.

In Section 4 we describe the design of a scalable content-aware request distribution mechanism that maintains centralized control. Results in Section 6 indicate that this cluster is capable of achieving an order of magnitude higher performance than existing approaches.

4 Scalable Cluster Design

This section addresses the scalability problem with content-aware request distribution. We identify the bottlenecks and propose a configuration that is significantly more scalable.

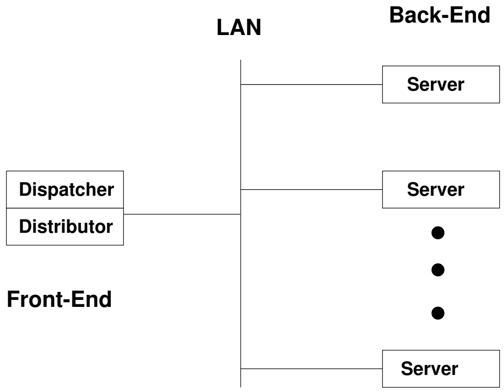

Figure 5: **Cluster components**

Figure 5 shows the main components comprising a cluster configuration with content-aware request distribution and a single front-end. The *dispatcher* is the component that implements the request distribution strategy; its task is to decide which server node should handle a given request. The *distrib-*

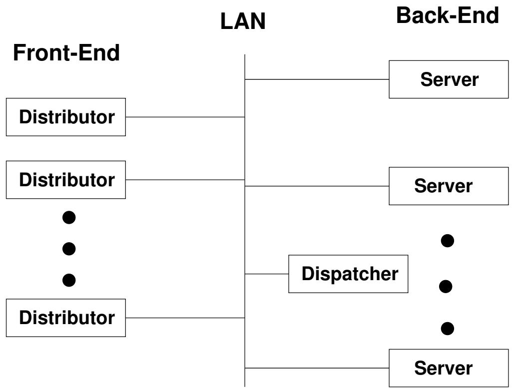

Figure 6: **Multiple front-ends**

utor component interfaces with the client and implements the mechanism that distributes the client requests to back-end nodes; it implements either a form of TCP handoff or the splicing mechanism. The *server* component represents the server running at the back-end and is responsible for processing HTTP client requests.

A key insight is that (1) the bulk of the overhead at the front-end node is incurred by the distributor component, not the dispatcher; and, (2) the distributor component can be readily distributed since its individual tasks are completely independent, while it is the dispatcher component that typically requires centralized control. It is intuitive then, that a more scalable request distribution can be achieved by distributing the distributor component over multiple cluster nodes, while leaving the dispatcher centralized on a dedicated node.

Experimental results show that for TCP handoff, the processing overhead for handling a typical connection is nearly 300 μsec for the distributor while it is only 0.8 μsec for the dispatcher. With splicing, the overhead due to the distributor is larger than 750 μsec and increases with the average response size. One would expect then, that distributing the distributor component while leaving only the dispatcher centralized should increase the scalability of the request distribution by an order of magnitude. Our results presented in Section 6 confirm this reasoning.

Figure 6 shows a cluster configuration where the distributor component is distributed across several front-end nodes while the dispatcher resides on a dedicated node. In such a configuration with multiple front-end nodes, a choice must be made as to which front-end should receive a given client request. This choice can be made either explicitly by the user with strategies like *mirroring*[3], or in a client transparent manner using DNS round-robin. However, these approaches are known to lead to poor load balancing [20], in this case among the front-end nodes.

Another drawback of the cluster configuration shown in Figure 6 is that efficient partitioning of cluster nodes into either front-end or back-end nodes depends upon the workload and is not known *a priori*. For example, for workloads that generate significant load on the back-end nodes (e.g, queries for online databases), efficient cluster utilization can be achieved by using a few front-end nodes and a large number of back-end nodes. For other workloads, it might be necessary to have a larger number of front-end nodes. A suboptimal partitioning, relative to the prevailing workload, might result in low cluster utilization, i.e, either the front-end nodes become a bottleneck when the back-end nodes are idle or vice versa.

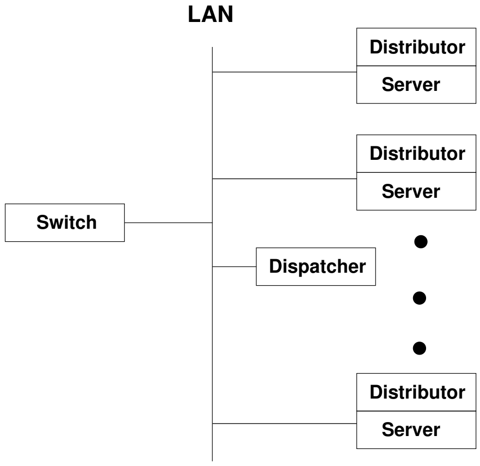

Figure 7: **Co-located distributors and servers**

Figure 7 shows an alternate cluster design where the distributor components are co-located with the server components. As each cluster node hosts both the distributor and the server components, the cluster can be efficiently utilized irrespective of the workload. As in the cluster of Figure 6, the replication of the distributor component on multiple cluster nodes eliminates the bottleneck imposed by a centralized distributor. In addition, a front-end consisting of a commodity layer-4 switch is employed

[3]With mirroring, the clients are aware of the multiple front-ends in the cluster and explicitly choose where to send their requests.

that distributes incoming client requests to one of the distributor components running on the back-end nodes, in such a way that the load among the distributors is balanced.

Notice that the switch employed in this configuration does not perform content-aware request distribution. It merely forwards incoming packets based on layer-4 information (packet type, port number, etc.). Therefore, a highly scalable, hardware based commercial Web switch product can be used for this purpose [21, 11]. In Section 6, we present experimental results with a software based layer-4 switch that we developed for our prototype cluster.

A potential remaining bottleneck with this design is the centralized dispatcher. However, experimental results presented in Section 6 show that a centralized dispatcher implementing the LARD policy can service up to 50,000 conn/s on a 300MHz Pentium II machine. This is an order of magnitude larger than the performance of clusters with a single front-end, as shown in Section 3.

In the next section, we provide a detailed description of our prototype implementation. In Section 6 we present experimental results that demonstrate the performance of our prototype.

5 Prototype Implementation

This section describes the implementation and operation of our prototype cluster. Section 5.1 gives an overview of the various components of the cluster. Section 5.2 provides details on the implementation of a software based layer-4 switch that we implemented for interfacing the cluster with the clients. Section 5.3 describes the sequence of operations in the cluster upon receiving a client request. Section 5.4 discusses techniques employed in our implementation to improve the scalability of the dispatcher by reducing the overhead of communicating with other cluster nodes.

5.1 Overview

The cluster consists of a number of nodes connected by a high speed LAN. As we mentioned in Section 4, there are three main components in the cluster – dispatcher, distributor and server. The interface design for these components is such that any cluster node can contain one or more of these components. This implies that any of the cluster configurations of Figure 5, Figure 6 or Figure 7 can be realized by placing the components appropriately on the cluster nodes.

The communication between components on different cluster nodes is realized using persistent TCP control connections that are created during the cluster initialization. A control connection, thus, exists between any two nodes of the cluster and multiplexes the messages exchanged between any two components. These connections also serve to detect node failures.

Owing to the superior performance afforded by the TCP handoff protocol as compared to splicing, our prototype distributor employs the handoff protocol. The server component consists of the user-level server application and an enhanced network protocol stack in the kernel capable of accepting connections using the handoff protocol. The server application can be any off-the-shelf web server (e.g., Apache [3], Zeus [30]) and requires no change for operation in the cluster. The distributor is also implemented as an enhanced protocol stack, and resides wholly in the kernel. Similarly, the dispatcher component also resides in the kernel for efficiency. The kernel changes required to implement the cluster components are added using a loadable kernel module for the FreeBSD-2.2.6 OS.

Our prototype implementation supports both HTTP/1.0 as well as HTTP/1.1 persistent connections (P-HTTP [24]). Support for P-HTTP is similar to that described in our earlier work [6]. As this paper focuses on scalability issues in the cluster, we only consider HTTP/1.0 connections in the experimental results presented in this paper.

A layer-4 switch is used that receives all requests from clients and forwards them to the distributors. With the switch, the distributed nature of the cluster becomes completely transparent to the clients. We next describe the implementation of this switch.

5.2 Layer-4 Switch

We implemented a fast software based layer-4 switch to be used as the front-end node. Even though hardware based, highly scalable layer-4 switches are commercially available, we decided to implement a software based switch for two reasons: we did not have a commercial layer-4 switch available to us, and we wanted to explore what level of scalability could be achieved with a software based switch.

The switch maintains a small amount of state for each client connection being handled by the cluster. This state is used to determine the cluster node where packets from the client are to be forwarded. Additionally, the switch maintains the load on each cluster node based on the number of client connections handled by that node.

Upon seeing a connection establishment packet (TCP SYN packet), the switch chooses a distributor on the least loaded node in the cluster. Subsequent packets from the client are sent to the same node unless an *update* message instructs the switch to forward packets to some other node (update messages are sent by the distributor after a TCP connection handoff). Upon connection termination, a *close* message is sent to the switch by the cluster node that handles the connection and is used for discarding connection state at the switch.

All outgoing data from the cluster is sent directly to the clients and does not pass through the switch. Only the packets sent by the clients are received and forwarded by the switch. To improve switch performance, the forwarding module in the switch avoids interrupts and uses soft timer based polling to receive network packets [5, 25]. In Section 6, we report the forwarding throughput of this switch.

Using a front-end layer-4 switch (as opposed to using, for instance, DNS round-robin to distribute requests among the server nodes) offers several important advantages. The first is increased security. By hiding the back-end nodes of the cluster from the outside world, would-be attackers cannot directly attack these nodes. The second advantage is availability. By making the individual cluster nodes transparent to the client, failed or off-line server nodes can be replaced transparently. Finally, when combined with TCP handoff, the use of a switch increases efficiency, as ACK packets from clients need not be forwarded by the server node that originally received a request.

Of course, a front-end switch forms a single point of failure in the cluster. There are, however, a number of possible solutions to this problem, such as using a hot-swappable spare.

5.3 Cluster Operation

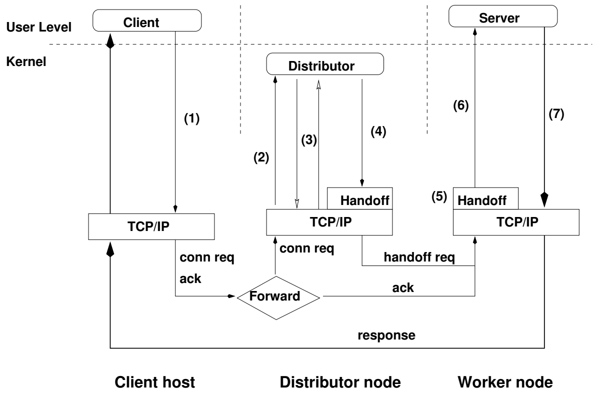

Figure 8: **Operation with DNS round-robin**

Figure 8 shows the operation of the cluster. For simplicity, we assume here that clients directly contact a distributor (for instance, via DNS round-robin). The figure shows the user-level processes and protocol stacks at the client and the distributor and server nodes. The client application (e.g., Web browser) is unchanged and runs on an unmodified standard operating system.

The figure illustrates the cluster operations from the time the client sends the request, to the instant when it receives a response from the cluster. (1) the client process (e.g., Netscape) uses the TCP/IP protocol to connect to the chosen distributor, (2) the distributor component accepts the connection and parses the client request, (3) the distributor contacts the dispatcher (not shown in the figure) for the assignment of the request to a server, (4) the distributor hands off the connection using the TCP handoff protocol to the server chosen by the dispatcher, (5) the server takes over the connection using its handoff protocol, (6) the server application at the server node accepts the created connection, and (7) the server sends the response directly to the client. Any TCP acknowledgments sent by the client to the distributor are forwarded to the server node using a *forwarding module* at the distributor node.

The operation in Figure 8 assumes that the server component chosen by the dispatcher resides on a different node than the distributor. If the server is on the same node as the distributor, then the handoff is accomplished efficiently using procedure calls in the kernel.

The operation of the cluster with a layer-4 switch is similar to that in Figure 8. However, there are some important differences. First, the choice of the distributor is made by the switch, using the WRR strategy. Second, after a connection is handed off by the distributor, the switch is notified and the subsequent forwarding of TCP acknowledgments to the corresponding server node is handled by the switch.

5.4 Batching Requests

As the dispatcher component is centralized, its performance determines the scalability of the cluster. We demonstrate in Section 6.1 that the bulk of the overhead imposed on the dispatcher node is associated with communication with other cluster nodes. The key to achieving greater cluster scalability, therefore, is to reduce this communication overhead.

The communication overhead can be reduced in two ways, (1) by reducing the size of the messages sent to the dispatcher, and (2) by limiting the number of network packets sent to the dispatcher. For (1), the distributor *compacts* the request URL by hashing it into a 32-bit integer before sending it to the dispatcher. (2) is achieved by *batching* together several messages before they are sent to the dispatcher. This enables several messages to be put in the same TCP packet and reduces interrupt and protocol processing overheads at the dispatcher node.

An interesting design choice is the degree of batching, i.e., the number of messages collected together before sending them to the dispatcher. Although the overhead on the dispatcher node is lower with a higher degree of batching, the response time is adversely affected because messages might be held for a long time before a batch of the requisite size is formed.

Rather than fixing the degree of batching, our implementation batches messages only as long as the replies for messages in the previous batch are outstanding. This has the beneficial effect of dynamically adjusting the degree of batching based on the load on the dispatcher node. If the dispatcher is heavily loaded, the time taken by it to respond to messages in an earlier batch is long, which in turn causes high degree of batching of the next set of messages. Similarly, a lightly loaded dispatcher would result in sets of messages where the degree of batching is low.

6 Experimental Results

In this section, we present performance results obtained with our prototype cluster. Section 6 presents performance results of the software layer-4 switch discussed in Section 5.2. In Section 6.1, we present performance characteristics of our cluster prototype such as scalability, throughput and latency. Finally, Section 6.2 provides experimental results on a real workload obtained from webserver logs. For all cluster experiments we used the configuration shown in Figure 7. The experimental setup used was the same as described in Section 3.

The first experiment determines the maximum throughput of our software layer-4 switch used as a front-end. The switch receives packets from seven client machines. The packet processing in the switch closely emulates the processing in actual operation with the cluster. Each packet was forwarded to another machine where it was discarded.

Our results show that the peak throughput afforded by the switch running on a 300MHz Pentium II machine is about 128,000 packets/s. In actual cluster operation, we have observed an average of about 5–6 packets for a typical HTTP/1.0 connection where the content size ranges from 5–13 KB. This implies that the maximum throughput afforded by the switch is about 20,000 conn/s.

Higher switch performance can be obtained by (1) using faster hardware for the switch, (2) using an SMP based machine for the switch, or (3) by using a commercial layer-4 switch. Hardware implementations of layer-4 switches are reported to achieve throughputs of over 70,000 conn/s [1].

6.1 Cluster Results

We repeated the experiments from Section 3 to demonstrate the scalability of our proposed cluster configuration. Figure 9 shows the throughput results with the Apache webserver as the number of nodes in the cluster (other than the one running the dispatcher) are increased. As shown in the figure, the throughput increases linearly with the size of the cluster. Figure 10 depicts the CPU idle time on the node running the dispatcher. The results clearly show that in this experiment, the dispatcher node is far from causing a bottleneck in the system.

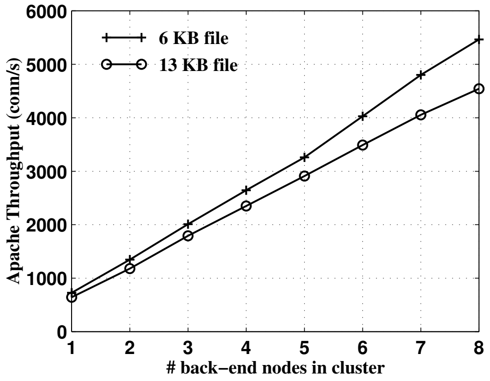

Figure 9: **Cluster Throughput**

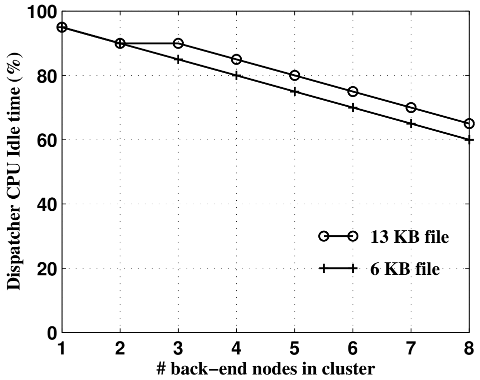

Figure 10: **Dispatcher CPU Idle Time**

A comparison with earlier results in Figure 2 and Figure 3 shows that the absolute performance in Figure 9 is less for comparable number of nodes in the cluster. In fact, the performance with N nodes in Figure 9 was achieved with $N-1$ back-end nodes in our earlier results with the conventional content-based request distribution based on handoff. The reason is that with the former approach, the front-end offloads the cluster nodes by performing the request distribution task, leaving more resources on the back-end nodes for request processing. However, this very fact causes the front-end to become a bottleneck, while our proposed approach scales with the number of back-end nodes.

Due to a lack of network bandwidth in our PIII cluster, we were unable to extend the above experiment so as to demonstrate when the dispatcher node becomes a bottleneck. However, we devised a different experiment to indirectly measure the peak throughput afforded by our system. We wrote a small program that generated a high rate of requests for the dispatcher node. Requests generated by this program appeared to the dispatcher as if they were requests from distributor nodes from a live cluster. This program was parameterized such that we were able to vary the degree of batching in each message sent to the dispatcher. Two messages to the dispatcher are normally required per HTTP request—one to ask for a server node assignment and one to inform the dispatcher that the client connection was closed. The degree of batching determines how many of these messages are combined into one network packet sent to the dispatcher.

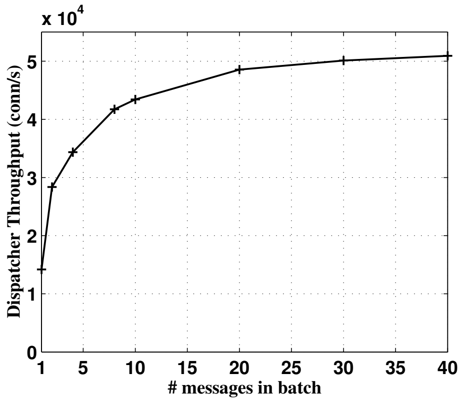

Figure 11: **Dispatcher Throughput**

Figure 11 shows the peak throughput afforded by the dispatcher as the degree of batching was increased from one to forty. These results show that the dispatcher node (300MHz PII) can afford a peak throughput of more than 50,000 conn/s. This number determines the peak performance afforded by our cluster and is an order of magnitude larger than that afforded by traditional clusters that employ content-aware request distribution. At this peak throughput, each HTTP connection imposes an overhead of about 20 μsec on the dispatcher node. The bulk of this overhead is attributed to the overheads associated with communication. The LARD request distribution strategy only accounts for 0.8 μsec of the overhead. Repeating this experiment using a 500MHz PIII PC as the dispatcher resulted in a peak throughput of 112,000 conn/s. Also, it is to be noted that the peak performance of the dispatcher is independent of the size of the content requested. In contrast, the scalability of cluster configurations with conventional content-aware request distribution decreases as the average content size increases.

We also measured the increase in latency due to the extra communication incurred as a result of the dispatcher being placed on a separate node. This extra hop causes an average increase of 170 μsec in latency and is largely due to round trip times in a LAN and protocol processing delays. When the layer-4 switch is used for interfacing with the clients, an additional 8 μsec latency is added due to packet processing delay in the switch. This increase in latency is insignificant in the Internet where WAN delays are usually larger than 50 ms. Even in LAN environments, the added latency is not likely to affect user-perceived response times.

6.2 Real Workload

We now present results from our prototype to demonstrate the scalability of the cluster when presented with real, trace-based workloads.

The workloads were obtained from logs of a Rice University Web site and from the www.ibm.com server, IBM's main Web site. The data set for the Rice trace consists of 31,000 targets covering 1.015 GB of space. This trace needs 526/619/745 MB of cache memory to cover 97/98/99% of all requests, respectively. The data set for the IBM trace consists of 38,500 targets covering 0.984 GB of space. However, this trace has a much smaller working set and needs only 43/69/165 MB of cache memory to cover 97/98/99% of all requests, respectively.

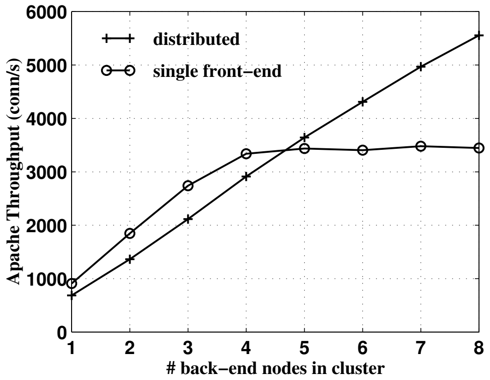

Figure 12: **Throughput, IBM Trace**

The results presented in Figures 12 and 13 clearly show that our proposed cluster architecture for content-aware request distribution scales far better than the state-of-the-art approach based on handoff at a single front-end node, on real workloads.

Note that for cluster sizes below five on the IBM trace, the performance with a single front-end node

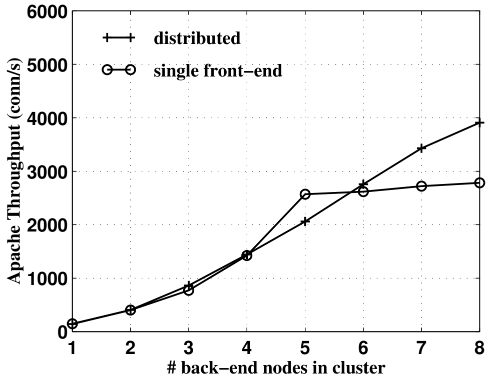

Figure 13: **Throughput, Rice Trace**

exceeds that of our distributed approach. The reason is the same as that noted earlier: with the former approach, the front-end offloads the cluster nodes by performing the request distribution task, leaving more resources on the back-end nodes for request processing. However, this very fact causes the front-end to bottleneck at a cluster size of five and above, while the distributed approach scales with the number of back-end nodes.

This effect is less pronounced in the Rice trace, owing to the much larger working set size in this trace, which renders disk bandwidth the limiting resource. For the same reason, the absolute throughput numbers are significantly lower with this workload.

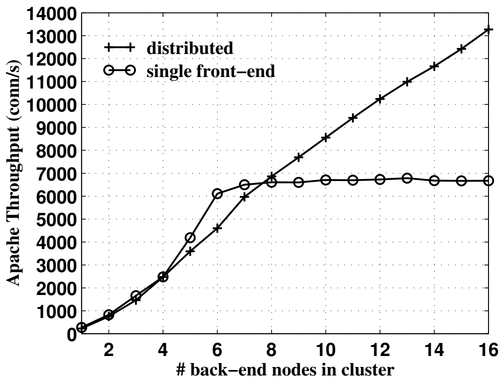

Figure 14: **Throughput, Rice Trace on 800MHz Athlon**

Finally, Figure 14 shows results obtained with the Rice trace on a new, larger cluster. The hardware used for this experiment consists of a cluster of 800MHz AMD Athlon based PCs with 256MB of main memory each. The cluster nodes, the node running the dispatcher, and the client machines are connected by a single 24 port Gigabit Ethernet switch. All machines run FreeBSD 2.2.6.

The results show that the performance of the cluster scales to a size of 16 nodes with no signs of slowing, despite the higher individual performance (faster CPU and disk, larger main memory) of the cluster nodes. The throughput obtained with 16 nodes exceeds 13,000 conn/s on this platform, while the single (800MHz) front-end based approach is limited to under 7,000 conn/s. Limited hardware resources (i.e., lack of a Gigabit Ethernet switch with more than 24 ports) still prevent us from demonstrating the scalability limits of our approach, but the measured utilization of the dispatcher node is consistent with our prediction that the approach should scale to at least 50,000 conn/s.

7 Related Work

A substantial body of work addresses the design of high-performance, scalable Web server clusters. The work includes cooperative caching proxies inside the network, push-based document distribution and other innovative techniques [7, 10, 13, 22, 23, 28]. Our proposal addresses the complementary issue of providing support for scalable network servers that perform content-based request distribution.

Web servers based on clusters of workstations/PCs are widely used [18]. Most commercial Web switch products for cluster servers use a request distribution strategy that does not require examining the content of the request [2, 20, 14, 9]. The most common such technique used for request distribution is some variant of weighted round-robin. Resonate, Inc. [27] is an exception in that their product offers content-aware request distribution using a method similar to TCP handoff.

Fox et al. [18] describe a layered architecture for building cluster-based network services. The architecture has a centralized load manager and several front-end and back-end nodes. This architecture is similar to the one shown in Figure 6; however, the request distribution strategy and the mechanisms employed are purely load-based and do not consider the content of the requests. Our work focuses on scalable cluster configurations for content-aware request distribution.

In [26], Pai et al. explore the use of content-based request distribution in a cluster Web server environment. This work presents an instance of a content-aware request distribution strategy, called LARD. The strategy achieves both locality, in order to increase hit rates in the Web servers' memory caches, and load balancing. Performance results with the LARD algorithm show substantial performance gains over WRR.

More recently, in [31], Zhang et al. explore another content-based request distribution algorithm that looks at static and dynamic content and also focuses on cache affinity. They confirmed the results of [26] by showing that focusing on locality can lead to significant improvements in cluster throughput.

The key to content-based request distribution is that a client's request is first inspected before a decision is made about which server node should handle the request. The difficulty lies therein, that in order to inspect a request, the client must first establish a connection with a node that will ultimately not handle the request. There are currently two known viable techniques that can be used to handle this situation. They are TCP splicing [15, 12, 29], and TCP handoff [6, 26, 19]. Our proposed approach offers a third alternative that scales well with the number of back-end nodes.

As mentioned earlier, the switch component of our cluster could easily be replaced by a commercial layer-4 switch. A number of layer-4+ network switch products [1, 16, 17, 11] are currently available on the market. These commercial products use specialized hardware and advertise high performance. A subset of these switches are also advertised to be layer-7 switches, which means they can perform URL-aware routing. We are not aware of any published performance results for these switches when used for URL-aware distribution. However, since software processing is involved in layer-7 switching, we expect that these products have similar limitations to scalability as software based content-aware front-ends when used for this purpose.

8 Conclusions

We have presented a new, scalable architecture for content-aware request distribution in Web server clusters. Content-aware distribution improves server performance by allowing partitioned secondary storage, specialized server nodes, and request distribution strategies that optimize for locality.

Our architecture employs a level-4 switch that acts as the central point of contact for the server on the Internet, and distributes the incoming requests to a number of back-ends. In particular, the switch does *not* perform any content-aware distribution. This function is performed by *each* of the back-ends, who may forward the incoming request to another back-end based on the requested content. In order to make their request distribution decisions, the back-ends access a dispatcher node that implements the request distribution policy.

In terms of scalability, the proposed architecture compares favorably with existing approaches, where a front-end node performs content-aware request distribution. In our architecture, the expensive operations of TCP connection establishment and handoff are distributed over all back-end nodes, rather than being centralized in the front-end node. Only a minimal additional latency penalty is paid for much improved scalability. Furthermore, the dispatcher module is so fast that centralizing it on a single 300MHz PIII machine scales to throughput rates of up to 50,000 conns/sec.

We have implemented this new architecture, and we demonstrate its scalability by comparing it to a system that performs content-aware distribution in the front-end, both under synthetic and trace-driven workloads.

9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Liviu Iftode, for their helpful comments. Thanks to Erich Nahum at IBM T.J. Watson for providing us with the www.ibm.com trace. This work was supported in part by NSF Grant CCR-9803673, by Texas TATP Grant 003604, by an IBM Partnership Award, and by equipment donations from Compaq Western Research Lab and HP Labs.

References

[1] Alteon WebSystems. ACEdirector. http://www.alteonwebsystems.com.

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

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

[4] 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.

[5] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.

[6] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-based Web Servers. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[7] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 USENIX Technical Conference*, Berkeley, CA, Jan. 1997.

[8] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 2(1), May 1999.

[9] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. In *Proceedings of the 6th International Conference on Network Protocols*, Austin, TX, Oct. 1998.

[10] 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*, San Diego, CA, Jan. 1996.

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

[12] A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicing for URL-Aware Redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

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

[14] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable Highly Available Web Server. In *Proceedings of the IEEE International Computer Conference*, San Jose, CA, Feb. 1996.

[15] K. Fall and J. Pasquale. Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, Jan. 1993.

[16] FORE Systems, Inc. ESX-2400/4800. http://www.fore.com.

[17] Foundry Networks. ServerIron. http://www.foundrynet.com.

[18] 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 Systems Principles*, San Malo, France, Oct. 1997.

[19] 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.

[20] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, Apr. 1998.

[21] IBM Corporation. IBM interactive network dispatcher. http://www.ibm.com/software/-network/dispatcher.

[22] 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.

[23] 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.

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

[25] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[26] 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 Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

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

[28] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Topic in Operating Systems (HotOS-V)*, Orcas Island, WA, 1995.

[29] C.-S. Yang and M.-Y. Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

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

[31] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.