Distributed Cooperative Apache Web Server [*]

Quanzhong Li
Department of Computer Science
University of Arizona
Tucson, AZ 85721
lqz@cs.arizona.edu

Bongki Moon
Department of Computer Science
University of Arizona
Tucson, AZ 85721
bkmoon@cs.arizona.edu

ABSTRACT

Given explosive data traffic in the world-wide web (WWW), it is crucial to achieve the scalable performance of web servers. The overall performance and resource utilization can be improved by spreading document requests among a group of web servers. This leads to the design and implementation of Distributed Cooperative Apache ($\mathcal{DC}$-**Apache**) web server. In this paper, we describe the unique features of the $\mathcal{DC}$-**Apache** system (1) to migrate and replicate documents among cooperating servers, (2) using dynamic hyperlink generation to distribute requests for documents to balance the load, and (3) to maintain replicated copies in a consistent state. We also address the issue of storage management for more effective document replication under limited capacity. In the experiments, the $\mathcal{DC}$-**Apache** system demonstrated its ability to achieve high performance and scalability by effectively distributing load among a group of cooperating Apache servers and by eliminating hot spots and performance bottleneck with replicated documents. The $\mathcal{DC}$-**Apache** system is an effective and practical solution to provide high performance and scalability to cope with ever increasing demands from clients all over the web.

keywords: WWW, Scalable Web server, Apache, DC-Apache, Distributed Web server, Replication, Load balancing

1. INTRODUCTION

With the explosion of data traffic on the World Wide Web (WWW), web servers are often experiencing overload from an increasing number of users accessing the servers at the same time. To address the performance and scalability problems of web servers, we propose a *Distributed Cooperative Apache* ($\mathcal{DC}$-**Apache**) web server solution. The $\mathcal{DC}$-**Apache** system can dynamically manipulate the hyperlinks embedded in web documents in order to distribute access requests among multiple cooperating web servers.

A collection of web documents can be viewed as a directed document graph, where each document is a node and each hyperlink is a directed link from one node to another. The $\mathcal{DC}$-**Apache** solution takes this *graph-based* approach and is built upon the hypothesis that most web sites only have a few *well-known entry points* from which users start navigating through the documents on these sites. Empirical studies performed on the current prototype system indicate that the $\mathcal{DC}$-**Apache** system has a high potential for achieving linear scalability by effectively removing potential bottlenecks caused by centralized resources[1].

The $\mathcal{DC}$-**Apache** solution poses the following benefits over traditional systems based on packet-level manipulation, domain name services (DNS) and distributed file systems:

- Network or packet level manipulation is not necessary. There is no entity (such as a router) that needs to touch every packet that is transferred between client and server. This avoids the possible bottleneck.
- The $\mathcal{DC}$-**Apache** system makes use of the connectivity of hyperlinks to directly control load balancing at the finer grained level of documents than using custom DNS servers. There is also no request redirection that requires client to make two connections for one request.
- By data replication, the $\mathcal{DC}$-**Apache** can effectively solve the hot spots problem caused by exceedingly popular web pages.
- Adding a new server is easy, flexible, and cost effective. Any available machine may be added as a cooperating server.

Apache [10] web server has been chosen as a software platform, on top of which the $\mathcal{DC}$-**Apache** system is designed and implemented. Apache is a high performance web server with a fully featured functionality. A recent web server site survey by Netcraft reports that over 60 percent of the web sites on the Internet are using Apache web server [17].

The experimental results of $\mathcal{DC}$-**Apache** server show that the system can effectively distribute load among multiple servers. It can also eliminate bottleneck by replicating hot spots while keeping the consistency of replicated documents.

[*]This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037) and Research Infrastructure programs EIA-9500991 and EIA-0080123. The authors assume all responsibility for the contents of the paper.

[1]The software distribution of the $\mathcal{DC}$-**Apache** prototype system (release 1.0) is available at http://www.cs.arizona.edu/dc-apache.

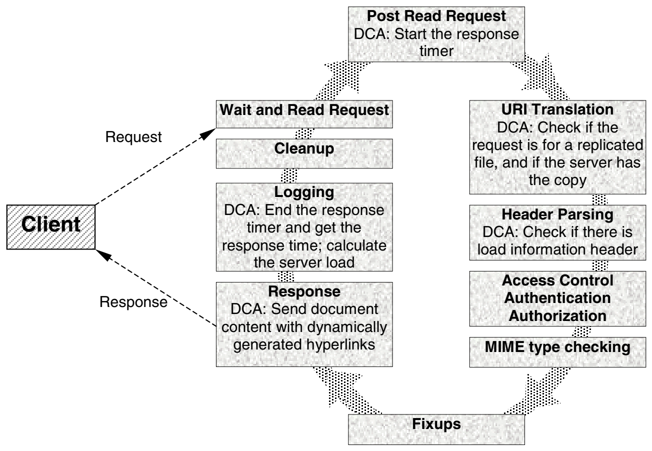

Post Read Request
DCA: Start the response timer
Wait and Read Request
Cleanup
Logging
DCA: End the response timer and get the response time; calculate the server load
Response
DCA: Send document content with dynamically generated hyperlinks
URI Translation
DCA: Check if the request is for a replicated file, and if the server has the copy
Header Parsing
DCA: Check if there is load information header
Access Control
Authentication
Authorization
MIME type checking
Fixups
Client
Request
Response

Figure 1: Apache request processing cycle and $\mathcal{DC}$-Apache module functionalities

Moreover, it is easy and flexible to build a scalable system with only a small amount of configuration work to the existing Apache server. We claim that $\mathcal{DC}$-**Apache** system can be put into practical use to build a scalable web server system.

In this paper, we introduce the design and implementation of $\mathcal{DC}$-**Apache** as well as several techniques we used. The remainder of this paper is organized as follows. Section 2 describes the overview of the $\mathcal{DC}$-**Apache** system, including the Apache module and how $\mathcal{DC}$-**Apache** module is incorporated in the Apache server. The structure of $\mathcal{DC}$-**Apache** system is also described in this section. In Section 3, we describe the mechanisms of load balancing and consistency of $\mathcal{DC}$-**Apache**. Also in this section, we discuss the issue of storage management on cooperating servers under limited capacity. A few implementation details are discussed in Section 4. Section 5 provides experimental results. We overview related work briefly in Section 6. In Section 7, we present concluding remarks and suggestions for future work.

2. OVERVIEW OF THE $\mathcal{DC}$-Apache SYSTEM

Apache allows to extend its functionality by linking new modules directly to its binary code. Using the API and module interface provided by Apache, the $\mathcal{DC}$-**Apache** module is incorporated into the Apache server and its request processing cycle. We call the Apache web server augmented with the new module a $\mathcal{DC}$-**Apache** system.

2.1 Apache Request Processing Cycle

The Apache request processing is divided into several phases and each of server processes repeatedly handles requests through these phases in a cycle. The diagram of the processing cycle and phases is shown in Figure 1.

During the start-up of an Apache server, its main process reads configuration information, initializes configured modules and forks several child processes. Each of these child processes then enters a request processing cycle to process incoming requests in several phases. Each phase corresponds to a processing handler. An Apache module can have its own handlers and register them to the Apache server. When a request arrives, the Apache server will call each handler to process the incoming request. Figure 1 describes the functionalities added by the $\mathcal{DC}$-**Apache** module.

2.2 The $\mathcal{DC}$-Apache Module

Figure 2 shows the functional structure of the $\mathcal{DC}$-**Apache** system. The main process of the Apache server dispatches requests to several child processes. Each child process services the request in the request processing cycle in several phases as described before. Using the handler mechanism, we implemented the $\mathcal{DC}$-**Apache** system as an Apache module that processes requests in the request processing cycle. The function of *pinger* process in Figure 2 is to compute and collect load information about participating servers. It also carries out the task of migrating and replicating documents. The shared memory contains the document graph and statistics information.

In the request processing cycle, the $\mathcal{DC}$-**Apache** module mainly provides the *URI translation handler* and the *content handler* (corresponding to **URI translation** phase and **response** phase) to deal with requests for documents that may be hosted by co-op servers, as described in Figure 1. In this paper, we call the server that has the original copy of a document *the home server* of the document, and call the cooperating server a *co-op server*. The purpose of URI translation handler of the $\mathcal{DC}$-**Apache** module is to intercept the requested URI and to check if this URI is for a replicated document. If it is the case, then check if the server has the document and if the document needs to be re-fetched for consistency reasons. The server will get the document from the home server of this document if it needs to. Then the request will be processed as a normal request. Another important handler is the content handler, which produces the content of the response. The $\mathcal{DC}$-**Apache** system will dynamically generate the hyperlinks in a document according to the current status of document migration and replication. This will be described in later sections.

3. REPLICATION AND LOAD BALANCING

Load balancing among multiple web servers can be achieved by migrating a document from a server to another and/or replicating a document across more than a web server. Figure 3 illustrates the difference between migration and replication. In the figure, if document D is migrated from server#1 to server#2, the hyperlinks will be modified so that documents B and E point to document D on server#2. On the other hand, if document D is replicated on server#1 and server#2, both the servers can provide service for document D. The hyperlink to document D in document B (or E) can point to *either of the two copies*, which will be determined dynamically.

There are a few advantages for a migration-only approach [2]. First, it is relatively easy to maintain the consistency of documents. Second, whenever a document is requested, the request will be handled by a server on which the single copy of the document currently resides, without a potentially complicated process for choosing a server from several servers that host a replicated copy of the document. At the presence of a few extremely popular web documents (*i.e.*, hot spots), however, it becomes very difficult to achieve even distribution of load particularly when the number of cooperating web servers grows beyond a few.

The $\mathcal{DC}$-**Apache** system is based on document *replication for load balancing*. This section describes the framework for

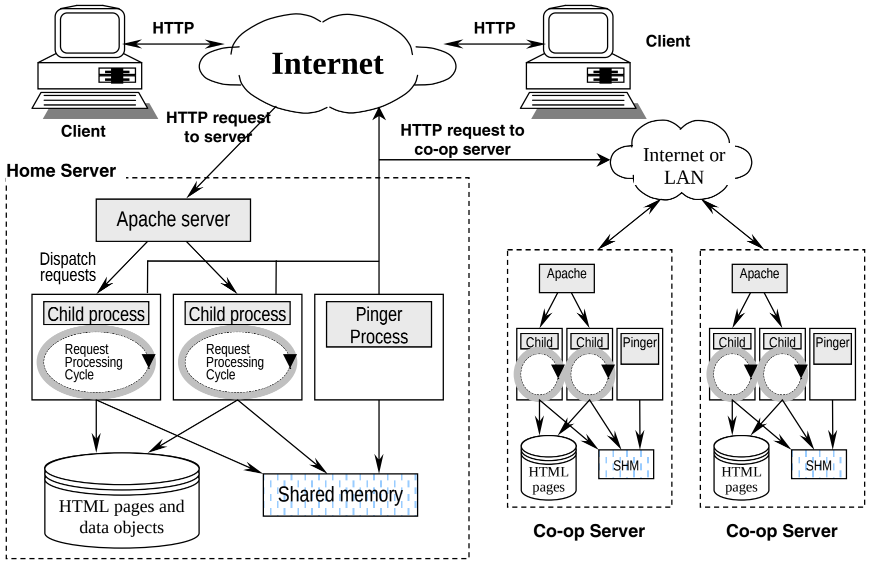


Figure 2: Functional Diagram of the $\mathcal{DC}$-Apache System.

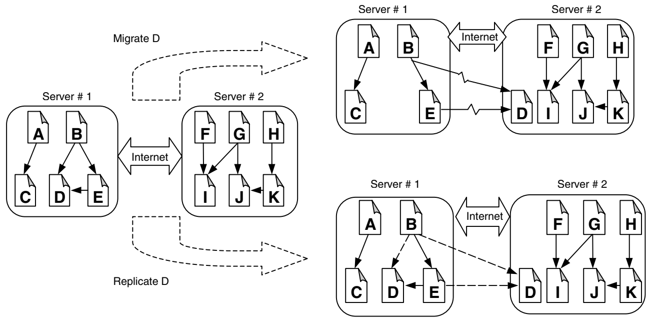


Figure 3: Migration and Replication

load balancing, document replication under limited storage, and consistency issues.

3.1 Framework for Load Balancing

3.1.1 Collect Load Information

To balance work load among cooperating servers, each server needs to know the current load situation of the other servers. This information is global in nature, but each server maintains its own local view of the global state. The best-effort global load information is stored on each server.

Since the network is already presumably filled with many user requests and responses, it is desired not to initiate any additional data transfers simply for the purpose of communicating load information. Such a solution would be wasteful in a system where network bandwidth is an important resource. Instead, the chosen solution is to *piggyback* the load information onto existing HTTP transfers. When a co-op server requests a copy of a document from a home server of the document, its load information will be piggybacked inside the HTTP packet, without incurring additional traffic between servers. The idea of piggybacking information has been used for cache coherency by server invalidation [13].

If load information is not being communicated frequently enough on rare occasions, then it is possible to insert an artificial transfer to communicate load information. This would incur additional overhead since the transfer would not have occurred normally. The $\mathcal{DC}$**-Apache** system uses the *pinger* process to watch for out-of-date information and automatically generate artificial transfers to bring the information up to date. It issues `HEAD` requests to other co-op servers periodically. Then, the co-op servers return their load information in the response HTTP header.

3.1.2 Dynamic Hyperlink Generation

When a document is replicated to any of co-op servers, all the subsequent requests for the replicated document need to be directed to one of the servers that host the document. Thus all the replicated copies should be kept track of and all hyperlinks pointing to the document should be updated dynamically between original and replicated copies in a way load is balanced among servers.

3.1.2.1 Client-Select.

One possible way to realize dynamic hyperlink generation is to embed alternatives for a hyperlink in a document itself and let clients make a selection. All or part of replicated copies can be listed as alternatives in the `href` tag. This method, however, increases the size of a document and requires clients (*i.e.*, browsers) to be able to handle additional attributes in `href` tags. More importantly, servers will have no control over the selection of hyperlinks, and clients will not be able to make informed decisions without knowing the load situation of servers. This makes the client-select an unrealistic approach and leads us to the following approach.

3.1.2.2 Server-Select.

The solution of our choice is to let home servers dynamically generate hyperlinks when serving each request. For example, if the requested document contains a hyperlink that can be chosen from several alternatives, the home server of the document makes a selection based on global load information and reconstructs the document by updating the hyperlink accordingly.

To speed up the processing, we do not parse documents on each request. Instead, documents are parsed once at the system start-up time. Then, the information about hyperlinks (*i.e.*, positions and lengths) is stored in the document graph. When serving a document upon its request, if the document contains a hyperlink pointing to a replicated document, the $\mathcal{DC}$-**Apache** system replaces the hyperlink in the document with the one generated at the service time, based on the global load information. This approach allows servers full control over load balancing by dynamically selecting a hyperlink to a replicated copy hosted by a lightly loaded server.

In order to balance load efficiently with minimal number of documents to replicate, the $\mathcal{DC}$-**Apache** system maintains a list of candidate documents for replication. This list contains frequently requested documents. Only documents in this list are considered for replication. The issue of selecting a server to host those replicated copies will be discussed in Section 3.2.

It should be noted that the physical copy operation to replicate a document is delayed until the replicated copy is actually requested by a client. When a document is chosen for replication, the decision is merely recorded in the document graph. This policy, which we call *lazy replication*, has been chosen to avoid replicating a document that used to be frequently requested but would never be requested due to the change in document request patterns.

3.1.3 Relative Hyperlink Resolution

If a document containing a relative hyperlink (*e.g.*, `href = "foo.html"`) is replicated to a co-op server, this relative hyperlink will be resolved incorrectly if a client gets the copy from the co-op server. Although the actual document pointed to by the relative hyperlink resides on the home server, the client may resolve this relative hyperlink as if it points to the co-op server. We may avoid this problem by changing all relative hyperlinks to absolute hyperlinks. However, the expansion of relative hyperlinks can make the document larger if there are many relative hyperlinks, thereby consuming more disk space and network bandwidth. In the $\mathcal{DC}$-**Apache** module, we solve this problem by using the `BASE` element of HTML. If a document does not have a `BASE` element, we will dynamically add one when sending out the document. The value of this `BASE` element is a hyperlink pointing to the home server. It indicates the original location of this document. With this information, a client can correctly resolve relative hyperlinks.

3.2 Replication and Replacement under Limited Storage

In the $\mathcal{DC}$-**Apache** system, the amount of storage set aside for replicated copies from other servers can be specified in the configuration. The $\mathcal{DC}$-**Apache** system will automatically check the disk usage and guarantee the aggregate size of replicated documents is no more than the assigned quota.

Due to the limited disk space, a co-op server may delete a replicated document from its local store and free space for newly replicated documents. If a deleted copy is requested later, the document needs to be fetched from its home server again. Deleting and re-fetching a document frequently can cause the system to thrash. In the rest of this section, we examine load balancing methods and present better ways to deal with such a situation.

3.2.1 Naive Initial Approach

Under this approach, when a home server is overloaded, the $\mathcal{DC}$-**Apache** system seeks to replicate documents from the home server to other servers. A co-op server with the least amount of load will be selected to host the replicated documents. If the aggregate size of replicated documents grows beyond the alloted disk quota on a co-op server, the co-op server will delete some of the replicated copies. This can cause the system to thrash with recurring requests for the deleted copies as explained above. It can also impose the home server with additional load to fetch deleted documents again and again. To make it worse, as the home server is already overloaded, more documents will be replicated from the home server to other servers, which will make the system more vulnerable to thrash. The following sections propose ways to break this vicious cycle.

3.2.2 Deletion-Aware Method

To prevent deleted copies from being requested frequently, a co-op server informs a home server when a replicated copy from the home server is deleted from its local store. On such a notice, the home server revokes the replicated copy from the co-op server by taking the replicated copy off the list of alternative hyperlinks. For implementation details, such deletion notices are sent to a home server in `UDP` packets. The *pinger* process of the home server is responsible for receiving and processing those messages.

If an overloaded co-op server has a relatively small disk space for replicated copies, the co-op server may need to evict and replace replicated copies frequently. Consequently, a home server has to deal with many deletion notices.

3.2.3 Resource-Aware Method

Recall all the decisions about document replication are made solely by a home server. Given the information as to how many documents have been replicated to which co-op servers is already available on a home server, each home server can be better poised with additional information about the disk quota of other servers. That is, when a document is being replicated, its home server can choose a co-op server which a replicated copy will be sent to, considering the amount of disk space available on the co-op server.

A home server maintains a list of replicated documents and the aggregate size of replicated documents for each co-op server. During load balancing process, a home server will replicate a document to or revoke one from a co-op server according to the following guidelines.

- **[Replication]** If a co-op server is lightly loaded and disk space is available for a document to be replicated, send the replicated copy to this co-op server.
- **[Revocation]** If a co-op server is heavily loaded, then revoke documents from this co-op server. Change the list of replicated documents and the aggregate size of replicated documents accordingly.
- **[Replacement Policy]** If a co-op server is lightly loaded, but there is no available space to host the replicated copy of a document, replace unpopular copies

with popular documents in an attempt to balance load by shifting more load to this lightly loaded co-op server without exceeding the disk quota.

Replicating, revoking and replacing a document are all *lazy* in the sense that the physical effects of these operations will be materialized only when the document copy on a particular server is actually requested by a client. Upon requests for a replicated copy, a co-op server will obtain the copy from its home server. When the copy is revoked from the home server, the co-op server is not notified. Instead, the revoked copy is removed from the list of replicated documents on the home server. It will become less popular on its current co-op server, and will be eventually evicted by the document replacement policy described above.

3.3 Consistency of Replicated Documents

The replicated documents on co-op servers and the original document on the home server must be in a consistent state in order to give users a consistent view of the whole web site. Two approaches can be used to maintain the consistency. One is the *home server invalidation approach*. The other is *co-op server validation approach*.

3.3.1 Home Server Invalidation

As described above, the home server has the replication information of documents. When a document is modified, the home server can use the following three methods to keep the consistency.

Active Update The home server sends the new copy to co-op servers that have an old copy. Subsequent client requests do not need to wait for the new copy to be fetched back. This method can maintain a high consistency. However, it may cause heavy load on the home server and consume too much network bandwidth in a short time.

Explicit Notification The home server sends notifications to co-op servers to invalidate old copies. A co-op server then deletes old copies upon receiving the notification. In the future, when there is a request for a new copy, the co-op server will fetch the new copy from the home server.

Implicit Notification In this method, there is no notification to co-op servers. The home server puts the version number or timestamp in the generated hyperlinks. For example, a possible hyperlink looks like `http://coop1/~migrate/timestamp/home/foo.html`. Upon receiving such a request and the timestamp in the hyperlink is greater than the last modification time of its local copy, the co-op server knows that it needs to get a new copy from the home server.

3.3.2 Co-op Server Validation

While the home server invalidation approach puts all consistency maintenance work on the home server, the co-op server validation approach can distribute the burden of consistency maintenance among co-op servers. In the $\mathcal{DC}$-**Apache** module, we use this approach based on time-out.

There is a time interval used to check whether a co-op server needs to update a replicated document. When there is a request for a replicated document and the difference between current time and the last check time of this copy is greater than this interval, the co-op server will use a conditional `GET` request of HTTP to see if the document should be retrieved again. Thus, only when there is a request for a replicated copy, will the co-op server try to update the copy. We call this *lazy update*. This means only client requests for a document can trigger its update. This is especially important for large documents (such as in Sequoia data set), since we do not want to consume too much bandwidth for the consistency maintenance.

4. IMPLEMENTATION DETAILS

The $\mathcal{DC}$-**Apache** system has been implemented as an Apache module on Unix platforms. In this section, we discuss a few implementation details of the $\mathcal{DC}$-**Apache** module.

4.1 Multi-Threaded vs. Pool-of-Processes

Most web servers are based on either a *multi-threaded* model (*e.g.*, `phttpd` [8]) or a *pool-of-processes* model (*e.g.*, Apache server [9]) to deal with concurrent requests from multiple clients. Although the $\mathcal{DC}$-**Apache** system can be implemented atop either of the models, the current prototype is based on the pool-of-processes model, because a multi-threaded version of Apache was not available at the time of implementing the $\mathcal{DC}$-**Apache** system.

The multi-threaded model is considered more efficient than the pool-of-processes model in processing concurrent requests, because it avoids the cost of creating and destroying processes and the overhead of switching contexts of processes. It is also easier to share information between threads than processes. The next release of the $\mathcal{DC}$-**Apache** system will support the multi-threaded model[2].

4.2 Management of Shared Information

A document graph needs to be built on each server. Since the $\mathcal{DC}$-**Apache** system is based on the pool-of-processes model, the document graph and other related information (such as statistics of co-op servers and replicated documents) should be shared by all Apache processes on a server. The information is stored in a shared memory and semaphore is used to coordinate accesses to the shared memory. The structure of shared memory is shown in Figure 4. The shared memory can grow incrementally by adding a block at a time to accommodate a varying number of documents.

Each document record contains data such as a document name, a list of outgoing hyperlinks, a `BASE` element position, servers hosting the document, and so on. A block table is used to index the blocks in the shared memory and to indicate the status of the blocks (*e.g.*, initialized or allocated). Each process maintains a separate table in its private memory. Each entry of this table indicates whether a block is associated with the process and it stores the address of the block in the private address space of the process. This information is used to translate a logical address (*i.e.*, a pair of *block* and *record*) into an address in the private memory space of a process.

When a document is requested, the $\mathcal{DC}$-**Apache** module

[2]On Unix platforms with POSIX threads support, Apache 2.0 can now run on a hybrid of multi-threaded and pool-of-processes models.

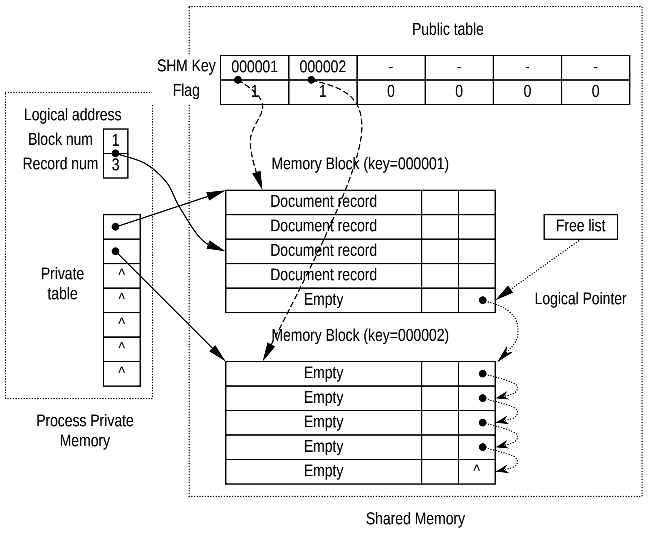


Figure 4: Shared memory structure.

uses the name of the requested document to locate the document record in the shared memory. A hash table is used to translate a document name into the logical address of its document record. Collisions in the hash table can be resolved by *linear hashing* [15], which is a dynamic open hashing scheme.

5. EXPERIMENTS

We benchmarked the $\mathcal{DC}$**-Apache** system on a cluster of 64 Intel Pentium workstations. Each workstation has a processor of 200 MHz clock rate and 128 MB memory, and runs Linux kernel version 2.2.14. They are connected by a Catalyst 5500 switch, which provides a 100 Mbps switched Ethernet connection. The switch can handle an aggregate bandwidth of 2.4 Gbps in an all-to-all type communication. Apache server version 1.3.6 is used as a software platform which the $\mathcal{DC}$**-Apache** module is incorporated into. In most experiments, the number of servers was varied from one to 16 workstations, and the rest of the workstations are used to run client processes.

5.1 Data Sets and Performance Metrics

We have chosen three data sets (Sequoia, MAPUG, LOD) from real-world applications and a synthetic data set generated from the SPECweb99 benchmark program [6]. These data sets are described in the following.

Sequoia Benchmark data: It contains 130 AVHRR image files from NOAA satellite [19]. The images are compressed and in the 1-2.8 Mbytes range. We created an HTML front-end page that includes a hyperlink to each image file.

MAPUG Mailing List Archive: The mailing list archive contains 1,534 documents, 28,998 links and 5,918 Kbytes aggregate size of all files. The data set is mostly text, each with 4-6 bit-mapped images, which have a high request rate.

LOD Role-Playing Adventure Guide: This data set is a graphical database for a popular computer role playing game, with 349 documents and 1433 links (240 of the 349 documents are images). The aggregate size of the documents is 750 Kbytes.

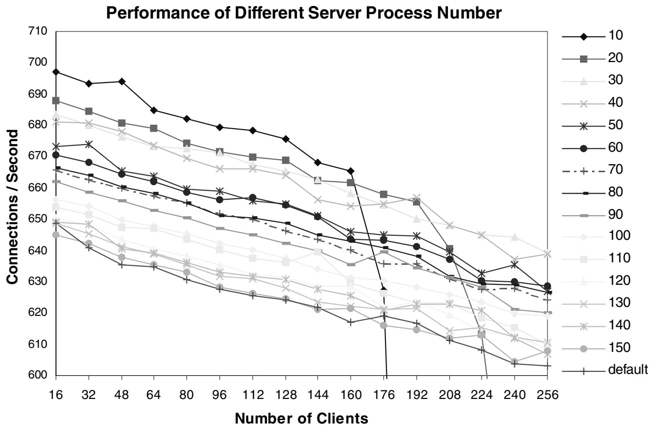


Figure 5: Apache configuration test result

SpecWeb: This data set is generated by SPECweb99 [6]. The total size is 1027897 bytes. There is a single index HTML file to index all data files. The hyperlinks in the index file pointing to data files can appear more than once according to the workload distribution of SPECweb99.

We have used three major performance metrics, namely *connections per second (CPS)*, *bytes transferred per second (BPS)* and *round-trip time (RTT)*. The round-trip time (RTT) is the interval between the moment when a client sends out a request and the moment when it receives the response. This includes the network overhead (connection and transfer time) and the server processing time. All the three metrics are measured at each client side.

Real-world web transactions are fairly small [1]. So we use CPS as a primary metric for the purpose of load balancing for making decisions regarding document replication. The *pinger* process computes a numeric value to indicate the work load of each server, using the following formula.

$$L_{new} = L_{old} \times \alpha + S_{current} \times (1 - \alpha)$$

where L_{new} and L_{old} are new and old measurements of load, $S_{current}$ is the current load sample, and α is a tunable parameter that reflects the relative weights between current and history load measurements. In our experiment, the value of α is set to 0.5.

5.2 Settings for a Stand-alone Apache Server

There are several run-time directives to configure the Apache server. Using the directives, an Apache server can be set up to start with a certain number of child processes, and the number remains unchanged during the experiments. The number of processes particularly has a substantial impact on the performance of a stand-alone Apache server. We carried out experiments with the LOD data set to evaluate the performance impact of these parameters.

Figure 5 shows a general trend that the performance becomes worse as the number of clients (and load) increases. More importantly, we made two interesting observations.

First, when work load is light (with a small number of clients), an Apache server with fewer processes outperforms those with more processes. Second, when work load is heavy, the performance of an Apache server with a few processes degenerate rapidly as the number of clients increases.

One plausible explanation for the first observation is that the overhead of context switch and process scheduling increases, as the number of server processes increases. For the second observation, we conjecture that a large number of TCP connection requests are queued in the system kernel, because those connection requests are not processed quickly enough by only a few processes. This large queue slows down the packet processing speed in kernel due to the increased cost of maintaining a large number of TCP states.

Using default configuration of the Apache software distribution, we observed that with 16 clients sending requests continuously, the number of server processes will soon reach 150 and remains at that number. This indicates that the limit of the processing capability of a stand-alone Apache server is reached, and the performance of the server may be improved by altering the configuration of the server. Based on the this observation, we used a fixed number of server processes throughout the experiments. Thus, the overhead of process creating and termination did not have much influence on the performance of the $\mathcal{DC}$-**Apache** system.

5.3 Evaluation of $\mathcal{DC}$-Apache System

In the experiments, the number of clients was varied from 16 to 544 with an increment of 16 or 32. The performance measurements were first logged at client side, and then collected to compute the averages.

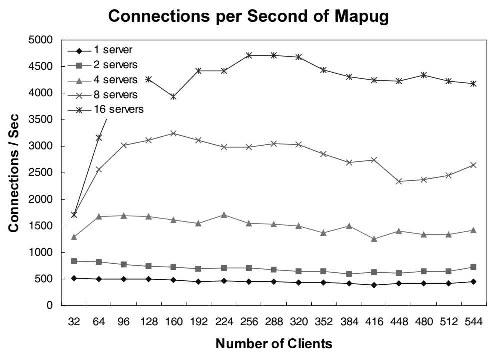
5.3.1 Scalability and Hot Spots

In Mapug data set, there are only a few images, but those images are pointed to by most of the documents. This makes the images hot spots. Under the migration only approach, it is hard to distribute these few hot spots among servers in such a way the load is evenly distributed. Those hot spots become the major limiting factor on the scalability of the entire system. In the $\mathcal{DC}$-**Apache** system, this problem is resolved by using the replication approach. Those popular images can be replicated to multiple co-op servers and the requests for them can be served by those servers at the same time, so that the scalability is not limited by hot spots.

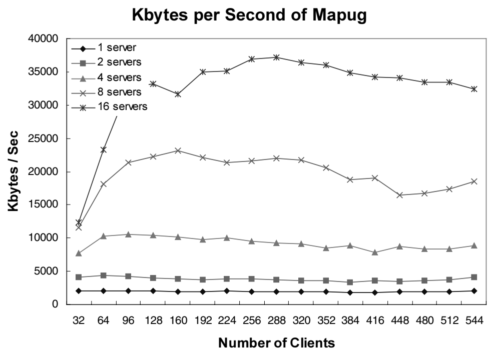
The experiment results from the Mapug data set are presented in Figure 6(a) and Figure 6(b). These two figures are for connections per second (CPS) and bytes transferred per second (BPS), respectively. This experiment examines the scalability of the $\mathcal{DC}$-**Apache** system for a different number of servers with a varying number of clients (from 32 to 544). In both figures, the performance improves in a scalable manner with an increasing number of servers, even at the presence of a few hot spots in the data set.

Figure 7(a) and Figure 7(b) show the performance measured in CPS and BPS respectively for all of the four data sets. In these figures, the number of servers was varied from 1 to 16, and the number of clients was 256. Those figures show that, for all data sets we used, the $\mathcal{DC}$-**Apache** system yielded near-linearly scalable performance.

The average response times measured at client side are shown in Figures 8(a) and 8(b) for SpecWeb and Sequoia data sets, respectively. As the number of servers increases, the response time decreases reversely proportionally. This

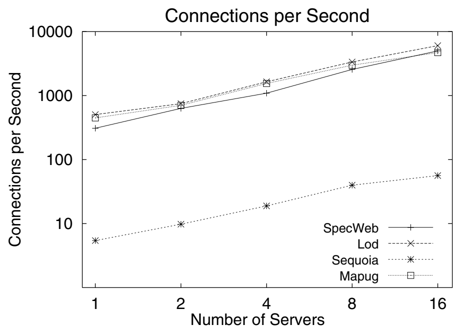


(a) Connections per second

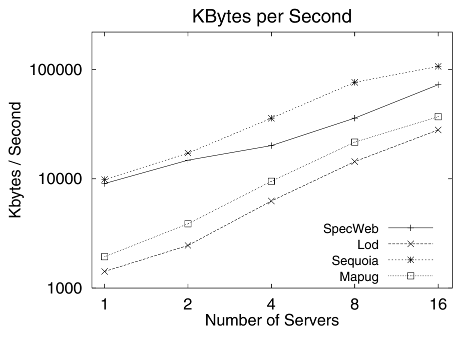


(b) KBytes per second

Figure 6: CPS and BPS from Mapug data set

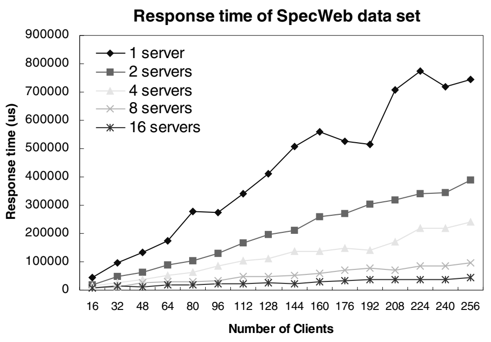


(a) Connections per second

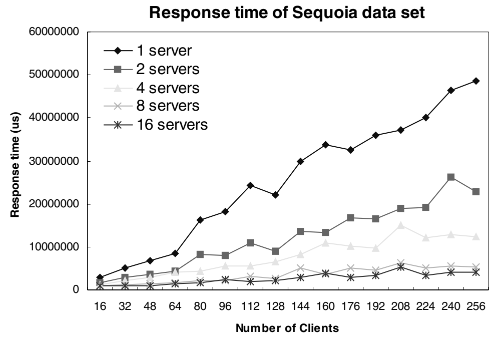


(b) KBytes per second

Figure 7: CPS and BPS from all data sets

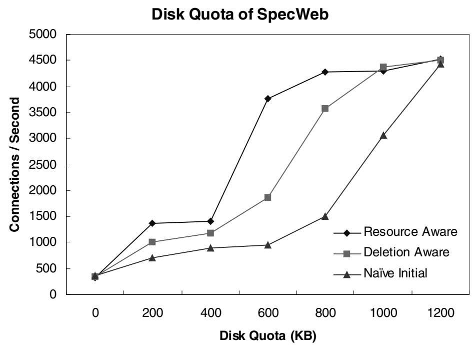


(a) Response time of SpecWeb

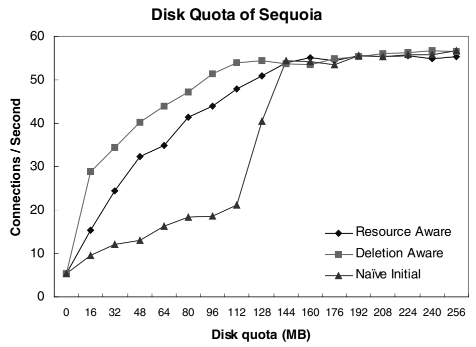


(b) Response time of Sequoia

Figure 8: Client response time



(a) Disk quota of SpecWeb



(b) Disk quota of Sequoia

Figure 9: Comparison of Three Methods for Replication under Limited Storage

is another evidence that the $\mathcal{DC}$**-Apache** system has a high potential to achieve near-linearly scalable performance.

5.3.2 *Replication under Limited Storage*

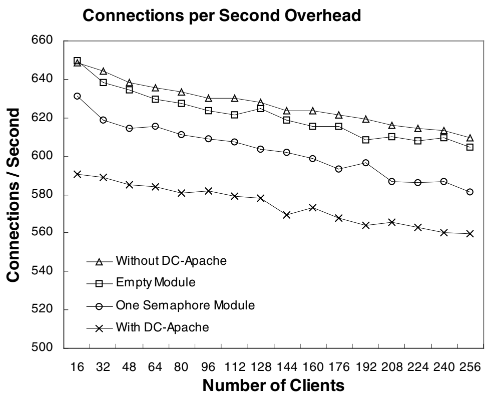
With SpecWeb and Sequoia data sets, we carried out experiments to show the effects of the limit on disk space of co-op servers. Three proposed methods were examined to deal with the problem of limited storage for document replication. One server was designated as a home server of all documents, while the other servers were used as co-op servers. The disk quota reserved for replicated documents on a co-op server was varied from zero byte to 1.2 MBytes for SpecWeb data set, and from zero to 256 MBytes for Sequoia data set.

As shown in Figure 9, for all three proposed methods, if the disk quota is set to zero (*i.e.*, no documents are replicated), the performance of the $\mathcal{DC}$**-Apache** system is reduced to one server case, obviously because no co-op server can share the work load. Again, for an obvious reason, if the disk quota is large enough to replicate an entire set of data, the $\mathcal{DC}$**-Apache** system yielded the best performance regardless of the methods for storage management. For SpecWeb data set, as shown in Figure 9(a), the Resource-Aware method outperformed the other methods over the entire spectrum of disk quotas. Then, it was followed by the Deletion-Aware and Naive methods. This was an expected result, because the Resource-Aware method could do better in replicating documents with knowledge about the disk quota of co-op servers. With the Naive method, the performance drops quickly, even when the disk quota is less than the size of an entire data set only by a small margin.

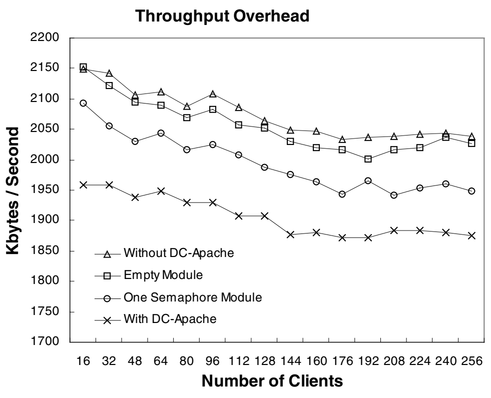
On the other hand, for Sequoia data set, as shown in Figure 9(b), the Deletion-Aware method was better than the Resource-Aware method. By the Resource-Aware method, if a home server decides to stop pointing to a replicated copy, the copy on a co-op server will be requested less and less frequently until it will be eventually deleted by the replacement policy of the co-op server. This implies that the copy and its disk space are not utilized for a certain period of time. Since individual file sizes are large in Sequoia data set, such loss of opportunity could be significant. However, for Deletion-Aware method, a home server does not stop pointing to the copy until the home server is notified from a co-op server that the copy is actually deleted.

5.3.3 *Overhead of* $\mathcal{DC}$**-Apache** *module*

The experiment results of the $\mathcal{DC}$**-Apache** module overhead are shown in Figure 10. This experiment was done using LOD data set with one server and a different number of clients. The server used the default configuration of the Apache server. There were 5 child processes at startup time, and the server could deal with at most 150 concurrent requests. We tested the server in four cases: 1) compiled without the $\mathcal{DC}$**-Apache** module; 2) compiled with an empty module, with only a return statement in each handler; 3) compiled with a module in which the content handler has one semaphore operation and other handlers are empty; 4) compiled with the $\mathcal{DC}$**-Apache** module. If the server is compiled with the $\mathcal{DC}$**-Apache** module, the $\mathcal{DC}$**-Apache** module will use its own content handler to send request response. In other cases, the server uses the default Apache handler to send response. The module interface overhead tested by the empty module case is around 0.9% for both CPS and throughput. Using the $\mathcal{DC}$**-Apache** module added an aver-



(a) CPS Overhead of LOD



(b) Throughput Overhead of LOD

Figure 10: Overhead of $\mathcal{DC}$-Apache Module

age overhead of 8.36% for CPS and 8.08% for throughput. The $\mathcal{DC}$**-Apache** module uses semaphore to coordinate the accesses of shared memory, so we tested one semaphore module case to see the effect of semaphore operation on performance. We found that the overhead of this one semaphore module was about 3.8%. If we took out the overhead of module interface and semaphore operations, the rest of the overhead will be at most 5%, which is used to send the document content and to do logging work. When sending the content, the $\mathcal{DC}$**-Apache** module checks every hyperlink of the document to see if a new hyperlink needs to be generated. Because we preprocessed each document and stored hyperlink information in memory, this dynamic processing of hyperlinks inside a document did not impose a significant overhead on the performance.

6. BACKGROUND AND PREVIOUS WORK

Because of the phenomenal traffic growth of WWW, much research work has been focused on the performance of web servers.

Various DNS based scheduling techniques have been proposed in the literature. The NCSA scalable web server [12, 14] is based on round-robin DNS to dispatch requests to web servers. In order to cope with heterogeneous web servers and the increasing complexity of the DNS scheduling, Adaptive TTL algorithms [4] have been proposed to adaptively set the TTL value for each address mapping request. It takes into account both the uneven distribution of client request rates and heterogeneity of web servers. The problem with this DNS based method is that clients may cache resolved IP addresses, which can cause load imbalance among servers. And when a server is no longer in service, clients may still try to use the cached IP address to visit this server.

The LocalDirector of Cisco [20] is an example of architectures dispatching requests in TCP/IP level. The LocalDirector appears as one "virtual" server to clients. All traffic is directed to a virtual IP address (virtual server) via DNS. Those requests are then distributed over a series of real IP address servers (real servers). The dispatcher rewrites the header of incoming IP packages and forwards the packages to the selected server. The dispatcher also needs to keep the TCP connection information and always forwards the IP packages of the same request to the same server. Because all client requests are routed through the dispatcher, this may cause network resource contention. The dispatcher itself can be a bottleneck that limits the scalability and the physical distribution of servers.

HTTP redirection based methods do the load balancing in HTTP level. There can be one or more redirection servers. The initial request is first dispatched to a redirection server, using the techniques we mentioned above, such as DNS rotation. The redirection server then dispatches the request using the HTTP redirection mechanism. A redirection-based server architecture has been proposed in [16], which falls in this category. The problem with this approach is that the response time is longer, because the client needs to initiate two TCP connections for one request.

A system model for dynamic replication and migration of Internet objects is proposed in [18]. In the model, a request is first directed to the "closest" distributor of a gateway, then this distributor forwards the request to a redirector, which forwards the request to a server hosting the requested object. The selected server then sends the object to the client via the distributor. In this model, one request needs several steps to travel through the system and the proposed algorithms use information from routing databases and IP headers.

There are other techniques based on the ability of client side. In the Dynamic Server Selection method [7], clients automatically determine the best server for a given file without a priori knowledge of server location or network topology. They assume that a client has been provided with a list of server addresses and the client can dynamically choose a server according to distance measured by hops and round-trip latency.

Several leading companies that support e-businesses like Alteon [21], ArrowPoint [3], and Resonate [11] have their own products to do content level switching. A web switch receives HTTP connections and looks into the request packet to get the knowledge of requested content. Then it forwards the request to a selected server using delayed binding technique. Because of the centralized nature of switching, all client requests need to go through the web switch and the web switch can easily become the bottleneck. The switch will first establish the connection with the client and then with the web server and it will perform NAT (Network Address Translation) for each packet. The response time will be longer than the direct connection with a web server[3]. Our approach proposed in this paper aims at eliminating possible hops and bottlenecks between web servers and clients while achieving the goal of load balancing among servers.

7. CONCLUSION AND FUTURE WORK

We have designed and implemented a prototype system of the Distributed Cooperative Apache ($\mathcal{DC}$-**Apache**) web server. The $\mathcal{DC}$-**Apache** system is based on the idea of dynamic manipulation of the hyperlinks embedded in web documents to distribute client requests among a group of cooperating web servers. The $\mathcal{DC}$-**Apache** system is built atop Apache web servers by augmenting them with new functionalities so that individual web servers can cooperate and share work load as a collective unit. In particular, the $\mathcal{DC}$-**Apache** system supports seamless document replication as a means of load balancing as well as document consistency. Additionally, the $\mathcal{DC}$-**Apache** system provides methods to deal with issues concerning replication under limited storage.

The experimental study shows that the $\mathcal{DC}$-**Apache** system can effectively distribute load among a group of cooperating servers. It can eliminate performance bottleneck effectively by replicating hot spots while keeping the consistency of replicated documents. Moreover, it is easy and flexible to build a scalable system with only a small amount of configuration work to existing Apache servers. We conclude that the $\mathcal{DC}$-**Apache** web server system presented in this paper is an effective and practical solution to provide high performance and scalability to cope with ever increasing demands from clients all over the web.

An issue that is not addressed in this paper is the use of geographic information of web servers and clients to make the most of document replication. For example, if a document is being requested frequently by clients located in a particular region, it would be beneficial to replicate this document to cooperating servers in or near the region. When a document is requested, a server that is geographically closest to a client can be chosen to serve the request. With the evolution of the Internet, new features and services will be added to the current network (*e.g.*, the *anycast* service). Further study about how to utilize them in the design and implementation of the $\mathcal{DC}$-**Apache** system will be necessary.

8. REFERENCES

[1] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, October 1997.

[2] Scott M. Baker and Bongki Moon. Distributed cooperative web servers. *Computer Networks*, 31(11-16):1215–1229, 1999.

[3] CISCO. ArrowPoint communications. *http://www.arrowpoint.com/index.html*, 2000.

[4] Michele Colajanmi and Philip S.Yu. Adaptive TTL schemes for load balancing of distributed web servers. *ACM Sigmetrics Performance Evaluation Review*, 25(2):36–42, 1997.

[5] CISCO-ArrowPoint Communications. ArrowPoint Web Network Services (WebNS). http://www.arrowpoint.com/solutions/white_papers/WebNS.html, 2000.

[6] Standard Performance Evaluation Corporation. SPECweb99 benchmark. http://www.specbench.org/osg/web99, August 2000.

[7] Mark E. Crovella and Robert L. Carter. Dynamic server selection in the internet. In *the Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95)*, August 1995.

[8] Peter Eriksson. Phttpd - a multithreaded web server. http://www.signum.se/phttpd.

[9] Roy T. Fielding and Gail E. Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, July/August 1997.

[10] The Apache Software Foundation. The Apache HTTP server. http://www.apache.org/, 1999.

[11] Resonate Inc. Keeping E-Business Open for Business. http://www.resonate.com, 2000.

[12] Eric Dean Katz, Michelle Butler, and Robert McGrath. A Scalable HTTP Server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[13] B. Krishnamurthy and C. E. Wills. Piggyback server invalidation for proxy cache coherency. *Computer Networks and ISDN Systems*, 30:185–193, April 1998.

[14] Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, November 1995.

[15] Witold Litwin. Linear hashing : A new tool for file and table addressing. In *Proceedings of the 6th VLDB Conference*, pages 212–223, Montreal, Canada, October 1980.

[16] Antoine Mourad and Huiqun Liu. Scalable web server architectures. *IEEE Symposium on Computers and Communications*, 1997.

[17] Netcraft. The netcraft web server survey. http://www.netcraft.com/survey/, 2000.

[18] M. Rabinovich and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. *IEEE Int. Conf. on Distributed Computing Systems*, May 1999.

[19] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The SEQUOIA 2000 storage benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference*, pages 2–11, Washington, DC, May 1993.

[20] Cisco System. Scaling the internet web servers. http://www.cisco.com/warp/public/751/lodir/scale_wp.htm, November 1997. White Paper.

[21] Alteon WebSystems. Alteon web switching. http://www.alteonwebsystems.com, 2000.

[3]Although the NAT peering of ArrowPoint acts as *triangulation protocol* allowing the response to be delivered directly to the user. One direction of flow still needs to go through the switch [5].