Making World Wide Web Caching Servers Cooperate

Radhika Malpani, Jacob Lorch, David Berger*

Abstract

Due to its exponential growth, the World Wide Web is increasingly experiencing several problems, such as hot spots, increased network bandwidth usage, and excessive document retrieval latency. The standard solution to these problems is to use a caching proxy. However, a single caching proxy is a bottleneck; there is a limit to the number of clients that can use the same cache, and thereby the effectiveness of the cache is limited. Also, a caching proxy is a single point of failure. We address these problems by creating a protocol that allows multiple caching proxies to cooperate and share their caches, thus increasing robustness and scalability. This scalability, in turn, gives each client an effectively larger cache with a higher hit rate. This paper describes a prototype implementation of this protocol that uses IP multicast to communicate between the servers. ***Keywords:*** *World Wide Web, WWW, proxy, cache, multicast, HTTP*

Introduction

The World Wide Web is a document distribution system based on a client/server model. Currently, the World Wide Web is experiencing exponential growth. According to [14], for example, in the first ten months of 1994 the amount of WWW traffic on the Internet doubled roughly every 11 weeks. This increasing use of the Web results in increased network bandwidth usage, straining the capacity of the networks on which it runs. It also leads to more and more servers becoming "hotspots," sites where the high frequency of requests makes servicing these requests difficult. This combination of increased network bandwidth usage and overloaded servers eventually results in increased document retrieval latency.

Caching documents throughout the Web helps alleviate the above problems. Due to the exponential growth of the WWW, considerable effort has been spent investigating caching of WWW objects. At first, caching meant that each client maintained its own cache. However, the benefits of caching grow with the number of clients sharing the same cache, so the *caching proxy* was developed and used. Such a proxy services client requests from its cache whenever possible, getting the objects from their home servers if required. Unfortunately, a single caching proxy introduces a new set of problems, namely those of scalability and robustness, since a single server is both a bottleneck and a single point of failure. Scalability to large numbers of clients is important because the more clients sharing a cache, the larger the probability of getting a cache hit. Keeping these considerations in mind, we have designed and implemented a protocol to allow multiple independent caching servers to cooperate and jointly service a set of clients.

The rest of the paper is structured as follows. First, we describe related work, indicating the state of the art in caching on the WWW. Then, we describe the problems with current caching techniques, and our solution to these problems. Next, we describe our prototype implementation and describe and discuss measurements of this implementation. Finally, we discuss future work and state our conclusions.

* This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

Page 1 of 11 **GOOGLE EXHIBIT 1011**

Related Work

As discussed before, the WWW suffers from problems of high latency, network congestion, and server overload. Hence, considerable effort has been spent investigating one solution to this problem: caching WWW objects. The fundamental issues that have been considered include cache topology, cache replacement policy, cache consistency, whether caching is server- or client-initiated, and cachability of different objects. In addition, work has been done on optimizing caching server implementations.

There are two basic approaches to caching that have been explored: client side and server side solutions. In the server side solutions, servers shed load by duplicating their documents at *caching servers* spread throughout the WWW [4, 11]. Client side solutions usually use some sort of caching proxy that fields requests from one or more clients and caches objects on the clients' behalf. Our work only involves client side solutions and is orthogonal to any server side solutions.

Many client side caching servers have been developed recently. The most prominent of these is probably CERN's httpd 3.0 [13]. All clients using it can have the benefits of a shared cache. Also, its caching policies are configurable; in the server configuration file one can indicate, for any string pattern, the caching policy to be applied to URLs fitting this pattern.

Other caching servers similar to CERN's have appeared recently. The Lagoon caching server [7], developed at the Eindhoven University of Technology, is quite similar to CERN's server. Guenther Fischer modified a patch, designed to convert a server into a proxy, into a patch that converts a server into a caching proxy [10]. Also, Gertjan van Oosten wrote a perl script that can be installed in the *cgi-bin* directory of a server to convert it into a caching proxy [15].

Another caching server system, developed concurrently with the work done for this paper, is the Harvest cache [6]. Developed at the University of Colorado and the University of Southern California, the Harvest cache is a proxy designed to operate in concert with other instances of itself. These servers are typically configured as a tree, with each server considering certain other servers to be parents and certain other servers to be siblings. When a server receives a request for data that it does not have cached, it can call upon its siblings and parents in the tree to find if any of them have the data cached. One disadvantage of the Harvest cache approach is that it uses unicast to communicate with these siblings and parents; more efficient communication may be possible using multicasting.

A group at Boston University developed a cache that could be configured to work at either the client, the host, or the LAN level [5]. Clients either have their own caches, share with other clients on the same host, or share a cache with clients on the same LAN. Surprisingly, according to their results, the number of cache hits did not vary from configuration to configuration. However, as one might expect, a LAN level cache made more efficient use of resources than a host level cache, which in turn made more efficient use of resources than a client level cache. Finally, they observed that while they were able to get a significant number of hits from the cache, hit documents tended to be small and thus the number of bytes served out of the cache was lower than one might expect.

System Design

In this section, we discuss certain problems that current caching solutions have, and how we chose to solve them.

Problems Statement

Most of the work in client side solutions has concerned clients sharing a single caching proxy. But, as explained before, this approach has several problems. First, it lacks robustness, since the proxy serving a set of clients becomes a single

point of failure. Second, it is a bottleneck, creating a limit on the number of clients that can share a cache. This, in turn, limits the effective cache size and hit rate that each client obtains.

For instance, here at the University of California at Berkeley, many different research groups use their own separate caching proxies. This is because there is no good way to share caching resources among different groups. Unfortunately, this means that the caching resources available to one group are limited to that group. If each client at U.C. Berkeley were able to access objects cached at *any* caching proxy on campus, they would make better use of the caching resources available. In other words, if all the existing individual caches could be combined to form a *global, distributed cache*, then we could improve the system without increasing the amount of resources used.

Solution

Our goal was to address the above-mentioned problems faced by single caching proxies. To obtain a system that was robust in the face of failures and that scaled well to a large population of clients, we needed to make multiple servers cooperate in such a way that they shared their individual caches to effectively create one large distributed cache. For robustness, all servers needed to be functionally equivalent so that any server could handle the request of any client. For scalability, we needed some means of distributing and balancing the load among the servers. At the same time, we needed to ensure that our protocol for making the servers cooperate did not significantly increase the network bandwidth usage. Therefore, we decided to use multicasting [8] wherever possible to make efficient use of network bandwidth.

Given the above considerations, we developed the following protocol. For each request, a client randomly picks a caching proxy server from a list of cooperating servers and sends its request to it. Let us refer to this proxy as the *master* for this request. If the master has the requested object in its local cache, it returns it to the client. Otherwise, it multicasts a query to the other cooperating servers asking if any of them has the object cached. If it receives no reply within a certain time, it acts as it normally would as a caching proxy, i.e., it contacts the host specified in the URL, requests the object, passes it on to the client, and caches it for future use. If any of the other proxies has the object cached, this caching proxy informs the master, so the master can *redirect* the client to this caching proxy. The client then makes a new request for the object to this caching proxy, and obtains the object from it. Note that in our protocol all servers are functionally equivalent in that any server can act as a master for any request. This protocol is presented graphically in Figure 1.

Most of the overhead of our technique comes from having the client make two requests in the case that it chooses the "wrong" proxy first: one to the master and another to the proxy to which it is redirected. The way we reduce this overhead is to have clients use the same proxy for retrieving inline images of a document that they use for retrieving the document to which these images belong. In this way, documents and their images tend to get cached at the same place. Furthermore, once the document is cached, a client requesting it will get redirected at most once during its requests for the document and its inline images.

Alternative Approaches

There are several other approaches and variants to our protocol that we considered but ultimately rejected. Here we discuss these variants and the reasons we rejected them.

In our protocol, the client makes a request to a single server and that server multicasts the request to other servers. An alternative approach is to have the client itself multicast its request to all the servers. The servers can then execute some distributed protocol to decide who should service the request, taking into account the contents of their caches and their loads. The advan-

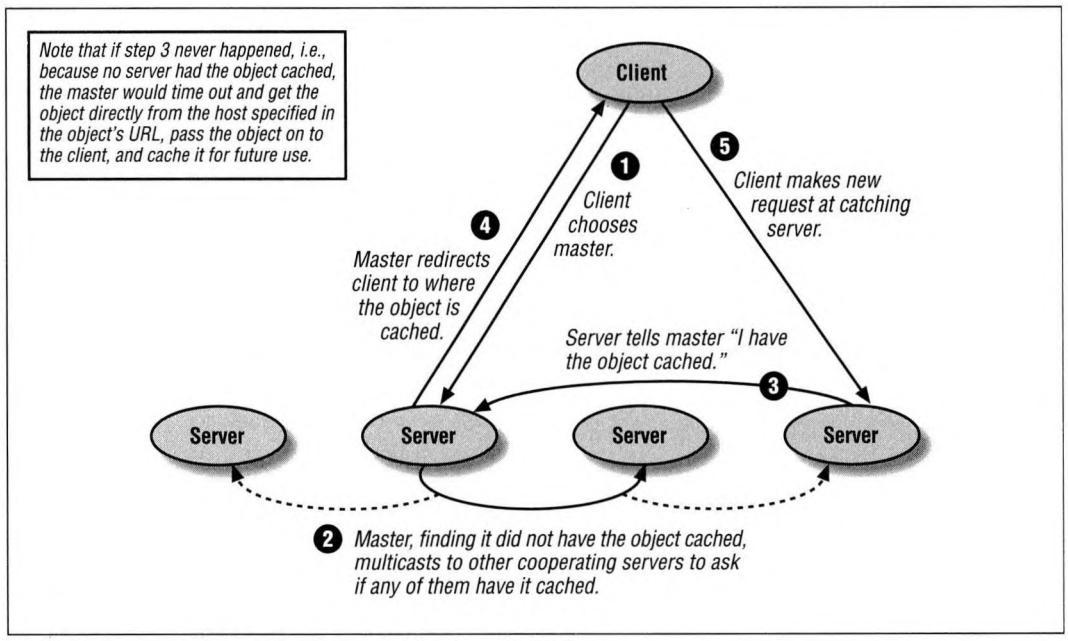

Figure 1: Illustration of the protocol used

tage of this method is that the client need not know all the servers—it just needs to know the multicast address. Hence, servers can be added and dropped dynamically. However, this method has some serious disadvantages. First, it requires major modifications to the client software. Given the multiplicity of browser implementations, it seems unlikely that the implementors of all the different browsers would be willing to make the appropriate modifications to their software. To maximize the likelihood of widespread acceptance of our techniques, we wanted to modify the client as little as possible. Another disadvantage is that it requires all client machines to support IP multicast, which is not a reasonable assumption. Hence, we felt it would be best to restrict the requirement of multicast capability to the machines on which the servers run.

Given the server multicast approach, there are still a few different approaches we could have taken. One variant on our protocol is to have the server with the cached object send the data directly to the master, and for the master to pass it on to the client. This approach was initially attractive as it requires no changes to the client. However, on further consideration, we decided against this approach as it loads two servers for a single request, and, depending on the relative positions of the client and the two servers, it can result in inefficient usage of network bandwidth. Furthermore, this makes the location of cached objects transparent to the client, making optimizations, such as always looking in the same place for inline images as for the document that contains them, impossible.

Another variant is to have the server with the cached object directly contact the client rather than having the master redirect the client. This is undesirable, since it requires the client to provide a mechanism for being contacted by a server other than the one to which it connects. This, in turn, requires major modifications to the way the client communicates with its proxies.

In our protocol, the server contacts the home server only after timing out on receiving no

responses to its multicast request. Another approach is to have the server contact the home server while waiting for responses from the other servers, so that if no proxy has the data cached the master already has a head start in getting the data. However, the reduction in latency that this yields is at the cost of eliminating some of the benefits of caching in the first place: reducing network bandwidth use and reducing load on the host specified in the URL, in the case that the object is cached locally. Thus, we do not contact the host specified in the URL until it becomes clear that the object is not locally cached.

Implementation

To build on existing work, we decided to implement our new client and server as modifications of widely available products. We selected NCSA's Mosaic [1] as the basis for our client because of the availability and simplicity of its source code. We selected CERN's httpd 3.0 [13] as the basis for our server because its source code was available and already implemented proxy caching.

Client Modifications

The basic modification we made to the client was to implement the redirection mechanism. This was done by extending HTTP to include a special proxy redirect result code, 317, and by modifying the client to interpret this code. Upon receipt of this code, the client changes the proxy it uses to the one specified by this message, and then sends its request to this new proxy. The other modification we made to the client was to have it select a random proxy from a list of proxies for each new document requested. It then uses the same proxy for all related objects, such as inline images.

Note that we had to slightly modify HTTP to accommodate our protocol, adding a code for proxy redirect messages. This change seems justified, because the proxy redirect message is a natural extension to HTTP.

Server Modifications

Modifications to the server were more extensive. First, we had to make it select and join a multicast group, and listen to this multicast group at the same time as it was listening for client requests. Second, we had to change the way it treats client requests, to satisfy the protocol. Thus, when it fails to find an object in its cache, it sends a multicast message asking if the object is cached elsewhere. If there is no response within a certain time, it proceeds, as usual, by getting the object from the host specified in the URL, caching it, and returning it to the client. However, if there is a response, it instead sends a proxy redirect message to the client. Third, we had to modify the server to process queries it receives from other servers asking whether certain objects are cached. We did not have the server fork a separate process to handle each such query, as the servicing of each query was not expected to take a great deal of time.

Analysis

Although the primary purpose of our research was to show that our scheme for cooperation among multiple proxies could be implemented, we were also curious about the performance of our system and the system on which it was based. For this purpose, we obtained and analyzed measurements of the time it took to service requests under various conditions.

Testing Methodology

We will use the following terminology to refer to the four types of latency we measured. *Direct latency* is the time it takes to receive a document and its associated images directly from the hosts specified in the URLs for those objects. For instance, the time it takes to get both *http://Wall-Street-News.com/forecasts/* and *http://Wall-Street-News.com/forecasts/images/wall-street-news.gif* from *Wall-Street-News.com* is considered direct latency. *Proxy latency* is the time it takes to retrieve a document and all its associated images

from a proxy when none of those objects have been previously cached locally. *Caching latency* is the time it takes to receive a document and all its associated images from a proxy that already has those objects cached. Finally, *redirection latency* is the time it takes to receive a document and all its associated images when they are all cached at the same proxy, but this is not the first proxy the client queries. Thus, it is the time it takes to receive a proxy redirect message from the "wrong" proxy, plus the caching latency at the "right" proxy. Note that, since our optimization attempts to cache the document and its images at the same proxy, we only incur the latency of receiving and processing a proxy redirect message once for each document, no matter how many images are associated with that document.

For our tests, we obtained a random set of URLs by invoking the "random link" feature of Yahoo [9] repeatedly. We eliminated from consideration any URLs that did not use the *http* scheme. We then wrote a simple program to fetch these documents, extract the URLs for their associated images, and determine the size of each document object and image object. For ease of analysis, we then threw out any documents that referenced the same image more than once, leaving us with 46 documents on which to perform the remainder of our measurements. Two of these turned out to be unreachable during our later experiments, so the results to follow concern 44 documents.

We wrote another program to determine the direct, caching, proxy, and redirect latencies for each document with associated images. We ran this program at night, when there would be less interference from other users of network bandwidth. The measurement program and two instantiations of our caching proxy ran on three different machines in our laboratory, all on the same subnet. We determined that in this configuration a 15 ms timeout on the multicast request was sufficiently long for the correct running of our protocol, but to be conservative we used a 40 ms timeout instead. In all our presentations of results, we have not made any effort to eliminate values that might be considered statistical outliers. This is because seemingly abnormal results are typically due to bursts of uncontrollable external network activity, which are an important aspect of the environment and should be taken into consideration.

Measurements

Each type of latency was measured fifty times for each document, and the average was taken of those fifty trials. All trials for a single document and its images were done consecutively, so that all data for any such document was taken under as similar network conditions as possible. Figure 2 shows the means of each type of latency for each of the documents studied. Note that the documents are numbered in order of increasing direct latency. So, for instance, the y-values plotted above the number 5 on the x-axis represent the average direct, proxy, caching, and redirection latencies for the document with associated images whose direct latency was the fifth smallest among all documents with associated images. Note that we have only plotted values for 43 of the 44 documents, as plotting the values for the document with the highest latency would render the scale too small. Figure 3 shows the mean of each type of latency across all documents with associated images, along with error lines proportional to the standard deviations.

Discussion

Note first that these latency values are only meaningful to average if one considers the workload to be one in which only the 44 documents studied are accessed, each of them is accessed with equal probability, and all of them are accessed at night from U. C. Berkeley. Any more general interpretation of one of our average values is not accurate unless one expects the distribution of latency values for our workload to be similar to that found in a realistic client workload. In the case of proxy and direct latencies, it is

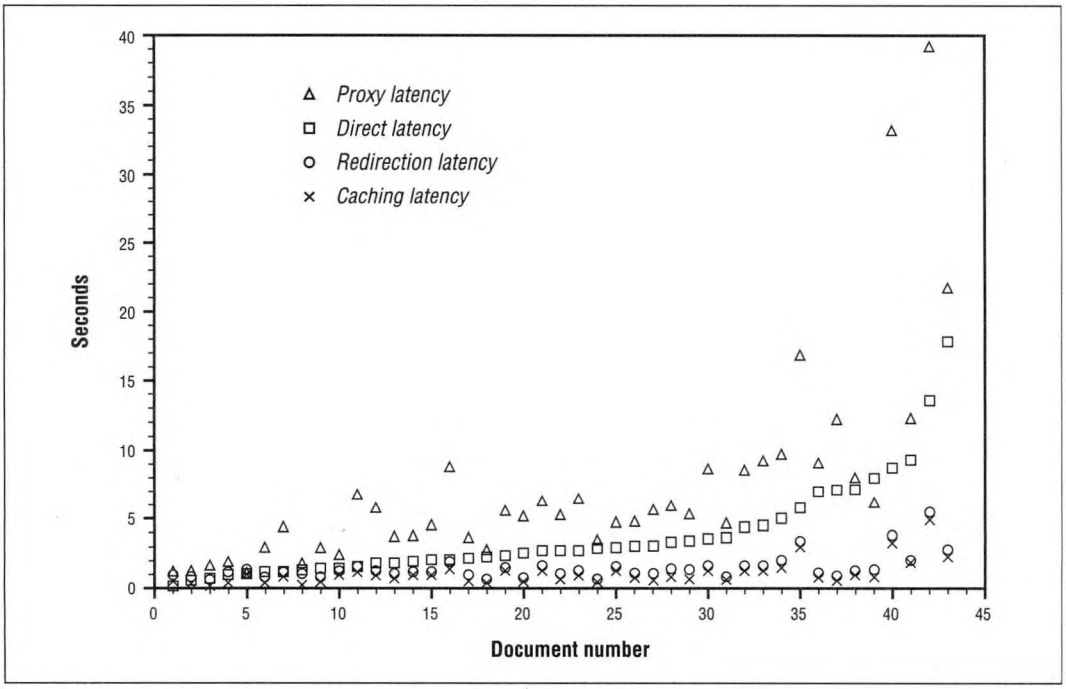

Figure 2: Latencies for various documents and their associated images

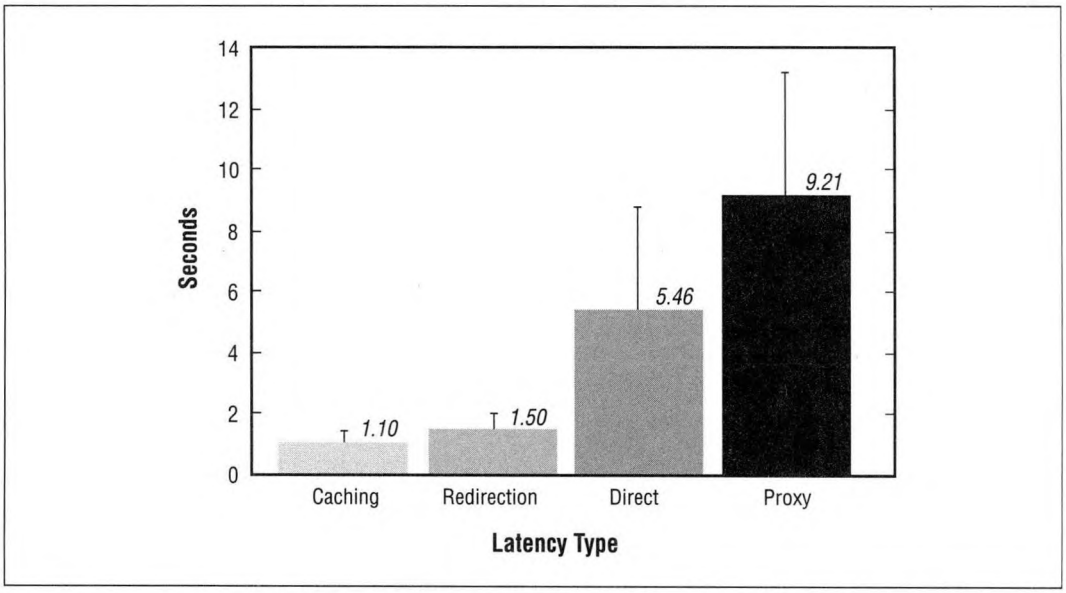

Figure 3: Means of different types of latency across all documents with associated images, with lines indicating the relative size of standard deviations

likely we have not achieved this goal, as these latencies are very dependent on the distance to the hosts and the time of access, and we have made no attempt to make the distribution of distances to hosts match that of any sort of typical client. On the other hand, since caching and redirection latency do not involve any communication with the host specified in the URLs, there should be no correlation between host distance and either type of latency. Also, there is little correlation between either type of latency and document size: both correlation coefficients have magnitudes under 0.05. Therefore, it seems likely that our caching and redirection latency values are appropriate to general workloads. However, one thing to be careful about is that the impact of redirection on latency is proportionally greater the fewer inline images a document has. If the distribution of number of inline images in documents in our workload is atypical of another workload, our redirection latency numbers may not be applicable to it.

The difference between the caching and redirection latency for a document and its images is equal to the amount of time it takes to receive a proxy redirect message. As expected, there was essentially no correlation between this value and the direct latency (correlation coefficient -0.01) or the size of the document (correlation coefficient -0.04). Therefore, we feel comfortable in averaging this number across all documents to determine an average latency for proxy redirect messages, obtaining the figure of 0.398 sec. The 95% confidence interval for this value is plus or minus 0.015 sec. Now, the average caching latency among all the documents with associated images we considered was 1.102 sec. Thus, in our workload, choosing the wrong proxy to service a request makes the request take on average 36% longer than choosing the right proxy. In a system of n cooperating servers, we can expect to incur this latency on cache hits (n-1)/n of the time. Note that the figure of 36% is specific to the document composition of our workload, but the figure of 0.398 sec is applicable to the retrieval of any document.

Most of the difference between proxy latency and direct latency is due to overhead of the original CERN server that we instrumented. Our figures show that, for our workload, the average proxy latency is 69% higher than average direct latency. However, the only part of this that is due to our technique is the 40 ms spent waiting for the multicast request to time out; thus, overhead from our protocol only accounts for 1.1% of the increase from direct latency to proxy latency. In other words, the CERN server alone takes 68% more time to service a cache miss than it would take without a caching proxy; with our modifications, this becomes 69%. Note, again, that these percentages are specific to our workload, but the figure of 40 ms overhead per cache miss is generally applicable.

Although proxy latency is significantly greater than direct latency, we believe that caching is still worthwhile. First, one must consider that although one pays a greater cost when a document misses in the cache, this is made up for when a document hits in the cache and the client experiences less latency. For our workload, average caching latency is 20% of average direct latency, and even average redirection latency is only 27% of average direct latency. Second, the benefits of caching go beyond reduction of document retrieval latency for cached objects. Caching also ensures reduced use of network bandwidth and reduced server load. These may, in turn, provide savings in document retrieval latency for all objects.

Thus, we can make the following comparison between our multiple proxy caching system and a single caching proxy. Our protocol permits sharing of multiple caches among many clients, hence we expect a higher cache hit rate. Since documents that hit in the cache take less time to retrieve than ones that miss, this will decrease our relative latency. The increased cache hit ratio will also translate into less use of network bandwidth and less load imposed on remote servers. The only overhead of our system over a single caching proxy is the 398 ms redirection delay if

the **wrong** proxy is contacted first, and the additional 40 ms timeout delay if the requested object is not cached at any proxy.

Future Work

There are several areas open for future work. One of these is the evaluation of protocols other than the one we implemented. We have presented arguments for why we chose our approach; however, to empirically evaluate how justified our decisions were, it is necessary to implement the alternative protocols we have eschewed and to compare their performance to that of our approach. Especially interesting to consider would be implementations of a protocol involving multicast messages by the client. In an environment where multicasting was more prevalent, and the problem of reliable multicast was solved, this could well be a better protocol than ours.

Furthermore, evaluation of our approach could be better done with extensive traces of document retrieval patterns, which we do not have. Such traces would provide a more realistic workload from which to get more meaningful averages of latency values.

Another avenue of investigation is the extent to which our server selection method provides satisfactory load balancing. Although it is clear that in the long run, the expected number of requests processed by each server is the same as that of all other servers with which it cooperates, it might turn out that the variance of the randomness in our system is high enough that it is likely some server will nevertheless wind up with the bulk of the processing load or disk space requirement.

Finally, we feel it would be useful to evaluate alternative caching policies to the ones currently in use. The CERN approach described essentially embodies the pinnacle of current work in this area, and in our opinion it is still not the best that could be achieved. One reason for this is that it requires manual intervention to decide upon and later tune such parameters as default expiry times for objects whose URLs conform to various patterns. More research is needed to determine how these parameters might be automatically set and modified by the server itself based on its ongoing experience. Improvement may also be possible in the estimation of expiry times for documents not containing "Expires" headers. We feel that there may be other information in a URL, besides the last-modified time, which is useful for this estimation.

Conclusion

The increasing popularity of the World Wide Web presents many challenges. A good solution to many of these challenges is caching, which reduces server load, document service time, and network load. However, the complexity of caching brings with it its own problems, some of which we have attempted to solve with our protocol for sharing caches among servers. The problems of robustness and scalability with number of clients can be solved by using multiple servers for a set of clients. Load balancing and sharing among such multiple servers can be achieved by ensuring that any server is equally likely to be chosen to satisfy any given client request. Finally, to solve the problem of communication overhead scaling poorly with communication among many servers, we propose the use of IP multicast to make this communication proceed efficiently.

We implemented the protocol that best met our needs to demonstrate the feasibility of such a protocol. We also performed measurements of this implementation to illustrate that the overhead involved in using multiple caching proxies instead of one is small, while the advantages obtained are several. We expect the sharing of caches will lead to higher hit ratios, with a corresponding decrease in network bandwidth usage, server load, and document retrieval latency.

Our system of using multiple servers to perform the work of one is generally applicable to most

current caching architectures. In a hierarchical cache structure in which clients share proxies, which in turn share proxies, etc., it would be straightforward to replace each cache server by a cooperating set of servers that use our protocol. ■

References

1. Andreesen, M., NCSA Mosaic home page, May, 1995.
2. Berners-Lee, T., Masinter, L., and McCahill, M., RFC 1738: uniform resource locators (URL), December, 1994.
3. Berners-Lee, T., Fielding, R., and Nielsen, H. F., Hypertext Transfer Protocol—HTTP/1.0, March, 1995.
4. Bestavros, A., Demand-based document dissemination for the World Wide Web, Technical Report BU-CS-95-003, Boston University Computer Science Department, Boston, MA, February, 1995.
5. Bestavros, A., Carver, R., Crovella, M., Cunha, C., Heddaya, A., and Mirdad, S., Application-level document caching in the Internet, Technical Report BU-CS-95-002, Boston University Computer Science Department, Boston, MA, March, 1995.
6. Chankhunthod, A., Danzig, P., Neerdales, C., Schwartz, M., and Worrell, K., A hierarchical Internet object cache, April, 1995.
7. De Bra, P. and Post, R., Information retrieval in the World Wide Web: making client-based searching feasible, *Proceedings of the First International Conference on the World Wide Web*, Geneva, Switzerland, May, 1994.
8. Deering, S., RFC 1054: host extensions for IP multicasting, May, 1988.
9. Filo, D., and Yang, J., Yahoo home page, July, 1995.
10. Fischer, G., *http://www.tu-chemnitz.de/~ftpadm/httpd/src/cache.html*
11. Gwertzman, J., and Seltzer, M., The case for geographical push-caching, Technical Report HU TR-34-94, Harvard University, DAS, Cambridge, MA, 1994.
12. Jain, R., *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, New York, NY, 1991.
13. Luotonen, A., and Berners-Lee, T., CERN httpd Reference Manual, July, 1995.
14. O'Callaghan, D., A central caching proxy server for WWW users at the University of Melbourne, *Proceedings of AusWeb95, the First Australian World Wide Web Conference*, March, 1995.
15. van Oosten, G., Article posted to *comp.infosystems.www*, February, 1994.

About the Authors

Radhika Malpani
[*http://http.cs.berkeley.edu/~radhika/*]
University of California at Berkeley
radhika@cs.berkeley.edu

Radhika Malpani is a Ph.D. student in the computer science department of the University of California at Berkeley. She is a National Science Foundation Fellowship recipient. She holds a B.E. in electrical engineering from the Victoria Jubilee Technical Institute in Bombay, India. Her current research interests include continuous media applications for the Internet and the Mbone.

Jacob Lorch
[*http://http.cs.berkeley.edu/~lorch/*]
University of California at Berkeley
lorch@cs.berkeley.edu

Jacob Lorch is a Ph.D. student in the computer science department of the University of California at Berkeley. He is a National Science Foundation Fellowship recipient, as well as a member of ACM and IEEE. He holds a B.S. in computer science and a B.S. in mathematics from Michigan State University. His current research interests include operating systems techniques for reducing the power consumption of laptop computers, and caching strategies for the World Wide Web.

David Berger
[*http://www.eit.com/~dvberger/*]
Enterprise Integration Technologies
dvberger@eit.com

David Berger is a software engineer at Enterprise Integration Technologies (a subsidiary of Verifone). He is conceiving and developing products for the global Internet and the World Wide Web. A former graduate student in computer science at

the University of California at Berkeley, he received an M.S. in 1995 while working on the Berkeley Video on Demand System. David also holds a B.A. in computer science from Rutgers University and is a member of the Phi Beta Kappa and Phi Eta Sigma national honor societies.