Distributed Browser Caching

Terry Koo, Steven Chan, Tsehsiang Teo, Matthew Brown
December 6, 2001

ABSTRACT

This is a simple architecture for a distributed web caching system that allows computers connected by a reasonably fast LAN to share locally cached web content. Since it is distributed across the individual computers, the system would be inexpensive, easy to maintain, and efficient. We also provide benchmarks for tests we performed that show the performance gained by using a distributed web cache.

1. INTRODUCTION

Cache systems generally reduce network traffic throughout large networks by minimizing the need to access the original server for a piece of static data. Typically, HTTP responses are either stored in a local browser cache, or in a central proxy server. The distributed caching system merely takes advantage of local networking capabilities to aggregate individual computer caches into more comprehensive cache.

Consider for a moment four roommates who share Internet access through a router. None of the roommates wants to be responsible for setting up and maintaining a proxy server, yet they want the performance advantages of a shared cache. We present an architecture for a decentralized client web cache that allows each client to access the cached documents of all clients. With a larger set of cached documents to draw upon, hit rates in the cache will increase. As long as communication between client computers is fast, performance is far better than that achieved by individual client caches, and is comparable to that of a centralized proxy's caching system. Moreover, the system requires far less maintenance than a centralized proxy, as it does not require any dedicated hardware and responds automatically to changes in network topology.

We will begin our discussion by examining the usefulness and need for such a system; we will compare our system to a single browser cache on a client machine and even to the cache on a central proxy server. The following discussion will focus on examining the design aspects of the distributed cache system, as well as the different implementation options available. Finally, we provide a test of a prototype implementation that shows the advantages of using a distributed caching system.

2. MOTIVATION

We propose an architecture for a distributed web browser cache and assess the feasibility and benefits of this system over existing web caching systems. The distinguishing feature of our system is that it allows users to pool their individual browser caches into a consolidated unit without requiring a single point of coordination or control.

Our system makes a few optimistic assumptions. Firstly, it assumes that users are likely to visit a large number of common sites. Hence, when a user fetches a web document through his browser, it is likely that another user has already fetched and cached that same document. Secondly, it assumes that fetching documents from cooperating nodes is much faster than fetching them directly from their original source. These two assumptions indicate that document retrieval time can be reduced by enabling computers interconnected by fast connections to share their browser caches. In our system, this is done by introducing an additional level of caching: if the document is not in the local cache, an attempt will first be made to fetch it from a peer's cache before contacting the original web server.

Additionally, our distributed web cache system will be able to make better use of unused resources. Considering that only half of all disk space available is actually in use, and most machines are idle most of the time [2], our system will not only cause minimal interference with most users, but will also make use of otherwise wasted drive space and CPU time. Since half of all machines are accessible 95% of the time [2], our system can assume that much of the network machines are available for caching. In view of these three findings, it appears that a distributed web cache system could provide broad performace gains, with minimal negative impact on its users. The overhead involved in operating the system could easily be covered in idle CPU time and disk space.

2.1 Distributed Browser Cache vs. a Single Browser Cache

A distributed browser cache performs the same basic function as a single browser cache; it speeds up the fetching of documents by loading files from the cache whenever possible. However, a distributed browser cache offers several distinct advantages over a single browser cache.

A distributed browser cache spans several users. Therefore, there is a greater chance that a requested document has previously been accessed and hence cached by another user. Additionally, there is also more memory and disk space available across a distributed web cache since the cache is not limited to a single computer. In fact, every new user on the system brings with him additional storage space for the cache. Both of these factors translate to a potentially higher hit rate for the distributed cache system resulting in a reduction of external web server accesses, and an increase in speed of web page retrievals.

Aside from this increased speed, the ability to fetch a document from a peer cache also results in increased availability of cached web pages. This is especially true when the connection to a particular web server is flaky or if the web server is experiencing an extremely high load. To illustrate this, let us consider a significant news event occured. Large numbers of people would simultaneously check the few authoritative news sites available such as cnn.com and bbc.co.uk for reliable information. These few news sites would then become badly congested. Without a distributed browser cache, every user would have a low probabability of receiving a timely http response even if he were on the same local network as another user who had previously managed to fetch the same document. With a distributed browser cache on the other hand, once a single user has managed to retrieve a particular news page, all his peers utilizing the same distributed browser cache would then be able to access that same document almost instantaneously in spite of congestion on the web server.

However, there are some drawbacks of using this system instead of a standalone browser cache. First of all, the additional network overhead may slow down the overall system if the number of cache misses is large. Secondly, it seems plausible that a malicious peer could insert his own documents into the shared cache in order to prevent other peers from getting their requested documents or to provide them with false information. Finally, privacy is also an issue since each browser's cache is essentially leaking information about its user's surfing habits.

2.2 Distributed Browser Cache vs. Centralized Proxy Server

Admittedly, a central proxy server could provide the same functionality as our distributed web cache. In fact, it even has a few advantages over our system. Most obviously, a central proxy server has lower latency than a distributed browser cache, since it does not have the overhead of querying peer caches for documents. In addition, security is also less of an issue since the integrity of the single proxy server is easier to verify than having to inspect every single peer in a distributed browser cache system. The central proxy server is also probably simpler to implement since it does not rely on cooperation of its users. Finally, a central proxy server also provides some additional functionality that a distributed browser cache system cannot provide. For example, many proxy servers can also filter traffic as well as bridge Intranets to the Internet.

However, a distributed browser cache still provides many advantages over a central proxy server. A central proxy server incurs maintenance costs. It requires assigned maintenance staff as well as expensive dedicated hardware. Because of this, a central proxy server may be inconvenient for small, informal networks such as small office or home networks.

In addition to these costs, a central proxy server must constantly be upgraded to keep up with an expanding network. On the other hand, the total storage space and computing power in a distributed browser cache grows with each additional user. In this sense, our system scales better than a central proxy server.

Also note that a centralized server is a single point of failure; if the single proxy server crashes or fails, the entire cache is lost and the web is inaccessible untl the server is brought back online. In contrast, a distributed browser cache allows individual nodes to crash or log off, without losing much functionality. In the worst case, a portion of the cache may be lost, but both web access and the caching system will continue to function.

Finally, the amount of replication of a particular document over a distriuted browser cache is also directly proportional to the popularity of the document. This is attractive since popular documents would not impose a high load on any particular machine.

3. SYSTEM DESIGN

3.1 Overview

Each node in our distributed browser cache plays a number of roles (see Figure 1). When the user wishes to fetch a docuemnt, the node serves as a *Requestor*. When responding to other users' requests, the node acts as a *Cache Server*. Finally, if a document request hits locally, the node functions as a single web cache.

In its role as a Cache Server, each node listens for URL document lookups and document transfer requests by Requestors. The mechanism of URL lookups is discuesed later in the Document Lookup section. Document transfer requests are fulfilled if the document is available on that Cache Server.

In its role as a Requestor, each node listens for http requests from the local browser. Before searching the cache system, it checks the request to determine if the document can be safely fetched from a cache,

If the document cannot be fetched from the cache, the Requestor then fetches the document directly from the web server. Otherwise, the web proxy performs a document lookup for the URL in the distributed cache. If the distributed cache does not posses the document, then the document is then fetched directly from the web server.

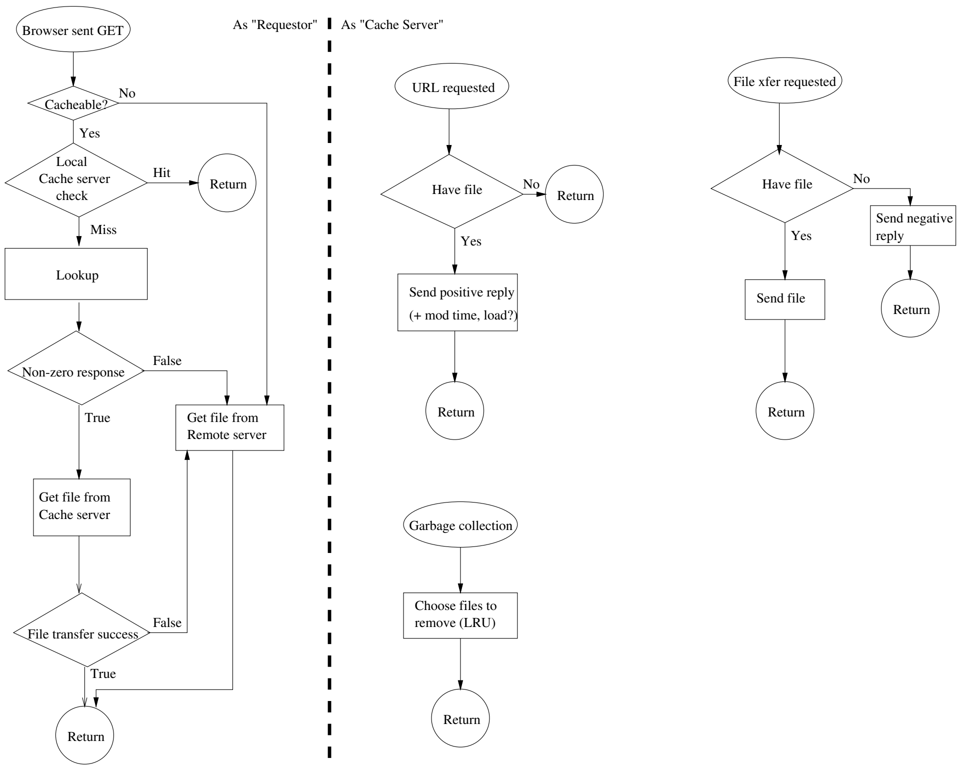

Figure 1: Program Flowchart of the distributed web cache running on a computer

As an illustration of how this would work in practice, suppose Bob wants to view the main MIT web page "http://web.mit.edu/index.html". Bob would type this URL into his browser. Subsequently, the web browser would submit a GET request to the Requestor, which performs a lookup within the distributed cache.

3.2 Optimizations

3.2.1 Assuming locality of reference

As an optimization, once a node has succesfully fetched a document from a Cache Server on the network, it assumes that the Cache Server also holds all documents from the same web server. This assumption is made on the basis that when users are surfing a particular page, they will also be downloading related content from the same web server such as the pages' images, and maybe even links from the original page.

As such, future requests for documents from the same web server would optimistically request the URL from the same Cache Server first until a miss occurs. When this happens, it is assumed that this particular Cache Server no longer holds content for the particular web server, and another document lookup request is sent out for new candidate Cache Servers.

3.2.2 Load Balancing

It is preferable that Cache Servers are loaded as evenly as possible in the network, to avoid bottlenecks developing on individual Cache Servers. As such, the current number of connections (or some other load measure) can be sent with the replies to requested URLs, so that Requestors can decide on a more lightly loaded machine to effect the transfer from.

3.2.3 Ignoring Freshness

Some sites, especially news sites, have their html text pages expire almost immediately. This could seriously constrain the benefit from caching, since this would force the browser to bypass the cache mechanism. Hence, we believe that freshness should not be strictly enforced for the best performance.

Rather than enforcing any particular policy towards freshness, we apply the End-To-End argument by informing the user that the document is supposed to be expired, and leave it to the user to force a reload. This might be done in a number of ways: by inserting HTML text, by altering the title of the document, or by displaying a message popup. This method would be especially useful if the Internet link is congested; users might prefer a stale document to no document at all.

3.2.4 Portability

Our distributed cache nodes are implemented as a loopback proxy server on each client machine. This confers the advantage of easily allowing any modern browser to make use of the distributed web cache by appearing to be a transparent HTTP proxy server on the localhost. In addition, this allows a common cache to be used by different species of browsers simultaneously.

3.3 Document Lookup

The preceding descriptions of our distributed web cache have assumed that the cache server(s) responsible for a given document are already known. In reality, however, this knowledge is unavailable and must be discovered using a document lookup subsystem. Here, we compare three alternatives we considered for document lookup: unicast, broadcast, and our chosen system, Chord lookup.

3.3.1 Unicast

Unicast is the most basic of the three lookup systems. It assumes that a list of cache servers is available through user input (for instance, via the command line or a graphical interface). Unicast lookup proceeds as follows:

1. The requesting computer chooses a cache server from its list and requests the desired document.
2. If the cache server has the document, lookup is concluded. Otherwise, remove the current cache server from the list and return to step 1.

Note that in step 2, to prevent unbalanced loading of cache servers, the servers are chosen randomly.

While a unicast system has an attractively simple design and implementation, it is impractical for a number of reasons. For instance, unicast lookup employs a polling algorithm to perform lookup. Suppose that there are n nodes in our system; this means that before a cache miss can be reported, $n-1$ servers must be queried. Even in the case of a cache hit, the number of servers queried, and thus the added latency and overhead of lookup, scales linearly with n. In addition, a unicast system assumes the user manages the list of known cache servers. This places unreasonable responsibility on the user and restricts the lookup system's ability to deal with the entrance and exit of cache servers.

There are other problems with unicast, but rather than explore these as well, we will simply stop here. Clearly, the benefits of unicast are already far outweighed by its drawbacks. A better lookup system is broadcast lookup, described next.

3.3.2 Broadcast

A broadcast lookup system assumes all of the cache servers are connected by a broadcast-capable medium, such as an Ethernet. This immediately places limitations on the usability of broadcast lookup; if broadcast communication is unavailable or inefficient, broadcast lookup is infeasible. However, if the benefits of broadcast lookup are great enough, then it may be worthwhile to use it.

A broadcast lookup proceeds as follows:

1. The requesting computer broadcasts the name of the desired document.
2. All cache servers hear the request, and any that have the desired document respond with a brief confirmation.
3. The requesting computer chooses one of the responding cache servers and downloads the document from it.

Unfortunately, beyond the problems with its usability, there are fundamental problems with broadcast lookup. To illustrate these problems, we will explore one specific problem with broadcast lookup in depth. Suppose a popular document is requested. Since it is popular, it is likely to be found on many of the cache servers; all of these cache servers will respond more or less simultaneously, briefly swamping the network. If popular documents are requested often (which seems likely, given the definition of "popular"), then the broadcast system will suffer from a constant flood of confirmation messages.

The intuitive workaround is to have cache servers cancel confirmation messages for a given document if they overhear confirmations for the same document from other servers. A direct implementation of this is more complex than it appears, however. Since the broadcast medium will deliver the desired document's name to all cache servers at approximately the same time, all cache servers will forward their confirmations to the link layer at approximately the same time. Thus, by the time the cache servers overhear each others' confirmation messages, it will be too late to cancel the messages.

An improved workaround is for cache servers to delay for some short random time interval before sending confirmation messages. While each cache server waits out its delay, it listens for competing confirmation messages, and cancels its own pending confirmation message if it overhears a competing one. This approach will work, but only at the cost of examining every message heard on the network; as the system grows, it is likely to suffer from receive livelock. Given the dangers of this possibility, it may be that allowing the large number of confirmation messages is the more efficient alternative after all.

A final workaround is for cache servers to delay as described above, but not eavesdrop to determine when competing confirmations are sent. Instead, the computer that broadcasted the original document request broadcasts an acknowledgment message once it receives a confirmation message. The cache servers cancel their confirmation messages if they hear this acknowledgment message. The advantage of this approach is that cache servers do not have to analyze every message on the network; the disadvantage is that the delay interval must be lengthened to allow time for the confirmation message to propagate. Another disadvantage is that even with this mechanism in place, the number of computers that respond to a document request will still scale linearly with the number of computers in the network, albeit at a slow rate. This prevents broadcast lookup from use with large numbers of computers. Of course, this may not prove directly restrictive, as efficient broadcast media, such as Ethernet or wireless LAN, already place restrictions on the number of participating computers. Thus, the poor scalability of broadcast lookup may never come into play in an actual application.

The above explanations have attempted to illustrate the two-edged nature of broadcast communications. On the one hand, they enable fast and cheap communication with large groups of computers, so that, in effect, every computer on the network can be polled simultaneously. On the other hand, each computer must process every broadcast message, which can easily spiral into receive livelock unless carefully dealt with.

Designing an efficient broadcast system is a difficult, though perhaps not impossible task. Nevertheless, even if an efficient broadcast lookup protocol could be designed, its dependence on broadcast communication would still render it unattractive. Chord lookup, which we will describe next, can provide at least as much efficiency without the dependence on broadcast communication.

3.3.3 Chord

We will begin by briefly describing the Chord system. The Chord lookup system maps identifiers to the nodes responsible for them. Chord provides two services: it presents a lookup method that returns the IP address of the node responsible for a given identifier, and it allows programs to be notified on node entry and exit events, allowing responsibilities to be shifted appropriately at the application level.

Conceptually, the Chord identifier space is arranged as a ring into which the nodes are scattered. Each node takes responsibility for the identifiers between its predecessor in the ring and itself; said another way, each identifier maps to its successor node in the Chord ring. In our system, the Chord identifiers are hashes of document names, and the Chord nodes are the cache servers.

A lookup proceeds as follows:

1. The requesting computer hashes the document name into a Chord identifier and uses it to perform a Chord lookup.

2. The lookup returns the IP address of the cache server that takes responsibility for the document. This computer is known as the document manager.

3. The requesting computer sends a query containing the desired document's name to the document manager.

4. The document manager sends back a list of IP addresses of cache servers that have the document.

5. The requesting computer chooses a cache server out of the list and proceeds to download the cached document.

Note that in the above, we create the role of the document manager; this is necessary for the following reason. Chord lookups are restricted to returning a single node as their result, yet we would like to allow replicas of documents, for load balancing and availability. We work around this limitation by adding a level of indirection: the document manager.

This lookup system has a few drawbacks, which we now describe. First, there is the problem of latency. Performing a Chord lookup entails several network hops, which slows the lookup process. The addition of the document manager's level of indirection introduces another network hop, adding more latency.

Another problem is the addition of complexity. Document managers maintain a mapping from document names to lists of cache servers that cache the document. This state must be carefully coordinated. For instance, when cache servers obtain new documents, they must inform the appropriate document manager, and conversely, when a cache server removes or loses a document, the document manager must again be informed.

In addition, if a document manager leaves the network, the cached documents it had been managing will become unreachable, even though the cache servers on which they are stored still on the network. Thus, the document manager's state itself must be replicated. This is accomplished by occasionally copying a version of the document manager's mapping to its Chord successor node. The Chord lookup mechanism is such that if a node vanishes, its responsibilities are passed on to its successor node. Thus, when a document manager exits the network, the Chord lookups that it normally would have received are received by its successor, which has the backup mapping.

We have seen that the Chord lookup system introduces both latency and complexity. Nevertheless, we feel that these drawbacks are an acceptable tradeoff in exchange for the benefits that Chord provides. We now explore those benefits.

First of all, the Chord lookup system scales efficiently. In a steady-state Chord ring of n nodes, a lookup requires $O(\log(n))$ network messages. The system can also cope with node entry and exit efficiently; returning to a stable state after a node entrance or exit requires, with high probability, $O(\log^2(n))$ messages. This is far superior to a broadcast lookup system, where the number of messages to be processed scales linearly with the number of nodes.

In addition, Chord lookup does not depend on any special properties of the communication medium, other than IP routing. Broadcast lookup, on the other hand, depends on broadcast communications, which restricts its use in a number of otherwise attractive scenarios.

For instance, a college campus generally contains several computer labs; each lab's computers are linked by a fast Ethernet, and the labs themselves are joined by high-speed connections. The labs, however, must all share the same link to the Internet. While broadcast lookup could locate documents within each lab individually, it would be unable to locate documents in other labs. Chord lookup, on the other hand, could locate documents in all labs. Thus, Chord lookup creates a distributed cache that is far greater in size than broadcast lookup. Assuming that web access patterns are sufficiently varied to benefit from the expanded cache, performance will increase with a larger cache.

It is for these benefits that we believe Chord lookup is the superior document lookup system.

4. DEMONSTRATION SYSTEM

Squid is a highly configurable open-source caching proxy for Unix systems. We used Squid as the base for our test implementation of a distributed browser cache using unicast lookup. Its ease of configuration and fantastic array of options made it easy to prototype a sample implementation of our system.

While Squid is usually used as a proxy server, we installed it on each test machine as a loopback server, effectively using it as a browser cache. Squid has a built-in ability to communicate with other Squid proxy servers, in order to create a distributed proxy structure. Since we are using Squid as a browser cache instead of a proxy server, this effectively confers on our browser caches the ability to intercommunicate.

A test script was written that fetched a list of URLs using *WGET*, with random delays of up to 20 seconds between each request, to simulate humans requesting the same documents at around the same time. *WGET* was invoked with the http_proxy environment variable set to point to the copy of Squid running on the local machine. Since *WGET* has no internal caching mechanisms, the local Squid proxy acts as the browser cache for *WGET*. The tests were run on a cluster of seven identical Linux machines in the same 10baseT Ethernet segment. In order to eliminate the effect of DNS caching on our results, we ensured that DNS names were cached on the individual machines by performing the lookup just before fetching each document.

4.1 Tests run

We ran four sets of tests. Each test was run once with Squid and once without any caching. Each Squid run was done on a clean cache, so the improvements over no caching would be directly due to cache intercommunication. Notice that separate testing with isolated individual caches is no longer necessary, because this would be identical to testing without any caching at all, with our clean-cache setup.

For the first three sets, we provided every machine with the same set of URLs do demonstrate the effect of a distributed browser cache when all the users are browsing the same set of documents. The first test involved pages residing on web servers that responded quickly, while the second test involved pages on slow servers (low bandwidth and high latency). The third test reused the fast links, but retrieved over a simulated 56K modem, while the fourth test retrieved documents unique to each computer so there was no request overlap.

The first two tests were meant to compare the effects of external network speed and latency on the performance of the distributed browser cache. We expected that as the response time from the remote server drops, the effect of the cache diminishes.

The third test simulated a slow modem connection to the Internet by by connecting the cache servers to a web proxy program which added a delay proportional to the number of bytes sent per burst. The delay was set to average to a 56kbps connection, so each computer appeared to have its own 56k modem line to the external network, but was also interconnected by 10base-T Ethernet.

The fourth test was done in order to show the worst case scenario in which all the users requested documents with no spatial locality. This would show up any overhead Squid adds to direct retrievals.

4.2 Results obtained

As you can see from table 4.2, Squid is clearly faster than directly accessing the remote server. The improvement is roughly commensurate with the slowdown from external access: high latency sites did not give as large an improvement as the 56Kbps restriction, the latter being 200 times slower than the full speed of the LAN and scaling with document size.

With our setup, it was possible for four different average retrieval times to occur. The longest would be over the direct Internet connection, if Squid was not in use. Second slowest would be if Squid was in use, and thus some documents were retrieved directly while the others were retrieved over the LAN. A faster retrieval time would be obtained if Squid was in use and one node of the distributed cache possessed the documents. Fastest would be if the documents were already in Squid's local memory or disk cache. The results obtained were as expected.

	min	avg	max
Fast sites:			
Direct	22.82	25.77	28.05
Squid	11.18	13.65	16.15
Slow sites:			
Direct	78.44	93.63	118.26
Squid	37.89	55.30	69.53
Fast sites (56k):			
Direct	183.59	200.68	209.69
Squid	10.08	23.48	54.28
Worst case:			
Direct	6.22	12.21	18.67
Squid	7.33	11.75	16.23

Table 1: Distributed browser caching using Squid showed a significant improvement over no caching, ranging from around 2x to almost 10x in the case simulating individual 56Kbps internet connections. Even in the worst case, where a cache could not help at all, Squid did not make things worse, which is probably a testimony to its skill of construction.

	min	avg	max
Slow sites:			
Direct	78.44	93.63	118.26
Squid (clean)	37.89	55.30	69.53
Squid (cached only at 1)	17.72	23.34	32.36
Squid (already cached)	3.24	4.77	8.9 (25.51)

Table 2: To see the effects of caching at various levels, this set of URLs was fetched with the above network conditions. They reflect normal expectations quite well, with a clear improvement from direct internet requests down to having a locally cached copy. The bracketed value in the last case was rejected from the average time computation, because it was very far away from the next highest value, and was probably due to an intermittent fault on the machine.

	min	avg	max
Slow sites:			
Squid (initial)	80.65	100.5	118.94
Squid (subsequent)	37.89	55.30	69.53

Table 3: Running test-slower.in for the first time using Squid gave unbelievably bad results. Subsequent runs (after resetting the cache) gave a much better score. We believe this reflects DNS lookups being cached by the OS, so allowing subsequent lookups to be resolved without any external network access. As a result, all tests using a new set of URLs were preceded by a "warm-up" run first.

During testing we realized that DNS lookups were slowing down our document retrieval, even if another computer had the document; the requesting computer still had to go to a remote server to do the DNS lookup, because Squid does not share DNS resolutions to our knowledge. Without the DNS lookup overhead, the Squid distributed cache consistently provides consistent performance gains without significant maintanence.

It should be noted that the tested situation is quite contrived. It is more likely that accesses will not overlap, and hence the earliest surfer will be "rewarded" with the longest latencies, while subsequent surfers at those sites will see an enormous speedup as all (static) requests are served locally.

Finally, this system may not perform optimally in an ad-hoc network, because the transient nature of connectivity may reduce the availability of cached content, making the distributed cache appear smaller than on an always-on, always-connected network. However, this system will always perform correctly, because it is not tied to network state and so is immune to an amorphous network.

4.3 Discussion

The improvement obtained by using Squid is sensitive to the cachability of data. For example, some pages expire almost immediately (www.cnn.com for example, expires in 1 minute), making Squid unable to cache the page because it must obey the header under the HTTP spec. This is especially noticeable for high-latency sites, since a single page that cannot be cached (such as a counter), will immediately increase the retrieval time from a couple of seconds (over the LAN), to 30 seconds or more (over the high-latency link).

5. RELATED WORK

The idea of a distributed web cache has been explored in previous studies, a few of which we will briefly describe. We also summarize systems with which our system shares common goals and design principles.

5.1 Serverless Network File Systems (xFS)

The xFS system is a network file system which, at its basic level, is similar to our own system. xFS distributes file management, data storage, and other duties among a set of computers sharing a fast LAN. However, its assumptions and constraints are in some ways quite different from our own; for instance, xFS assumes trust between the cooperating machines, whereas we cannot, and xFS must preserve files persistently, whereas we can allow an occasional cached document to slip through the cracks. Despite this, the xFS system provides several insights into the tradeoffs involved in our system. [1]

5.2 Feasibility of a Serverless Distributed File System

This paper presents a preliminary study done by researchers interested in implementing a fully cooperative network file system. In order to ascertain the feasibility of their proposed system, they collected statistics on computer usage, uptime, and access patterns in a variety of computing environments.

We used their data and analysis to guide the design of our own system. [2]

5.3 Design Issues of a Cooperative Cache with no Coherence Problems (PAFS)

This paper describes the caching mechanism of PAFS, a parallel/distributed file system, which takes an alternative approach to cooperative caching. In a distributed cache, data is typically replicated to provide performance, availability, and reliability. However, in solving those problems, this strategy creates the new problem of cache consistency and adds complexity to the system. The PAFS approach does away with replication, yet maintains high performance by streamlining the process of retrieving data from a remote cache. Because their system is targeted for a highly controlled environment (i.e. a multiprocessor computer), reliability and availability are not essential issue. As such, it is not directly applicable to our system, but the design nevertheless provides meaningful insights. [3]

5.4 Hint-Based Cooperative Caching

This paper describes an alternative approach to locating cached data. Rather than attempting to locate data exactly, which generally involves several expensive network hops, locally-stored and possibly incorrect "hints" quickly give a probable location. Their research has shown that hint-based cooperative caching performs as well as tightly coordinated caching, but without the complexity of synchronizing caches. Our system's lookup procedure incorporates hints: when it needs to find a cached document, it first asks the cache server it last received a document from. Thus the last document's Cache Server serves as a hint for the location of the next document. [4]

5.5 Consistent Hashing and Random Trees

This paper introduces the use of "consistent hashes" to organize data in a distributed system. We considered using a consistent hash-based lookup, such as Chord, in our system, but decided the added functionality was not worth the added latency and complexity. [5]

6. CONCLUSION

A distributed browser cache does indeed speed up retrievals. Unicast, and probably broadcast, lookups work fine over small networks, but we expect large networks will require Chord to avoid lookup congestion. This system promises to be especially useful in small networks, due to its simplicity, very low cost, and usage of off-the-shelf components.

7. REFERENCES

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang, *Serverless Network File Systems*, University of California at Berkeley, 1996 ACM

[2] W. Bolosky, J. Douceur, D. Ely, M. Theimer, *Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs*, Microsoft Research, 2000 ACM

[3] T. Cortes, S. Girona, J. Labarta, *Design Issues of a Cooperative Cache with no Coherence Problems*, Universitat Politecnica de Catalunya, Barcelona, 1997 ACM

[4] P. Sarkar, J. Hartman, *Hint-Based Cooperative Caching*, IBM Almaden Research, University of Arizona, 2001 ACM

[5] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, R. Panigrahy, *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, Laboratory of Computer Science, MIT, 1997 ACM