Directory Based Multi-tier Internet Architectures

Sinisa Srbljic, Dalibor F. Vrsalovic*, and Andro Milanovic [1]

University of Zagreb, Zagreb, Croatia
http://www.zemris.fer.hr/{~sinisa, ~andro}

* Intel Corporation, New Business Group, Santa Clara, CA, USA
dalibor.f.vrsalovic@intel.com

ABSTRACT

Traditional two-tier client/server architectures are sometimes replaced by more scalable multi-tier architectures based on caches. However, the effectiveness of these caches, proxy caches, is often not very high. One of the possible ways to improve their effectiveness is to connect them together to create a distributed cache system. A Distributed Cache Management (DCM) protocol provides the communication infrastructure for such a distributed cache system.

A version of a DCM protocol was introduced in Harvest, and improved in Squid. After that, the Berkeley protocol further attempted to improve the performance of DCM protocols. We introduce a directory-based DCM protocol that attempts to provide additional improvements. We compare the performance of Squid, Berkeley, and the directory-based DCM protocol, with measurements from real operational systems and analytical performance prediction for large-scale distributed caches.

Keywords: World Wide Web proxy cache, distributed cache, distributed cache management protocol, performance measurement, analytical performance prediction, performance comparison.

1. INTRODUCTION

The rapid growth of the Internet has brought about the problems of maintaining low latency and avoiding congestion in networks and server machines. One possible solution is rather simple and consists of extending the capacity of networks and server machines. However, this solution is not always practical, because it introduces large costs. Another solution was proposed in the form of the Internet proxy cache. The purpose of the proxy cache is to a certain extent similar to the purpose of a processor cache. Client software contacts the proxy cache and requests an object from the cache, instead of sending the request to the server. The proxy cache will service the request from its local storage for both possible cases that may occur. For the first case, the proxy cache examines its local storage and does not find the object there. The proxy cache then contacts the origin server and fetches the object from the server. For the other case, the object is found in the proxy cache's local storage and is served to the client without any communication with the server. The latter case presents the improvement that results from the use of a proxy cache.

Unlike a processor cache, the Internet proxy cache has a low effectiveness. This is caused by a low hit rate. The hit rate is the probability that the object that the client requests is stored in the proxy cache. There are two major reasons that make this probability low for Internet systems. The first reason is that there exist a large number of different Web pages and documents on the Internet today, while the local storage of proxy caches is limited. The second reason is that users usually look for diverse information. One way to increase the hit rate is to connect more users to the proxy cache. If a larger number of users are connected to the proxy cache, the probability they will request the same document increases. However, this solution is not realistic, because a larger number of users will also increase congestion on the proxy machine.

Another possible solution to this problem is to connect multiple proxies together and thus create a distributed cache. This solution is viable if two requirements are satisfied. The first requirement is that the network connections between the proxy cache machines have lower latency than the connections to the servers. The second requirement is that the proxy machines have a low response time. What makes this solution viable is the fact that while the total number of clients connecting to the distributed cache grows, the number of clients per proxy cache remains constant. In order to control such distributed cache systems, a distributed cache management (DCM) protocol is introduced.

The DCM protocol controls the communication between proxy caches. One of the first proxy caches that introduced a DCM protocol was Harvest [1] as well as its successor Squid [2]. A new problem arises as the number of proxy caches in a distributed cache grows. With the growth of the distributed cache, the network traffic and proxy machine load rise as well, due to the communication overhead introduced by the DCM protocol. This problem limits the scalability of the distributed cache. The cache system proposed by Malpani, Lorch, and Berger [3] (referred to hereafter as the *Berkeley* protocol), was introduced to solve this problem. Since the Berkeley protocol only partially solves the problem of large traffic, we developed the directory-based DCM (dbDCM) protocol that attempts to address both the problem of traffic and proxy load of a large-scale distributed cache[1].

We run tests and compare measurement results for both the Squid DCM protocol and our dbDCM protocol. Implementation of the support for complete Squid cache in an open IP platform makes these tests and comparison possible. In addition, we develop an analytical performance prediction model to evaluate DCM protocols for large-scale distributed caches, because it was difficult to perform measurements on more than fifty proxy machines. In Section 2, we describe the DCM protocols and make a simple comparison of the Squid and Berkeley protocols. Section 3 describes the dbDCM protocol, the main design issues in designing a large-scale directory, and the implementation of the dbDCM protocol. The results of performance measurement and analytical performance prediction for Squid, Berkeley, and the dbDCM protocols, are presented in Sections 4 and 5, respectively. Final comments on the scalability of these DCM protocols are given in Section 6.

[1] The directory-based DCM protocol was designed in 1995 while the authors were with AT&T Labs, San Mateo, CA. It was coded and integrated into the Harvest cache [1] at the end of 1995. The Harvest cache was replaced with the Squid cache [2] in 1999. The work on the patents started at the beginning of 1996, resulting in patents issued in 1999 [4] and 2000 [5].

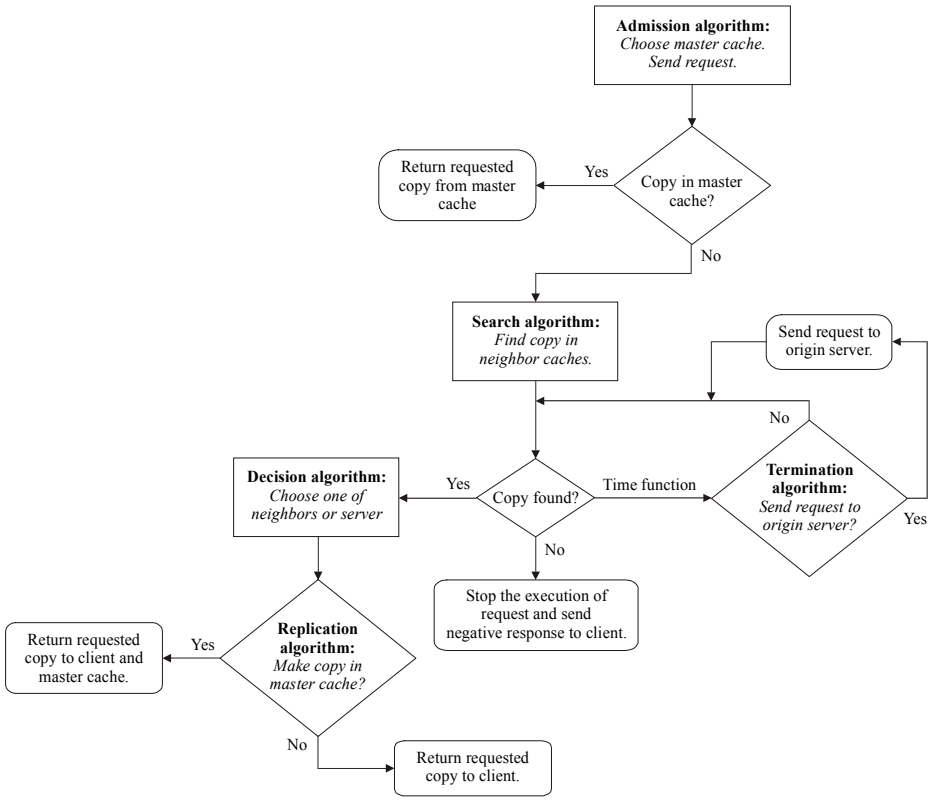


Figure 1: Distributed cache management (DCM) protocol

2. DCM PROTOCOLS

A distributed cache management protocol unifies cooperating caches to serve as a single cache that is distributed over multiple proxy machines. It can be understood as having five major algorithms: admission, search, decision, termination, and replication. The general flow diagram of DCM protocol is presented in Figure 1. The *admission algorithm* determines to which cache a client issues its request. The selected cache is known as the *master* cache. The *search algorithm* locates a valid copy in one or more of the other cooperating caches (the *neighbors* of the master cache) if the master cache does not have a copy of the object requested by the client. The *decision algorithm* selects, from among the neighbors holding a copy of the requested object, which neighbor should deliver the object. The *termination algorithm* halts the searching and preemptively forwards the request to the origin server. The *replication algorithm* decides if the master cache should retain a copy of the object.

We discuss several cache systems that implement DCM protocols, while the performance of two of them, *Squid* [2] and *Berkeley* [3], is evaluated and compared to the dbDCM protocol. These protocols were designed to be simple, stateless and to perform well in small configurations, where less than ten proxy machines cooperate.

Squid [2], and its predecessor *Harvest* [1], are widely deployed distributed caches whose design exemplifies the framework outlined above. Since both cache systems use similar algorithms, we briefly describe only the algorithms of the Squid DCM protocol. The *admission algorithm* is determined by the client and is trivial. For each client the master cache is fixed until the user manually reconfigures the browser.

If a Squid master cache does not itself contain a copy of the requested object, then it performs a logical broadcast[2] of *cache requests* to each of its neighbor caches. A cache request contains the URL of the desired object and the IP address of the issuing cache, in this case the IP address of the master cache. The *search algorithm* is very simple: the Squid master cache transmits a cache request to all of its known neighbors (we ignore the Squid access control mechanisms that can, in an URL-dependent manner, moderate the behavior of the Squid search algorithm). Upon receiving a cache request, each neighbor cache responds with a *hit* reply if the requested object resides in its cache and a *miss* reply otherwise.

Once all of the cache requests are issued, the *termination algorithm* is initiated. Squid waits a timeout period for a reply to a cache request. Within that time window, the Squid *decision algorithm* has control - the request is forwarded to the *first* neighbor cache that replies with a hit. At the expiration of the window, when either one or more miss replies were received, or no replies were received at all, the request is forwarded to the origin server.

Finally, the Squid *replication algorithm* is as straightforward as its companion algorithms - the master cache always retains a copy of the data object.

The Squid algorithms posses two important virtues: simplicity (hence easy implementation) and lack of states (hence scalability in some dimensions). However, the Squid algorithms were not designed to scale with respect to network traffic or

[2] A *logical broadcast* from host x to N hosts is a sequence of N UDP transmissions in which x transmits the same information to each of the N hosts. Note that the logical broadcast differs from the broadcast used at the network layer. For example, Ethernet broadcast assumes that only one packet is sent to all participating hosts connected to the same Ethernet subnet.

proxy load. A client request will generate 2*N* cache protocol network messages, where *N* is the number of neighbors of the client's master cache. One request message is generated to each of the *N* neighbors and one hit or miss reply is generated from each of the *N* neighbors. Squid administrators scale their cache meshes by structuring them hierarchically (as trees) and by limiting the number of neighbor caches at any one level. However, increasing the number of levels in the cache hierarchy also increases cache latency, therefore techniques that allow the size of any one level to increase gracefully are welcome.

One such approach is explored within the *Berkeley protocol* [3]. We summarize their approach using the "algorithm framework" that we applied to Squid. The Berkeley *admission* algorithm is more sophisticated than that of Squid, because a Berkeley client randomly chooses a cache from a set of cooperating proxies and sends the request to the selected *master* cache. If the master cache does not itself contain a copy of the requested object, the *search* algorithm multicasts[3] the request to other cooperating caches. Note that, because of IP multicast semantics, the master cache has no idea which proxy, or how many, will receive its request for assistance. In sharp distinction to the Squid protocol, a cooperating cache in the Berkeley protocol responds with a hit reply if the requested object resides in its cache and remains silent otherwise.

Once the cache request has been multicast, the *termination* algorithm is initiated. The master cache waits for a limited period for a reply from some cooperating cache. Within that window, the Berkeley *decision* algorithm redirects the client to the first cache that replies. The client, upon receipt of the redirect, reissues its request directly to a cooperating cache. At the expiration of the window, the master cache, not having received a reply from any neighbor, forwards the request to the origin server, and passes the returned object back to the client.

The Berkeley *replication* algorithm deals with two cases. If the master cache redirects the client to another proxy, it will never see, and consequently not replicate, the requested object for itself. If there is no redirection, then the master cache will retain a copy of the object it received from the origin server.

Although the Berkeley protocol potentially reduces network traffic by using IP multicasts for request dissemination, it does nothing to reduce the overall number of incoming requests to which a neighboring proxy cache must respond. In addition, the master cache must still cope with all of the cache responses. Since the master cache cannot easily determine the membership of the multicast group to which it sends the request, it cannot predict in advance a tight upper bound on the number of respondents. If nothing else, the caches are condemned, at the interrupt level, to process all of the requests and the responses that they receive - a task that consumes resources best put to other uses. From this perspective, the exploitation of IP multicasting by the Berkeley protocol is only a partial solution to the scaling problem, as IP multicasting reduces the amount of network traffic, but does not reduce proxy load in any way.

In order to reduce network traffic and proxy load, the CARP (Cache Array Routing Protocol) DCM protocol [7] does not implement the search algorithm, while the AWC (Adaptive Web Caching) DCM protocol [8] replaces the hierarchy of the cache groups with the self-organized cache groups. In the CARP protocol, each proxy machine is responsible for caching a determined set of objects, where the set is calculated from the URLs of the objects. Since the object can be cached only by one proxy machine determined uniquely by some hash function, this eliminates need for the search algorithm, which significantly reduces network traffic and proxy load imposed by this algorithm. However, this solution is not appropriate for large-scale distributed caches, because it is almost impossible to determine in advance an appropriate hash function that will provide satisfactory load balancing among the proxy machines. Moreover, the popular objects should be cached by more than one proxy machine in order to serve large number of requests[4].

Although the self-organization proposed by the AWC protocol is an appropriate way to resolve the scaling problem on coarse grain structures like cache groups, there still remains the scaling problem within the cache group. Next section describes our solutions of scaling problem for large distrusted cache.

3. DIRECTORY-BASED DCM PROTOCOLS

Our principal goal was to design a distributed cache system for a medium to large Internet service provider with tens of proxy machines and thousands of clients per proxy machine [9]. Since other DCM protocols were not designed to scale with respect to network traffic and proxy load, we decided to design a new DCM protocol, called the directory-based DCM protocol. In order to reduce both the network traffic and the proxy load, our DCM protocol uses a custom-designed distributed directory. The directory information allows us to build scalable search algorithms based on a directory-limited unicast. Detailed description of the dbDCM protocol is given in [4, 5], while in this paper we provide a brief description of the dbDCM protocol and describe the performance evaluation results.

Designing Internet-Scale Directories

Since the fundamental design issue is the scalability of a directory and not its precision, the directory is (1) *distributed across all of the neighbors*, (2) it is *structured like a cache*, (3) it is *never locked*, and (4) it *never issues acknowledgements*. Since each neighbor maintains a distinct portion of the directory, the total size of the directory scales linearly with the number of proxies. Since caches are by definition and design incomplete, structuring the directory like a cache further reduces the per neighbor cost of the directory. Forgoing locks permits simultaneous reads and updates of the directory at the cost of an occasional cache miss. Eschewing acknowledgements minimizes the traffic overhead and response time.

Let a set $P=\{p_1, p_2, \ldots, p_N\}$ of proxies be given, where the proxies $p_2, \ldots, p_N$ are all neighbors of p_1. Let $O=\{o_1, o_2, \ldots\}$ be the set of all data objects that are uniquely denoted by their URLs. A directory is a partial mapping $M : O \rightarrow Pr(P)$, where $M(u) = \{p_{j1}, \ldots, p_{jm}\}$ is the set of proxies that might have a copy of data object o denoted by URL u. Hereafter we assume that object o has URL u. Each proxy p maintains a directory M_p and $\bigcup_{p \in P} \mathrm{dom}(M_p) \subseteq O$, where $dom(M_p)$ denotes the domain of function M_p.

Directory scaling is improved by ensuring that for any two distinct proxies p and q $dom(M_p) \cap dom(M_q)$ is small. In other

[3] A *multicast* from host x to a multicast group of N hosts requires that x issues just one packet which is then replicated as required by intermediate routers for each of the $1 \le D \le N$ distinct subnets on which N hosts appear [6]. The total number of UDP packets required for the multicast is D and, in many cases (depending upon the network topology), $D \ll N$. However, due to deployment issues with multicast servers, the systems usually use *logical multicast* that actually sends N unicast messages.

[4] These are not the issues with our dbDCM protocol. The hash function is not used to distribute Web objects, but to distribute contents of the directory. Also, Web objects are distributed according to client requests allowing each Web object to be cached by more than one proxy.

words, two distinct proxies maintain comparatively distinct portions of the directory. Finally, all of the proxies $p \in P$ share a common total hash function $H : O \rightarrow P$ that maps URLs to proxies. This shared hash function allows us to distribute the directory entries among multiple proxy hosts. By computing $y=H(u)$, host x can discover that neighbor proxy y maintains the directory entry for the object whose URL is u.

Each directory M_p is implemented as a bucket hash of key/value pairs, where the keys are URLs and the values are finite sets of proxies. Given an URL u, $M_p(u) = \{p_{j1}, \ldots, p_{jk}\}$ signifies to proxy p that data object u is replicated at proxies $p_{j1}, \ldots, p_{jk}$. A fixed amount of memory is allocated for directory M_p, which effectively limits the total number of key/value pairs contained in M_p at any one time. If the memory space for M_p is exhausted, then one or more key/value pairs will be flushed from the directory in order to make room for new updates. Any one of several standard replacement algorithms, such as Least Recently Used or Least Frequently Used, can be used to select candidate key/value pairs for flushing.

When a proxy p receives a request, it first consults its own cache for the requested object u. Failing that, it computes $H(u)=q$ and passes the directory lookup request for u onto proxy q, where q could be p itself. At this point, q can update its directory $M_q(u) \leftarrow M_q(u) \cup \{p\}$ (the *optimistic* approach) assuming that proxy p will obtain a copy of object u. If p fails to obtain u, it must request q to delete p from $M_q(u)$, that is $M_q(u) \leftarrow M_q(u) / \{p\}$. In the *pessimistic* approach, only when p has u in hand it notifies q, which sets $M_q(u) \leftarrow M_q(u) \cup \{p\}$. When the probability of satisfying a request is high, the optimistic approach is preferable as it reduces network traffic. If the probability is low, the pessimistic approach is preferred as it ensures higher accuracy for directory entries. In either approach, p will request q to delete p from $M_q(u)$ whenever p ejects u from its cache and p will request q to add it to $M_q(u)$ if it receives object u directly from the origin server.

Later on, independently of our work, other authors have also used a directory to reduce the network traffic and proxy load in a distributed cache. However, they did not take into the account all four features listed at the beginning of this section that allow for the design of the large-scale distributed cache. The scalability of the RPD-SD protocol [10] is limited by a centralized directory manager implemented on one server machine and the scalability of DCM protocols described in [11, 12] is limited by a fully replicated directory at each proxy machine.

Implementation of Cache Management Protocol

We now revisit the search algorithm, this time incorporating directories to reduce the network traffic and proxy load. Since the dbDCM protocol is implemented on top of the Squid distributed cache, the remaining algorithms are identical to the algorithms of the Squid DCM protocol.

First, by narrowing the set of candidate proxy caches likely to possess a fresh copy of a desired data, object directories can reduce network traffic. Second, requests are directed toward just those proxy caches most likely to respond positively, thereby reducing the overall workload for any proxy host. To illustrate this, assume that the master proxy cache *mc* receives a request for a data object o denoted by URL u. Assuming that *mc*'s cache does not contain o, let $x=H(u)$ be the proxy identified by *mc* (x could be *mc* itself) as the proxy responsible for maintaining a directory entry for u. That is, x's directory contains a mapping $D_x(u)=\{nc_1, \ldots, nc_m\}$ - the set of all neighbor proxies that contain a fresh copy of o. At this point in the search algorithm two variants are possible: (1) x returns the set of neighbor proxies to *mc* and *mc* completes the execution of the search algorithm or; (2) x executes the remainder of the search algorithm on *mc*'s behalf. In either case let α, a small positive integer, be given. Of the set of candidate neighbor proxies $\{nc_1, \ldots, nc_m\}$, $A=min(\alpha, ||D_x(u)||)$ of them will be selected and each so selected will be sent a cache request containing u (the URL of the desired object) and the IP address of the requesting proxy *mc*. As in the Squid algorithm, each neighbor proxy nc_i replies to *mc* with a hit if its cache contains o, and a miss otherwise. In the rest of the paper, we evaluate variant (1).

Variant (1) of the search algorithm requires that an IP packet be transmitted from x to *mc* containing the set of candidate proxies $\{nc_1, \ldots, nc_m\}$, so that *mc* can select candidates and issue cache requests. Variant (2) eliminates the transmittal by having x issue cache requests to the candidate proxies. Note that in both cases, the cache requests, whether issued by *mc* or x, specify *mc* as the requesting proxy. Consequently, the replying neighbor proxies always direct their replies to *mc* although, in variant (2) proxy x issues the requests.

Parameter α may be one, equal to $||D_x(u)||$, or it may have some constant value. Whenever α is less than $||D_x(u)||$, the proxy must decide to which of the proxies from the directory list it will send cache requests. Either the proxy uses a Round Robin method to advantageously distribute the load among the proxies on the list, or, the proxy randomly selects α proxies to send the request to.

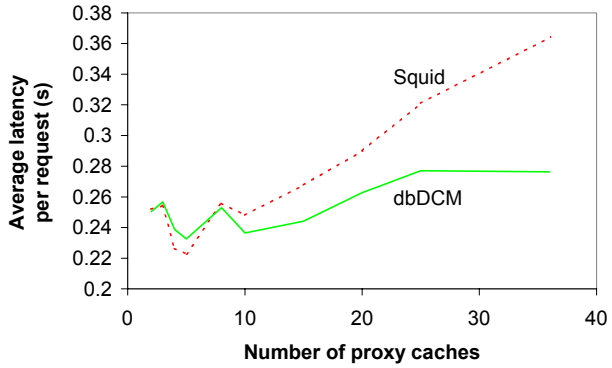
The parameter α set to one is appropriate for a highly reliable environment, because it reduces the network traffic and proxy load. However, if the response is a miss, or the directory is inaccurate, the repetitions of the cache request may increase the network latency. Choosing $\alpha>1$ alleviates these problems for an unreliable environment. For highly unreliable networks, the best solution is to have the value of α equal to $||D_x(u)||$.

In order to implement the dbDCM protocol [4, 5], we extend the Internet Cache Protocol (ICP) [13, 14] by adding the following messages: *ICP_DIR_GET* to send the directory lookup request from the requesting proxy to the hashed proxy, *ICP_DIR_DATA* to send back the directory information from the hashed proxy to the requesting proxy, *ICP_DIR_DEL* to remove the requesting proxy from a given directory entry, and *ICP_DIR_ADD* to add the requesting proxy to a given directory entry. These four messages comprise the complete mechanism for retrieving information from the directory and maintaining its accuracy.

4. COMPARISON OF MEASUREMENT RESULTS

For the purpose of measurement, we use three Sun Ultra 10 machines, four Sun SparcStation 20 systems and one personal computer (PC) based on Intel Pentium III 500. All machines are connected by a 100 Mb/s Ethernet local area network. We install the Apache Web server on one of the Ultra 10 machines, while the remaining six Sun machines run multiple copies of the proxy caches. We use the PC to run the multithreaded client software.

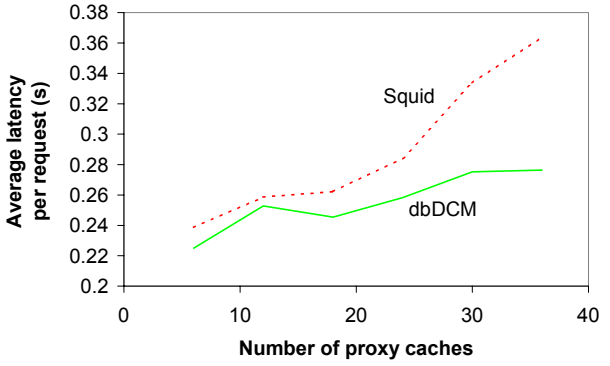
The test program runs a configurable number of client threads and the main, scheduler thread. Each client thread sends a request to the proxy, waits for the response and measures the time it took to complete the entire request. The scheduler thread controls the client threads and distributes the requests among them. As soon as the client thread finishes the request and becomes available, it will be assigned new request by the scheduler thread. A small number of our client threads create the load that is comparable to the much larger number of Web clients. For example, 10 client threads might represent as much as 1,000 Web clients. During the tests, we vary the number of proxy caches, the number of simultaneous client threads, and the Web object size. In order to increase the number of proxy



Number of machines	2	3	4	5	4
Number of proxy caches per machine	1	1	1	1	2
Total number of proxy caches	2	3	4	5	8

Number of machines	5	5	5	5	6
Number of proxy caches per machine	2	3	4	5	6
Total number of proxy caches	10	15	20	25	36

Figure 2: Average latency for an increasing number of proxy machines and an increasing number of proxy caches per machine



No. of machines	1	2	3	4	5	6
No. of proxy caches per machine	6	6	6	6	6	6
Total number of proxy caches	6	12	18	24	30	36

Figure 3: Average latency for variable number of proxy machines and six proxy caches per machine

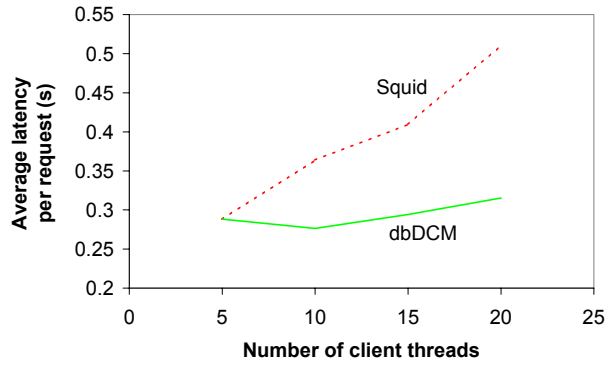


Figure 4: Average latency for variable number of client threads

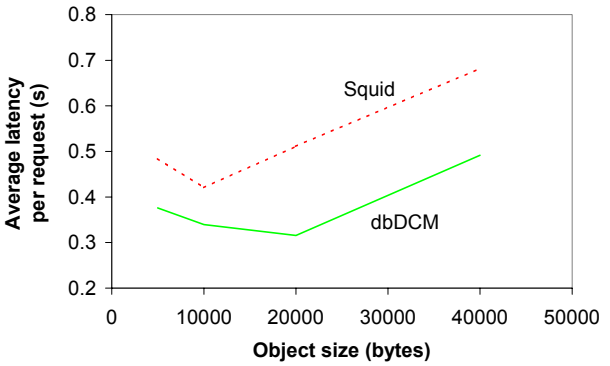


Figure 5: Average latency for variable Web object size

caches, multiple copies of the proxy cache processes run on each machine. We run up to six proxy caches per machine, giving us total of 36 cooperating proxy caches. As a performance measure, we choose the average latency per client request.

Figure 2 presents the results of measurements for increasing number of proxy machines and increasing number of proxy caches per machine. The Web object size is 10 kB and ten client threads simultaneously request the objects. For a small number of proxy caches, the dbDCM protocol shows slightly higher latency than the Squid DCM protocol, which is imposed by the directory lookup request that is sent from one proxy machine to another[5]. In addition, the directory-limited unicast does not significantly reduce the network traffic and proxy load for small number of proxy machines. However, if the number of proxy caches increases over ten, the dbDCM protocol shows lower latency than the Squid protocol. The directory-limited unicast reduces the network traffic and proxy load in comparison to the logical broadcast performed by the Squid protocol. The difference between the Squid protocol and the dbDCM protocol increases, reaching the value of 25% for 36 proxy caches. As we can see from Figure 2, the slope of the curve for the dbDCM protocol is lower than the slope of the curve for the Squid protocol. Moreover, if the number of proxy machines increases over 30, the latency for the dbDCM protocol does not increase any more, because of directory-limited unicast.

As Figure 3 shows, similar results are obtained with a constant number of proxy caches per machine. Each test machine runs six proxy caches. As previously discussed, the slope of the curve for the Squid protocol is higher than for the dbDCM protocol, while the directory-limited unicast straightens the slope for the number of machines greater than 30.

If we increase the number of client threads, the difference between the Squid protocol and the dbDCM protocol significantly increases. Figure 4 shows the measurement results achieved with 36 proxy caches and an object size of 20 kB. While the slope of the curve for the Squid protocol significantly increases with the number of client threads, the slope of the curve for the dbDCM protocol shows a very slow increase. For 20 simultaneous client threads, the average latency of the dbDCM protocol is 40% lower than the latency of the Squid protocol.

[5] As we expect, the average latency per request decreases if the same amount of requests is distributed among larger number of proxy machines (for example, if the number of proxy caches increases from two to five, the number of proxy machines increases from two to five, and the average latency decreases). However, the average latency per request increases if the same amount of requests is performed by a lower number of proxy machines (for example, if the number of proxy caches increases from five to eight, the number of proxy machines is reduced from five to four, and the average latency increases).

Figure 5 shows the results of measurements with variable Web object size. Since we can use the artificial load as well, it is possible to change the Web object size in a simple way. The measurement results in Figure 5 are achieved with 36 proxy caches and 20 client threads that simultaneously request the objects.

As expected, the right portion of both curves shows the constant growth due to the increasing size of the object. The latency is mostly affected by the data transfer of large objects. However, the latency in the left portion of the graph in Figure 5 is significantly affected by the small communication packets incurred by the proxy-to-proxy communication. Since the requests for smaller objects increase the number of small communication packets and the communication with small packets cannot utilize the full bandwidth of the network [15, 16], the latency in the left portion decreases as the objects become larger.

As presented in Figure 5, the dbDCM protocol shows better results than Squid DCM protocol for all object sizes. Both the Squid and the dbDCM protocols have the same rate of increase of the average latency per request for large objects. However, as the object size becomes smaller, the latency of Squid grows faster than the latency of the dbDCM protocol. Since the minimum latency of the dbDCM protocol is located more to the right than the minimum latency of Squid DCM protocol, the optimum object size for the dbDCM protocol is larger than the optimum size for Squid.

5. COMPARISON OF ANALYTICAL RESULTS

We developed sophisticated analytical models for comparing competing DCM protocols [17]. Our models are intended to capture *first-order differences* among these protocols for *large-scale distributed caches*. The *average latency per request* for each of the three DCM protocols is calculated based on the analytical expressions for the following parameters: network traffic, average host load, average client load, average network latency, average host latency, and average client latency. The analytical expression for the average latency per request is derived in three main steps. *In the first step*, we precisely analyze the communication on the IP packet level for the master cache hit, the neighbor cache hit, and the distributed cache miss. Based on this analysis, we derive the analytical expressions for network traffic, average host load, and average client load for each of the three DCM protocols. *In the second step*, we incorporate the analytical expressions for the network traffic,

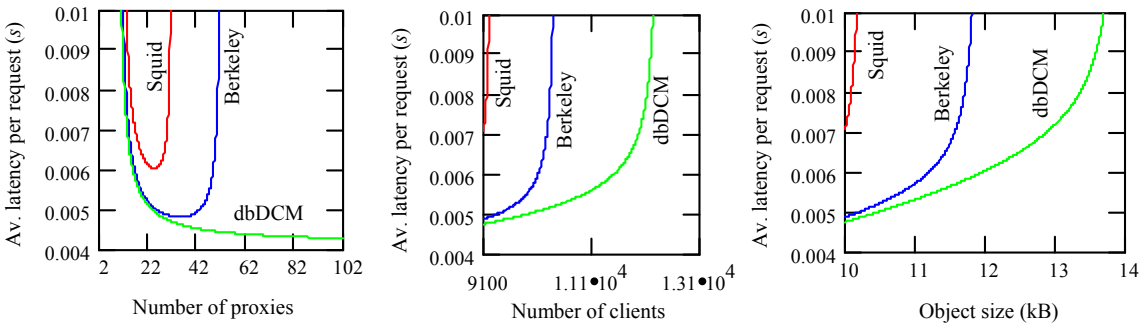
Distributed Cache Parameters	***Value***
Total number of proxy hosts in the distributed cache	$N = 30$
Average cross-sectional bandwidth of the network	1 Gb/s
Average bandwidth of proxy host	100 Mb/s

Communication Protocol Parameter	***Value***
Size of a small IP packet	40 B
Size of a medium IP packet	180 B
Size of a large IP packet	$L = 1024$ B

Application Parameters	***Value***
Total number of client hosts	9100
Average number of requests per client per second	0.53
Average bandwidth of client	10 Mb/s
Master cache hit rate	0.25
Neighbor cache hit rate	0.30
Average size of Web object	$O = 10$ kB
Total number of acknowledgement packets to IP data packets	$max(1, 0.75(O/L))$
Percentage of data packets that cause the TCP window to fill	10%
Total number of server hosts per distributed cache	3
Average bandwidth of a server host	100 Mb/s

DCM Protocol Parameter	***Value***
Number of subnets that connect the proxy hosts	$max(1, 0.20\ N)$
Number of the positive responses from neighbor proxies	$max(1, 0.30(N\text{-}1))$
Maximum number of dbDCM neighbor caches to which a master cache will broadcast a request for a Web object	$min(5, N)$
Probability that a dbDCM directory entry is accurate	0.75
Probability that a dbDCM master cache does not hold the particular directory entry	$(N\text{-}1) / N$
Timeout period used by termination algorithm	40 ms

Table 1: Analytical model parameters



(*a*) Distributed cache size **(*b*)** Clients **(*c*)** Object

Figure 6: Analytical performance comparison of DCM protocols for large-scale distributed caches

average host load, and average client load, into the M/G/1 queuing model [18] in order to obtain the analytical expressions for average network latency, average host latency, and average client latency. *In the third step*, we analyze timing diagrams that describe the propagation of the IP packets during the master cache hit, the neighbor cache hit, and the distributed cache miss. Finally, based on the timing analysis and on the analytical expressions for average network latency, average host latency, and average client latency, we derive the analytical expressions for the average latency per request.

The parameters included in the model are listed in Table 1. Table 1 also presents the values of the parameters we use to generate the results presented in this paper, except the parameters we explicitly vary in a given diagram.

Figures 6*a-c* show how the average latency per request varies with three parameters: distributed cache size, number of clients, and Web object size. The graph in Figure 6*a* shows the average latency per request for a variable number of proxy hosts, while all other parameters have fixed values that are given in Table 1. At lower numbers of proxy machines (see Figure 6*a*), the DCM protocols saturate because of the overloaded proxy hosts. At higher numbers of proxy machines, the Berkeley and Squid protocols saturate because of network congestion, while the dbDCM protocol does not saturate due to the directory-limited unicast.

The other graphs show the average latency per request for variable values of two other parameters, while keeping the number of proxy host machines fixed (N=30). From the graphs, it is clear that the dbDCM protocol contributes to the slowest increase in latency per request for both parameters: increasing number of clients and increasing Web object size. Additionally, the dbDCM protocol performs adequately (does not saturate) over a wider range of parameter values than either of the other two DCM protocols.

6. CONCLUSION

The results in this paper indicate that the dbDCM protocol scales more gracefully than the Berkeley and the Squid protocol. If either the number of proxy caches, the number of clients, or the object size is increased, the dbDCM protocol has a lower increase of average latency per request. We show by measurement that the difference between the DCM protocols can be over 40%. Moreover, the directory-limited unicast limits the amount of generated network traffic and proxy load. This keeps the average latency per request constant regardless of the number of clients and regardless of the number of proxy caches. In contrast, the Squid and Berkeley protocols saturate if the number of proxy machines or the number of clients becomes large; this extremely increases the average latency per request and prevents the distributed cache from operating normally. In addition, our comparison of analytical results shows that the dbDCM protocol scales better than the two other protocols over a broader range of parameter values.

7. ACKNOWLEDGMENT

The authors wish to thank AT&T Corporation for supporting this work, and Z. G. Vranesic, A. Grbic, I. Skuliber and K. Kelley for providing valuable comments and editorial help.

8. REFERENCES

[1] "**Harvest: A Scalable, Customizable Discovery and Access System**", *Technical Report* CU-CS-732-94, Department of Computer Science, University of Colorado - Boulder, Revised March 1995.

[2] D. Wessels: "**The Squid Internet Object Cache**", *National Laboratory for Applied Network Research*, http://squid.nlanr.net/Squid

[3] R. Malpani, J. Lorch and D. Berger: "**Making World Wide Web Caching Servers Cooperate**", *World Wide Web Journal*, Vol. I, Issue 1, Winter 1996.

[4] S. Srbljic, P.P. Dutta, T.B. London, D.F. Vrsalovic, and J.J. Chiang: "**Scalable Distributed Caching System and Method**", *United States Patent* 5,933,849, issued August 3rd, 1999, filed April 10th, 1997, App. No. US1997000827763, http://www.uspto.gov; also submitted to European Patent Office and Mexico Patent Office

[5] S. Srbljic, P.P. Dutta, T.B. London, D.F. Vrsalovic, and J.J. Chiang: " **Scalable network object caching**", *United States Patent* 6,154,811, issued November 28th, 2000, filed December 4th, 1998, App. No. US1998000204343, http://www.uspto.gov

[6] S. Deering: "**RFC 1054: host extensions for IP multicasting**", May 1988.

[7] V. Vallopillil and K. W. Ross: "**Internet Draft: Cache Array Routing Protocol v1.0**", February 1998. (http://www.web-cache.com/Writings/Internet-Drafts/draft-vinod-carp-v1-03.txt)

[8] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson: "**Adaptive Web Caching: Towards a New Global Caching Architecture**", 3rd International WWW Caching Workshop, June 1998.

[9] G. Vanecek, N. Mihai, N. Vidovic, and D. Vrsalovic: "**Enabling Hybrid Services in Emerging Data Networks**", *IEEE Communication Magazine*, July 1999, pp. 102-109.

[10] S. Gadde, J. Chase, and M. Rabinovich: "**Directory Structures for Scalable Internet Caches**", *Technical Report CS-1997-18*, Department of Computer Science, Duke University, November 1997. (http://www.research.att.com/~misha/crisp/distrProxy/lazyCrisp.ps.gz)

[11] L. Fan, P. Cao, J. Almeida, and A. Z. Broder: "**Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol**", *Proceedings of the ACM SIGCOMM '98*, Vancouver – Canada, September 1998.

[12] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay: "**Design Considerations for Distributed Caching on the Internet**", *Proceedings of International Conference on Distributed Computing Systems*, Austin - Texas, May 1999.

[13] D. Wessels and K. Claffy: "**Internet Cache Protocol (ICP), version 2**", RFC-2186, September 1997.

[14] D. Wessels and K. Claffy: "**Application of Internet Cache Protocol (ICP), version 2**", RFC-2187, September 1997.

[15] A. Milanovic, S. Srbljic and J. Radej: "**Performance of Distributed Systems Based on Ethernet and Personal Computers**", *Proceedings of the IEEE International Symposium on Industrial Electronics*, Bled - Slovenia, 1999, vol. 1, pp. 79-83

[16] A. Milanovic, S. Srbljic and V. Sruk: "**Performance of UDP and TCP Communication on Personal Computers**", *Proceedings of the 10th Mediterranean Electrotechnical Conference, Volume I, Regional Communication and Information Technology*, Lemesos, Cyprus, May 29-31, 2000, pp. 286-289

[17] S. Srbljic and A. Milanovic, "**Analytical Prediction of Performance for Distributed Cache Management Protocols**", *Technical Report*, School of Electrical Engineering and Computing, Department of Electronics, Microelectronics, Computer and Intelligent Systems, Zagreb, Croatia, May 1998.

[18] L. Kleinrock: "**Queuing Systems; Volume I: Theory**", *John Wiley & Sons, Inc*, 1975.