Reducing Outgoing Traffic of Proxy Cache by Using Client-Cluster

Kyungbaek Kim and Daeyeon Park

Abstract: **Many web cache systems and policies concerning them have been proposed. These studies, however, consider large objects less useful than small objects in terms of performance, and evict them as soon as possible. Even if this approach increases the hit rate, the byte hit rate decreases and the connections occurring over congested links to outside networks waste more bandwidth in obtaining large objects. This paper puts forth a client-cluster approach for improving the web cache system. The client-cluster is composed of the residual resources of clients and utilizes them as exclusive storage for large objects. This proposed system achieves not only a high hit rate but also a high byte hit rate, while reducing outgoing traffic. The distributed hash table (DHT) based peer-to-peer lookup protocol is utilized to manage the client-cluster. With the natural characteristics of this protocol, the proposed system with the client-cluster is self-organizing, fault-tolerant, well-balanced, and scalable. Additionally, the large objects are managed with an index based allocation method, which balances the loads of all clients well. The performance of the cache system is examined via a trace driven simulation and an effective enhancement of the proxy cache performance is demonstrated.**

Index Terms: **Client-cluster, peer-to-peer, replacement algorithm, web caching.**

I. INTRODUCTION

The recent increase in popularity of the World Wide Web has led to a considerable increase in the amount of Internet traffic. As a result, web caching has become an increasingly important issue. Web caching aims to reduce network traffic, server loads, and user-perceived retrieval delays by replicating popular content on caches strategically placed within the network. Web caches are often deployed by institutions (corporations, universities, and ISPs) to reduce traffic on access links between the institution and its upstream ISP.

By caching requests for a group of users, a web proxy cache can quickly return objects previously accessed by other clients. As a proxy cache handles a greater number of requests by clients, it can further reduce network traffic and response delays. Though a proxy cache tries to handle as many requests as possible, the storage of a proxy cache is limited and it can not store all of requested objects in its local storage. If a proxy cache is full and needs space for new objects, it evicts the other objects which are not useful for cache performance; this is the replacement policy of a web proxy cache [1]–[5]. Generally, these policies evict least recently used objects and large objects first. According to this behavior, a proxy cache obtains more small objects, achieves a higher hit rate and reduces more access links between an institution and its upstream ISP.

A proxy cache with these policies either stores large objects for little time or does not obtain them. This means that it has less of a chance to further reduce the outgoing traffic for the large objects. Due to this, though these policies increase the hit rate, they decrease the byte hit rate, which is the number of bytes that hit in the proxy cache as a percentage of the total number of bytes requested. This degradation in the byte hit rate causes more outgoing traffic of a proxy cache even if it handles more requests for small objects. As is the case for the web proxy cache, it is assumed that an intra-community file transfers occur at relatively faster rates, whereas file transfers into the community occur at relatively slower rates. As an example, the community may be a university or corporate campus, with tens of thousands of peers in the campus community interconnected by high speed LANs, but with connections to the outside world occurring over congested campus access links. Consequently, if the proxy cache wastes more outgoing traffic, the connections to the outside become more congested and the response delay for a request increases.

To prevent this degradation, it is necessary to store large objects and maximize chances of hitting these objects. As a naive approach, a proxy cache simply increases its local storage. However, this approach is only a temporary solution and continues to be affected by the general replacement policies. For this reason, exclusive storage for large objects is needed. The content delivery network (CDN) services can be utilized for this purpose; however, these services are expensive. Moreover, these approaches incur high administrative costs owing to the the frequent variation of clients. For example, a growth in client population necessitates increasing the storage and updating the system information.

In this paper, a new web proxy cache system that uses the residual resources of clients is suggested. Essentially, a web proxy cache stores only small objects, and the resources of clients are used to store only large objects. This separation of storage causes a proxy cache to store a greater number of small objects, as it does not need to store any large objects. The large objects are stored in an exclusive storage area which is supplied by clients; this is termed the *client-cluster*. According to this action, a proxy cache maintains or improves its performance, through such parameters as the hit rate, as well as the byte hit rate, the response delay and the usage of the outgoing bandwidth. Furthermore, the size of the exclusive storage increases as more clients use the proxy cache, reducing the administrative cost and allowing the proxy cache to be more scalable.

The client-cluster is composed of the client's residual re-

Manuscript received November 21, 2003; approved for publication by Edward Chow, Division III Editor, April 5, 2006.

The authors are with the Department of Electrical Engineering and Computer Science, the division of Electrical Engineering, KAIST, Daejeon, Republic of Korea, email: kbkim@sslab.kaist.ac.kr, daeyeon@ee.kaist.ac.kr.

sources. As clients join and leave dynamically, in order to use its storage efficiently, the client-cluster must be self-organizing and fault tolerant and the load of each client should be balanced. To cope with these requirements, the client-cluster is managed using the distributed hash table (DHT) based peer-to-peer protocol. By using this protocol, all clients receive a nearly identical load, as the hash function balances the load with a high probability. Moreover, the proxy cache does not need to manage information about these clients and administrative costs can be saved.

This protocol matches an object with a client. When storing large objects in the client-cluster, it is difficult, as well as unfair to a single client, for that client to store the entire large object. Therefore, the large object is then broken into many smaller blocks and these blocks are stored with many clients by using the index based allocation method. All of the blocks are distributed in the client-cluster and the storage overhead for each client is reduced as well as balanced. However, when a proxy cache sends requests to a client-cluster and the requested objects are not stored in that client-cluster, the proxy cache takes on additional latency. To prevent this latency, a cache summary with a Bloom filter is used, which determines whether the requested objects are in the client-cluster.

This paper is organized as follows. In Section II, we describe web caching and peer-to-peer lookup algorithm briefly. Section III introduces the detail of the client-cluster storing method for large objects. The simulation environment and the performance evaluation are given in Section IV. We mention other related works in Section V. Finally, the study is concluded in Section VI.

II. BACKGROUND

A. Web Caching and Replacement Policy

The basic operation of web caching is simple. Web browsers generate HTTP GET requests for Internet objects such as HTML pages, images, or mp3 files. These are serviced from a local web browser cache, web proxy caches, or an original content server—depending on which cache contains a copy of the object. If a cache closer to the client has a copy of the requested object, bandwidth consumption is reduced and network traffic decreases. Hence, the cache hit rate and byte hit rate should be maximized and the miss penalty, which is the cost when a miss occurs, should be minimized when designing a web cache system.

If a web cache has infinite storage, caching objects poses few problems, and a web cache achieves the maximum of hit rate and byte hit rate. A web cache, however, has size-limited storage and if a cache needs space for new objects, it evicts the other cached objects which are deemed not useful for cache performance. In this case, the policy of selecting object is a replacement policy.

Many replacement policies have been proposed [1]–[5]; generally these evict large objects first when new objects enter. A number of these policies do not store large objects at all. As a result of these policies, a web cache stores more objects, the hit rate increases and the number of access links between the institution and its upstream ISP decreases. However, because these policies evict large objects earliest, the large objects are not cached for extended periods of time, and there is little chance for these objects to be hit in the cache. Although a web cache can achieve a high hit rate with these policies, it can not achieve high byte hit rate, thus wasting further upstream bandwidth while retrieving large objects. If the requests of large objects increase, this degradation can be extreme.

Even if local storage for the web cache increases due to the caching of large objects, it is only a temporary solution, requiring additional costs related to cache management, such as new hardware and reconfigurations. In other words, this approach is not viable; nor is it scalable.

B. Peer-to-Peer Lookup

Peer-to-peer systems are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality; this includes redundant storage, selection of nearby servers, anonymity, search, and hierarchical naming. Among these features, lookup for data is an essential functionality for peer-to-peer systems.

A number of peer-to-peer lookup protocols have been recently proposed, including Pastry, Chord, CAN, and Tapestry [6]–[9]. In a self-organizing and decentralized manner, these protocols provide a DHT that reliably maps a given object key to a unique live node in the network. Because DHT is made by a hash function that balances load with high probability, each live node has the same responsibility for data storage and query load. If a node wants to find an object, a node simply sends a query with the object key corresponding to the object to the selected node determined by the DHT. Typically, the length of routing is about $O(\log n)$, where n is the number of nodes. According to these properties, peer-to-peer systems balance storage and query load, transparently tolerate node failures and provide efficient routing of queries.

III. PROPOSED IDEA

A. Overview

As was described in the previous section, a web proxy cache evicts large objects first in order to obtain free space, which is then used to store a new cached object. This feature reduces cache performance, especially the byte hit rate. Accordingly, large objects are the main obstacles of cache performance. To address this, the residual resources of clients for a proxy cache are exploited. To be precise, any client who wants to use the proxy cache provides small resources; instead of this being the local storage of the user's browser caches, it is what is known as residual storage. The proxy cache uses these additional resources to store large objects. This group of client residual resources is termed a *client-cluster*.

The client-cluster is utilized as exclusive storage for large objects. Generally a web proxy cache stores all requested objects in the local storage, but in the proposed system, local storage by the proxy cache involves only small objects while large objects are stored in the client-cluster which is distributed among the clients. From another vantage point, the size of a requested object can be said to decide its cache location. If the size is small, the proxy cache assign the object to its local cache, otherwise,

it allocates the object to the client-cluster. When a proxy cache receives a request, it checks its local storage. If a hit occurs, it returns the requested object; otherwise, it sends a lookup message to the client-cluster and this message is forwarded to the client having the responsibility to store the object. By maintaining these activities, the proxy cache system can store a greater number of large objects for relatively a longer time without incurring a high cost, making it possible to achieve the high byte hit rate and further save outgoing bandwidth. Moreover, as the proxy cache does not need to process large objects, it can store additional small objects; thus, the hit rate of the system also increases.

B. Management of Client-Cluster

In the proposed scheme, a proxy cache uses the resources of clients that are in the same network. Generally, if a peer wants to use the resources of other peers, the requesting peer should have information about the others. This approach is viable when these other peers are available and reliable. However, the client membership can be very large and it changes dynamically. If the proxy cache manages the states of all clients, too much overhead can be created to effectively manage the client information, and complex problems such as fault-tolerance, consistency, and scalability can arise. In consideration of these issues, in this study, the proxy cache is established such that it has no information about the clients. Furthermore, the client-cluster manages itself.

The client-cluster is designed using DHT-based peer-to-peer protocol [6], [7]. To use this protocol, each client needs an application called *Station*. *Station* is not a browser or a browser cache, but is instead a management program enables the use of client resources for the proxy cache. A client can not use the resources of a Station directly; a proxy cache sends requests issued from clients to Stations in order to use the resources of a client-cluster. When a Station receives a request from a proxy cache, it forwards the request to another Station or verifies the presence or absence of the requested objects. Each Station has a unique node key and a DHT. The unique node key is generated by computing the SHA-1 hash of a client identifier, such as an IP address or an Ethernet address, and the object key is obtained by computing the SHA-1 of the corresponding URL. The DHT describes the mapping of the object keys to responsible live node keys for the efficient routing of request queries. This is similar to a routing table in a network router. A Station uses this table with the key of the requested object to forward the request to the next Station. Additionally, the DHT of a Station has the keys of *neighbor Stations* that are numerically close to the Station, similar to the leaf nodes in PASTRY or the successor list in CHORD.

The basic operation of the lookup in a client-cluster is shown in Fig. 1. When a proxy cache sends a request query to one Station of a client-cluster, the Station obtains the object key of the requested object and selects the next Station according to the DHT and the object key. Finally, the *home Station*, a Station having the numerically closest node key to the requested object key among all live nodes at this point, receives the request and verifies the presence or absence of the object in the local cache. If a hit occurs, the home Station returns the object to the proxy cache; otherwise, it returns only a null object. In

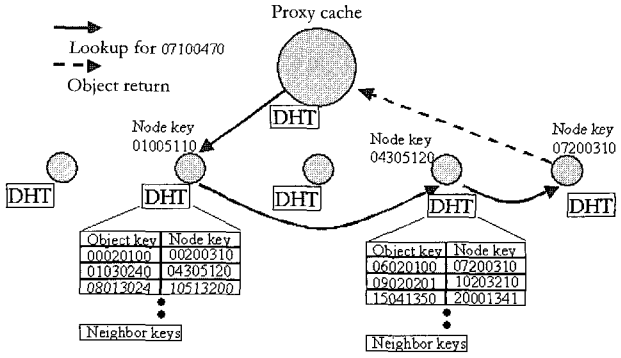


Fig. 1. Basic lookup operation in the client-cluster. In this figure, total hop count is 3 for an object.

Fig. 1, the node whose key is 07200310 is the home Station for the object whose key is 07100470. The cost of this operation is typically $O(\log n)$, where n is the total number of Stations. If 1,000 Stations exist, the cost of the lookup is 3 (approximately), and if there are 100,000 Stations, the cost is nearly 5. As the RTT for any server on the Internet from one client is 10 or 100 times larger than that for another client in the same network, the latency for an object can be reduced by 2 or 20 times when the object is obtained from the client-cluster.

The client-cluster can cope with frequent variations in client membership by using this protocol. Though clients dynamically join and leave, the lazy update that manages the small amount of information used by membership changes does not interfere with the lookup operation of this protocol. When a Station joins the client-cluster, it sends a join message to any one Station in the client-cluster and receives new DHT, and other Stations update their DHT for the new Station slowly. Conversely, when a Station leaves or fails, other Stations having a DHT mapping with the departing Station slowly detect the failure of the departed station and repair their DHT. According to these procedures, the client-cluster is self-organizing and fault-tolerant.

All Stations have approximately the same amount of objects, as the DHT used for the lookup operation provides a degree of natural load balance. Moreover, the object range, which is managed by one Station, is determined by the number of live nodes. That is, if there are few live nodes, the object range is large; otherwise, it is small. Due to this, when the client membership changes, the object range is resized automatically and the home Stations for every object is changed implicitly.

C. Storage for Large Sized Objects

In the proposed system, large objects are stored in the client-cluster. Essentially, the client-cluster stores the object in the corresponding node which has the closest node key numerically to the object key. However, each node in the client-cluster supports residual resources that are not used by a node and that are too small to facilitate the storing of the whole large object. To solve this problem, the large object is broken into many small blocks and these blocks are stored at many nodes. Each block has a block key obtained by hashing the block itself, and the home node that has the closest node key numerically to the block key stores the block. According to this, all of blocks for a large ob-

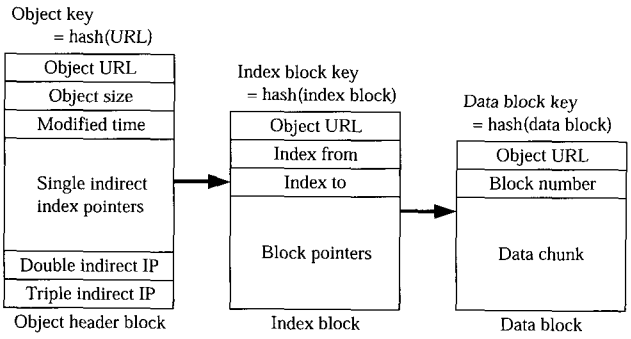


Fig. 2. The structure of our index based allocation method.

ject are distributed in the client-cluster and the storage overhead for each client is reduced and balanced.

The index-based allocation method to store large objects is used, as this method is simple, cost-effective functions simply for random access. Fig. 2 shows the simple structure of the proposed index-based allocation method. Initially, the *object header block* is required. This contains basic information about a large object, such as its URL, size and time of modification, and index pointers, such as single, double, and triple indirect index pointers. Direct index pointers are not used in the object header block. In the general indexed method, a direct index pointer is used to store small files and to avoid making unnecessary index blocks. In the client-cluster, however, the size of the stored objects is large enough so that the overhead of the index blocks can be neglected. The home node for a large object stores this object header block instead of storing the object itself, and manages these header blocks as a LRU list separately from the data blocks.

An index pointer indicates an index block using an index block key, which is the hashed value of the index block itself. The index block is composed of a URL, as well as block pointers which address data using a data block key, and the range of the block pointers. The data block is the leaf block of this method, and stores the actual data chunk. Each data block has a URL as well as a block number, which is assigned continuously from the start of the object to the end.

The basic operation for requesting an object is shown in Fig. 3. In this figure, Client A wants to acquire an object, and sends a request to the proxy cache. The proxy initially checks its local storage. If a hit occurs, the proxy cache returns the object to Client A, the object either being a small object or the object header block for a large object. Otherwise, if a miss occurs, as the proxy cache has no information about the size of the requested object, it sends a lookup message to the client-cluster. In the figure as shown, Client B receives this lookup message first and forwards it to Client C, with the message finally arriving at Client D, which is the home node for the requested object. This home node returns a lookup result indicating whether the node has the object header block or not. If the object header block exists, the proxy stores the new header block in local storage and returns a redirection class response, most likely the "302 Moved Temporarily" response, to the Station of Client A. The Station of Client A obtains the object header block from Client D and

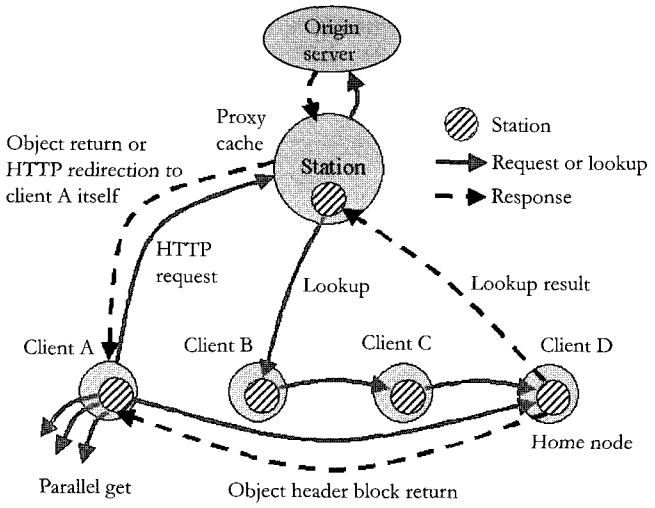


Fig. 3. Operation of the client-cluster, when Client A wants to get an object.

receives data blocks by using parallel connections. In the alternative case, if the header block does not exist at the home node, the proxy sends a request to the origin server for the object and obtains the object. After obtaining the object from the origin server, if the size of the object is small, the proxy cache stores this object in its local storage and returns the object to Client A; otherwise, if it is a large object, the proxy cache only relays the object, and Client A takes charge of storing this object by creating data blocks, index blocks and the object header block, and distributes these blocks into the client-cluster.

D. Client-Cluster Summary

When the proxy misses the object in its local storage, it always checks the client-cluster without regard to the size of the object. If the object is the large sized object, this overhead is negligible, because the transfer overhead for the large sized object is much more than this. However, if the object is the small sized object, this lookup behavior is unnecessary and we get additional latency by this behavior and waste the internal bandwidth. To prevent this leakage, the proxy can have a summary of the objects in the client-cluster.

We use Bloom filter [10] as the summary of the client-cluster. A Bloom filter is a method for representing a set $A = a_1, a_2, \cdots, a_n$ of n elements to support membership queries. The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $h_1, h_2, \cdots, h_k$, each with range $1, \cdots, m$. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), \cdots, h_k(a)$ in v are set to 1. Given a query for b we check the bits at position $h_1(b), h_2(b), \cdots, h_k(b)$. If any of them is 0, then certainly b is not in the set A. For a Bloom filter to represent the large sized objects in the client-cluster, when the proxy cache obtains an large sized object from the outside network, we insert a key for the object to the summary; when the objects in the client-cluster are removed we delete the key from the summary. Using to this summary, we can know whether the requested object is in the client-cluster or not and reduce the unnecessary lookups.

E. n-chance Replacement on Client-Cluster

Each client whose key range does not overlap stores index blocks and data blocks as an LRU list. This behavior distributes a large object over the client-cluster very well, but if just one block is missed, it corrupts the entire object. The missing of a block occurs when a client leaves/fails or if a client evicts blocks to store new blocks (replacements). It is possible to overcome client failures or departures by applying a simple replication strategy to the p2p protocol for the client-cluster [7], [11]. To solve the problem of a replacement, blocks to be evicted are transferred to other clients before they are evicted. When a client evicts a block, it first regenerates a different block key by hashing the block and an optional suffix having a random value. In order to move the block correctly, the client finds the object header block, index block and the block number of the evicted block through the URL, and updates the block pointer with the new block key. The possibility of moving blocks for one large object is permitted until the number of chances is larger than a threshold value, n. If a large object uses all n chances, all of the blocks of the object are removed from the client-cluster. This is known as the n-chance replacement policy.

F. Cache Refreshness

All cached objects can be refreshed to contain the latest data. Typically, an (if modified since) IMS message is used to check if an object has the latest content. If only the proxy cache is used, the validation of the objects is checked by simple IMS methods. If the proxy cache uses the client-cluster, the validations of the objects in the client-cluster are checked only when the proxy cache looks up these objects. According to this procedure, if a proxy cache looks up an object, and the object is deemed to be 'stale' in the client-cluster, the home Station returns this object with an IMS query and the proxy cache sends the IMS query to the original server. If the object is not changed, the proxy cache keeps the object. Alternatively, if the response confirms that it has been modified and is now a new object, the proxy cache stores this new object and sets the *backup bit* to 0 in order to update this modification to the old object in the client-cluster slowly. The backup bit is used to prevent the duplicated storage of an object that is already in the client-cluster. If the backup bit is set to 1, the proxy cache recognizes that the client-cluster has the evicted object and drops this object immediately. If the bit is set to 0, the proxy cache backs up the evicted object to the client-cluster. When the proxy cache obtains the object from the client-cluster, this bit is set to 1. When the object is refreshed or returned from the original server, this bit is set to 0. Through this scheme, the clients do not have to be concerned about the 'staleness' of objects.

G. Analysis of the Response Time

In web caching systems, the hit ratio, the byte hit ratio, and the perceived response time are all important parameters. If the response time is too long for the user to wait for a requested object, though a p2p-based web caching systems achieves a high hit ratio and a high byte hit ratio, these systems do not excel at web caching.

To determine how the proposed system affects the response time of web caching systems, simple models for the systems are utilized. Additionally, a number of simplifying assumptions regarding the p2p substrate are made. First, the data is assumed to be always available. Even if clients dynamically join or leave, the availability of the data is preserved by the fault-tolerant p2p substrate, and this fault handling does not affect response time. Second, the total number of clients is assumed to remain static, and the p2p lookup cost is limited and manages the average value by the ideal formula, $O(\log N)$.

For a set of N identical clients interconnected by high speed LANs whose bandwidth is B_i, the average RTT of the connection from a client to another client is R_i. There is a proxy cache which is placed at the front end of the intra network. The bandwidth of the outside link is B_o, and the average RTT of an internet connection is R_o. The hit ratio of the web caching system is H_r and the hit ratio of the proxy cache is H_{pr}, which does not include hits from the client-cluster. The object size is S. According to these parameters, simple models of the expected latencies were made for the various p2p based web caching systems.

$$\begin{aligned} P &= H_r(R_i + S/B_i) \\ &\quad + (1 - H_r)(R_i + R_o + S/B_i + S/B_o) \\ PC &= H_r(H_{pr}(R_i + S/B_i) \\ &\quad + (1 - H_{pr})(\log(N)R_i + S/B_i)) \\ &\quad + (1 - H_r)(R_i + R_o + S/B_i + S/B_o). \end{aligned}$$

These equations represent the response times for the sole proxy cache (P) and the web caching system with the client-cluster (PC). In essence, when the miss occurs, additional transfer time is needed to obtain the objects from outside links which likely have less bandwidth (B_o) than what is available on the inside links (B_i). Moreover, when the p2p method is used, additional lookup time, such as $\log(N)R_i$, is needed. In particular, though a hit may occur, the lookup time continues to elapse, and this lengthens the response time. In order to overcome this lookup overhead, both the hit ratio and the byte hit ratio should be maximized, and the transfer time from outside links should be reduced. When the web caching interacts with the client-cluster, both the hit ratio and the byte hit ratio increase effectively, helping reduce the external traffic. Consequently, the proposed new system reduces the response time and encourages web caching systems to adopt the p2p protocol. The results for the response time are shown in Section IV-E.

IV. EVALUATION

In this section, the results of extensive trace-driven simulations that evaluated the performance of the proposed system are presented. The proxy cache simulator was designed to conduct the performance evaluation. This simulator illustrates the behavior of a proxy cache and client-cluster. It was assumed that the simulation of the behavior of a proxy cache was effective. The proxy cache is error-free and does not store non-cacheable objects such as dynamic data or control data. It is also assumed that problems such as congestion and overflowing buffers do not exist in the network. The size of a proxy cache ranges from 0.5 MB to 500 MB. Each client uses one Station having storage

Table 1. Traces used in our simulation.

Traces	Trace 1	Trace 2
Measuring day	2001.10.08	2001.10.09
Requests size	9.02 GB	11.66 GB
Objects size	3.48 GB	1.38 GB
Request number	699280	698871
Object number	215427	224104
Hit rate	69.19%	67.93%
Byte hit rate	63.60%	57.79%

of approximately 40 MB. The client-cluster stores large objects whose size is larger than 1 MB, and the size of each block for large objects is 32 kB. The basic replacement policy of every cache is LRU. Moreover, to prepare the simulator the parameters were measure after one million requests.

A. Traces Used

In the trace-driven simulations, traces from KAIST [12] were utilized. KAIST uses a class B IP address for its network. Traces from the proxy cache at KAIST contain over 3.4 million requests in any single day. The simulations were run with traces from this proxy cache starting in October, 2001. A number of the characteristics of these traces are shown in Table 1. It is interesting to note that these characteristics result when the cache size is infinite. However, the simulations assume limited cache storage, and ratios including the hit rate and byte hit rate can not be higher than the *infinite-hit rate* and *infinite-byte hit rate*, denoting the hit rate and the byte hit rate, respectively, when an infinite amount of storage is used.

B. Preliminary Inspection

In order to estimate the performance of the proxy cache, observations were made that measured the number of objects handled by the proxy cache. If more hits occurred in a proxy cache, the proxy cache was said to achieve better performance. However, every hit does not have same weight. A good example of this is the byte hit. As the size of a requested object is variable, both the large number of hits for a small object and a small number of hits for a large object achieve similar byte hits. In Fig. 4, the distribution of the hits which occurs in a proxy cache is shown when the aforementioned traces are used, simulates a proxy cache whose storage is infinite. The hits are distributed by the size of a file. The maximum number of hits obtained takes place for files whose sizes are approximately 256 bytes, and for a minimum value for 64 MB (approximate size) files. However, in terms of the byte hit, the byte hits on files close to 64 MB are larger than those for files of approximately 256 MB.

According to these results, if the general replacement algorithms evict large objects to achieve a high hit rate, they must sacrifice a high byte hit rate. To prevent this degradation of the byte hit rate, it is necessary to store large objects in an exclusive storage area having no relationship to the proxy cache. Large objects can be stored in the client-cluster, which is composed of clients and required no management cost for a proxy cache or other storage

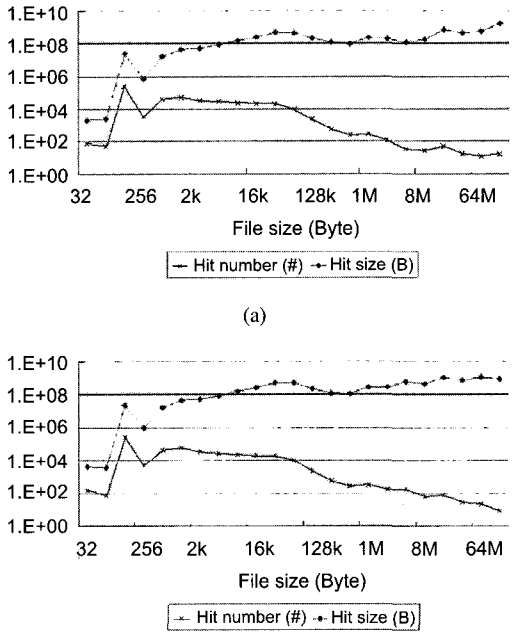


Fig. 4. Hit distribution in the proxy cache which has the infinite storage: (a) Trace 1, (b) Trace 2.

Additionally, in Fig. 4, hits decrease rapidly for files of nearly 1 MB. This value, 1 MB, is used as the threshold value for selecting large objects.

C. Hit Rate and Byte Hit Rate

Figs. 5 and 6 show comparisons of the hit rate and the byte hit rate. The hit rate denotes the number of requests that hits the proxy cache as a percentage of the total requests. The higher the hit rate, the more requests the proxy cache can handle. The original server must then handle a proportionally lighter load of requests. The byte hit rate is the number of bytes that hits the proxy cache as a percentage of total number of bytes requested. A higher byte hit rate results in a greater decrease in network traffic on the server side.

In the figures, *proxy only* indicates using only a proxy cache and *client-clustering* denotes using the client-cluster to store large objects. When the client-cluster is used, a hit not only occurs at the local storage of a proxy cache but also at the client-cluster. To separate the two types of hits, the notation *pure hit* is used, indicating that the hit or byte hit rate obtained only at the local storage of a proxy cache. The difference between *client-clustering* and *pure hit* is that the hit or byte hit rate is measured at the client-cluster only. *Infinite* is the rate when a proxy cache has infinite storage.

Fig. 5 shows the effect of using a client-cluster for the hit rate. When a client-cluster is used to store large objects, the hit rate increases by nearly 10% regardless of the size of the proxy cache. Moreover, in every case, the hit rate of a *pure hit* is similar to the hit rate of *client-clustering*. Numerically, the difference of these two values is approximately 0.2%. Most of hits occur in the local storage of a proxy cache, and few hits (nearly 0.2%) occur in the client-cluster. When using a client-

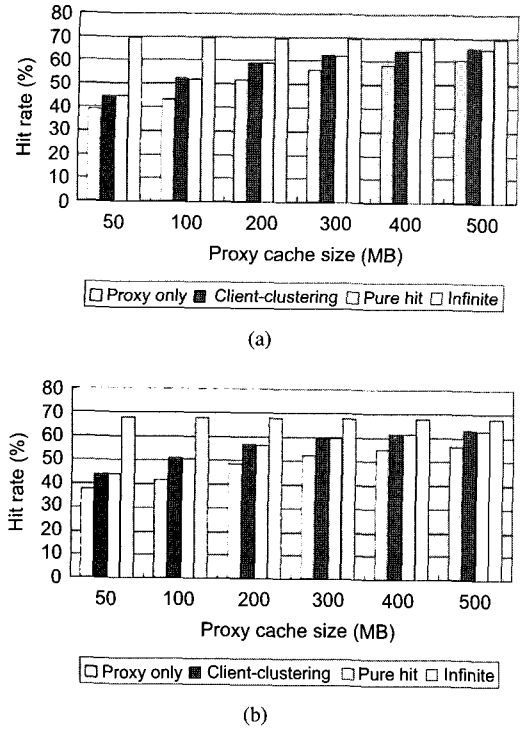


Fig. 5. Hit rate comparison between only proxy cache and client-clustering (100 clients): (a) Trace 1, (b) Trace 2.

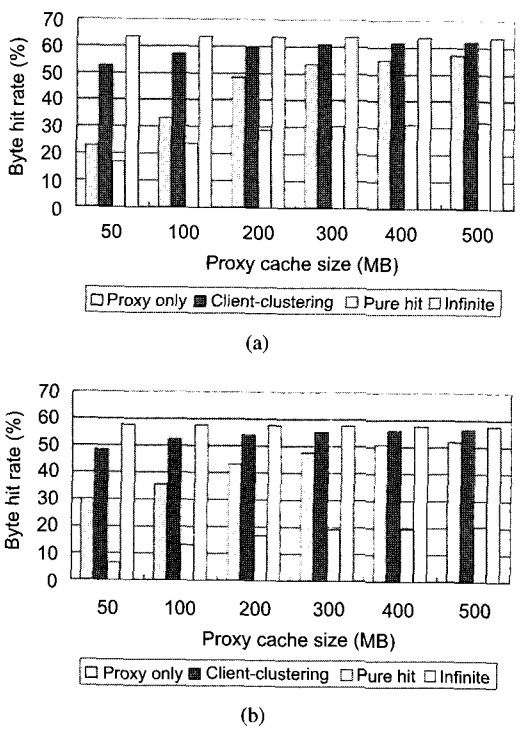


Fig. 6. Byte hit rate comparison between only proxy cache and client-clustering (100 clients): (a) Trace 1, (b) Trace 2.

cluster, the increasing effect of the hit rate is due to the increase in the hit rate in the local storage of a proxy cache. That is, the proxy cache does not handle large objects anymore, and it can store a greater number of small objects. The hit rate of the whole system then increases.

From the results of the hit rate, when the client-cluster is used, only a small number of hits occur in the client-cluster for large objects. However, these hits bring very large byte hits due to the large sizes of the requested objects. In Fig. 6, when using a client-cluster, the byte hit rate increases remarkably, achieving a similar value to the infinite-byte hit rate while staying unrelated to the proxy cache size. In contrast to the hit rate, the byte hit rate of *pure hit* is much smaller than the rate of *client-clustering* or the rate of *proxy only*. Chiefly in the result of Trace 2, the byte hit rate obtained from the local storage of a proxy cache is smaller by a third than the byte hit rate from a client-cluster. According to this result, though a proxy cache achieves high hit rate, small objects in the proxy cause the low byte hit rate. This weak point can be managed by storing large objects in a client-cluster. Consequently, to use a client-cluster, which is an exclusive storage for large objects, not only a high hit rate is achieved, but also a high byte hit rate. It is possible to preserve and improve the performance of a proxy cache without incurring expensive management costs.

D. Client Size Effect

Additionally, to show scalability of the proxy cache which uses the client-cluster, it is assumed that every 100 clients make 0.35 million requests and that they simulate with a variable number of clients. The results are shown in Figs. 7 and 8, where the cent n indicates the use of only a proxy cache whose size is n hundreds MB, and the back n denotes that a proxy cache and a client-cluster are used. In every case, the client-cluster is used, the hit rate increases by nearly 10% and the byte hit rate increases by nearly 10–15%. It is interesting to note that in Fig. 8(b), the byte hit rate achieves a high value regardless of the number of clients. This indicates that a proxy cache using a client-cluster copes with the growth of the client population without incurring management costs; that is, the proposed system is scalable.

Unlike Trace 2, even though the hit rate is high in Fig. 7(a), in Fig. 8(a), the byte hit rate decreases drastically. The characteristics of Trace 1 cause this appearance. In Table 1, the request size of Trace 2 is nearly eleven times the object size; however, the request size of Trace 1 is only approximately three times the object size. In other words, Trace 1 has many more requests for small objects than it does for large objects, and the effect of storing large objects is insufficient to compensate for the degradation of the byte hit rate. In order to prevent this from happening, the proxy cache should have some backup storage, such as the proposed system in [13], and should maximize the hit rate for small objects.

E. Response Time

Even though the hit rate and byte hit rate increase by using the client-cluster, if it causes any harm to the response time, it is not viable for the web caching system. The client-cluster has a small number of factors which can increase the response time, such as the lookup cost. The lookup cost of the client-cluster is heavy, as the lookup of the p2p substrate requires multiple routing hops.

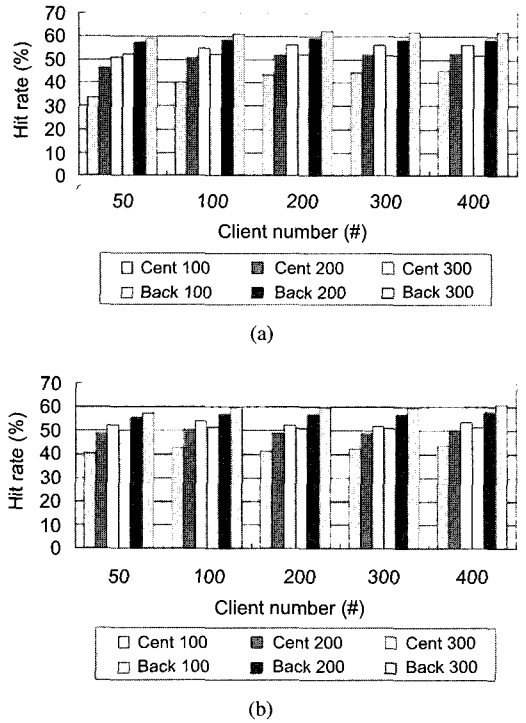


Fig. 7. Hit rate comparison with various client number: (a) Trace 1, (b) Trace 2.

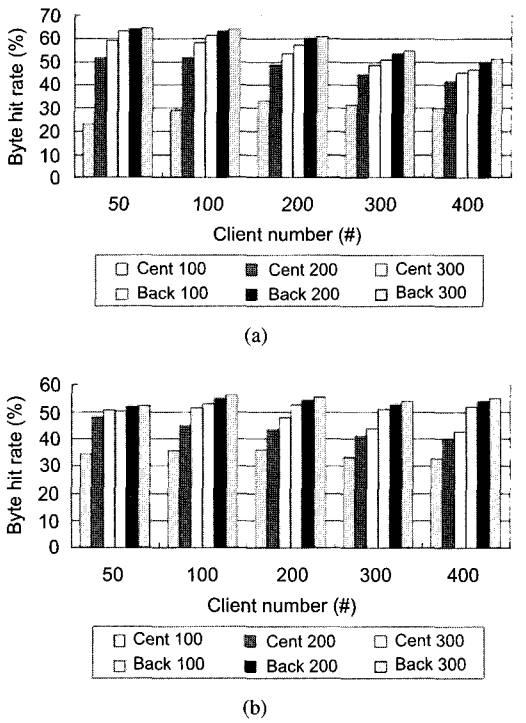


Fig. 8. Byte hit rate comparison with various client number: (a) Trace 1, (b) Trace 2.

General DHT-based p2p protocols need $O(\log N)$ messages for a lookup, where N is the number of live nodes. To analyze how the client-cluster affects the response time, the response time of each request is measured using the network model which was introduced in Section III-G. In Fig. 9, the comparison of the average response time is shown. It is surprising that the client-cluster decreases the response time by about 20 ms. Moreover, if more clients join the client-cluster, the web caching system can further reduce the response time. The reason for this result has to do with efficient caching for large objects. When only large objects are requested, the heavy lookup cost adds to the response time. However, the number of the requests for large objects is not many, and most of these hit on the client-cluster. According to this, the client-cluster acting as the exclusive storage for large objects can help reduce the response time.

F. Client Load

The client loads are examined, including the request number, storage size, and stored object, in order to verify that the client-cluster balances storage and requested queries. Table 2 shows a summary of the supported storage size, the requested number and the requested byte of clients. According to this table, in order to store entire large objects in the client-cluster, each client should supply about 10–20 MB of storage to the proxy cache and handle about 1000–4000 requests per day. These loads are enough for any client to handle considering today's technology, and if the number of clients is 300 or more, these loads are negligible. The deviation value for each metric is less than 4%, and each client receives roughly the same load. Furthermore, when the client number increases, the load of each client decreases.

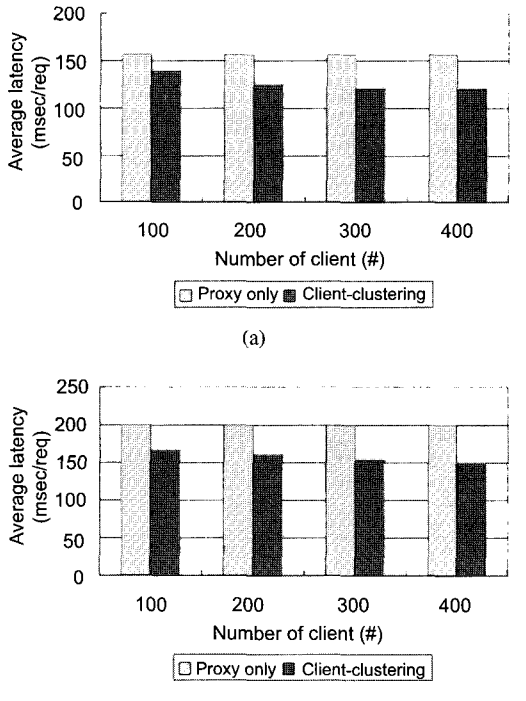


Fig. 9. Response time comparison between the only proxy cache and client-clustering: (a) Trace 1, (b) Trace 2.

The scalability of the proposed system is verified according to these results.

Table 2. Summary of client loads for Trace 1 with the 200 MB proxy.

Client number	Mean size	Max size	Dev.
100	20967kB	21659 kB	1.65
200	10586 kB	11141 kB	2.48
300	7081 kB	7372 kB	1.98
	Mean req.	Max req.	Dev.
100	3662	3899	2.12
200	1842	2001	2.99
300	1230	1375	4.1
	Mean BReq.	Max BReq.	Dev.
100	120025 kB	127762 kB	2.1
200	60374 kB	65568 kB	2.98
300	40327 kB	45056 kB	4.09

V. RELATED WORK

Many peer-to-peer applications such as Napster, Kazza, and Morpheus have become popular. Additionally, large area file systems using peer-to-peer have been proposed, including PAST [14], CFS [11], and Oceanstore [15]. The target of these systems, however, is a wide area network, and they address issues of the characteristics of web objects, such as size, popularity, and update frequency.

A similar proposal for the outlined approach appears in [13] and [16]. The web browser caches of clients themselves are used in [16]. This approach does not consider large objects, and the load of each client is not balanced well. Moreover, when the availability of clients is asymmetric, some of the clients decrease the total performance of the cache system. In [13], the residual resources of clients are used as backup storage of the proxy cache and the hit rate is maximized, making it similar to the infinite-hit rate. [17] shows a new approach which uses 'superclients,' which makes the cluster of clients hierarchical in structure. However, this proposed system is affected by large objects. Therefore, they cannot achieve high byte hit rate and cannot balance the load in regards to the byte requests. Moreover, in [18], files are cut into pieces of fixed sizes for parallel downloading, similar to the proposed approach for a large object. While this approach focuses the point of the improvement on the download time, the proposed system here considers not only the download time but also system performance such as the hit rate, as well as the byte hit rate and storage utilization.

VI. CONCLUSION

In this paper, the peer-to-peer client-cluster is proposed and evaluated, which is used as exclusive storage for a web proxy cache. The proxy cache with this client-cluster achieves not only a high hit rate but also a high byte hit rate. This behavior reduces the outgoing traffic occurring over congested links and improves the performance of the connections to the outside world. Moreover, the client-cluster supported by the clients using the proxy cache is highly scalable, and the proxy requires only a low administrative cost. With this system, even if the clients take on a load, this load is verified on a range of real workloads to be low and well balanced. Additionally, if this client-cluster is used not only as the exclusive storage of a proxy cache but also as the backup storage of a proxy cache, a high value can be achieved for both the hit rate and the byte hit rate, which is similar to the value when the infinite cache is used.

REFERENCES

[1] S. Williams, M. Abrams, R. Standbridge, G. Abdulla, and E. A. Fox, "Removal policies in network caches for world-wide web documents," in *Proc. ACM SIGCOMM'96*, Aug. 1996.

[2] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proc. USITS'97*, Dec. 1997.

[3] C. Aggarwal, H. L. Wolf, and P. S. Yu, "Caching on the world wide web," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, Jan. 1999.

[4] K. Kim and D. Park, "Least popularity per byte replacement algorithm for a proxy cache," in *Proc. ICPADS 2001*, June 2001.

[5] J. Wang, "A survey of web caching schemes for the internet," *ACM Computer Commun. Rev.*, vol. 29, Oct. 1999.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proc. ACM SIGCOMM 2001*, Aug. 2001.

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proc. MIDDLEWARE 2001*, Nov. 2001.

[8] B. Y. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," *UCB Technical Report UCB/CSD-01-114*, 2001.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM 2001*, 2000.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *Proc. SIGCOMM'98*, 1998.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. ACM SOSP 2001*, 2001.

[12] Korea Advanced Institute of Science and Technology, http://www.kaist.ac.kr.

[13] K. Kim and D. Park, "Efficient and scalable client clustering for web proxy cache," *IEICE Trans. Inform. Syst.*, vol. E86-D, Sept. 2003.

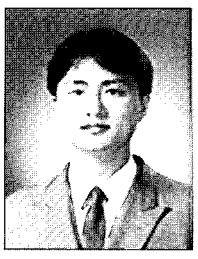
[14] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Proc. HotOS VIII*, 2001.

[15] J. Kubiatowicz, D. Binder, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proc. ACM ASPLOS 2000*, Nov. 200.

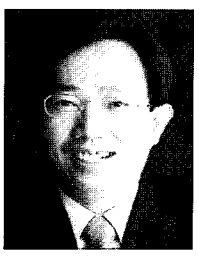
[16] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A decentralized peer-to-peer web cache," in *Proc. PODC 2002*, 2002.

[17] Z. Xu, Y. Hu and L. Bhuyan, "Exploiting client cache: A scalable and efficient approach to build large web," in *Proc. IPDPS 2004*, Apr. 2004.

[18] Bittorent, http://bittorent.com.



Kyungbaek Kim received his B.S. and M.S. degrees in Electrical Engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1999 and 2001, respectively. Currently, he is a Ph.D. candidate at KAIST in Korea. His research interests include operating system, distributed system, world wide web, peer-to-peer algorithm/network, and overlay multicast.



Daeyeon Park received his B.S. and M.S. degrees in computer science from University of Oregon, USA, in 1989 and 1991, respectively and Ph.D. degree in computer science from University of Southern California, USA, 1996. He worked at Hankuk University of Foreign Studies from 1996 to 1997. He joined the Department of Electrical Engineering at KAIST in 1998, where he is currently an Assistant Professor. His major interests include operating system, distributed system, parallel processing, and computer architecture. He is a member of KIEE, KISS, IEICE, and IEEE.