PAPER *Special Issue on Parallel and Distributed Computing, Applications and Technologies*

Efficient and Scalable Client Clustering for Web Proxy Cache

Kyungbaek KIM[†] ***and*** **Daeyeon PARK**[†]**,** ***Nonmembers***

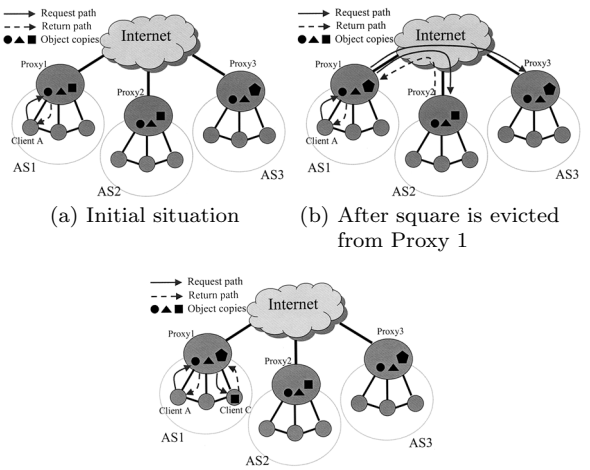
SUMMARY Many cooperated web cache systems and protocols have been proposed. These systems, however, require expensive resources, such as external bandwidth and CPU power or storage of a proxy, while inducing hefty administrative costs to achieve adequate client population growth. Moreover, a scalability problem in the cache server management still exists. This paper suggests peer-to-peer client-clustering. The client-cluster provides a proxy cache with backup storage which is comprised of the residual resources of the clients. We use DHT based peer-to-peer lookup protocol to manage the client-cluster. With the natural characteristics of this protocol, the client-cluster is self-organizing, fault-tolerant, well-balanced and scalable. Additionally, we propose the Backward ICP which is used to communicate between the proxy cache and the client-cluster, to reduce the overhead of the object replication and to use the resources more efficiently. We examine the performance of the client-cluster via a trace driven simulation and demonstrate effective enhancement of the proxy cache performance.

key words: *peer-to-peer, clustering, web caching, cooperated caching*

1. Introduction

The recent increase in popularity of the Web has led to a considerable increase in the amount of Internet traffic. As a result, the Web has now become one of the primary bottlenecks to network performance and web caching has become an increasingly important issue. Web caching aims to reduce network traffic, server load, and user-perceived retrieval delay by replicating popular content on caches that are strategically placed within the network. Browser caches reside in the clients' desktop, and proxy caches are deployed on dedicated machines at the boundary of corporate network and Internet service providers.

By caching requests for a group of users, a proxy cache can quickly return documents previously accessed by other clients. Using only one proxy cache has limited performance, because the hit rate of the proxy is limited by the cache storage and the size of the client population. That is, if a cache is full and needs space for new documents, it evicts the other documents and it will retrieve the evicted documents from the Internet for other requests. In Fig. 1 (a), if the square object is evicted, the proxy cache obtains it from the Internet. But if the near proxy cache has a square object like that in Fig. 1 (b), Proxy 1 can obtain it from Proxy 2 and reduce the latency and the Internet traffic. According to this procedure, multiple proxies should cooperate with each other in order to increase the total client population, improve hit ratios, and reduce document-access latency; that is the cooperative caching.



(a) Initial situation (b) After square is evicted from Proxy 1

(c) After square is evicted from Proxy 1 and stored at client C

Fig. 1 Request and response path when client A requests the square object in Proxy 1.

Various cooperative caching systems have been proposed in [1]–[5]. However, these techniques need high bandwidth, expensive infrastructure and high administrative cost. ICP-based cooperative caches communicate with other caches that are connected by busy core-links, which are the inter-proxy links, to find and obtain requested objects in other caches. Even if the requested objects are not in these caches, they spend bandwidth of core-links in order to find the objects. Some cooperative caches use the proxy cluster, as a single large cache so as to be overprovisioned to handle bursty peak loads. However, this approach still needs too much administrative cost for the frequent variation of clients. For example, a growth in client population necessitates increasing the cluster size and updating the cluster information.

In this paper, we suggest a new web caching system which uses the residual resources of clients. In Fig. 1 (c), not only the proxy cache but also the clients are respon-

Manuscript received December 4, 2002.
Manuscript revised March 11, 2003.

[†]The authors are with Department of Electrical Engineering and Computer Science [Division of Electrical Engineering], Korea Advanced Institute of Science and Technology, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305–701, Republic of Korea.

sible for storing objects; the proxy cache stores more popular objects and the client-cluster stores evicted objects from the proxy cache. That is, the client-cluster is used as a backup storage for the proxy cache. In this case, Client A can get the square object from Client C, which is inside the network, not outside of it. This behavior reduces the usage of core-links and improves the performance of the proxy cache, in terms of the hit rate, the byte hit rate and the reduced latency. Furthermore, the size of the backup storage of the proxy increases as more clients use the proxy. According to this feature, this approach reduces the administrative cost and makes the proxy cache more scalable.

The client-cluster is composed of the clients' residual resources. Since the 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 meet these requirements, we manage the client-cluster by using DHT (Distributed Hash Table) based peer-to-peer protocol. By using this protocol, all clients receive roughly the same load because the hash function balances load with high probability. Additionally, the proxy cache does not need to gather the client information and we reduce administrative cost.

This protocol is responsible for the routing of the object, but it needs to cope with updating the object whenever clients join or leave. Typically, we can replicate the object. However, this approach leads to extremely large traffic overhead and wasted storage. To reduce this overhead, we suggest the *Backward ICP* which is responsible for storing and finding objects in a manner similar to replication. A proxy saves objects to a client-cluster and gets objects from it by using this protocol.

When a proxy cache sends requests to a client-cluster and the requested objects are not stored in it, the proxy cache takes on additional latency. To prevent this latency, we use the cache summary with a Bloom filter, which determines whether the requested objects are in the client-cluster.

This paper is organized as follow. In Sect. 2, we describe cooperated web caching and peer-to-peer lookup algorithm briefly. Section 3 introduces the detail of the peer-to-peer client-clustering. The simulation environment and the performance evaluation are given in Sect. 4. We mention other related works in Sect. 5. Finally, we conclude in Sect. 6.

2. Background

2.1 Cooperated Web Caching

The basic operation of the web caching is simple. Web browsers generate HTTP GET requests for Internet objects such as HTML pages, images, mp3 files, etc. 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 requested objects are in a web browser cache, the client does not need to send GET requests. Otherwise, the client sends GET requests to web proxy caches. If a cache closer to the client has a copy of the requested object, we reduce more bandwidth consumption and decrease more network traffic. Hence, the cache hit rate should be maximized and the miss penalty, which is the cost when a miss occurs, should be minimized when designing a web caching system.

The performance of a web caching system depends on the size of its client community. As the user community increases in size, so does the probability that a cached object will soon be requested again. Caches sharing mutual trust may assist each other to increase the hit rate. A caching architecture should provide the paradigm for proxies to cooperate efficiently with each other. One approach to coordinate caches in the same system is to set up a caching hierarchy. With hierarchical caching, caches are placed at multiple levels of the network. Another approach is a distributed caching system, where there are only caches at the bottom level and there are no other intermediate cache levels. In a hybrid scheme, caches may cooperate with other caches at the same level or at a higher level.

ICP (Internet Cache Protocol) [1] is a typical cooperating protocol for a proxy to communicate with other proxies. If a requested object is not found in a local proxy, the proxy sends ICP queries to neighbor proxies; sibling proxies and parent proxies. Each neighbor proxy receives the queries and sends ICP replies without concern about existence of the object. If the local proxy receives an ICP reply with the object, it uses that reply. Otherwise, the local proxy forwards the request to the parent proxy. ICP wastes expensive resources; core-link and cache storage. Even if the neighbor caches do not have the requested object, ICP uses the core-links between proxies, which are used for many clients and are bottlenecks of the network bandwidth. Another protocol for cooperated caching is CARP (Cache Array Routing Protocol) [2], which divides the URL-space among an array of loosely coupled caches and lets each cache store only the objects whose URL are hashed to it. For this feature, every request is hashed and forwarded to a selected cache node. In this scheme, clients must know the cache array information and the hash function, making the management of CARP difficult. Additionally, there are other issues such as load balancing and fault tolerance.

Another problem of CARP, as well as ICP, is scalability of management. Large corporate networks often employ a cluster of machines, which generally must be overprovisioned to handle burst peak loads. A growth in user population creates a need for hardware upgrades. This scalability issue cannot be solved by ICP or CARP.

2.2 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 a data is an essential functionality for peer-to-peer systems.

A number of peer-to-peer lookup protocols have been recently proposed, including Pastry [6], Chord [7], CAN [8] and Tapestry [9]. In a self-organizing and decentralized manner, these protocols provide a DHT (distributed hash-table) 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.

3. Peer-to-Peer Client-Clustering

3.1 Overview

As we described in the previous section, the use of only a proxy cache has a performance limitation because of potential growth in client population. Even if proxy caches cooperate with each other to enhance performance, high administrative cost and scalability issues still exist. To improve the performance of the cache system and solve the scalability issues, we exploit the residual resources of clients for a proxy cache. That is, any client that wants to use the proxy cache provides small resources to the proxy and the proxy uses these additional resources to maintain the proxy cache system. This feature makes the system resourceful and scalable.

We use the residual resources of clients as a backup storage for the proxy cache. While a conventional proxy cache drops evicted objects, our proxy cache stores these objects to the backup storage, which is distributed among the client-cluster. When a client sends a GET request to a proxy cache, it checks its local storage. If a hit occurs, it returns the requested object; otherwise, it sends a lookup message to the backup storage and this message is forwarded to the client that has responsibility for storing the object. If the client has the object, it returns the object to the proxy; otherwise, the proxy gets the object from the original server or other proxy caches. This interaction between the proxy cache and the backup storage decreases the probability of sending requests outside the network, reduces the usage of inter-proxy links, and increases the performance of the proxy cache.

3.2 Client-Cluster Management

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

We design the client-cluster by using DHT (distributed hash table) based peer-to-peer protocol [6], [7]. To use this protocol, each client needs an application whose name is *Station*. A Station is not a browser or a browser cache, but a management program to provide clients' resources for a proxy cache. A client can not use resources of a Station directly, while a proxy cache sends requests issued from clients to Stations in order to use resources of a client-cluster. When a Station receives requests from a proxy cache, it forwards requests to another Station or checks whether it has 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 the 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 efficient routing of request queries. It 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* which are numerically close to the Station, like the leaf nodes in PASTY or the successor list in CHORD.

The basic operation of the lookup in a client-cluster is shown in Fig. 2. When a proxy cache sends a request query to one Station of a client-cluster, the Station gets the object key of the requested object and selects the next Station according to the DHT and the object key. Finally, the *home Station*, which is a Station having the numerically closest node key to the requested object key among all currently live nodes, receives the request and checks whether it has the object in local cache. If a hit occurs, the home Station returns the object to the proxy cache; otherwise, it only

returns a null object. In Fig. 2, 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 1000 Stations exist, the cost of lookup is about 3, and if 100000 Stations, the cost is about 5. Since the RTT for any server in the Internet from one client is 10 or 100 times bigger than that for another client in the same network, we reduce the latency for an object by 2 or 20 times when we obtain the object in the client-cluster.

The client-cluster can cope with frequent variations in client membership by using this protocol. Though the clients dynamically join and leave, the lazy update for managing the small information of the membership changes does not spoil 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 gets new DHT and other Stations to update their DHT for the new Station lazily. On the other hand, when a Station leaves or fails, other Stations, which have a DHT mapping with the departing Station, detect the failure of it lazily and repair their DHT. According to this feature, the client-cluster is self-organizing and fault-tolerant.

The proxy cache stores the evicted objects to a particular Station in the client-cluster by using this lookup operation. All Stations have roughly the same amount of objects, because 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. According to this, when the client membership changes, the object range is resized automatically and the home Stations for every object are changed implicitly.

As described, the routing information and the object range are well managed by this protocol. Consequently, after updating the information for variation in the client membership, future requests for an object will be routed to the Station that is now numerically closest to the object key. If the objects for the new home Station are not moved, subsequent requests miss the objects. According to these misses, the performance of a client-cluster decreases remarkably. We can replicate the objects to neighbor Stations to prevent such misses. This approach ensures the reliability of the objects, but leads to serious traffic overhead and inefficient storage usage. To reduce this overhead and use the storage efficiently, we store and lookup objects using the *Backward ICP*. This is described in the next section.

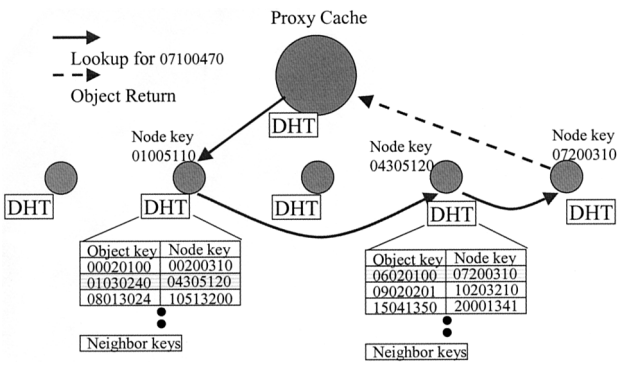
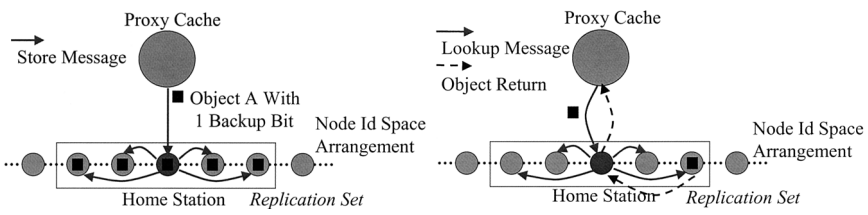


Fig. 2 Basic lookup operation in the client-cluster. In this figure, total hop count is 3 for an object.

3.3 Backward ICP

The Backward ICP, which is a communication protocol between the proxy cache and the client-cluster, is similar to the ICP used between the proxy caches. However, the Backward ICP uses a local area network rather than an inter-proxy link.

There are two types of messages in the Backward ICP, as shown in Fig. 3. One is a *Store* message and the other is a *Lookup* message. A Store message is used to store evicted objects from a proxy cache. The proxy cache sends a Store message and the evicted object to the home Station and the home Station replicates the objects to the replication set, which is composed of neighbor Stations. Before sending a Store message for an evicted object, the proxy cache checks the *Backup bit* of the evicted object. This Backup bit is used to prevent duplicated storage of an object that is already in the client-cluster. If the Backup bit is set to 1, the proxy cache knows that the client-cluster has this 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 gets the object from the client-cluster, this bit is set to 1. When the object is refreshed or returned from the original



(a) Store message (b) Lookup message

Fig. 3 Two types of Backward ICP message.

server, this bit is set to 0.

A Lookup message is used to find objects in the client-cluster. When the proxy cache sends a Lookup message to the home Station, this Station returns the object to the proxy cache if it has the requested object. Otherwise, if a miss occurs, it sends Lookup messages to the replication set simultaneously and waits for a response from any Station. If the object is somewhere among the replication set, the home Station stores this object and returns this to the proxy cache; otherwise, it returns a null object. Following this, the home Station replicates the object to the replication set, except the responding Station.

This protocol replicates objects only at the time when they are stored or a lookup miss occurs. It reduces traffic overhead incurred by object replications. Moreover, it uses storage efficiently by giving more opportunities to retrieve popular objects. The first time when any object is stored, the object is replicated to increase the probability of accessing the object. As time goes by, popular objects are requested more than other objects and they are replicated again to increase the probability.

3.4 Cache Summary

When the proxy cache does not have an object in local cache, it finds the object in the client-cluster. If the object is not in the client-cluster, the proxy cache suffers from additional latency by routing in the client-cluster. To prevent this latency, the proxy can have a summary of the objects in the client-cluster. We use a Bloom filter as the summary of the client-cluster. A Bloom filter is a method for representing a set $A = a_1, a_2, \ldots, a_n$ 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, $h_1, h_2, \ldots h_k$, each with range $1, \ldots, m$. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$ in v are set to 1. Given a query for b we check the bits at position $h_1(b), h_2(b), \ldots, h_k(b)$. If any of them is 0, then certainly b is not in the set A. More details and other applications can be found in [5].

For a Bloom filter to represent the objects in the client-cluster, when the proxy cache evicts the object to the client-cluster we insert a key for the object to the filter; when the proxy cache misses the object in the client-cluster we delete the key from the filter. We maintain for each location l in the bit array a count $c(l)$ of the number of times that the bit is set to 1. All counts are initially 0. When a key a is inserted or deleted, the counts $c(h_1(a)), c(h_2(a)), \ldots, c(h_k(a))$ are incremented or decremented accordingly. When a count changes from 0 to 1, the corresponding bit is turned on. When a count changes from 1 to 0, the corresponding bit is turned off.

3.5 Cache Refreshness

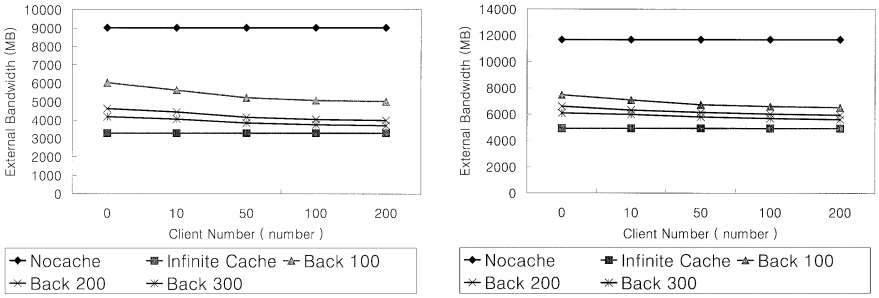
All cached objects can be refreshed to contain the latest data. Typically, an IMS (If Modified Since) message is used to check if the object has the latest content. If only the proxy cache is used, the validation of the objects is checked by simple IMS methods. If the proxy cache uses the client-cluster, the validation 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 that is stale in the client-cluster, the home Station returns this object with the IMS query and the proxy cache sends the IMS query to the original server. If the object is not changed, the proxy cache keeps this object. Otherwise, if the response notifies that it is modified with the 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 lazily. By using this scheme, the Stations do not have to be concerned about staleness of objects.

4. Performance Evaluation

In this section, we present the results of extensive trace driven simulations that we have conducted to evaluate the performance of our cache system. We design our proxy cache simulator to conduct the performance evaluation. This simulator illustrates the behavior of a proxy cache and client-cluster. We have assumed that we simulate the behavior of a proxy cache effectively. The proxy cache is error-free and does not store non-cachable objects: dynamic data, larger size data than total cache storage, control data, etc. We also assume that there are not any problems in the network, such as congestions and overflow buffers. The size of a proxy cache is in the range from 0.5 MByte to 500 MBytes. Each client uses one Station which has the storage, from 10 MBytes to 30 MBytes.

4.1 Traces Used

In our trace-driven simulations we use traces from KAIST, which uses a class B ip address for the network. The trace from the proxy cache in KAIST contains over 3.4 million requests in a single day. We have run our simulations with traces from this proxy cache since October, 2001. We show some of the characteristics of these traces in Table 1. Note that these characteristics are the results when the cache size is infinite. However, our simulations assume limited cache storage and ratios including hit rate and byte hit rate cannot be higher than *infinite-hit rate* and *infinite-byte hit rate*, which are the hit rate and the byte hit rate when the infinite cache is used.



(a) Trace 1 (b) Trace 2

Fig. 4 Total external bandwidth usage with various client size.

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%

4.2 External Bandwidth

We define *external bandwidth* as the number of bytes transfered between the proxy cache and the original servers. This bandwidth, which uses the core-link of the proxy cache, should be of a small value for good performance of the Internet and the proxy cache. We compare external bandwidth savings achieved by using the client-cluster with a variable proxy cache storage. We assume that each client provides 10 Mbyte storage to the proxy server.

Figure 4 shows the external bandwidth in MBs over the entire period of each trace against different sizes of the client number. In this figure, back n means that the proxy cache has a storage of n hundred MBs and uses the client-cluster. No Cache and Infinite Cache represent the external bandwidth if no cache is used, and if a cache which has infinite storage is used, respectivley. The difference between the two values denotes the maximum external bandwidth reduction that web caching obtains.

When there is no member in the client-cluster and the proxy cache only serves clients, we can reduce the external bandwidth. However, if we want to reduce the bandwidth further, we should increase the size of the proxy cache storage. On the other hand, by using the client-cluster, if more clients provide resources to the proxy cache, external bandwidth is further reduced. These results strongly indicate the client-cluster is scalable and reduce the administrative cost of the proxy cache.

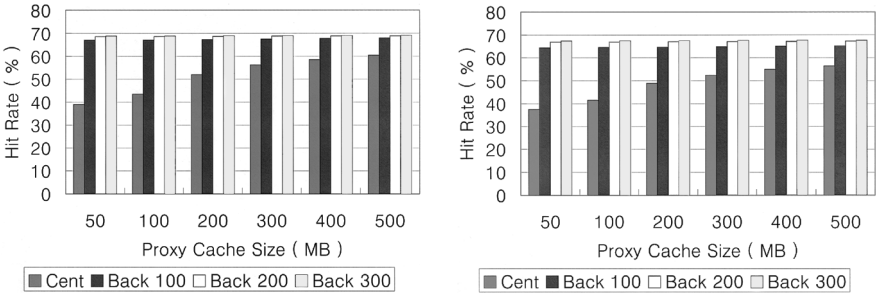
4.3 Hit Rate and Byte Hit Rate

Figures 5 and 6 show a comparison of the *hit rate* and the *byte hit rate*. By the hit rate, we mean the number of requests that hit in the proxy cache as a percentage of total requests. A higher the hit rate means the proxy cache can handle more requests and the original server must deal with proportionally lighter load of requests. The byte hit rate is the number of bytes that hit in 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, cent means using only a proxy cache and back n means using the client-cluster with n hundreds clients. The hit rate of only the proxy cache is greatly affected by the cache size, but the hit rate of using the client-cluster achieves nearly an infinite-hit rate without any relationship to the proxy cache size. This is achieved by the plentiful resources provided by the clients. That is, though the proxy cache size is limited, the storage of the client-cluster is sufficient to store evicted objects and the proxy cache gets almost all requested objects from the client-cluster.

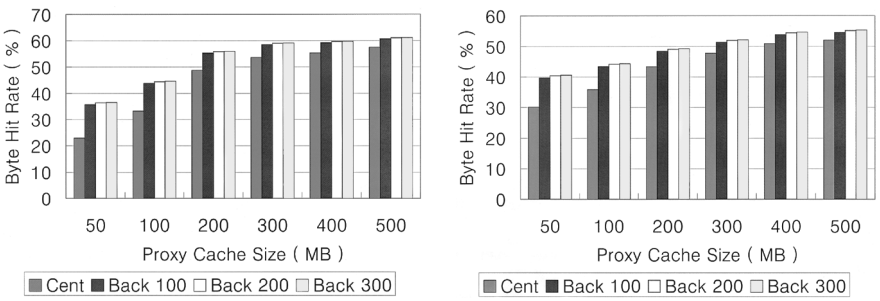
For the byte hit rate, we can obtain a similar result as that for the hit rate. However, in this case, using the client-cluster does not yield infinite-byte hit rate, particularly with a small proxy cache size. The reason for this result is the different sizes of the objects. Though each client have roughly the same amount of objects, some clients that usually have large objects cannot store many objects, and the hit rate and the byte hit rate decrease. In particular, large size objects whose size is bigger than that of one client storage, which is the Station's storage, 10 MB, are not stored on the client-cluster and the byte hit rate decreases remarkably.

To examine this feature, we run a simulation with a fixed client number, 100, and various sizes of client storage, 20 MB and 30 MB. The results of this simulation are shown in Fig. 7, which provides a comparison of the byte hit rate. If an individual client storage increases, the byte hit rate increases slightly. This strongly indicate that the different sizes of objects, and especially



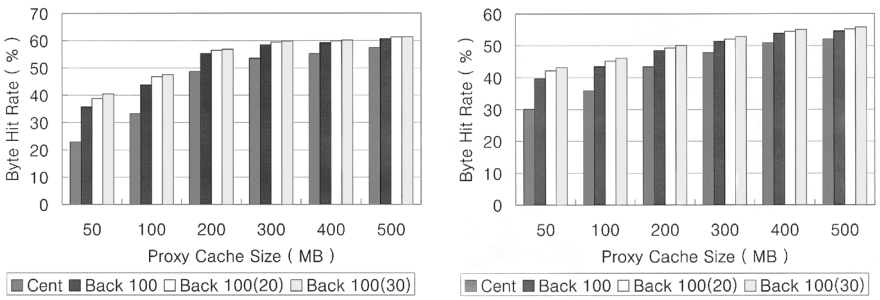
(a) Trace 1 (b) Trace 2

Fig. 5 Hit rate comparison between only proxy cache (cent) and client-cluster (back-n).



(a) Trace 1 (b) Trace 2

Fig. 6 Byte hit rate comparison between only proxy cache (cent) and client-cluster (back-n).



(a) Trace 1 (b) Trace 2

Fig. 7 Byte hit rate comparison with various client storage.

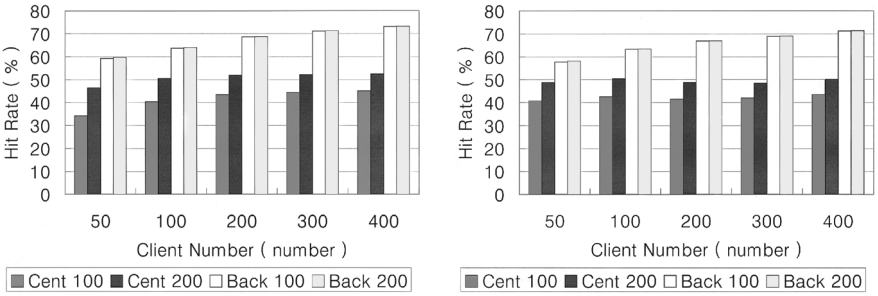
large objects, affects the byte hit rate. If the client-cluster stores these large objects well, both the hit rate and the byte hit rate achieve an infinite-hit rate and infinite-byte hit rate for a proxy cache.

4.4 Client Size Effect

In this section, we show scalability of the client-clustering. We assume every 100 clients makes 0.35 million requests and simulate with variable client number. The results are shown in Fig. 8 and Fig. 9 where the cent n indicates use of only a proxy cache whose size is n hundreds MB and the back n means the proxy cache and the client-cluster are used.

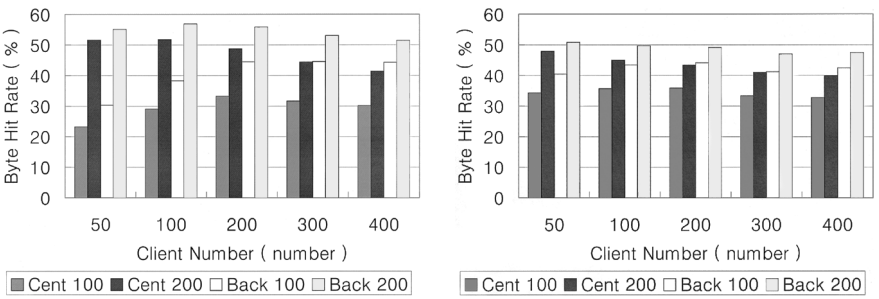
In the results, the hit rate when only the proxy cache is used does not increase markedly. Even in Trace 2, the hit rate decreases. However, when a proxy cache uses the client-cluster, the hit rate increases by 30-40% over that when only the proxy cache is used. Additionally, as the client number increases, the hit rate increases accordingly. For the byte hit rate, when the proxy cache uses the client-cluster, the byte hit rate increases by 20-30% over that when only the proxy cache is employed. When the client number is 300, the byte hit rate with the client-cluster is the same or higher than that with only the proxy cache and when the client number is 400, the byte hit rate with the client-cluster is much higher.

According to these results, when client population grows, using only a proxy cache should take on administrative cost to provide sufficient service to clients. However, using the client-cluster does not need any management cost to handle the growth in client population.



(a) Trace 1 (b) Trace 2

Fig. 8 Hit rate comparison with various client number.



(a) Trace 1 (b) Trace 2

Fig. 9 Byte hit rate comparison with various client number.

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

Client number	Mean Req.	Max Req.	Dev.
100	1024	1369	2.2
200	602	733	2.4
300	401	510	2.5
	Mean Byte Req.	Max. Byte Req.	Dev.
100	13422KB	316805KB	11.1
200	6711KB	315158KB	12.1
300	4474KB	314197KB	12.9

Consequently, the client-cluster is scalable.

4.5 Client Load

We examine the client loads, which include the request number, storage size, stored objects, hit rate, etc, to verify that the client-cluster balances the storage and request queries. Table 2 shows a summary of the request number and the sizes of the requested objects. Each client receives roughly the same load, and when the client number increases the load of each client decreases. The properties of DHT-based peer-to-peer protocols account for these findings. For the byte request, we again see the effect of the different sizes of the objects, which we strongly believe account for the performance degradation.

5. Related Works

Cooperative web caching has been found to be useful in improving hit rate in a group of small organizations. There are many forms of cooperative caching, such as hierarchical web caching [1], hash-based clustering [2] and directory-based scheme [5]. These methods are efficient but they need high resources and induce high administrative costs to improve the utility and the scalability of the caching system. On the other hand, in our scheme, the client-cluster is composed of the residual resources of the clients and scalability is a natural characteristic of the client-cluster.

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 [12], CFS [13] and OceanStore [11]. 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.

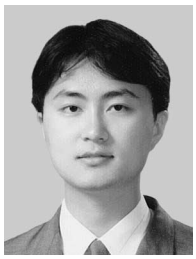
A similar proposal for our approach appeared in Squirrel [10], which described a decentralized web browser cache. Squirrel fully distributes the web caches storage among the browser cache of clients. Hence, when the availability of clients is asymmetric, some clients decrease the total performance of the Squirrel network. Additionally, all contents are distributed and it is hard to manage the objects according to the characteristics of web objects. In our scheme, a web object is assigned to the proxy cache or the client-cluster according to the popularity of the object, which optimizes the overall performance of the proxy cache.

6. Conclusions

In this paper, we propose and evaluate peer-to-peer client-clustering, which is used as a backup storage for the proxy cache. The proxy cache with this client-cluster is highly scalable and more efficient, and has low administrative cost. Even if the clients take the load, this load has been verified on a range of real workloads to be low. Moreover, the utility of the client-cluster can be improved by managing objects according to their properties such as size, popularity and update frequency. We can extend the usage of the client-cluster to other proxy systems. If a proxy performs demanding jobs such as encoding/decoding and complex calculation for many clients, it can use the residual resources of the clients to accomplish these tasks.

References

[1] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell, "A hierarchical internet object cache," Proc. 1996 USENIX Technical Conference, pp.153–163, Jan. 1996.
[2] J. Cohen, N. Phadnis, V. Valloppillil, and K.W. Ross, "Cache array routing protocol v1.0," http://www.ietf.org/internet-drafts/draft-vinod-carp-v1-03.txt, Sept. 1997.
[3] A. Wolman, G.M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H.M. Levy, "On the scale and performance of cooperative web proxy caching," Proc. 17th ACM Symposium on Operating Systems Principles, pp.16–31, Dec. 1999.
[4] J. Wang, "A survey of web caching schemes for the Internet," ACM Computer Communication Review, pp.36–46, Oct. 1999.
[5] L. Fan , P. Cao, J. Almeida, and A.Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," Proc. ACM SIGCOMM 1998, pp.254–265, Sept. 1998.
[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," Proc. 18th ACM Conference on Distributed Systems Platforms, pp.329–350, Nov. 2001.
[7] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," Proc. ACM SIGCOMM 2001, pp.149–160, Aug. 2001.
[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," Proc. ACM SIGCOMM 2001, pp.161–172, Aug. 2001.
[9] 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.
[10] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A decentralized peer-to-peer web cache," Proc. Principles of Distributed Computing 2002, pp.213–222, July 2002.
[11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gumadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," Proc. ACM ASPLOS 2000, pp.190–201, Nov. 2000.
[12] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," Proc. HotOS VIII, pp.75–80, May 2001.
[13] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," Proc. 18th ACM Symposium on Operating Systems Principles, pp.202–215, Oct. 2001.



Kyungbaek Kim received his B.S. degree and M.S. degree 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 Califonia, 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, and IEEE.