Secure and Reliable Decentralized Peer-to-peer Web Cache

Bo Sheng and Farokh B. Bastani
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75080, USA
Email: bxs028000@utdallas.edu, bastani@utdallas.edu

Abstract

Client side caches form a large space for web caching that, if properly utilized, can significantly improve the hit ratio. Conventional peer-to-peer web caching schemes fail to consider several critical issues. Some rely on a centralized proxy server for coordination. Most do not consider the frequent arrival and departure of client platforms. In this paper, we propose a decentralized, peer-to-peer web caching scheme in a corporate network. Without proxy servers, each client shares its local cache contents with the others via structured peer-to-peer routing protocols. We design a new replacement policy specific to peer-to-peer systems and consider security and fault tolerance in our approach. Trace-driven simulations show that our scheme is more efficient, secure, and reliable than the existing approaches.

1 Introduction

Web caching is a well developed scheme for improving the performance of web browsing. Users' requests can be satisfied by the local cache or the cache of a nearby proxy, instead of a remote web server. Performance studies [3,11,22] show that proxy caching is very effective in reducing the response latency of web accesses. Various proxy caching techniques and replacement policies have been proposed to improve the cache hit ratio [1, 4, 12, 14, 22]. Cooperative caching architectures and mechanisms have been investigated to achieve the same goal [5, 15, 21, 23].

One class of the cooperative caching strategies is based on the client-side caching. Generally, web browsers cache web contents on client platforms and these client caches form a large cache space. In a LAN environment, the clients' cache space can be organized and utilized to reduce the total outgoing traffic. A client may get web contents from other clients. Based on this peer-to-peer concept, sharable web caching strategies among the clients have been investigated [9, 19, 24]. Simulation results indicate that sharing cache among clients can significantly decrease the external traffic [24].

However, client-side caching has some fundamental problems. First, the availabilities of the client platforms are unpredictable. A site may be turned on or off frequently. Second, the privacy of user's access activities should be protected. Finally, it has to be efficient with low latency. In [24], a centralized peer-to-peer web caching protocol has been proposed. However, centralized approaches incur a higher management cost, especially in a large domain. They also have the problem of single point of failure. In contrast, peer-to-peer solutions can offer better scalability and reliability. Squirrel [9] is a decentralized peer-to-peer web cache mechanism. It considers two document look-up schemes, home store and directory. However, privacy issues are not considered in either of the look-up protocols. Also, though the home store scheme yields a lower external bandwidth, it incurs a much higher internal traffic. More specifically, the data size transferred between clients is greatly increased. Furthermore, the high hit ratio in this scheme is obtained based on some idealistic assumptions that may not be satisfied in a real environment, such as low failure rate and some necessary operations when nodes join or leave the system. Otherwise, the hit ratio drops steeply.

This paper presents a web caching model based on the decentralized peer-to-peer architecture. We use a hybrid policy that combines existing home store and directory approaches. The combined approach eliminates the problems in the original home store and directory schemes. We develop new algorithms and policies to resolve certain problems that arise due to the combination. A new cache replacement policy based on the hybrid approach has been developed. Unlike the replacement policies in other peer-to-peer systems, our policy specifically considers the peer-to-peer environment. It tries to retain high-hit objects by replicating the directory information. Thus, high-hit objects in client local caches can be used effectively. Coordinated replacement is considered to minimize unnecessary duplications. The hybrid approach also reduces unnecessary data transfers. An object can be kept in the local client cache and referenced through the directory without being actually transferred to the home node. As a result, our approach yields a high hit ratio with a relatively low inner traffic. Moreover, our algorithm considers privacy, anonymity, and security issues. We encode the web contents by symmetric encryption and hide IP information to protect clients' access histories from being traced by other users. Furthermore, reliability of nodes (nodes fail or leave the system) and load smoothing among peers are also considered in our scheme.

The rest of the paper is organized as follows. In the next section, we review the related work in peer-to-peer systems. Our decentralized peer-to-peer caching scheme is presented in Section 3. In Section 4, we present our experimental study setup and performance results. Section 5 concludes the paper and outlines some areas of future research.

2 Background and Related Work

2.1 Peer-to-peer Routing Protocol

Document look-up is a critical function in decentralized peer-to-peer web caching. Traditional systems use centralized directory or flooding models [6, 8, 10]. Recently, some structured document routing protocols have been proposed, such as Pastry [16, 17], CAN [13], Chord [7, 18] and Tapestry [25]. These models provide efficient and scalable 'lookup' functionality to support large-scale applications.

Pastry is a structured self-organizing peer-to-peer protocol. Each node in Pastry has a unique nodeID. The nodeIDs and the keys are assigned in the same 128-bit ID space by a suitable hashing function, such as *SHA*. Given a key, the associated query will be forwarded to a live node whose nodeID is closest to the key. For the purpose of efficient routing, each node maintains a routing table which includes information about a set of other nodes. At each step, a node searches its own routing table and finds a node whose nodeID shares with the key a prefix that is at least one bit longer than that of the current node. A Pastry node also has a leaf set to store the nodes whose nodeIDs are numerically closest to the current nodeID. This set ensures reliable delivery and can be used for replication. Thus, the expected number of forwarding hops in Pastry is around $\lceil \log_{2^b} N \rceil$, where b is a configurable parameter. When nodes join the system, there are some special messages delivered to build the states of new nodes and let others know their presence.

Other ongoing research projects provide similar functionalities by different design views. In Chord [18], nodeIDs are assigned in a circular identifier space. A message is routed to the successor of the associated key k. There are $\log N$ entries in the routing table. The i-th peer is the node that has the smallest nodeID among those nodes whose nodeID is greater than $2^i - 1$. The routing length is thus $\log N$. CAN [13] has a multi-dimensional identifier space. The routing table maintains information about the neighbors in each dimension. Each node is responsible for a set of information and the responsibility can be split and transferred.

We use Pastry as the routing protocol. But we only utilize the function of mapping an application object to the corresponding node. Therefore, our design can be implemented on other peer-to-peer substrates with some minor modifications.

2.2 Peer-to-peer Web Caching

A centralized peer-to-peer caching scheme is proposed in [24]. Each proxy server maintains an index file of all web objects cached by a group of clients. When a proxy cache miss occurs, instead of forwarding the request to upper level proxies or remote servers, the proxy server first checks the index file to see if another client has a fresh cache. If the web object is found in some client's browser cache, it can be transferred to the requesting client directly or through the proxy server. They compared the schemes listed in Table 1, and found that Scheme 3 is superior with highest hit bytes ratio. The simulation results demonstrate the importance of the peer-to-peer sharing of clients' cache contents and the existence of proxy cache. In fact, the hit ratio in the centralized caching with an infinite proxy cache space is the highest we can get from peer-to-peer systems. They also present anonymity and security protection through the proxy server. Briefly, a trusted proxy server plays a critical role in their approach. Our purpose is to build a web caching scheme on a self-organizing peer-to-peer system, where no server is involved. Therefore, we can inherit the design ideas and remove the proxy role from the system. Basically, in peer-to-peer systems, a trusted proxy server provides three functionalities. First, the proxy provides an extra large cache space to store the web objects. As in the traditional proxy caching system, the proxy cache leads to increased hit ratio when a local miss occurs. Secondly, the proxy maintains a directory of a group of clients' local cache contents. If a proxy miss occurs, but the object can be found from another client's local cache, then the requesting client is directed to fetch the data from that peer. Finally, as a transfer tunnel, the proxy can provide anonymous communications for both sender and receiver. Therefore, to design a decentralized system, we have to distribute these jobs among the peers.

Scheme 1	no client cache share, proxy cache
Scheme 2	client cache share, no proxy cache
Scheme 3	client cache share, proxy cache

Table 1. Centralized peer-to-peer web caching schemes.

Squirrel [9] is built on self-organizing peer-to-peer protocols. When a web browser requests an object, Squirrel first checks the local cache. If the object cannot be found, Squirrel will send out a query message with a key that is the hash value of the URL. Then the message will be forwarded to the live node whose ID is closest to the key. This node is called the home node of that web object. If the home node finds a cache miss, it has two approaches to deal with the request. In the home store approach, the home node contacts the original web server to get the object. It then caches the object and sends a copy to the client. In contrast, in the directory scheme, the home node only keeps a directory of the nodes which have the fresh copies of the object. When a request arrives, the home node randomly picks a node from the directory and lets it transfer the object to the requesting client. The home store approach performs better than the directory scheme in terms of hit ratios. As a matter of fact, these two algorithms fall in the category of distributing the proxy's jobs among the peers. In the home store scheme, each node participates in the distribution of proxy spaces. Clients cache the objects that they host but may never access, and share them with other clients. The directory scheme takes care of the distribution of the directories. Every client maintains a directory of objects it hosts and forwards the request randomly. If we reconstruct the centralized proxy caching from these two approaches, the home store method represents scheme 1 and the directory approach yields scheme 2 in Table 1. Our design is based on scheme 3. The proxy server maintains cache space and directory. At the same time, the proxy's responsibility is distributed among the clients. We will show that a combination scheme is more suitable for peer-to-peer systems.

3 Decentralized Web Cache

We use a combination of home store and directory schemes to achieve a better performance. The major advantage of peer-to-peer web cache is the highly increased hit ratio. In the proxy-based peer-to-peer web caching, a cache hit may occur at the local cache, proxy cache, or other client cache. In decentralized caching schemes, we also consider these three types of hits. *Local hit* is always the same in all approaches. In the decentralized approaches, the home node scheme can be used to realize the proxy's caching task. Each object has a home node and it may be cached at the home nodes. Thus, a proxy hit becomes a *home node hit*. The home node of an object also keeps a directory of other clients that have a copy of the object. Thus, a *directory hit* results in getting the object from other client's local cache.

The system we consider consists of a set of clients with limited cache spaces. In the directory scheme, all the space is used to store the local cache and *directory hit* is the major improvement. However, for an active client that requires a large local cache space, many local contents recently accessed will be evicted, leading to a loss of *directory hit*. This problem is mentioned by Squirrel developers [9] also. On the other hand, the home store scheme provides *home node hit* instead of *directory hit*, but it may reduce local hit ratio. Due to home store, each client has to store some objects it hosts and they are treated by the uniform replacement policy. Thus, the space for caching local contents becomes smaller. This problem becomes worse for a high load system, where a node may be the home of many hot web objects. Though the *home node hit* increases, but this may not yield an overall performance gain, because the local hit is a major part in the overall hit ratio.

In our design, we split the client cache space into two parts. Some of the cache space is allocated to the browser, i.e., used as the local cache. The other part of the client cache space is used for the home node scheme. All the objects hosted by the home node are cached in this space. Every object has a directory maintained at its home node. The directory points to the local caches of the clients. Let $psize$ denote the client cache size that is allocated for the home node and $lsize$ denote the local cache size. Both $psize$ and $lsize$ are arguments that can be set by users. They also could be dynamically adjusted for different conditions. Each part of the cache space has its own replacement policy so that *local hit* and *home node hit* do not affect each other. We also use an adaptive scheme to increase $psize$ or $lsize$ when the other part is idle. For example, consider a node that has heavy local cache accesses, but hosts only a few objects or the objects hosted have not been accessed for a long time. In this case, we can increase $lsize$ and reduce $psize$ for more efficient usage of the space. When the home node access frequency is back to a normal level, we can gradually adjust the space allocation to its initial setting.

3.1 Access Request Processing

Let C denote the client that issues a web request and H denote the corresponding home node for the request. The home node also maintains a directory for each object it hosts. Let $D(k)$ represent the directory of the object whose URL hash value is k. Also, assume that node D is one of the delegates that have a fresh copy of the object. The following is the pseudo code for the requesting client and the home node.

Requesting Client C:

```
1:  issue a request $r$(URL);
2:  check local cache;
3:  if local_hit then
4:     retrieve the object;
5:  else
6:     $key$=Hash(URL);
7:     route($key$,$lookup$);
8:     if remote_hit then
9:        get the object from other nodes
10:    else
11:       send $r$ to the original server
12:    end if
13:    cache the object;
14:    ......
15: end if
```

When a client C requests a web object, it first checks its local cache. If the object is not found, a request message will be forward to the home node H. If the home node has cached a copy, it sends the object to C. Otherwise, H randomly pick a delegate D from the directory for the transferring. If the directory is empty, the requesting client will fetch the object from the original server. Finally, the home node can add C into the directory. When nodes join or leave the system, directories are transferred to new home nodes. But it is not necessary to transfer all hosted objects. Only those objects whose directories are empty need to be transmitted.

Home Node H:

```
1:  receive a lookup request $key$;
2:  check local cache;
3:  if key is found then
4:     send remote_hit;
5:     send the object;
6:  else
7:     checks $key$'s directory $D(key)$;
8:     if $D(key)$ is empty then
9:        send remote_miss;
10:    else
11:       select one node $D$ from $D(key)$
12:       while $D$ has no valid copy do
13:          remove $D$ from $D(key)$
14:       end while
15:       if $D(key)$ is empty then
16:          send remote_miss;
17:       else
18:          send remote_hit;
19:          let $D$ transfer the object back to $C$;
20:       end if
21:    end if
22:    cache the object;
23:
24: end if
25: update directory $D(key)$;
26: ......
```

In this design, there are three types of messages. The first one, which is between C and H, is to ask the home node if a fresh copy is available or transfer requested objects from H to C. The second type is between H and D. The home node initializes it when H has no cache of the object and tries to find it via directory. The last one is to transfer web objects from D to C.

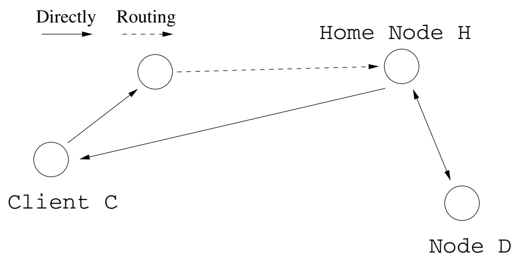

Figure 1. Directory caching scheme.

Obviously, the first message from C to H is delivered by the routing protocol. Since it is a control message, we assume the reply is instantaneous. This message includes source IP information so that H can reply to C directly. Similarly, the second type of message also contains the source H's IP address. The difference is that H knows D's IP information, because D is in H's directory. That means, for the second messages, bidirectional deliveries are instantaneous. In our design, the third type of message is delivered via the home node H. The object is sent to H first and then to C directly. In this way, the home nodes can cache the objects if they need without extra communication and we get the lowest latency(two hops) from D to C.

3.2 Cache Replacement

The cache replacement policy is a key algorithm in caching systems. Many approaches [1, 4, 12, 14, 21, 22] have been proposed. In current peer-to-peer web cache schemes, traditional proxy replacement policies are used. In traditional proxy caching, clients' local actions are not visible to the proxy server. But in our model, the home node can at least partially know the clients' local cache states through the directory. We are able to make the home node and delegates cooperate with each other to get a better performance.

The major goal of our cache replacement policy is to make a hot page stays in the system as long as possible. In the home node cache, we still use the classical LRU policy. When eviction occurs, we first replace older pages. We give a time-to-live (TTL) parameter and remove those pages with expired TTL. This TTL is different from TTL for web objects. If there are no expired pages, we will pick the objects with the maximum number of copies in the whole system. We can obtain this information from the directory of the home node. Replacement of popular pages in the home node cache space does not reduce the hit rate, because the directory hit can make up for the home node miss. When the home node evicts a web object it hosts, it also informs all holders of that object so that they can increase the priority of that object in their local cache. Therefore, the object stays in the clients' local cache longer when it is replaced out of the home node's cache. On the other hand, when a client evicts a hot page, if it is the last copy or one of the last few copies, it sends the page to the home node. By doing so, a directory miss becomes a home node hit.

3.3 Arrival and Departure of Clients

In peer-to-peer system, every node is unreliable. The clients may join or leave the system fairly frequently. In the pure home store scheme, web objects are cached on their home nodes and the requesting clients. However, only the copy on the home node is sharable. If a node fails or leaves, all contents it hosts will be gone. Even if the home node is still alive, some web objects might be replaced. Irrespective of whether the next request for one of these web objects is forwarded to the original or a new home node, a cache miss will occur. Even though there might be fresh copies in some client caches, the request will be forwarded to the web server. This is due to the loss of *directory hit*. The consequence is unnecessary increase in external traffic. Furthermore, when nodes join or leave the system, some objects have to be transferred between peers to make them available at new home nodes. Otherwise, the hit ratio will be decreased. In a highly dynamic system, this transfer operation yields a significant increase in internal traffic. If a client fails or leaves the system without advance notification, then the cached objects will not be forwarded to the new home node. Subsequently, the hit rate will be reduced.

Our approach combines the directory scheme with the home node scheme. So, a home node also maintains and replicates the directories. If a home node fails, the directory information still can be obtained from other nodes. A node can periodically replicate its directories to its immediate neighbors in the name space and some other nodes in the leaf set. From the view of each node, it keeps directories of objects hosted by a small set of nodes. In this way, the access to the referenced node is more reliable.

We also consider the use of more reliable platforms to cache frequently accessed objects. Reliability can be measured based on statistical data. When a home node H identifies an object that is frequently accessed, it can replicate the object on a more reliable platform S and add S to its directory.

3.4 Node Load Smoothing

Since all clients have similar and limited resources, it is important to keep the load evenly distributed among them. The load on each node can be measured in terms of the number of objects served and the size of data transferred. In the directory approach, the number of served objects per second can be bursty. When a client C visits a web page, it probably accesses a set of objects O contained in the page (such as images and audio files). In the directory scheme, C will be referenced by the home nodes for all objects in O since C caches fresh copies of all these objects. When another client visits the same page, the requests for all objects in O will be forwarded to C. It leads to a high load problem. Due to the use of the home store scheme, our approach will not incur this bursty workload problem. The objects in O may be stored on different home nodes and the load for accessing objects in O is distributed. The bursty workload problem will occur only if many of the home nodes of the objects in O evict out the corresponding objects.

A stricter solution for load balancing is to use a reference counter. Each node keeps an upper bound *rlimit*, which represents the maximum number of nodes it can be referenced by. When a home node H wants to add a client C to its directory, H first sends a message to C. C checks its current value of the reference counter. If it is less than *rlimit*, C sends an 'accept' response to H and increases the reference counter. Otherwise, C sends 'deny' response to H. In this design, referring is not a single party operation, but an agreement between a home node and a client node.

```
procedure join directory:
1: if ref_counter < rlimit then
2:     send join message ($k$, IP) to the home node;
3:     ref_counter++;
4: else
5:     send deny message to the home node;
```

```
6:     return;
7:  end if
```

rlimit is a configurable parameter which could be decided by each node based on its own state. Also *rlimit* could be adaptive and adjusted dynamically. Thus we can take advantage of the directory scheme without worrying about the bursty workload issues.

3.5 Privacy and Security Issues

In web caching systems, a node may not want the others to trace its access history. This means, for a requesting client C, physical information (IP address) and the web object (URL or the content) cannot be exposed to other nodes. In the centralized scheme, this could be done by the trusted proxy servers. However, it is more complex in this self-organizing approach. Generally, in a peer-to-peer system, these information appears as either plain contents or the corresponding hash values. Let P be the content of the web object and $(P)_k$ represents the encrypted version of P with key k. The strict model requires that only $(Hash(\text{IP}), Hash(URL)/(P)_k)$ may be known by the other nodes. However, this model incurs some problems. For example, only $Hash(\text{IP})$, not IP, can be stored in the directory. As mentioned previously, the messages between H and D have to be routed by several hops. We use a less strict model, that if $(\text{IP}, Hash(URL)/(P)_k)$ or $(Hash(\text{IP}), URL/P)$ is exposed, we consider that the privacy is still protected.

Here we discuss our approach that satisfies the less strict model. First, when the requesting client C sends the query message, the URL cannot be included. Otherwise, every node in the routing path knows the URL information. As mentioned before, the first message from C to H carries the IP information. Every node in the routing path can get it so that the privacy can be compromised. Actually, the URL information can be removed. For routing purpose, only key k which is the hash value of the URL is needed. Therefore, without the URL, the message can still be forwarded to H, the home node of the web object. In the home store scheme, H needs the URL information, because if a cache miss occurs H needs to contact the original server. However, in our design, C will connect to the original server only if both home node miss and directory miss occur. After that, client C can send a copy to the home node. Therefore, URL is not a necessary information for H in our approach.

The second problem is the storage of the web object. If the home node caches the plain content P, it breaks the rule, since the home node always knows the IP of the requesting client. Therefore, only encrypted version $(P)_k$ can be stored in the home node. The encoding could be done with a symmetric key cipher. We use a certain function on the URL string to generate the key, e.g., the first several characters of the URL. In fact, any simple function works, because the home node only knows the hash value $k = Hash(URL)$ while requesting clients know the original URL. The first requesting client sends $(k, (P)_{f(URL)})$ to the home node H. The subsequent requesting users get this encoded version and they can get the key $f(URL)$ if f is well known or f can be included in the original message to H. Then the requesting client is able to decode $(P)_{f(URL)}$. We assume the conversion of hashing is very hard. Therefore, H cannot get the exact content. In this way, we expose $(\text{IP}, Hash(URL)/(P)_{f(URL)})$ to the home node H and the privacy is protected. If a remote miss occurs and the directory is not empty, H will select a delegate D from the directory and transfer the object from D to C through itself. In this scenario, H knows C's IP address and D caches the wanted web object. Similarly, let D send $(P)_{f(URL)}$ to H and then to the client. The delegate knows nothing about the requesting client.

4 Performance Evaluation

In this section, we use trace-driven simulation to evaluate the performance. We compare our combined approach with the directory and the homes store schemes.

The traces we used are provided by NLANR (National Lab of Applied Network Research) [20]. They have several proxy servers running Squid and record sanitized log files. The following is the summary of accesses on 07/21/03.

	Bo1
Number of nodes	80
Number of requests	281066
Number of distinct requests	188031
Total size	3.266G
Infinite cache size	2.055G
Maxim hit bytes	1.211G

Table 2. NLANR bo1 Trace 07/21/03

4.1 Access Frequency

As mentioned in previous section, our design tries to retain a hot page stay in the system as long as possible. We analyzed the data and found that the access frequencies of web objects follow a Zipf-like distribution as reported in other research [2]. Figure 2 shows the probability for a page of to be accessed in the future, given its historical access frequency. To simplify the problem, if

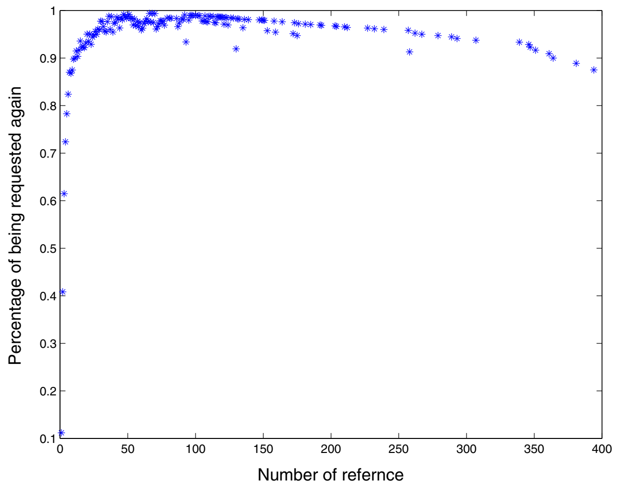

Figure 2. Requests Access

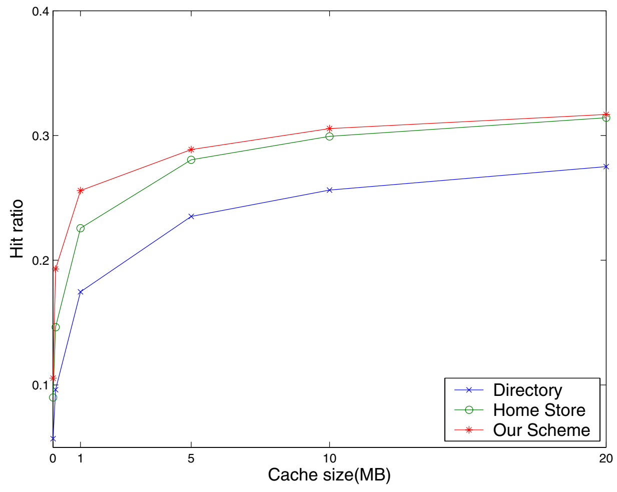

Figure 4. Total data transferred

it has been visited more than once, we consider a web object a hot page. Since these pages have more than 40% probability of being accessed again, they are worth keeping in the system as long as possible.

4.2 Performance

We compare our combined scheme with the home store and the directory schemes. In the experiments, we assume that each node has the same size of cache space and we assign the size to 10K, 100K, 1M, 5M, 10M and 20M. For our approach, we set $psize$ to 30% of the total cache size and TTL to one hour. The following figures present byte hit ratio and hit ratio.

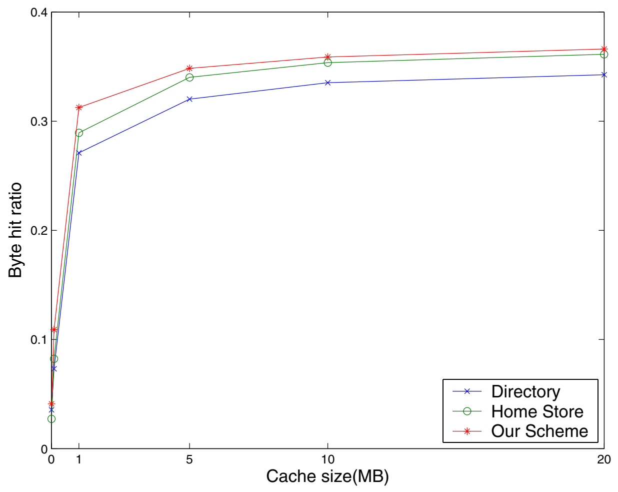

Figure 3. Total data transferred

The results indicate that our approach has a higher hit ratio than the other two, even if the cache space assigned is modest. As discussed previously, in the directory scheme, active nodes often evict fresh pages and the same requests issued later have to be forwarded to remote servers. The home store scheme resolves this problem and yields a better performance, but incurs another problem, namely, that the local cache and the hosted objects cache may compete for the same cache space. We use parameters $psize$ and $lsize$ to address this problem. This distinct space limit prevents them from affecting each other too much. Also, the adaptive arrangement yields a more efficient usage of idle space. In our approach, within TTL, a web object stays much longer in the whole system, either in the home node or some other clients' cache. When a request for a web object P is issued, a hit miss happens only when

1. Every hosted objects cached in the home node is the only copy extant, and

2. Every client which accessed P has evicted P out of its cache space.

Meanwhile, our algorithm has a lower internal data transfer traffic than the home store scheme. Figure 5 is the results of the total data size transferred which includes the external and internal data. Comparing with the home store scheme, our approach reduces about $10\% - 25\%$ data size. The first reason is that we get a higher hit ratio. In the home store scheme, one hit miss leads to double size data transfer, i.e. from the server to the home node then from the home node to the requesting client. In addition, our scheme eliminates some unnecessary home node cache transfers. The home store scheme always caches a missed object both in the client and the home node irrespective of whether this object

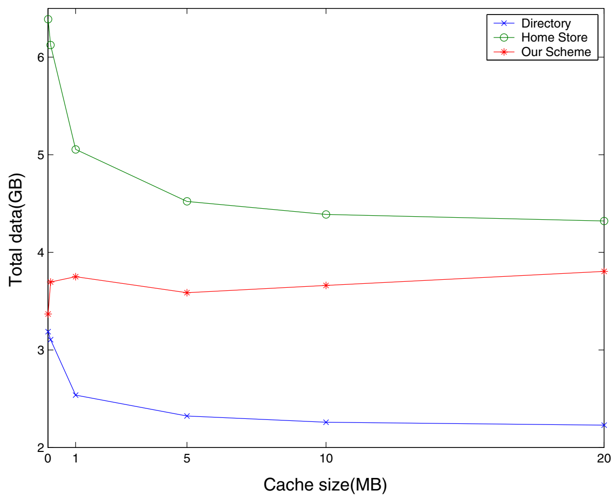

Figure 5. Total data transferred

will be accessed again or not. In our approach, the requesting client obtains the object from the remote server and the home node may reject the cache request from the client. It happens when all objects a home node currently hosts are the only copies of the objects in the system. Hence, many one-time pages are never cached in the home nodes. Also, when a node leaves, we do not have to transfer all hosted objects to the adjacent nodes. Since we have replicated directories, only those one-copy pages need to be transferred. Though our approach imposes extra communications between clients for the replacement policy, the internal traffic is still lower than the home store scheme.

5 Conclusion

We have proposed and evaluated a novel web cache scheme based on the decentralized peer-to-peer structure. Our experimental study shows that it performs better than the existing schemes in term of hit ratio and node load. Meanwhile, we take security issues into consideration and our design is more feasible. Also, this approach is resilient to node failures and can be scaled up for a large community. Finally, it is easy to implement our design on different peer-to-peer routing protocols in a corporate network environment.

Our future research work includes several directions. First, we consider caching large size media objects, such as images and videos. It can bring significant benefit in network traffic reduction if cached copies are located and used. However, it may be necessary to partition these objects and cache them on multiple client platforms to avoid occupying a large cache space of a single client and facilitate load smoothing. We will consider coordinated caching schemes for these large media objects in peer-to-peer caching systems. We will also consider applying peer-to-peer caching schemes to other applications, such as pervasive systems and sensor networks.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. In *WWW-4*, Boston, Dec. 1995.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[3] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details, June 1998.

[4] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[5] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[6] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[7] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Eighth Workshop on Hot Topics in Operating Systems*, May 2001.

[8] Gnutella. http://www.gnutella.com/.

[9] S. Lyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *ACM Symposium on Principles of Distributed Computing*, Monterey, California, USA, 2002.

[10] Napster. http://www.napster.com.

[11] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *Peer-to-Peer Systems: First International Workshop, IPTPS 2002*, pages 178–190, Cambridge, MA, USA, Mar. 2002.

[12] K. Psounis and B. Prabhakar. A randomized web-cache replacement scheme. In *INFOCOM*, pages 1407–1415, 2001.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[14] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[15] K. W. Ross. Hash-routing for collections of shared Web caches. *IEEE Network Magazine*, - 1997.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM Press, 2001.

[19] T. Tay, Y. Feng, and M. Wijeysundera. A distributed internet caching system. In *25th Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, USA, 2000.

[20] N. trace files. http://www.ircache.net.

[21] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 25(9):36–46, 1999.

[22] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Procedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[23] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.

[24] L. Xiao, X. Zhang, and Z. Xu. On reliable and scalable peer-to-peer web document sharing. In *International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, USA, Apr. 2002.

[25] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.