Distributed Caching Strategies in Peer-to-Peer Systems

Guoqiang Gao[†], Ruixuan Li[†*], Weijun Xiao[‡], Zhiyong Xu[§]

[†]School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[‡]Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities, USA
[§]Department of Mathematics and Computer Science, Suffolk University, Boston, USA
Email: ggq@smail.hust.edu.cn, rxli@hust.edu.cn, wxiao@umn.edu, zxu@mcs.suffolk.edu

***Abstract*—Today, P2P system is one of the largest Internet bandwidth consumers. In order to relieve the burden on Internet backbone and improve the user access experience, efficient caching strategies should be applied. However, due to its autonomous nature, a fully distributed caching scheme is very difficult to design and implement. Most current P2P caching approaches are using Client/Server architecture by deploying dedicated proxy servers on the edge of networks. Such architecture is expensive. It also incurs single point of failure and hot spot problems. Furthermore, it violates P2P principle and failed to utilize vast available resources on individual peers.**

In this paper, we investigate the techniques for efficient distributed P2P caching. We propose novel placement and replacement algorithms to make caching decisions. For each object, an adequate number of copies are generated and disseminated on topologically distant locations. Combined with the underlying hierarchical query infrastructure, our strategies relieve the over-caching problems for popular objects, and provide more cache space for other objects. This resolution greatly reduces WAN traffic for P2P applications. We conduct simulation experiments to compare our approaches with several common caching strategies. The results show that our algorithms can achieve higher cache hit rates and superior load balance property.

I. Introduction

For distributed applications, caching is an effective mechanism to reduce the amount of data transmission on the Internet backbone and improve user access experience. For example, in web applications, it plays a critical role for efficient web services and has been widely used in organizations (universities, government agencies, corporations and ISPs). The most popular mechanism used in web caching is dedicated servers. In this approach, organizations deploy dedicated proxy caches at the edge of networks which are close to the clients. In case a client requests an object, if such an object has been requested by another client in the past and has a copy on the proxy, no remote data transmission will be needed. The client can get the object directly from the proxy. This mechanism can also couple with server-side and client-side caching to further improve the performance. There are substantial literatures on web caching techniques [1] [2] [3] have been published.

With the increasing popularity of P2P file sharing applications, research and industry communities have proposed various solutions to adopt web caching techniques on P2P file sharing systems [4] [5]. Most of them use the same approach by deploying dedicated cache servers on the edges of ISP or network boundaries. All incoming and outgoing P2P data packages are intercepted. In case a new request for a cached object comes, the data can be retrieved directly from the cache server instead of the original peer.

*Corresponding author.

However, such benefits come with costs. First, it results in high investment cost, since each ISP has to buy and install expensive dedicated servers. Second, it causes a hotspot and single point of failure problem since all P2P traffic has to go through the proxy. If a system failure occurs, no peer can receive the service. It also results in slow response time if large number of requests come simultaneously. Third, available cache space on the proxy is limited. The objects shared in P2P applications are typically audio and video files ranging from several megabytes to hundreds of megabytes. These objects are much larger than conventional web objects. A proxy server can not hold too many files and the caching benefit can be obtained with a centralized proxy server is dubious. Apparently, such a caching structure violates the principle of P2P networks by using Client/Server model. The scalability/availability is questionable. It is easy to control and manage P2P accesses with a dedicated proxy server, but it fails to utilize vast available storage space on individual peers to achieve satisfactory performance.

Furthermore, P2P workloads have several characteristics which distinguish from web applications. In web applications, file accesses follow Zip-f distribution [6] which represents extreme high access frequencies for the most popular files. Thus, most caching algorithms proposed for centralized proxy servers are popularity based approach. The more popular the object, the higher the number of copies exist in the system. moderate and less popular objects are not given enough consideration. Even if they are cached somewhere, these objects have higher chances to be evicted. Such an approach is not adequate for P2P workload. In P2P file sharing applications, user access behaviour follows Mandelbrot-Zipf distribution [7] which has a flatter header. Thus, the most popular objects do not have as high access frequencies as in web applications. Another difference is the relatively bigger sizes of objects in P2P systems. Apparently, applying web caching techniques directly onto P2P file sharing applications can not yield a satisfactory performance.

We conduct a comprehensive study on distributed caching strategies to be deployed in P2P systems. Assume in these systems, each peer contributes its partial storage space to cache objects. In case a peer P1 issues a query for an object, and the query message passes through another peer P2, and P2 happens to have the copy of that object, a cache hit occurs. Then the object will be served from P2 to P1 directly. In this paper, we propose a hierarchical caching infrastructure and investigate various cache placement and replacement algorithms including probability-based and greedy based algorithms. We observe that most well-known caching strategies allocate too much resources for the most popular objects, such as Least Recently Used (LRU), Least Frequently Used (LFU), and etc. These traditional cache replacement algorithms do not work very well. A popular file could have a large number of cached copies and not all of them will be used for future queries. We design some experiments to compare LFU, LRU and PMRU (Probability-based Most Recently Used, proposed in this paper). we found that the top 5% most popular files occupy 49%, 47%, 29% of the cache

space in LFU, LRU, PMRU respectively. However, the cache hit rates for these files are very similar: 27%, 26%, 26%. We call such a scenario the over-caching problem. The cache space occupied by these useless copies also decreases chances for other files to be cached, and reduces the effectiveness of the cache space. If applying traditional cache algorithms directly on P2P traffic, it results in an over-caching problem for the most popular objects. For other objects, especially moderate popular objects, the current web caching algorithms can only create limited number of copies. In addition, those copies are easily to be evicted in case of a cache full. It greatly reduces caching effectiveness for P2P workload since the access frequency difference in P2P workload is much smaller compare to web workload. To remedy this deficiency, in our design, we compromise and balance the resources allocated for objects with different popularity. We also carefully choose the locations of cached copies for each object by placing them in topologically distant sub-networks using a hierarchical infrastructure. Our simulation results show that by taking all these factors into consideration, our strategies achieve great cache performance improvement over common caching algorithms.

The rest of the paper is organized as follows. In Section II, we present our the underlying routing infrastructure, caching placement and replacement algorithms. In Section III, we discuss and analyze the results obtained from the experiments. Related works are presented in Section IV. Finally, in Section V, we conclude the paper.

II. Caching Algorithms

We aim to design efficient caching algorithms for structured P2P systems. The caching placement and replacement algorithms to be used are highly dependent on the underlying routing algorithms. Most structured systems use Distributed Hash Tables (DHTs). In this section, we describe the underlying routing infrastructure used in our design. We also address cache placement strategy and cache replacement strategy, which are two important aspects of caching algorithms.

A. Underlying Routing Infrastructure

We use Hieras as the underlying routing infrasturece in this paper. Hieras is a multi-layer DHT based P2P routing algorithm. Like other DHT algorithms, all the peers in Hieras system form a P2P overlay network on the Internet. However, Hieras contains many other P2P overlay networks (sub-networks) in different layers inside this global layer P2P overlay. Each sub-network contains a subset of system peers. These sub-networks are organized in such a strategy: the lower the layer of a ring, the smaller the average link latency between two peers inside it. In Hieras, a routing procedure is first executed in the lowest layer P2P sub-network which the request originator is located in, it moves up and eventually reaches the global overlay. In Hieras, a large portion of the routing hops are taken in lower layer sub-networks which have relatively smaller network link latencies. Thus an overall lower routing delay is achieved. Hieras is a routing algorithm like Chord, it does not provide any caching functionality. In this paper, we use Hieras as the fundamental infrastructure to build our own hierarchical caching scheme. We also study the optimal caching policies to be applied on such a hierarchical architecture. More details about Hieras can be found in [8].

B. Cache Placement Strategy

Initially, when a new peer joins the overlay, it builds up necessary DHT data structures to support query and other services. However, its cache space is empty and has no copies of any file. Next, when a query is fulfilled, our system has to make the decision that how many copies of the requested objects should be generated, and which peers along the routing (query) path should be chosen for caching. This work is done by the cache placement algorithm. We introduce two novel algorithms based on Hieras routing. In our algorithms, we take the object popularity, query distribution, and network topology into consideration for object copy distribution. Our goal is to set an adequate number of copies for both popular and unpopular objects, and carefully select a set of peers to store these copies. Our algorithms avoid over-caching problems for the most popular objects. Furthermore, the copies of each file are well disseminated on different parts of the overlay to better serve local requests and reduce global traffic.

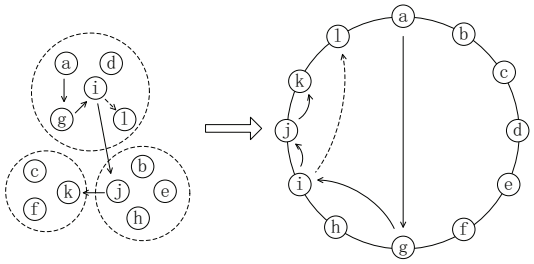

Fig. 1. Instance of searching an object

We also take some heuristic algorithms for comparison purpose. In All Peers (AP) cache placement strategy, the system selects all peers along the query path and creates a copy of the requested object on each of them. While, in Interval Peers (IP) strategy, the system only selects one of every two peers along the query path for object caching. This algorithm can be extended to select one of every three peers and so on. However, the principle is the same as IP. Through theoretical and experimental analysis, we propose two new strategies: Probability-based Sub-network Peers (PSP) cache placement strategy and the Last Sub-network Peer (LSP) cache placement strategy to support our hierarchical infrastructure. In this paper, we use a multi-layer Hieras routing infrastructure. The maximum number of layers is six in our experiments.

1) PSP: In PSP placement strategy, each peer in a sub-network tries to cache the objects locally to support potential queries from other peers in the same sub-network. In the following discussion, we will use "object" and "file" interchangeably to represent a file in P2P networks. In our hierarchical routing, a query is first executed in the sub-network of the lowest layer which contains the query originator peer. If a copy of this object is found within the sub-network, a cache hit happens. Then the query originator can download the object from this nearby peer. If no target can be found in this sub-network, the query is forwarded to the sub-network of higher layers until to the global DHT.

An example of such a query can be seen in Figure 1, and this example is a 2-layer Hieras network. Peer *a* initiates a query for object *k* (or the object is published at peer *k*). At the beginning, it starts searching within the sub-network it belongs to using Hieras routing. After several hops, the query reaches peer *i*. Peer *i* then forwards the query to the peer in the next hop (ie. peer *j*) and a query failure occurs. It means no copy can be found inside the sub-network. Therefore, peer *i* moves to the upper layer global DHT and continue the query process. In each layer, the corresponding routing data structure is used as described in [8]. Eventually, the query will be fulfilled at the peer whose peerid is the closest to the fileid. After the query finishes, we choose only one peer inside the sub-network for caching. The following strategy is used.

We define the sub-network query path as R=$(r_1, r_2, ..., r_n)$. As shown in the example of Figure 1, the query path is (g, i, l). We remove the source peer from the list, and the external peers of the sub-network are not included as well. We specify a factor $f(i)$ for each peer r_i in R, and define weight w_i for peer r_i based on Equation (1).

$$w_i = \frac{f(i)}{\sum_{j=1}^{n} f(j)} \tag{1}$$

According to the peers' weight, we select a peer and cache the target on it. The weight is determined by various factors such as available storage space, network bandwidth, location, and how many

objects are cached on each peer. By adapting different terms, such a strategy allows us to select the most adequate peer for caching. For example, we can always avoid the overloaded peers or choose the peer with the largest free cache space. In our experiments, we set $f(i) = i^2$ (peer location in query) to generate the weights. This strategy can ensure the last peer in the query path has more opportunities to cache the target, and the other peers have chances to cache the object as well. Thus, the system can obtain both good cache hit ratio and balanced load.

2) LSP: In PSP, the peer within the sub-network we choose for file caching is not fixed. It could change from time to time. Such a strategy has a drawback. For example, if a new query for the same object is issued by another peer. Depends on its ID, a different query path is constructed, and some of the peers in the previous query route might not be in this path. If this object is cached on one of those peers, we lose the opportunity to serve the object from a local peer within the sub-network, and it affects the cache effectiveness. To solve this problem, we propose LSP algorithm. In LSP, we always select the last hop peer along the query path for caching. In another word, the peer used to cache a file within the sub-network is determined. Because of the feature of the DHT routing, inside a sub-network, queries for the same object will always converge on this last hop peer. Thus, a high cache hit ratio can be expected. As shown in Figure 1, a copy of object *k* will be cached on peer *i*. However, in LSP, when we make a cached copy, if the cache space on the last hop peer is full, LSP has to evict an existing object, even if there's other peers along the query path in the sub-network has available cache space. This might affect the efficiency of LSP, especially when the allocated cache space on each peer is not big enough.

C. Cache Replacement Strategy

The second algorithm we have to design is the cache replacement strategy. In our system, when a peer is chosen to cache the current object, we examine the available cache space first. If it is full, one or several objects have to be evicted from the current cache to make room for the coming object. Here, cache replacement algorithms make the decision which objects should be replaced.

There are many classical cache replacement algorithms, such as Least Recently Used (LRU), Least Frequently Used (LFU) and Most Recently Used (MRU). LRU evicts the least recently used object in the cache. LFU keeps the most frequently used objects. For both algorithms, the intention is to keep the most popular objects. While MRU replaces the most frequently accessed object in the cache. Its incentive is to keep as many distinct objects as possible with limited space. We also propose our own Greedy-Dual Request (GDR) cache replacement strategy and Probability-based Most Recently Used (PMRU) cache replacement algorithms. We compare them with classical algorithms in our simulations.

1) GDR: LFU keeps those files which are frequently accessed to increase the cache efficiency. However, once the query popularity on an object changes, LFU has to take a long time before expelling the out-dated files out of the cache. It affects the system overall performance. On the contrary, LRU can adapt to the changes in file popularity very well. LRU does not only consider the query frequencies of the objects, but also the recent accesses on each object. However, these two algorithms allocate way too many resources to support the most popular objects, the remaining resources are not enough to support efficient caching for the moderate and less popular files. To combine the advantages of both LFU and LRU and avoid their shortcomings, we design our own cache replacement algorithm for cache re-construction.

Our GDR algorithm is based on the Greedy-Dual algorithm. The original Greedy-Dual algorithm introduced by Young [9] deals with the case when pages in a cache (memory) have the same size but have different costs to fetch them from secondary storage. A good feature of a greedy algorithm is: in the absence of any reference correlations, it can compute the utility value $u(p)$ and sort the objects in decreasing order according to it. Then it tries to keep as many objects as possible following this order. With the reference correlation, the Greedy-Dual Size (GDS) algorithm [10], a very effective web caching replacement algorithm, takes $cost/size$ as $u(p)$, and uses an inflation value L to age the objects. On retrieval or on a hit, the key of an object $H(p)$ is set to $L + u(p)$. On each replacement, L is set to be the value $H(p)$ of the evicted object p.

However, distributed P2P caching has different characteristics. The workload is well distributed on every peer. Thus, we modify GDS algorithm to improve the cache efficiency, and propose our own Greedy-Dual Request (GDR) algorithm. In GDR, we take the request time of files $r(p)$ as $u(p)$. To lessen the weight of $r(p)$, the square root of $r(p)$ is used to replace $r(p)$ as utility value. This strategy allows GDR put more weights on moderate and less popular files, and avoid over-caching on the most popular files. Meanwhile, it can also let us keep the number of copies for the most popular files above a certain level which is enough to support queries. Such a strategy used in GDR improves the cache efficiencies for all files with different popularity. The details of our GDR cache replacement algorithm is depicted as Algorithm 1.

Algorithm 1 GDR

```
1:  L ← 0;
2:  for each request for object p do
3:     if p is in cache then
4:        r(p) + +;
5:        H(p) ← L + √r(p);
6:     end if
7:  end for
8:  if build cache for object p then
9:     if cache has not space to put p then
10:       L ← min{H(q)|q is in cache};
11:       evict the minimum q;
12:    end if
13:    r(p) = 1;
14:    H(p) ← L + 1;
15:    insert p into cache;
16: end if
```

2) PMRU: Although MRU provides good support for the moderate and less popular files, its overall performance is very poor due to the following reason. MRU always removes copies of the most popular objects from the cache. Thus, the system might not have enough cached copies of a popular object. A large percentage of queries are for this popular object can not be fulfilled with a cached copy. Clearly, MRU allocates way too much resource for moderate and less popular files and affect the caching performance for the most popular files.

We propose PMRU algorithm to relieve this problem in MRU by using the replacement strategy based on probability for cache update. PMRU agrees with MRU that it is not necessary to keep too many copies for the most popular files. However, when making the replacement decisions, the system should not always evict the copies of those objects in order to avoid a resource shortage to satisfy the requests. Thus, PMRU evicts an object based on its popularity in the reversed order. To further reduce the chance a popular file is evicted, we take the square root of object popularity to calculate the replacement probability. By this mechanism, we avoid remove too many copies of the most popular objects, and still ensure other objects have more chances to stay in the cache.

In PMRU, let $T = (t_1, t_2, ..., t_n)$ be the set of cache objects on a certain peer, and $F = (f_1, f_2, ..., f_n)$ indicates the popularity of the object *T*. We define the replacement probability P_i of a file t_i in Equation 2. Let $P = (P_1, P_2, ..., P_n)$ indicate the replacement probability of *T*.

$$P_i = \frac{f_i^{\frac{1}{2}}}{\sum_{j=1}^{n}(f_j)^{\frac{1}{2}}} \quad (2)$$

In PMRU, when the system decides to cache a file on a peer, if the cache is full, it computes the query probability P_i for each object t_i. Then PMRU selects one from *T* for eviction based on P_i. The files with higher P_i are more likely to be selected for replacement. The details of PMRU cache replacement algorithm is described in Algorithm 2.

Algorithm 2 PMRU

```
for each request for object t_i do
    if t_i is in cache then
        f_i++;
    end if
end for
if replication for object t_i then
    if cache has not space to put t_i then
        compute P_i for each t_i;
        select one from cache according to P;
        evict the selected ojbect;
    end if
    f_i = 1;
    insert t_i into cache;
end if
```

D. Proactive Clean of Extra Cache

The over-caching problem discussed in Section I will generate a lot of unnecessary cache for the most popular objects. Even the system uses the cache algorithms proposed in this paper, this problem still exists. These cache will take a lot of cache space, which make the other lower popularity objects to not be cached by P2P systems. In order to cache more different objects, we design a proactive clean approach to evict the extra cache. From the previous discussion, we know that the top 5% most popular files occupy 49% of the cache space in the system with LFU replacement strategy, however, the cache hit rates for these files is only 27%. We call rarely visited cache for most popular files extra cache. However, in a distributed environment, each node only has the partial information, so to obtain the object popularity is very difficult. To address this issue, this paper presents a simple solution: a node initiates a query for its cached objects, we call this approach proactive probe, and determine whether the cache for an object is the extra cache or not according to the return value of proactive probe. The node *i*'s proactive probe function for object *R* $proactiveProbe(R)$ is refined in Equation (3).

$$proactiveProbe(R) = hops\ hit\ R \quad (3)$$

If $proactiveProbe(R)$ is lower than a predefined threshold T_p, the object *R* in node *i* is considered as high popularity object. If the request times of the object *R* $request_R$ is below a threshold T_r, the cache for *R* is extra cache and R should be evicted from the cache by node *i*. In our strategy, each peer performs proactive probe and clean extra cache to cache more distinct objects, and we call this strategy proactive clean (PC). In order to effectively improve the cache efficiency, PC is run periodically. Node *i*'s PC algorithm is shown in Algorithm 3.

Algorithm 3 PC

```
for each cached object R do
    if proactiveProbe(R) < T_p and request_R < T_r then
        evict R out of the cache;
    end if
end for
```

III. Performance Evaluation

In this section, we present and analyze the simulation results. We divide our experiments into three major sets. In the first set, we compare DHT and HDHT. The second set compares the caching performance of various placement strategies, including proposed PSP and LSP strategies, together with IP and AP algorithms. Finally, in the third set, we evaluate the efficiency of various replacement strategies, including our GDR and PMRU policies. We also take well-known algorithms such as LFU and LRU for comparison purpose. Before the analysis of experimental results, we will introduce performance metrics.

A. Performance Metrics

We use the following metrics to evaluate and compare the caching performance of various algorithms.

1) Query Delay: We use two terms for query delay comparison: query hops and query distance. The *query hops* is the average number of query hops to find a requested object. Although the query hops can be used to represent the query performance somehow, it is not very accurate. This is because a query Q1 with smaller query hops may contain some hops between distant peers, while another query Q2 with larger query hops only contains hops between nearby peers. This phenomenon may result in a longer query delay in Q1 than Q2. In our experiments, each peer is assigned to a position in Cartesian space. We group peers into sub-networks and use them as the second layer DHT in Hieras. The peers in the same sub-network are much closer to each other than the peers in different sub-networks. Let $Q_P = \{p_1, p_2, ..., p_n\}$ be the query path between peer p_1 and p_n, D_{ij} as the actual network distance between peer p_i and peer p_j. We define the query distance between peer p_1 and peer p_n D_P in Equation 4. To better reflect the actual query delay, in our experiments, we present query delay results in both query hops and query distance.

$$D_P = \sum_{i=1}^{n}\{D_{ij}|p_i, p_j \in Q_P\ and\ j = i+1\} \quad (4)$$

2) Query Cache Hit Ratio: In order to evaluate the cache efficiency for query results, we define query cache hit ratio as the proportion of the queries which result in a cache hit to the total number of submitted queries. Obviously, the higher the query cache hit ratio, the better cache utilization an algorithm can achieve.

3) Load Balance: Load balance is an important metric to measure the effectiveness of the caching algorithm. We define the number of cache hits on a peer p_i as p_{ri}, and we denote the average number of cache hits on all peers as p_r. The variance of the number of cache hits is used as the load balance performance metrics. It is calculated as:

$$\frac{1}{n}\sum_{i=1}^{n}(p_{ri} - p_r)^2 \quad (5)$$

.

Clearly, the smaller the variance, the better load balance an algorithm can achieve.

4) File Cache Hit Ratio: File cache hit ratio is the most important metric to evaluate the efficiency of proposed fully distributed cache. To evaluate its performance, we define the file cache hit ratio as the proportion of the number of file accesses which are served with a cached object to the total number of accessed objects. The larger the cache ratio, the better performance in file caching.

In our experiments, we use above metrics to evaluate our strategies with various network architecture, cache placement and replacement algorithms.

TABLE I
COMPARISON OF DHT AND HDHT NETWORK ARCHITECTURE

	Chord				Hieras			
	placement/replacement				placement/replacement			
metrics	IP/LRU	IP/GDR	LSP/LRU	LSP/GDR	IP/LRU	IP/GDR	LSP/LRU	LSP/GDR
Avg Query Hops	5.65	5.31	5.39	5.12	4.92	4.62	4.54	4.32
Avg Query Distance	2900	2689	2769	2673	691	636	635	612
Query Cache hit ratio (%)	54.9	61.8	64.1	69.6	43.2	50.7	53.4	55.4
Load balance	3119	2431	3480	2915	517	504	408	397
File Cache Hit ratio (%)	51.2	52.9	52.4	54.3	61.7	65.4	72.0	74.5

B. The effect of Network Architecture

To measure the caching performance with different network architecture, we perform the following experiments. We use DHT and HDHT with different placement and replacement strategies. In P2P systems, large files can be divided into a group of files with the same size. Most P2P applications (P2P file sharing and P2P streaming) are using this strategy to distribute data. Therefore, in our experiments, we set the sizes of each object are the same, and denote it as 1. In this set of experiments, the cache size is set to 15. The network size is 10,000, the number of HDHT layer is 3, and the number of objects in the system is 50,000. The popularity of objects follows a Zipf-like distribution. This distribution states that a small set of objects are extremely popular and the rest of the objects have relatively low popularity. For all experiments with HDHT architecture, PC strategy is used in which T_p is set to 3 and T_r = 2. Unless noted, we use the same settings for all the experiments.

In order to conduct a fair comparison for a conventional DHT overlay and a hierarchical DHT overlay. We take IP and LSP as cache placement strategies, LRU and GDR as cache replacement strategies in the experiments because they show better performance than other algorithms. The results are shown in Table I. Combining these strategies can produce four different configurations, together with two different network architecture, we obtain eight results for each metric. The first four columns are the results of DHT architecture, and the next four columns are the results of HDHT.

As we can observe from Table I, HDHT has much better performance in query delay than DHT under all scenarios. In terms of average number of query hops, the smallest number in HDHT category is 83.8 % of the smallest number in DHT. While in terms of average number of query distance, the ratio is only 22.8%. This is because in HDHT, more than half of query hops are completed in lower layer sub-networks, which results in a significant reduction on search distance. Clearly, HDHT shows excellent query performance.

According to the results, we found that DHT has better performance in query cache hit ratio. This is because we take IP and LSP as placement strategy in the experiments. In DHT, because only one global layer overlay exist, these algorithms make the peers near the target peer have higher chances to create copies of the requested objects. Consequently, it results in a higher cache hit ratio on those peers. However, such an approach can easily lead to load imbalance.

As we can observe, Table I also reveals the load balance performance for different caching strategies. Unlike DHT which could keep a copy in a distant peer, HDHT tries to disseminate copies of objects in different sub-networks. Thus, in HDHT, the new coming requests for the same object is likely to be absorbed by the cache located in the same sub-network as the query initiator. The load is well balanced. While in DHT, queries are likely to converge to a small number of peers who have the copy of the requested objects, and result in serious load imbalance. As we can see, the average variance in HDHT is only 15.3% of that in DHT.

Another beneficial consequence of HDHT architecture is that a peer can always find the closest peer who has the copy of the requested object. This is because the query in HDHT is conducted in the lowest layer sub-network first, with high probability, a query will not move up to the global layer if one or several peers in the same lower sub-network have cached copy of the object. In case of a cache hit, the peer who has the copy is in the same sub-network as the query initiator will provide the download service. They are very close to each other. Thus, a large percentage of object transmissions happen inside lower layer sub-networks. As shown in Table I, HDHT has better performance than DHT on file cache hit ratio in all scenarios, and the average improvement in HDHT is 130% of that in DHT. This is because HDHT take PC strategy to cache more distinct objects. Moreover, the available bandwidth in HDHT is much higher than in DHT for a file downloading. The query initiator can retrieve the file much faster in HDHT. Furthermore, it can also reduce the amount of inter-ISP or global Internet traffic since the peers in the same sub-network are likely belong to the same ISP as well. It can decrease the costs ISPs paid for Inter-ISP traffic.

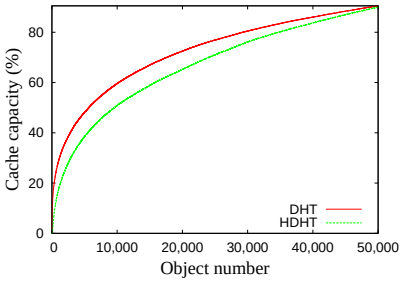

Fig. 2. The cache capacity CDFs for DHT and HDHT

To check the effectiveness of cache size, we draw the Cumulative Distribution Functions (CDFs) of cache capacity for DHT and HDHT. The experiments we conducted use LSP and GDR as placement and replacement algorithms respectively. Figure 2 shows the results. As we can observe, HDHT has better performance which allows it to keep more individual objects. Basically, HDHT will generate less copies for the most popular objects but the number of copies is still enough to serve the user requests. In addition, those copies are well distributed in topologically distant locations. Such a strategy gives the system extra space to keep moderate popular files for better performance. While in DHT, system generate too many copies of the most popular files, and the locations of these copies are not well chosen. It results in load imbalance problem as we observed in previous experiments, and reduce the effectiveness of a fully distributed P2P cache.

C. The Performance of Cache Placement Strategies

In these experiments, we compare and evaluate the performance of various cache placement strategies including AP, IP, PSP and LSP. We use HDHT as the underlying network architecture, GDR as the cache replacement strategy (because it achieves the best performance), and all the other settings are the same as the previous experiments. Table II shows the results of all the four strategies with five metrics evaluated.

The metrics Avg Query Hops (AQH) and Avg Query Distance (AQD) are used to measure the query delay performance. As can be seen from Table II, all the four cache placement strategies achieve good performance. LSP is slightly better than other algorithms in

TABLE II
SIMULATION RESULTS FOR CACHE PLACEMENT STRATEGIES

metrics	AP	IP	PSP	LSP
Avg Query Hops	4.78	4.62	4.35	4.32
Avg Query Distance	672	636	603	612
Query Cache Hit Ratio (%)	44.6	50.7	53.4	55.4
Load Balance	611	504	410	397
File Cache Hit Ratio (%)	51.4	65.4	73.8	74.5

terms of AQH and PSP has the best performance in terms of AQD. AP has the worst results for both AQH and AQD. This is because that AP makes copies of a requested file on all the peers along the query path. For the most popular objects, such an algorithm will create too many copies and occupy too much cache space. Many peers who have a copy of these objects will have no chance to serve requests generated by other peers because those requests are likely served by other peers already. Furthermore, this approach leaves too little cache space available for moderate and less popular objects, and affects the caching efficiency for them. Another issue in AP is that AP will generate too much network traffic as well as disk I/Os. The cache spaces are frequently replaced by new contents, thereby affecting the overall performance. Compared to AP, the caching frequency of IP replacement algorithm decreases by a half, and therefore, we observe better performance. However, it is still much worse compare to PSP and LSP. Among all the four algorithms, LSP builds cache at the last forwarding peer of the sub-network and it ensures a good query efficiency. The probability-based cache placement approach of PSP can also improve the hit ratio in sub-networks, thus reducing the query distance.

As shown in Table II, LSP placement strategy has the best performance in query cache hit ratio, PSP and IP have similar efficiency, while AP is still the least effective. For load balance, PSP and LSP show big advantages over AP and IP. The results again prove that our PSP and LSP algorithms can distribute the copies of objects more evenly onto different sub-networks. Furthermore, PSP and LSP only generate adequate number of copies for each file. For each individual file, the number of copies to be created is not only related to its popularity, but also related to the network architecture, such as the number of sub-networks. Such a well balanced object distribution mechanism results in a much higher file cache hit ratio in PSP and LSP compare to AP and IP. From this point, we conclude that choosing an adequate number of copies for each object is critical in the design of a fully distributed P2P cache.

To evaluate the efficiency of the four placement strategies for moderate and less popular objects, we also compute CDFs for cache capacity like we did in previous experiments. In the experiments, we take HDHT as network architecture and GDR as replacement strategy. Figure 3 shows the results. We can observe that PSP has the best performance. PSP uses 48% overall cache space to store top 10,000 popular files, while AP and IP use 61% and 59% space to cache the same files respectively. This is because the probability-based placement policy of PSP plays an important role. AP and IP algorithms allocate too much cache space for the most popular objects, and there's little room left for moderate and less popular objects. In LSP, when choosing the peers used to cache an object, it is always fixed to the last hop peer. If the cache space is full, we have no choice but evict an existing object even in case another nearby peer has available cache space. While in PSP, we have much more freedom to choose the peer to cache an object. Thus, PSP displays the best cache capacity performance.

D. The Efficiency of Cache Replacement Strategy

We also perform the experiments to analyze various cache replacement strategies including LFU, LRU, PMRU and GDR. Again, we use HDHT as the underlying network architecture, and PSP as the cache placement strategy. All the other settings are the same as previous experiments. Table III compares four strategies with five metrics.

TABLE III
SIMULATION RESULTS FOR CACHE REPLACEMENT STRATEGIES

metrics	LFU	LRU	PMRU	GDR
Avg Query Hops	4.75	4.84	4.80	4.33
Avg Query Distance	675	693	694	621
Query Cache Hit Ratio (%)	46.6	44.7	48.7	55.5
Load Balance	460	416	472	396
File Cache Hit Ratio (%)	66.3	68.8	89.9	74.3

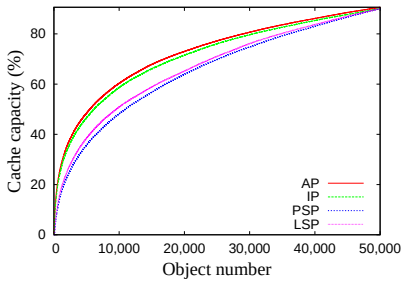

Fig. 3. The cache capacity CDFs for different placement strategies

As we can observe, our GDR replacement strategy has the best performance in terms of query delay. For the other three algorithms, LFU represents the best performance. This is because our queries are generated according to object popularity. Once the query popularity changes, LFU results in a worse performance which will be discussed in the following experiments. In terms of query cache hit ratio, GDR has the best result and PMRU has the worst. We believe the reason behind this fact is that PMRU removes popular files in case of a cache full problem occurs. It affects the cache hit ratio, especially for queries of popular objects. Fortunately, this difference among PMRU, LFU and LRU is not great. The rationale behind PMRU is to avoid the excessive number of copies for the most popular objects. On the other hand, because PMRU does not provide too much resources for these files, it alway has enough cache space for moderate and less popular objects, which results in higher overall file cache hit ratio. As we can see from Table III, the file cache hit ratio with PMRU is 87.9%. It is much higher than any other algorithms. This proves that it might be a good choice for Mandelbrot distribution P2P traffic where not too much very hot files exist. For load balance property, LRU displays the best performance and our GDR is in the second place. However, LFU and PMRU also has good results.

In summary, we believe that the cache placement policy has higher influence than cache replacement policy in fully distributed P2P cache. As we can observe from Table III, because we used PSP cache placement algorithm, all the four cache replacement algorithms show reasonably good performance on all the metrics.

To evaluate the performance of the four replacement strategies for unpopular objects, we also compute CDFs for cache capacity like we did in previous experiments. In the simulation, we use HDHT as network architecture and PSP as placement strategy. Figure 4 shows the results. PMRU only uses 45% overall cache space to store top 10,000 popular files compared to the worst 63% in LFU. Clearly, PMRU has the best performance since it tends to keep as many different objects as possible.

E. The Optimal Number of Layers

To study the cache performance under different number of layers in HDHT, we design experiments to assess the average query hops and load balance properties. The network size is 10,000, the maximum number of layers is 6, and the cache size is 10 on each peer. All the other configurations are the same as previous experiments. Figure 5 shows the results of the combination of PSP placement and GDR replacement under different layers. When we increase the number of layers, the average query hops decrease sharply until it reach to

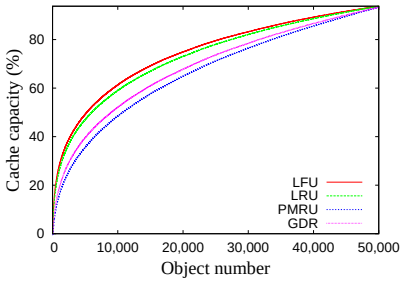

Fig. 4. The cache capacity CDF for different replacement strategies

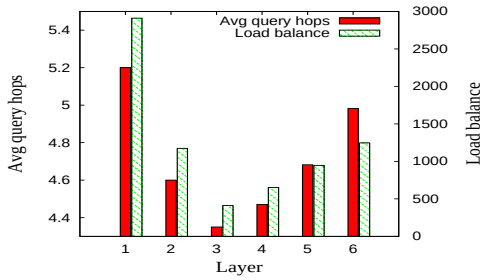

Fig. 5. The optimal number of layers

four. When the number of layers is one, both avg query hops and load balance are the worst. This is because the network architecture in this situation is DHT which only has a global layer. The network architectures in which the number of layers is bigger than one are all HDHT. When the number of layers is small, such as two, there are too many peers in each sub-network and this situation will result in higher query delay. However, when the number of layers is greater than a certain value, the sub-networks of lower layers contain a small amount of peers. Searching in these sub-networks is always failed. This is why the query delay become worse when the number of layers is larger. To achieve the best performance, we have to choose a right size. As we can see from Figure 5, a 3-layers HDHT system represents the best performance.

Also shown in Figure 5, at first, the load balance performance is worse in an overlay with a smaller number of layers than in an overlay with a larger number of layers. This is because the smaller the number of layers, the smaller number of copies we can generate for an individual object, and load imbalance problem can easily happen. As more sub-networks are used, load balance is improved. We found that the system with 3-layers has the best load balance performance. We found, under such a circumstance, we can achieve a good compromise between the maximum number of copies for each object as well as the number of queries which could reach a certain copy. After that, as more sub-networks are generated with the increase of layers, too many copies for an object will be generated and as we discussed before, it will reduce the effectiveness of our caching algorithms. Clearly, choosing the right number of layers could have great impacts on the system overall performance.

F. Different Cache Size

We perform experiments to analyze the performance under different cache sizes. We run the simulation for LSP, PSP, LFU, GDR with HDHT architecture, and the settings are same as previous experiments. Figure 6 shows the results for the combination of LSP/PSP placement and LFU/GDR replacement algorithms.

As can be seen from Figure 6(a), as the cache size increases, the query delay becomes smaller for all scenarios. The reason is that more requested objects can be found during the query processing, and the query path is shortened. Figure 6(b) shows that the larger the cache size, the higher query cache hit ratio we can achieve. Obviously, this is because we have rooms to cache more distinct objects. However, the performance difference between cache size 25 and cache size 30 is very small. It indicates that total cache capacity for size 25 on each peer is large enough to hold almost all query traffic in our current configuration. No further improvement can be obtained by unceasing the cache size. As shown in Figure 6(c), load balance is mainly determined by the caching algorithms we use. The cache size has little influence on load balance performance. Overall, the combination of LSP and GDR achieves the best performance.

TABLE IV
COMPARISON OF PC AND NO-PC

metrics	PC	no-PC
Avg Query Hops	4.35	4.82
Avg Query Distance	603	669
Query Cache Hit Ratio (%)	53.4	69.1
Load Balance	410	415
File Cache Hit Ratio (%)	73.8	53.2

G. Analysis of Over-caching Problem

In this section, we implement some experiments to evaluate system performance about over-caching problem. As previous discussion, hot resources will occupy a lot of cache space and many established caches have rarely been used. This makes many other low popularity objects have no chance to be established cache, thus reduces cache efficiency of the system. In this paper, we propose proactive clean (PC) to eliminate those unnecessary cache for hot objects. We use PSP as placement strategy and GDR as replacement strategy to evaluate PC performance. In the simulation, we set T_p to 3, T_r = 2. The experimental results are shown in Table IV. As we can see, taking PC strategy is better than not using PC in all metrics. PC has the biggest improvement compared to no-PC in file cache hit ratio, and it caches more 20.6% distinct objects. The extra cache cleaned by PC is rarely used, so it will not affect system performance. The space cleaned by PC can store the cache for other objects, thus this strategy improves query delay and query cache hit rate. We can see these improvements in Table IV.

IV. RELATED WORKS

To relieve the burden imposed by P2P traffic, design and implement an effective caching infrastructure in P2P systems attracted great interests from both industry and academia [11] [12]. However, it is difficult due to the unique features such as self-governing, and dynamic membership, large number of peers, and even larger amount of shared files in P2P applications.

A. Wierzbicki et al. discussed different features of P2P traffic and web traffic in [13]. They proposed specialized cache replacement policies such as MINRS and LRSB. They also conducted simulation experiments to evaluate the performance of various replacement policies for fasttrack traffic. However, they still used the same architecture as web caching by only considering caching policies for a dedicated proxy server (providing cache services for peers within the boundary of an Intranet). The clients themselves are not participated in the caching service.

In [7], O. Saleh et al. conducted a passive measurement study on Gnutella file sharing network, and showed that the popularity of P2P objects can be modeled by a Mandelbrot-Zipf distribution. They examined the impacts, and found that relying on object popularity alone may not yield high hit rates/byte hit rates. Finally, they designed a new caching algorithm for P2P traffic that is based on segmentation, partial admission and eviction of objects. Their work is very helpful to understand P2P workload characteristics. However, it is still considering a dedicated approach instead of a fully distributed scenario.

In [5], M. Hefeeda et al. proposed similar idea by deploying caches at or near the boarder of the ASs (autonomous systems), pCache will

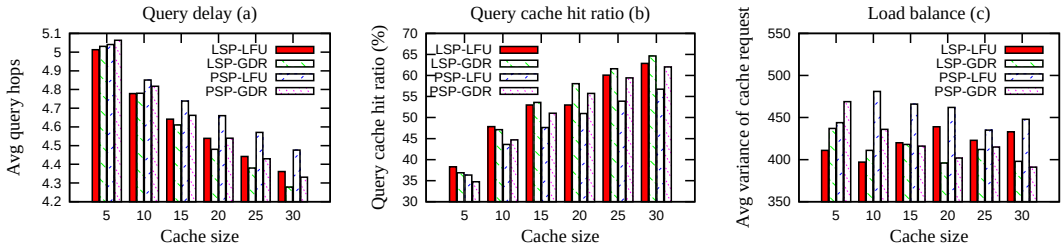

Fig. 6. The Simulation Results for cache strategies with different cache size

intercept P2P traffics go through the AS, and try to cache the most popular contents. The cache size is relatively small, and the objects in P2P applications are very big, so the effectiveness of this approach is doubtable. Furthermore, pCache itself became a bottleneck and a single point of failure which affect its efficiency. To solve this problem, G. Dan proposed collaborating relay caches among ISPs in order to minimize the peer-to-peer traffic costs and reduce WAN traffic [14]. Their concentrations are focus on cache coordination, few discussion are related to exploiting the resources on ordinary peers.

In PROD [15], we proposed a novel and efficient algorithm to improve the file retrieving performance in DHT based overlay networks. In PROD, when a file or a portion of a file is transferred from a source peer to the client, instead of creating just one direct link between these two peers, we build an application level connection chain. Along the chain, multiple network links are established. Each intermediate peer on this chain uses a store-and-forward mechanism for the data transfer. Thus, it can greatly reduce the user perceived retrieving performance. PROD can be combined with the caching strategies to further improve the performance.

Squirrel [16] is a fully decentralized, peer-to-peer web cache. Web caching workloads are taken by all the clients and the dedicated proxy server is eliminated. The key idea in Squirrel is to facilitate mutual sharing of web objects among client nodes. It enables web browsers on desktop machines to share their local caches, to form an efficient and scalable web cache, without the need for dedicated hardware and the associated administrative cost. It uses a self-organizing, peer-to-peer routing substrate called Pastry [17] for its object location service, to identify and route to nodes that cache copies of a requested object. In Squirrel, each node performs both web browsing and web caching. However, it is not a caching algorithm for P2P traffic.

V. Conclusions

Caching techniques are widely used to boost the performance in large-scale distributed applications. However, as one of the most bandwidth consuming applications on the Internet, not enough efforts have been conducted on P2P caching. In this paper, we propose novel and effective cache placement and replacement algorithms for P2P caching. Unlike previous works, our algorithms can be applied to build a fully distributed cache in P2P systems. We use a hierarchical query infrastructure to select an adequate number of cached copies for the object with different popularity and determine the locations of these copies carefully in order to improve cache effectiveness. To the best of our knowledge, this is the first work to address these issues using this approach. We compare our design with various common and heuristic caching algorithms by conducting extensive simulation experiments. We observe that combined with the hierarchical query infrastructure, our caching strategies can deliver lower query delay, better load balance and higher cache hit ratios. Our algorithms effectively relieve the over-caching problems for the most popular objects and offer satisfactory caching performance for other types of objects.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grants 60873225, 60773191, 70771043, National High Technology Research and Development Program of China under Grant 2007AA01Z403, Natural Science Foundation of Hubei Province under Grant 2009CDB298, Wuhan Youth Science and Technology Chenguang Program under Grant 200950431171, Open Foundation of State Key Laboratory of Software Engineering under Grant SKLSE20080718, Innovation Fund of Huazhong University of Science and Technology under Grants 2010MS068 and Q2009021.

References

[1] G. Barish and K. Obraczka, "World Wide Web Caching: Trends and Techniques," May 2000.

[2] L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," *IEEE/ACM Trans. on Networking*, vol. 8, no. 2, pp. 158–170, 2000.

[3] M. Busari and C. L. Williamson, "On the Sensitivity of Web Proxy Cache Performance to Workload Characteristics," in *Proceedings of IEEE INFOCOM*, (Anchorage, AL), pp. 1225–1234, 2001.

[4] PeerApp, "http://www.peerapp.com."

[5] M. Hefeeda, C.-H. Hsu, and K. Mokhtarian, "pcache: A proxy cache for peer-to-peer traffic," in *SIGCOMM 2008*, p. 539, ACM, August 2008.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 126–134, 1999.

[7] O. Saleh and M. Hefeeda, "Modeling and caching of peer-to-peer traffic," in *ICNP '06: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, (Washington, DC, USA), pp. 249–258, IEEE Computer Society, 2006.

[8] Z. Xu and Y. Hu, "HIERAS: A DHT-Based Hierarchical Peer-to-Peer Routing Algorithm," in *ICPP*, (Kaohsiung, Taiwan), October 2003.

[9] N. E. Young, "On-line caching as cache size varies," in *SODA*, pp. 241–250, 1991.

[10] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms," in *Proc. of the 1997 USENIX Symposium on Internet Technology and Systems*, 1997.

[11] T. Stading, P. Maniatis, and M. Baker, "Peer-to-peer caching schemes to address flash crowds," in *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, (Cambridge, MA), pp. 203–213, 2002.

[12] M. Sanchez-Artigas, P. Garcia-Lopez, and A. G. Skarmeta, "On the relationship between caching and routing in dhts," in *WI-IATW '07: Proceedings of the 2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops*, (Washington, DC, USA), pp. 415–418, IEEE Computer Society, 2007.

[13] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache replacement policies revisited: the case of p2p traffic," in *CCGRID*, pp. 182–189, 2004.

[14] G. Dan, "Caching and relaying strategies for peer-to-peer content delivery," tech. rep., School of Electrical Engineering, KTH, 2007.

[15] Z. Xu, D. Stefanescu, H. Zhang, L. Bhuyan, and J. Han, "Prod: Relayed file retrieving in overlay networks," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, (Miami, FL), pp. 1–11, April 2008.

[16] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A Decentralized Peer-to-Peer Web Cache," in *Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC)*, (Monterey, CA), pp. 213–222, July 2002.

[17] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, (Heidelberg, Germany), pp. 329–350, Nov. 2001.