Computers and Electrical Engineering 40 (2014) 688–703

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Distributed caching in unstructured peer-to-peer file sharing networks ☆

Guoqiang Gao [a,b], Ruixuan Li [a,*], Heng He [a], Zhiyong Xu [c]

[a] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[b] School of Media and Communication, Wuhan Textile University, Wuhan, China
[c] Department of Mathematics and Computer Science, Suffolk University, Boston, USA

ARTICLE INFO

Article history:
Available online 6 January 2014

ABSTRACT

Nowadays, the peer-to-peer (P2P) system is one of the largest Internet bandwidth consumers. To relieve the burden on Internet backbone and improve the query and retrieve performance of P2P file sharing networks, efficient P2P caching algorithms are of great importance. In this paper, we propose a distributed topology-aware unstructured P2P file caching infrastructure and design novel placement and replacement algorithms to achieve optimal performance. In our system, for each file, an adequate number of copies are generated and disseminated at topologically distant locations. Unlike general believes, our caching decisions are in favor of less popular files. Combined with the underlying topology-aware infrastructure, our strategy retains excellent performance for popular objects while greatly improves the caching performance for less popular files. Overall, our solution can reduce P2P traffic on Internet backbone, and relieve the over-caching problem that has not been properly addressed in unstructured P2P networks. We carry out simulation experiments to compare our approaches with several traditional caching strategies. The results show that our algorithms can achieve better query hit rates, smaller query delay, higher cache hit rates, and lower communication overhead.

1. Introduction

The peer-to-peer (P2P) application is one of the killer applications, which contributes a large portion of Internet traffic. According to a Cisco [1] report, although online streaming applications have overtaken P2P systems and became the No. one bandwidth consumer on Internet, P2P file sharing systems still accounted for 39 percent of Internet traffic at the end of 2009. P2P systems are now carrying 3.5 exabytes traffic each month and this number will continue to increase by 16 percent each year until 2014. Clearly, reducing P2P traffic is still of great importance. Most popular P2P applications, e.g., FreeNet [2], Edonkey [3], Gnutella [4], and BitTorrent [5], are unstructured overlay networks since they only enforce minimal constraints on the network topology. Compare to structured P2P infrastructures, such as Chord [6], CAN [7] and Pastry [8], unstructured P2P can reduce maintenance overhead substantially, which is critical to achieve scalability in a large-scale and highly dynamic environment. On the other hand, since no routing and searching infrastructure is built, a peer has virtually no knowledge about which peers have the answer to a special query. Therefore, the query mechanism for a file is exhaustive more or less. Normally, it starts from an initiating peer and spread over the overlay network either through a breadth-first style (e.g., broadcast in Gnutella) or a depth-first manner (e.g., the sequential search in FreeNet). In practice, a TTL (Time-To-Live) is set to limit the search, and avoid flooding the entire network. However, these searching algorithms

☆ Reviews processed and recommended for publication to Editor-in-Chief by Guest Editor Dr. Jia Hu.
* Corresponding author. Tel.: +86 2787544285.
E-mail addresses: ggao@wtu.edu.cn (G. Gao), rxli@hust.edu.cn (R. Li), willam1981@gmail.com (H. He), zxu@mcs.suffolk.edu (Z. Xu).

0045-7906/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compeleceng.2013.12.001

are inefficient and unscalable. For an unpopular object, a large number of query messages will be generated, and the system might not be able to find the answer before the TTL expires. Some strategies based on heuristics have been proposed to improve this performance [9–11], however, the poor search efficiency problem still exists.

Our former study [12] observed that, in unstructured P2P networks, it is actually very effective when queried files have high popularity. The popularity of a file is defined as the proportion of the number of copies in the network to the network size (the total number of peers). For a particular file, the number of its copies will directly affect the search efficiency because that the search mechanism used in unstructured P2P networks is a blind random probe. Thus, the proposed solution aims to generate more copies and increase the popularity of files to improve search and retrieve performance, especially for original less popular files.

Caching has already been proven to be an effective mechanism to reduce the amount of data transmission on Internet backbone. For example, there are substantial work have been conducted on web caching techniques [13,14]. With the increasing popularity of P2P applications, research and industry communities have proposed various solutions to migrate web caching techniques to P2P systems [15,16]. Most of them use the similar approach as web caching by deploying dedicated servers on the edges of Internet Service Provider (ISP) or network boundaries. However, such an approach has several drawbacks. First, it has high cost, since ISPs have to invest on expensive dedicated servers. Second, it causes a hotspot and a single-point-of-failure problem since all P2P traffic has to go through the proxy. Actually, we can utilize P2P architecture to solve this problem. Third, the available caching space on the proxy is limited, and the files in P2P applications are much larger than web objects. Thus a proxy server cannot hold large quantities of files and the caching benefit can be obtained with a centralized proxy server is dubious. Furthermore, the vast idle space on general peers is not utilized.

In [17], we investigate the techniques for efficient distributed caching algorithm in structured peer-to-peer systems to relieve these problems. In this paper, we propose another novel distributed caching algorithm to improve the overall caching efficiency and decrease the communication overhead in unstructured peer-to-peer file sharing networks. We build a distance-based network infrastructure with preference-based three-way random walk and investigate various caching placement and replacement algorithms including probability-based and greedy-based algorithms. We observe that traditional strategies, such as Least Recently Used (LRU), and Least Frequently Used (LFU), do not work very well and allocate too much space to the most popular files. If they are applied to distributed applications directly, a popular file could have a large number of cached copies and not all of them are used to serve future queries. The proposed solution addresses this issue with an effective mechanism.

After launching the experiments to compare LFU, LRU and HPLR (High Popularity and Least Request, proposed in this paper), we found that the top 5% most popular files occupy 56%, 55%, 31% of the caching space in LFU, LRU, HPLR respectively. However, the caching hit rates for these files are very similar: 33%, 34%, 32%. Clearly, in LFU and LRU algorithms, storage space is not well utilized. We call such a scenario over-caching. In short, each peer has the caching buffer in a distributed environment. The over-caching problem is that the popular files are cached by too many peers, while the system does not need to build so many caches for popular files for good efficiency in fact. Furthermore, because the overall caching space is limited, the over-caching will make the other files (not very popular) to lose cached opportunity. If the traditional caching algorithms are applied directly on P2P traffic, over-caching problems for the most popular files will occur in general. The caching space occupied by useless copies decreases chances that the other files are cached, and reduces the effectiveness of the caching. For other files, under such strategies, only limited number of copies can be created, including the files with moderate popularity. In addition, those copies are easily to be evicted in case that the cache space is full.

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. The results of our simulation show that, by taking these factors into consideration, our strategies achieve better performance over other caching algorithms.

In summary, compared to the previous works, our contributions mainly focus on:

- A novel distributed caching algorithms in unstructured peer-to-peer file sharing networks.
- A distance-based network infrastructure with preference-based three-way random walk.
- Probability-based and greedy-based caching placement and replacement algorithms.
- Compromising and balancing the resources allocated for objects with different popularity to remedy the over-caching problem.

The rest of the paper is organized as follows. In Section 2, we present the related works. In Section 3, we introduce the underlying network infrastructure, especially the query mechanism used in our distributed P2P caching algorithms. In Section 4, we describe our caching placement and replacement algorithms. In Section 5, we discuss and analyze the experimental results. Finally, in Section 6, we conclude the paper and lay out the future work.

2. Related works

To relieve the burden imposed by P2P traffic and improve search performance, designing and implementing an effective caching infrastructure in P2P systems attract great interests from both industry and academia [18–21]. However, it is a very

challenging topic because of unique characteristics such as self-governing, dynamic membership, large number of peers, and even larger amount of shared files in P2P applications.

IAC [22] is an interest-aware caching for unstructured P2P networks. In IAC, Each peer advertises its resource list to other peers. If a peer is interested in the resource advertisement received, it then caches the advertisement. IAC uses the probabilistic broadcast algorithm to propagate the advertisement. Although this strategy makes a great improvement compared to flooding, it still results in a great deal of communication overhead for maintaining the caching. Some studies [23,24] use Bloom Filter to index the resources of the peers, and take the gossip protocol [25] to exchange the index among the peers in order to reduce network traffic.

PROD [26] 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, PROD builds 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 will greatly enhance the user perceived retrieving performance. PROD is also useful to design the caching in unstructured P2P networks.

In [15], Hefeeda et al. proposed similar idea by deploying caches at or near the borders of the ASs (autonomous systems), where pCache will intercept P2P traffics going through the AS, and try to cache the most popular contents. This mechanism is suit for small web objects, while the size of files in P2P applications is very big generally. Therefore, the effectiveness of this approach is doubtable. Furthermore, pCache itself may become a bottleneck and a single point of failure which affect its efficiency. Shen et al. proposed a HTTP-based Peer-to-Peer (HPTP) framework [27]. The key idea is to exploit the widely deployed web caching proxies of ISPs to trick them to cache P2P traffic. This is achieved via HTTPifying segment large P2P files or streams into smaller chunks, encapsulate and transport them using the HTTP protocol so that they are cacheable. Such an approach avoids the investment for new proxy caches but failed to utilize the distributed peer resources.

In [28], Shen presents an Efficient and Adaptive Decentralized file replication algorithm (EAD) that achieves high query efficiency and high replica utilization at a significantly low cost. EAD enhances the utilization of file replicas by selecting query traffic hubs and frequent requesters as replica nodes, and dynamically adapting to non-uniform and time-varying file popularity and node interest. We study replica strategies for the rare objects in P2P networks in [12], and the proposed proactive probe can be used in our network design in this paper. In the study [17], we investigate the techniques for efficient distributed caching in structured P2P systems, which includes placement and replacement algorithms to make caching decisions. The resolution can reduce the network traffic for structured P2P applications. In this paper, we propose another novel distributed caching algorithm to improve the overall caching efficiency and decrease the communication overhead in unstructured P2P file sharing networks.

3. Underlying network infrastructure

Network topologies and search algorithms, which play the key roles for the caching strategies, will be discussed in this section. We perform our presentation form overlay topology infrastructure, search mechanisms, and peer join/leave strategies in unstructured P2P networks.

3.1. Distance-based network topology

A P2P network is an overlay network where two adjacent peers in it may be far away from each other in the physical networks. The longer the distance between two peers is, the larger the transmission latency is. In order to improve the search efficiency, we propose a peer clustering technique, which groups the adjacent peers in the physical topology together to form multiple sub-networks. Such network topology is called Distance-based Network Topology (DNT). Fig. 1 is an example of DNT. As we can see, the peers in a same sub-networks keep more edges between each other. There are also less connection links between sub-networks, that ensure the connectivity of the entire overlay network. If a query is hit in the cache of its initiator's sub-network, it will result in a smaller subsequent transmission overhead. If a query cannot be solved in its initiator's sub-network, it can be routed to other sub-networks and the search continues. Furthermore, it is easy to apply DNT in real networks to improve efficiency without destroying the original topology because it is a logical topology.

In unstructured P2P networks, it is difficult to guarantee a peer is always classified to the most appropriate sub-network in terms of its physical location. Therefore, some physically close peers may belong to different sub-networks, rather than stay in the same one. In the later section, we will discuss peers join/leave and sub-networks creation mechanism in detail to understand how to form the sub-networks.

3.2. Preference-based three-way random walk

In order to reduce query delay and data transmission overhead, the optimal strategy is to, for a query, answer it in its initiator's sub-network. If the target cannot be found in itself sub-network, the query has to be forwarded to other sub-networks. Given that a query is further spread to other sub-networks only after it searches all the peers in its initiator's sub-network, which will generate large number of flooding messages and result in high cost. In this paper, we propose

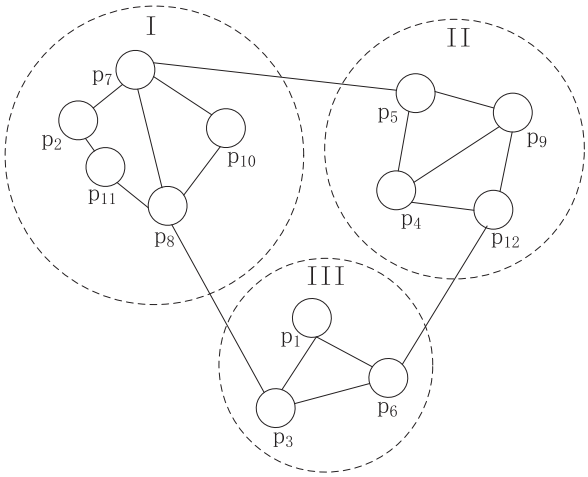

Fig. 1. An example of DNT.

the preference-based three-way random walk (PTRW) algorithm to relieve this issue. In PTRW, a peer forwards the query to only two of its neighbor peers in the same sub-network and one peer in another sub-network. This ratio can be adjusted if necessary. We define routing table structure as shown in Table 1. Each peer has two types of neighbors, where the peers that are relatively close in terms of physical distance are called local neighbors, and all the other peers are called remote neighbors. If a peer receives a query and cannot fulfill the request, it will select two peers from its own local neighbor table and one from its remote neighbor table to forward the query. For a peer, its routing table is updated periodically in order to ensure successful forwarding.

With this strategy, although only one remote neighbor is chosen to forward the query in each round, a query might still be forwarded to a large number of remote peers if there are no proper control strategies enforced. This is contradictory to our basic idea that queries should be processed by local peers for better caching performance if possible. To relieve this issue, we use the following strategy. The forwarding times (FT) with *TTL* of the standard three-way random walk (STRW) is defined as $FT(TTL) = \sum_{i=0}^{TTL} 3^i$, where *TTL* is the maximum query hops. Suppose that a query does not result in a hit until its *TTL* is over, forwarding times for remote peers (FTR) of STRW are defined in Eq. (1) which is quantified by Theorem 1. For example, suppose $TTL = 7$ and the query does not result in a hit (i.e., each querying branch will run out its *TTL*), 3025 remote peers will be searched which account for 92% of the total number of forwarded messages. Moreover, even if the query hits in the local sub-network, it cannot prevent the query messages from being forwarded on the remote peers, that will generate enormous communication cost.

Theorem 1. *For a query, assume all forwarding branches of the query do not stop until its TTL is equal to 0, and the remote peers do not forward the query back to the sub-network of its initiator. The number of message forwarded on remote peers (FTR) of standard three-way random walk (STRW) is defined in Eq. (1), where TTL is the maximum query hops.*

$$FTR(TTL) = \sum_{i=1}^{TTL} 2^{i-1} FT(TTL - i) \tag{1}$$

Proof. The forwarding can be considered as a complete triple tree t, and *TTL* is the layer of tree t. A node in t can be taken as a forwarding peer. The root of t is the querying initiator, and the depth of t is *TTL*. When $TTL = 1$, $FTR(1) = 1 = FT(0) = \sum_{i=1}^{1} 2^{i-1} FT(1 - i)$. When $TTL = k$, supposing $FTR(k) = \sum_{i=1}^{k} 2^{i-1} FT(k - i)$. To better illustrate the problem, we unfold $FTR(k)$ in Eq. (2).

Table 1
The routing table of peer p_7.

Local neighbors			Remote neighbors		
Peer	Distance	IP	Peer	Distance	IP
p_2	12	211.69.205.35	p_5	100	56.10.2.7
p_{10}	34	211.68.122.9	...	...	...
...	...				

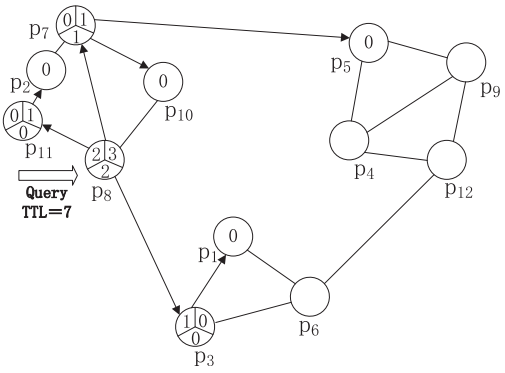

Fig. 2. An example of PTRW.

$$FTR(k) = \sum_{i=1}^{k} 2^{i-1} FT(k-i) = FT(k-1) + 2^1 FT(k-2) + \cdots + 2^{k-1} FT(0) \qquad \square \tag{2}$$

Each item of Eq. (2) is the number of nodes belonging to any sub-tree of t for remote peers. For example, $FT(k-1)$ is the number of nodes of a direct sub-tree of t with depth k, and $2^{k-1}FT(0)$ is the number of nodes of the bottom sub-trees of t for remote peers. When TTL is increased to $k+1$ from k, the depth of each sub-tree is increased by one. All these sub-trees will continue to forward the query to remote peers. When $TTL = k$, the number of sub-trees with only one node for remote peers is $2^{k-1}FT(0)$. Thus, the total number of their brother nodes for local peers is $2 * 2^{k-1}FT(0) = 2^k FT(0)$ because a peer always forwards the query to its two local neighbors and one remote neighbor. As a result, When TTL is increased to $k+1$ from k, the number of newly joined remote peers is $2^k FT(0)$. Therefore, when $TTL = k+1$, we have the conclusion shown in Eq. (3).

$$FTR(k+1) = FT(k) + \cdots + 2^{k-1}FT(1) + 2^k FT(0) = \sum_{i=1}^{k+1} 2^{i-1} FT((k+1)-i) \tag{3}$$

Therefore, $FTR(TTL)$ of STRW shown in Eq. (1) is correct.

In order to reduce communication overhead, PTRW first divides TTL of a query message evenly, and then forwards it with a new TTL. Suppose a peer is ready to forward a query q with TTL t, it will generate three queries from q to forward, which are q_i with TTL t_i where $i = 1, 2, 3$. In PTRW, $t = 1 + \sum_{i=1}^{3} t_i$. Fig. 2 shows an example of PTRW. At peer p_8, the query is first divided into three queries with sub-TTLs. However, the division is not the absolute average, where the sub-TTLs have different values. Then P_8 chooses the minimum sub-TTL to forward the query to one remote neighbor and others for local neighbors.

PTRW has the same communication overhead with one-way random walk (ORW). However, their message forwarding strategies are different. ORW tends to forward the query farther away from the initiator, but PTRW is somehow in favor of local peers. Using PTRW strategy in DNT not only obtains lower query delay, but also decreases the data transmission overhead for relatively physically close peers. The forwarding algorithm of PTRW is shown in Algorithm 1.

Algorithm 1. The forwarding algorithm of PTRW

Forward(peer p, query q, TTL t)
1: search q at peer p;
2: **if** do not hit q **then**
3: $t = t - 1$;
4: **if** $t <= 0$ **then**
5: return;
6: **end if**
7: split t evenly, obtain three sub-hops t_i and $t = \sum_{i=1}^{3} t_i$;
8: choose one peer p_r from remote neighbors of p;
9: choose the min t_{min} from t_i;
10: Forward(p_r, q, t_{min});
11: choose two peers p_1, p_2 from local neighbors of p;
12: forward q to p_1, p_2 with rest hops of t_i;
13: **else**
14: send hit results to the initiator of q;
15: **end if**

3.3. Peer join/leave strategy

In most unstructured P2P systems, when a peer joins a network, it receives guidance from the bootstrap servers [29], and obtains the information of an initial set of peers in the network. Then the peer explores more peers' information by launching peers finding procedure on these initial peers. It takes returned peer lists as candidate peers. Finally, the peer chooses some peers from the candidate peer set and keeps them as neighbor peers, and the join process finishes. In this paper, we use the same strategy. However, to make a balanced distribution of peers in DNT, we apply the random walk as search strategy to find peers. For a peer p_i in the candidate peer set, we first compute the distance d_i (round-trip access delay) between p_i and the newly peer p. We only choose those peers whose d_i is less than the threshold D_t as local neighbors of p, and the peers whose distance is greater than D_t as remote neighbors of p. In our strategy, for each peer, the ratio of local neighbors to remote neighbors is set to 4. Again, this ratio is adjustable depending on the characteristics of query distributions and network topology, etc. The details of peer join algorithm are presented in Algorithm 2.

Algorithm 2. Peer join algorithm for peer p

```
Join(p)
1: get some peers as seed neighbors from the boot strap servers;
2: while the neighbor set in p is not full do
3:    use random walk to find more candidate peers;
4:    compute d_i for each candidate peer p_i;
5:    choose the peers whose d_i < D_t as the local neighbors;
6:    choose the peers whose d_i > D_t as the remote neighbors;
7: end while
```

When a peer leaves the network, it sends offline messages to its neighbors. Once its neighbors receive the messages, they will remove this peer from their neighbor sets, and update their routing tables accordingly. To prevent the damage caused by the peer failure, each peer should keep sending periodical heartbeat messages to its neighbors. If a neighbor peer does not send any heartbeat message for a certain amount of time, it will be considered as a failed peer and has to be removed from the neighbor peer set. In order to reduce the overhead, we propose a one-way heartbeat mechanism. Each neighbor of a peer has a *heartbeat* value to represent its live status. The strongest *heartbeat* is 5, and *heartbeat* is reduced by 1 after a certain time interval. If peer i observes that the *heartbeat* of its neighbor peer j is reduced to 1, it will send a probe message to peer j. If the probe is successful, the *heartbeat* of peer j at peer i is reset to 5, and the *heartbeat* of peer i at peer j is set to 4. This can ensure that the next round of probe will be initiated by peer j for the link (i,j). The strategy can prevent two peers from sending probe messages to each other at the same time, thus it can reduce communication overhead caused by heartbeat mechanism. We also define the minimum number of links l_{min} to offset the impacts of failed peers. If the number of a peer's neighbor peer set is less than l_{min}, it will launch a peer finding process to gain more neighbor peers. The details of one-way heartbeat algorithm are to be presented in Algorithm 3.

Algorithm 3. one-way heartbeat for peer p

```
Heartbeat(p_i)
1: for each neighbor p_i in p do
2:    p_i's heartbeat_i − 1;
3:    if heartbeat_i <= 1 then
4:       p probes p_i;
5:       if probe is successful then
6:          heartbeat_i = 5;
7:          at p_i, p's heartbeat is set to 4;
8:       else
9:          remove p_i from p's neighbors set;
10:      end if
11:   end if
12:   pause for a while and then probe heartbeat for each neighbor;
13: end for
```

3.4. Caching policy

Our proposed caching algorithms are built on top of the underlying network infrastructure. Basically, the peers we choose to store the copies of an object are determined by DNT and PTRW. In general, one or several peers along the query path will be selected for caching. The locations of peers as well as the access frequency of requested objects will be used to make caching decisions, such as the number of copies should be created as well as the locations of these copies. In the following section, we will describe caching algorithms in details.

4. Caching algorithms

We assume that, on each peer, a dedicated storage space is allocated for caching. For distributed P2P caching, there are two important aspects needing to be addressed: caching placement and replacement strategies, which will be discussed in details in the following.

4.1. Caching placement strategy

In a centralized caching system, the objects are only cached on a single dedicated proxy server, thus the caching placement strategy is not necessary. However, for P2P, the system can choose any peer to cache an object, and the locations of these copies will have great impacts on the future accesses as well as caching replacement algorithms. Therefore, efficient caching placement strategy is of great importance. To achieve the optimal performance, it is critical to choose which peers to store a copy when an object needs to be cached. A problem in DNT is that, if a query cannot be satisfied at the peers close to the initiator, we have to increase TTL to search the query in other sub-networks, that will generate large amount of wide-area network traffic. In this paper, we design an effective caching placement strategy based on DNT and PTRW proposed in the previous sections to relieve this problem.

Our basic idea is that a peer prefers to cache an object whose owner is topologically separate from it. I.e., when a query is fulfilled at the owner of the object, we choose the peer who has the largest delay from the owner along the query routing path and keep a copy in its caching space. We denote this strategy as remote peer (RP) caching placement strategy. Suppose a query has the routing path $R = (r_1, r_2, \ldots, r_n)$, where r_i is a peer forwarding the query. In order to facilitate the analysis, the target peer is not included in R and defined as r_t. To select one peer to cache the target object, we set a weight w_i for the peer r_i in R. The weight w_i in Eq. (4) is defined to:

$$w_i = \frac{D(r_t, r_i)}{\sum_{j=1}^{n} D(r_t, r_j)} \tag{4}$$

where $D(r_i, r_j)$ is a function that computes the distance between peer r_i and peer r_j.

When a query reaches the destination (the target peer that can answer the query), the system calculates the weight w_i of all the peers in the query routing path R. The caching placement operation is triggered at the target peer. The peer with the largest w_i is chosen to cache a copy for the target object t. The details of RP caching placement algorithm is shown in Algorithm 4.

Algorithm 4. RP caching placement algorithm

```
RP(route R, target t)
1: for each r_i in R do
2:    compute w_i;
3: end for
4: choose the peer p with max w_i;
5: build caching for t at peer p;
```

4.2. Over-caching problem

The distributed P2P caching strategies are passive caching mechanisms generally. That is the caching decision is triggered only when a query hits the target. Like the degree distribution, the queries also follow the power-law distribution, that means most queries concentrated on a small number of popular objects [30]. This feature results in serious unbalanced caching result. Most of the total caching space are occupied by popular objects because that they have higher query rates, which had been evaluated by our formal study [17]. The experimental results, only taking LRU as the caching replacement strategy, show that the top 5% of popular objects occupy nearly 55% of the caching space, while the cache hit rates for these popular files only reaches 25%. It reveals that many cached copies of the top 5% popular objects are not used at all. We call it over-caching problem for popular objects. Clearly, over-caching reduces the effectiveness of caching space, and decreases chances

for other objects to be cached, especially for moderate popular objects. Thus, the efficiency of the entire P2P distributed caching system is greatly reduced when the over-caching problem is not properly addressed.

4.3. Caching replacement strategy

In order to relieve the over-caching problem, we propose a novel caching replacement strategy called High Popularity and Least Request (HPLR). In HPLR, a file's popularity is denoted as the proportion of the number of copies to the network size. The larger a file's popularity is, the more popular (hot) the file is. HPLR makes an excellent tradeoff. It does not create too many copies for high popular objects, and meanwhile it still keeps an adequate number of copies for those objects to serve large number of requests. Let $T = (t_1, t_2, \ldots, t_n)$ be the cached objects on a peer's caching space, f_i is the request frequency of the file t_i, and p_i is the popularity of t_i. It is difficult to get the popularity of an object in P2P networks because each peer only knows the local information. In this study, we use a simple mechanism to estimate the popularity. When a peer needs to cache a file, it launches a standard random walk to search the file, and the file's popularity is defined according to the query hops. The popularity p_i of t_i is defined in Eq. (5), where *hops* is the number of the hops for a successful query, and N is the network size. We present further illustration in Theorem 2.

$$p_i = \begin{cases} \frac{1}{hops} & hit\ target \\ \frac{1}{N} & not\ hit \end{cases} \tag{5}$$

Theorem 2. *In an unstructured P2P network with the standard random walk search algorithm, a file's popularity is equal to the inverse of query hops under hitting query, or the inverse of the network size under failure query.*

Proof. Suppose the cached file f has popularity p_f. In unstructured P2P networks, the job of search algorithms is an exploratory attempt. Therefore, the popularity of target files directly affects search efficiency. If the file f is searched only at one peer, then the search hit probability for f is p_f. If the standard random walk algorithm hits f with hops h, that is $h * p_f = 1 \Rightarrow p_f = \frac{1}{h}$. If the query does not result in a hit under a large enough TTL, it indicates the target is unpopular. However, if f is a cached object, at least one copy exists. We consider f has only one copy in this situation, as a result, $p_f = \frac{1}{N}$ with failure query probing, where N is the network size. Although only one probe cannot obtain a file's popularity accurately, it has some representation in the probability. Therefore, we consider the conclusion in Theorem 2 is correct. □

Let Pf_i be the replacement probability based on request frequency, and Pp_i be the replacement probability based on popularity, that are defined in Eqs. (6) and (7), respectively. The replacement strategy based on frequency is to keep files with high request because these cached files are useful. The replacement strategy based on popularity is to evict the objects with the highest popularity to leave more space for low popularity files. This strategy could relieve the over-caching problem and improve the performance of distributed caching systems.

$$Pf_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{6}$$

$$Pp_i = \frac{1/p_i}{\sum_{j=1}^{n} (1/p_j)} \tag{7}$$

The caching replacement popularity P_i of t_i is defined in Eq. (8), where γ is an adjustable parameter with the initial value being 0.5. When the caching space is full, the system chooses one file with minimum P_i to evict. The parameter γ can be automatically adjusted depending on the applications. For example, we can choose a smaller γ in P2P file sharing systems to increase the weight of caching replacement based on popularity because the rare objects are more valuable in those scenarios.

Table 2
The experimental parameters.

Parameter	Description	Value
N	Network size	10,000
R	The number of distinct resources	50,000
TTL	Maximum forwarding hops	10–400
n	Minimum number of neighbors	5
T_D	Distance threshold	50
m	The number of resources a peer holding	5–20
C_S	The caching size	20
γ	The parameter for replacement probability	0.5
α	The exponent of power law distribution	0.7

$$P_i = \gamma Pf_i + (1-\gamma)Pp_i \tag{8}$$

If a new object needs to be cached and the caching space is full, the caching replacement operation is triggered. First, the peer that implements the caching task calculates the replacement probability P_i for each cached files. Then, the peer chooses the file with minimum P_i to evict. Finally, the new file is put into the caching space with initial request frequency $f = 1$, and its popularity is also updated. When a query hit occurs in the caching space, the requested number of the corresponding file is incremented by 1. The details of HPLR caching replacement algorithm are shown in Algorithm 5.

Algorithm 5. HPLR caching replacement algorithm for object t

```
HPLR(object t)
1: if t in the caching then
2:    f_t = f_t + 1;
3: else
4:    compute P_i for each cached file;
5:    choose the file t_i with min P_i and evict it;
6:    put t into the caching;
7:    compute p_t;
8:    f_t = 1;
9: end if
```

5. Performance evaluation

In this section, we present the simulation results. The metrics we explore include over-caching, query hit rates, query delay, system overhead, cache hit rates, caching contribution breakdown and system scalability.

5.1. Experimental setting

P2P networks contain a large number of peers, that can join/leave the system at any moment. Dealing with such a dynamic environment is very challenging. In our experiments, we use the popular PeerSim [31] simulator as the driven kernel. PeerSim has been designed to simulate large-scale P2P networks. It has great scalability and can support thousands of peers. For all experiments, the network is generated by using Brite [32] model, that is a topology generation tool, and can construct the overlay network based on the AS (autonomous system) Model. We build an 10000-node overlay network for most of the experiments. In order to better represent the real world P2P systems, we set the node placement strategy of Brite to Heavy Tail. Thus, the topology generated with this strategy will have the power-law distribution, which is more in line with the real environment.

In the experiments, the object popularity also follows a Zipf-like distribution that is similar to power-law distribution. This distribution presents that a small number of objects have very high access frequencies and most other objects have the same low popularity more or less. In Eq. (9) the Zipf-like probability mass function [33] is depicted, where C denotes the number of personal content items and α is the exponent characterizing the distribution.

$$P_{Zipf-like}(x) = \frac{x^{-\alpha}}{\sum_{j=1}^{C} j^{-\alpha}} \tag{9}$$

$P_{Zipf-like}(x)$ determines the probability that a personal content object having rank x is requested, where $x \in \{1, \ldots, C\}$. In [34], Backx et al. show, with a number of practical experiments using popular P2P file sharing applications, that α is usually between 0.6 and 0.8. To assign terms (files) for each of peers in the network, we generate 50,000 distinct terms and assign each term a frequency according to Eq. (9). For queries, based on the terms' frequency, each peer gets query keywords from generated term set randomly. Thus the terms with high-frequency are more likely to be selected as the query keywords.

The experimental parameters for most of the experiments are shown in Table 2. As indicated in Table 2, the network has 10,000 peers, and these peers are distributed in an area of $1000 * 1000$. If the distance between two peers is less than 50, they may belong to the same sub-network. The caching size indicates the number of objects the caching space on a peer can store, rather than the actual storage capacity. To simplify the discussion, we assume all the objects have the same size. For large files, they can be divided into a group of files with the same size. This is the strategy used in most P2P file sharing and P2P streaming applications as well as research simulation environments. If P2P caching is used to guide the routing procedures, the caching space is used to place the index or the reference to peers which has much smaller sizes, and those indexes can be considered as the same size objects as well.

5.2. Analysis of over-caching problem

We carry out some experiments to verify the over-caching problem. As discussed in early sections, the hot objects occupy a large portion of the caching space, while many of cached copies are rarely been used. Even worse, it makes many other mid and low popular files have no chance to be cached, thus the system caching efficiency is dropped. In order to comparative analysis, we compare LFU, LRU and HPLR on over-caching problem. Fig. 3 shows the results. We collect the caching information for the top 5% most popular objects, including the percentage of caching space used and caching hit rates for these objects. As we can see from Fig. 3, the top 5% hot objects of both LFU and LRU account for a large portion of caching space. For the two algorithms, more than 55% caching space is occupied by the copies of the top 5% objects. As for HPLR, it only uses 31%. However, three different strategies have similar cache hit rates for the top 5% hot files, all at about 33%. This shows that LFU and LRU waste 25% more caching space than HPLR. From the results, we can conclude that HPLR relieves this issue and achieve much more efficient caching space utilization than LFU and LRU.

5.3. Query hit rates

In order to evaluate the performance of the distributed caching strategies proposed in this paper, we chose other two P2P strategies for comparative analysis. One is Gnutella network without caching mechanism, that uses standard random walk as search algorithm. We call this scheme Gnutella with Random Walk (GRW). Another one is Adaptive Resource Indexing (ARI) proposed by Lerthirunwong [23]. In ARI, each peer creates a Bloom Filter [35] for a set of its owning objects. Bloom Filter can be used to answer whether or not an object is available in a peer. ARI iteratively distributes indices over the network using the resource indexing technique to reduce the number of messages. To replace a cached object, ARI adjusts the weight of Bloom Filter index using usage and integration rules. Although the content of ARI's caching is index which has a small size, the maintain cost for index is high. If ARI does not set the storage limit, it will generate enormous communication overhead. Therefore, no matter the cached object is an index or the object itself, it is necessary to design caching replacement strategies.

Fig. 4 shows the query hit rates for different P2P strategies. In our configuration, ARI searches local indexes first, and then uses random walk to forward the query if it cannot be satisfied by local indexes. Gnutella without caching mechanisms has the worst performance in query hit rates, and the performance of our caching mechanism is slightly better than ARI. When the maximum number of forwarding hops is equal to 10, the query hit rates of Gnutella is only 3%. As the number of forwarding hops increases, the query hit rates rapidly increases until the hops reaches 200. This is because the targets of some queries are rare objects and random walk cannot find them. For both our strategies and ARI using caching mechanisms, as a result, the query hit rates are much higher compared to Gnutella. Although ARI has good search efficiency, it introduces significant network communication overhead which is discussed in the following subsection.

5.4. Query delay

The query delay is very important to evaluate the efficiency of search algorithms. In unstructured P2P networks, for a file, its query hops are directly affected by its popularity. The higher the popularity is, the lower the query hops are. We implement some comparative experiments to analyze the query delay of the proposed distributed caching strategies. We compute average query hops for those successful queries according to their popularity. Fig. 5 shows average query hops for successful queries with different strategies, and the horizontal axis indicates the popularity of files. Gnutella has the worst performance in query hops because it does not use caching mechanisms. Average query hops of Gnutella have exceeded 200 when the popularity for a file is equal to 0.5%. Both our strategies and ARI have much lower number of query hops compared to Gnutella. For our strategies, the average number of query hops only reaches 5 under 5% files' popularity, while Gnutella is 21. Our

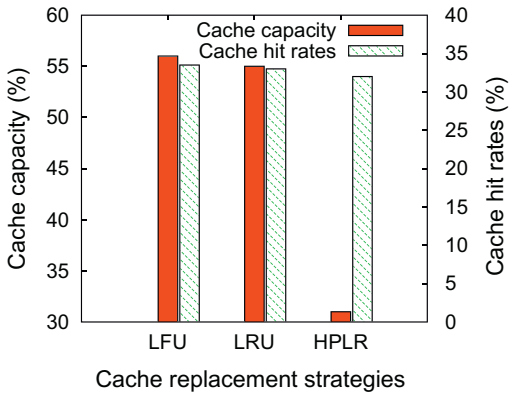

Fig. 3. Over-caching.

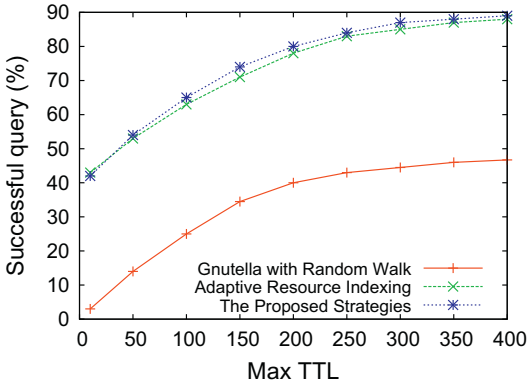

Fig. 4. Query hit rates for different P2P strategies.

caching mechanism, in fact, aims to increase the popularity of files to improve query performance and reduce communication overhead.

Although the query hops can be used to represent query performance somehow, it is not very accurate. For example, a query Q_1 with smaller query hops may contain some hops between distant peers, while another query Q_2 with larger query hops only contains hops between nearby peers. This may result in a longer query delay in Q_1 than Q_2. In our experiments, each peer is assigned to a position in Cartesian space. The peers in the same sub-network are much closer to each other than the peers in different sub-networks. Let $Q_P = \{p_1, p_2, \ldots, p_n\}$ be the query path between peer p_1 and peer p_n, D_{ij} as the actual physical network distance between peer p_i and p_j. We define the query distance D_P between peer p_1 and p_n in Eq. (10).

$$D_P = \sum_{i=1}^{n} \{D_{ij} | p_i, p_j \in Q_P \; and \; j = i + 1\} \tag{10}$$

Fig. 6 shows the results of average query distance for successful queries with different P2P strategies. In order to better express the experimental results, the logarithmic scale is used for the vertical axis. The peers in the experiments are distributed into a simulated network whose area is 1000 square. Our strategies have the best performance than others, where the query distance is only 227 under 5% popularity, while that of ARI reaches 1500 and Gnutella even exceeds 6300. This is because most of queries under our strategies will result in cache hits in local sub-networks. Hence, the average query distance is much lower. Suppose that we use kilometers to measure the simulated topology, even without considering the bandwidth and processing delay, the query delay of Gnutella will reach up to 0.5 s under 0.5% objects popularity. The real query propagation time will be far higher than this delay. Moreover, the longer the query distance is, the more traffic generated on Internet, because the query needs to across more ISPs. Therefore, the proposed distributed caching mechanisms not only improve query performance, but also reduce the Internet traffic caused by P2P applications.

5.5. System overhead

To evaluate system overhead of different strategies, we collect the number of messages forwarded by peers under different concurrent query sizes in the experiments. In the simulations, we do not consider the traffic generated by network topology maintenance, but only count the messages from the query and caching operation. ARI regularly uses an iterative approach to spread Bloom Filter for index updating, and utilizes local flooding to adjust the index weights for caching

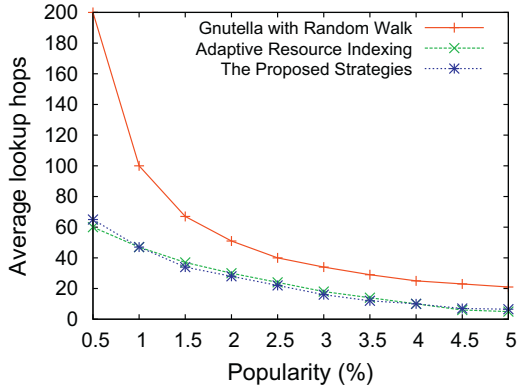

Fig. 5. Average query hops for successful queries.

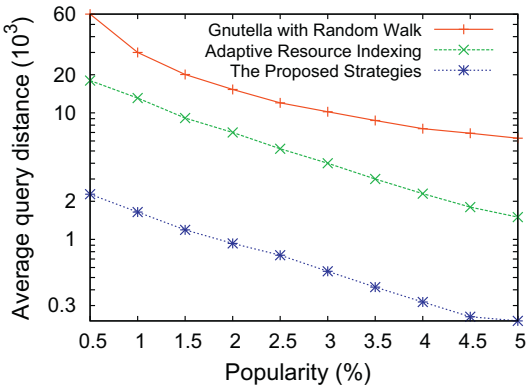

Fig. 6. Average query distance for successful queries.

replacement. This leads ARI to have a large number of basic communication messages. In the experiments, we suppose only 1% peers in ARI need to update the index and adjust the weight when a group concurrent queries are launched. Our strategy introduces necessary communication as well, such as the traffic generated by caching placement and popularity probing.

Fig. 7 shows the results of message overhead under different concurrent query sizes. When the number of concurrent queries is equal to 100, Gnutella has the best performance. This is because Gnutella does not perform the caching and does not need to use extra system overhead to maintain routing information. However, with the increase in the number of concurrent queries, the communication overhead in Gnutella increases rapidly because the query delay of Gnutella is high. This leads to Gnutella's performance begins to deteriorate, and it becomes much worse than that of the proposed strategies when the number of concurrent queries reaches 300. ARI uses the iterative communication and local flooding mechanism to maintain the caching. Although ARI has good query performance, the communication overhead is huge, which approaches 53,600 on average. Our strategies have less communication overhead compared to other caching algorithms, and in addition, they have better performance than the strategies without caching mechanism under the scenarios with high concurrent requests.

5.6. Cache hit rates

The cache hit rate is another important factor to evaluate the caching performance, where we denote it as the percentage of the number of caching hits and query hits. The caching placement strategy RP proposed in this paper always selects the peers being far away from the current peer to cache the target. The proposed caching replacement strategy HPLR uses both the popularity and request frequency to replace the cached files, which tries to keep the valuable cached objects and minimize the impact of over-caching problem at the same time. For comparative analysis, we use Random algorithm, that randomly selects a peer from the query routing to cache the target, as caching placement strategy, and LRU as caching replacement strategy. In the experiments, the network size is 10,000. We launch a query at 10% peers in each time unit, and the query terms are randomly selected from the set of generated terms based on their popularity. That is the objects with high frequency are more likely to be selected as the query targets. The simulation lasts 200 time units, and the cache hit rates are collected every 5 time units.

Fig. 8 is the experimental results of two caching strategies. When the simulation begins, the cache hit rate is zero because the caching space is empty. As the simulation goes on, the cache hit rates increase for both two caching strategies. The caching performance becomes stable when the simulation reaches three-quarters of total time units, and the caching hit rates remain stable as well. From the results, we can observe that the proposed algorithms can achieve the highest cache hit rate

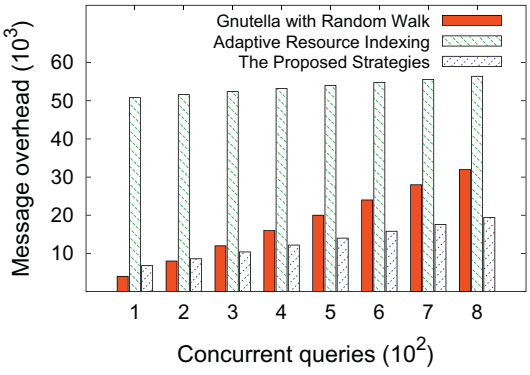

Fig. 7. Message overhead under different concurrent query sizes.

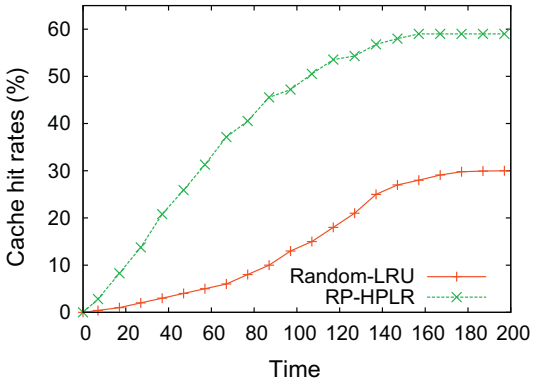

Fig. 8. Caching hit rates for different caching strategies.

of 60%, while Random-LRU only has 30%. In RP, a peer always caches the objects coming from remote peers as possible, and PTRW prefers to search files in local peers. These combined strategies can greatly improve the query hit probabilities. For the replacement strategy, HPLR not only keeps the objects whose request frequencies are high, but also relieves the over-caching problem by reducing the number of cached copies for highly popular objects. In contrast, Random does not provide a good guidance in message routing, and LRU leaves too much caching space for hot objects. Therefore, our strategies have better performance.

5.7. Caching contribution breakdown

Another important metric we want to discuss is the Cumulative Distribution Function (CDF) for cached objects distribution. In the experiments, the total number of distinct objects in the network is set to 50,000, and their distribution follows the power-law property. We count the number of objects emerging in the caching according to the descending order of their popularity. LRU is used to conduct a comparative analysis. Fig. 9 shows the results for LRU and HPLR. As we can see from Fig. 9, HPLR has the best performance. For LRU, the top 5% most popular objects occupy nearly 70% of total caching space, while HPLR only has 34%. HPLR evicts those hot objects and leaves more free space to cache other objects with mid or low popularity. This strategy makes the system store more distinct objects, which can significantly enhances the caching efficiency.

5.8. System scalability

To evaluate the scalability of our strategies, we implement the experiments with different network size for four metrics: query hit rates, query delay, cache hit rates and system overhead. We present three networks with our strategies whose size are 1000, 10,000 and 50,000 peers respectively. Each peer owns 5 distinct original objects in average. From Fig. 10(a), we can find that query hit ratio decreases slightly as the network size increasing. Even in the biggest gap (about 200 hops), the gap for the query hit ratio is only 6%. This shows that our algorithms have good scalability to the metric query hit rates. Fig. 10(b) presents the query delay with different networks size. For three different networks, they have similar performance regardless of their size. The average number of query hops is about 5 under 5% files popularity. As the network size increasing, the proposed network architecture can still effectively distribute the cached objects that ensures a better query delay.

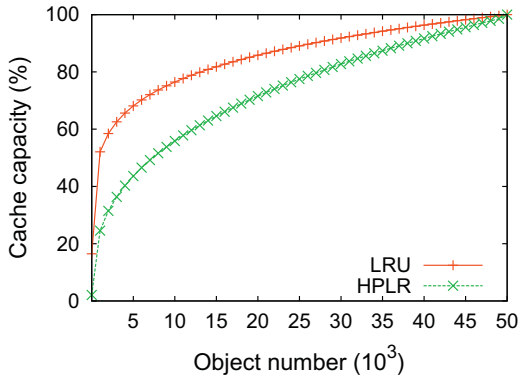

Fig. 9. Cached resources CDF.

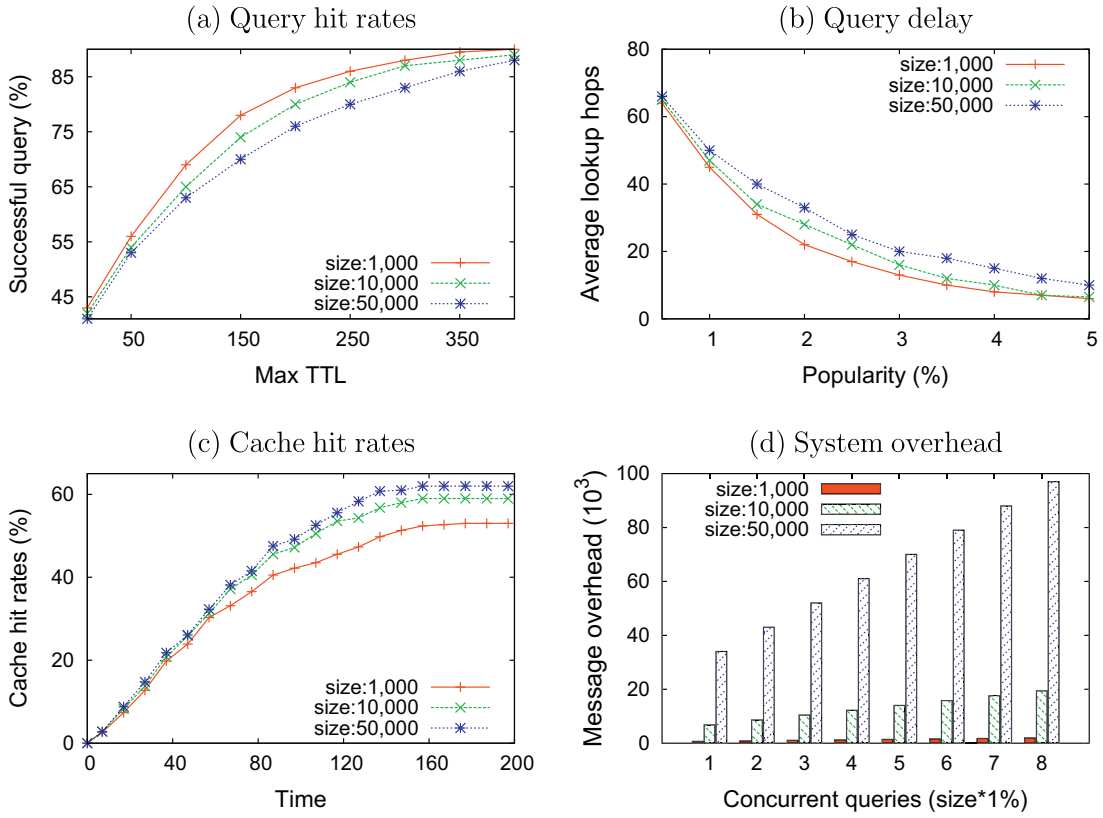

Fig. 10. Scalability of systems with proposed strategies.

Fig. 10(c) shows the cache hit ratio of the systems using proposed strategies with different size. The simulation lasts 200 time units, and the cache hit rates are collected every 5 time units. In each time unit, 10% peers will launch a query, and the query terms are randomly selected from the set of generated terms based on their popularity as previous experiments. We find that the query hit ratio can reach about 60% when the systems are stable. Furthermore, with the increasing of network size, the query hit ratio increases as well, while only slightly increase. These result indicates that our strategies are scalable at the metric cache hit rates.

In order to evaluate the scalability in system overhead, we design the experiments, in which we collect the number of messages forwarded by peers under different concurrent query sizes, for the networks with different size. Fig. 10(d) presents the results. We find that the total overhead increases rapidly as the increasing of the network size. Under (network size) $*$ 5% concurrent queries, that is 50 queries for the network with 1000 peers and 2500 queries for the network with 50,000 peers, the network with 1000 peers has only total forwarding messages 1441, while the network with 50,00 peers reaches at 70,100. However, in average, their overhead are similar, about 1.4 messages for each peer. Therefore, Our strategies have better scalability in system overhead.

6. Conclusions and future work

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, insufficient efforts have been made on P2P caching. In this paper, we design an efficient network topology and search algorithm, and propose novel and effective caching placement and replacement strategies for P2P caching. Unlike previous works, our algorithms aim to build a fully distributed caching infrastructure in unstructured P2P networks. We use Distance-based Network Topology (DNT) to cluster the topologically close peers, and take Preference-based Three-way Random Walk (PTRW) to increase local queries and achieve better searching performance. We also utilize Remote Peer (RP) caching placement approach to determine the locations of the objects needing to be cached in order to improve caching effectiveness. Our High Popularity and Least Request (HPLR) caching replacement algorithm effectively relieves the over-caching problems for popular files and offers satisfactory caching performance for other files. We compare our design with various common and heuristic caching algorithms and the results show that combined with proposed underlay network infrastructure, our caching strategies can deliver better query hit rates, smaller query delay, higher cache hit rates, lower communication overhead and good system scalability. In the future, we will investigate other algorithms to better predict user query pattern, and remove the caching pollution problems caused by the least popular objects.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grants 61173170 and 61300222, National High Technology Research and Development Program of China under Grant 2007AA01Z403, Innovation Fund of Huazhong University of Science and Technology under Grants 2012TS052, 2012TS053 and 2013QN120, and Natural Science Foundation of Hubei province of China under Grants 2013CFB310.

References

[1] Cisco Inc., Cisco visual networking index: forecast and methodology; 2009–2014.
[2] Clarke I, Sandberg O, Wiley B, Hong TW. Freenet: a distributed anonymous information storage and retrieval system. In: International workshop on design issues in anonymity and unobservability; 2000. p. 311–20.
[3] Edonkey. <http://www.edonkey2000.net>.
[4] Gnutella. <http://www.gnutella.wego.com>.
[5] BitTorrent. <http://www.bittorrent.com>.
[6] Stoica I, Morris R, Karger D, Kaashoek M, Balakrishnan H. Chord: a scalable peer-to-peer lookup service for internet applications. Tech. rep.; March 2001.
[7] Ratnasamy S, Francis P, Handley M, Karp R, Shenker S. A scalable content addressable network. Tech rep; 2000.
[8] Rowstron AIT, Druschel P. 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; November 2001. p. 329–50.
[9] Sripanidkulchai K, Maggs BM, Zhang H. Efficient content location using interest-based locality in peer-to-peer systems. In: Proceeding of the 22nd annual joint conference of the IEEE computer and communications societies (INFOCOM'03), San Franciso, CA, USA; March 2003.
[10] Yang B, Garcia-Molina H. Improving search in peer-to-peer networks. In: Proceeding of the 22th international conference on distributed computing systems (ICDCS'02). Vienna, Austria: IEEE; 2002. p. 5–14.
[11] Shen HH. IRM: Integrated file replication and consistency maintenance in P2P systems. IEEE Trans Parallel Distrib Syst (TPDS) 2010;PDS-21:100–13.
[12] Gao G, Li R, Wen K, Gu X. Proactive replication for rare objects in unstructured peer-to-peer networks. J Netw Comput Appl 2012;35(1):85–96.
[13] Wang J. A survey of web caching schemes for the Internet. ACM Comput Commun Rev 1999;25(9):36–46.
[14] Busari M, Williamson CL. On the sensitivity of web proxy cache performance to workload characteristics. In: Proceedings of IEEE INFOCOM, Anchorage, Alaska, USA; April 2001. p. 1225–34.
[15] Hefeeda M, Hsu C-H, Mokhtarian K. Design and evaluation of a proxy cache for peer-to-peer traffic. IEEE Trans Comput 2011;60:964–77.
[16] Wierzbicki A, Leibowitz N, Ripeanu M, Wozniak R. Cache replacement policies revisited: the case of P2P traffic. In: Proc 4th IEEE/ACM international symposium on cluster computing and the grid (CCGRID'04), Chicago, Illinois, USA; April 2004. p. 182–9.
[17] Gao G, Li R, Xiao W, Xu Z. Distributed caching strategies in peer-to-peer systems. In: Proceedings of the 13th IEEE international conference on high performance computing and communications (HPCC 2011), Banff, Canada; September 2011. p. 1–8.
[18] Stading T, Maniatis P, Baker M. Peer-to-peer caching schemes to address flash crowds. In: Proceedings of the international workshop on peer-to-peer systems (IPTPS), Cambridge, MA; 2002. p. 203–13.
[19] Shen H, Liu G. A lightweight and cooperative multi-factor considered file replication method in structured p2p systems. IEEE Trans Comput (TC) 2013;66:1–14.
[20] Meng X, Chen X, Ding Y. Using the complementary nature of node joining and leaving to handle churn problem in p2p networks. Comput Electr Eng 2013;39(2):326–37.
[21] Amoza FR, Rodríguez-Bocca P, Romero P, Rostagnol C. A new caching policy for cloud assisted peer-to-peer video-on-demand services. In: Proc of the 12th IEEE international conference on peer-to-peer computing (P2P'12), Tarragona, Spain; September 2012. p. 43–9.
[22] Luo X, Qin Z, Geng J, Luo J. IAC: Interest-aware caching for unstructured P2P. In: SKG. IEEE Computer Society; 2006. p. 58.
[23] Lerthirunwong S, Maruyama N, Matsuoka S. Adaptive resource indexing technique for unstructured peer-to-peer networks. In: Proc 9th IEEE/ACM international symposium on cluster computing and the grid (CCGRID'09), Shanghai, China; May 2009. p. 172–9.
[24] Cheng A-H, Joung Y-J. Probabilistic file indexing and searching in unstructured peer-to-peer networks. Comput Netw 2006;50(1):106–27.
[25] Frey D, Guerraoui R, Kermarrec A-M, Monod M. Boosting gossip for live streaming. In: Peer-to-peer computing. IEEE; 2010. p. 1–10.
[26] Xu Z, Stefanescu D, Zhang H, Bhuyan L, Han J. Prod: relayed file retrieving in overlay networks. In: Proceedings of the international parallel and distributed processing symposium (IPDPS), Miami, FL; April 2008. p. 1–11.
[27] Shen G, Wang Y, Xiong Y, Zhao BY, Zhang Z. HPTP: relieving the tension between ISPs and P2P. In: Proceedings of the international workshop on peer-to-peer systems (IPTPS), Bellevue, WA; 2007.
[28] Shen H. An efficient and adaptive decentralized file replication algorithm in P2P file sharing systems. IEEE Trans Parallel Distrib Syst 2010;21:827–40.
[29] Wolinsky DI, Juste PS, Boykin PO, Figueiredo RJO. Addressing the P2P bootstrap problem for small overlay networks. In: Peer-to-peer computing. IEEE; 2010. p. 1–10.
[30] Sripanidkulchai K. The popularity of Gnutella queries and its implications on scalability. Tech. rep., O'Reilly's; 2001. <www.openp2p.com>.
[31] Jelasity M, Montresor A, Jesi GP, Voulgaris S. The Peersim simulator <http://peersim.sourceforge.net/>.
[32] Medina A, Lakhina A, Matta I, Byers JW. BRITE: an approach to universal topology generation. In: MASCOTS. IEEE Computer Society; 2001. p. 364.
[33] Breslau L, Cao P, Fan L, Phillips G, Shenker S. Web caching and zipf-like distributions: evidence and implications. In: Proceedings of the eighteenth annual joint conference of the IEEE computer and communications societies; 1999. p. 126–34.
[34] Backx P, Wauters T, Dhoedt B, Demeester P. A comparison of peer-to-peer architectures. In: Proceedings of Eurescom 2002 powerful networks for profitable services; 2002. p. 1–8.
[35] Bloom B. Space/time trade-offs in hash coding with allowable errors. Commun ACM 1970;13:422–6.

Guoqiang Gao received the M.S. and Ph.D. in Computer Science from Huazhong University of Science and Technology, China in 2007 and 2011 respectively. He is currently a Lecturer in School of Media and Communication at Wuhan Textile University. His research interests include peer-to-peer computing, cloud computing, and data mining. He is a member of IEEE.

Ruixuan Li received the B.S., M.S. and Ph.D. in Computer Science from Huazhong University of Science and Technology (HUST), China in 1997, 2000 and 2004 respectively. He is currently a Professor in School of Computer Science and Technology at HUST. His research interests include cloud computing, big data management, and system security. He is a member of IEEE and ACM.

Heng He received the B.S. in Computer Science from Wuhan University of Technology in 2004 and the M.S. in Computer Science from Huazhong University of Science and Technology (HUST) in 2007. He is currently a Ph.D. candidate in the School of Computer Science and Technology at HUST. His research interests include peer-to-peer computing, cloud computing, network coding and network security.

Zhiyong Xu received the Ph.D. in Computer Engineering from University of Cincinnati in 2003. He is currently an Associate Professor in the Department of Mathematics and Computer Science at Suffolk University. His research interests include peer-to-peer computing, high performance I/O and file systems, multimedia applications, parallel and distributed computing. He is a member of IEEE.