DUP: Dynamic-tree Based Update Propagation in Peer-to-Peer Networks

Liangzhong Yin and Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
E-mail: {yin, gcao}@cse.psu.edu

***Abstract*—Peer-to-peer networks have received considerable attention due to properties such as scalability, availability, and anonymity. In peer-to-peer networks, indices are used to map data id to nodes that host the data. Previous work showed that the performance of locating data in peer-to-peer networks can be improved by passively caching passing-by indices. The performance can be further improved by actively pushing indices to interested nodes. This paper proposes a Dynamic-tree based Update Propagation (DUP) scheme to propagate the indices in peer-to-peer networks. DUP dynamically builds the update propagation tree to facilitate the propagation of indices. Because the update propagation tree only involves nodes that are essential for update propagation, the overhead of DUP is very small. Simulation results show that DUP not only reduces the overall cost of the index propagation and index access, but also results in much lower index query latency compared to existing schemes.**

I. Introduction

In peer-to-peer networks, one important problem is to locate data (content) among nodes throughout the Internet. This problem is complicated by the facts that the number of nodes in peer-to-peer networks is huge and their contents change from time to time. Traditional peer-to-peer networks such as Napster [1] use centralized servers to map a data id to the node that hosts the data. Clearly, this approach creates a single point of failure and does not scale well. To address this problem, distributed approaches such as CAN [2] and Chord [3] have been proposed. These approaches distribute the mapping throughout the network so that locating data can be performed in a distributed fashion. When a node needs to locate a data object, its request is routed through the network to search for the node that is maintaining the mapping information for that object. An index that indicates the address of the node hosting the data is then sent back to the requesting node, which retrieves the data from the hosting node directly.

This work was supported in part by the National Science Foundation (CAREER CCR-0092770 and ITR-0219711).

Peer-to-peer networks can be divided into two categories according to their index searching methods: structured networks [2], [3], [4], [5] and unstructured networks [6], [7], [8], [9]. In structured networks, the queries for indices are routed along a well-defined path to reach the node which maintains the mapping information for the requested data. These search paths form a tree, which is called the *index search tree*. In unstructured network, there does not exist well-defined query search paths. Requests are sent out using schemes such as flooding. This paper is based on the structured peer-to-peer networks.

To reduce the index query latency, the index can be cached by intermediate nodes along the query path so that intermediate nodes can serve queries for the index later [2], [3], [6], [10], [11]. There are two widely used cache consistency models: weak consistency [12] and strong consistency [13], [14]. In the weak consistency model, stale indices might be returned to the requesting node. In the strong consistency model, after an update, no stale copy of the modified data will ever be returned to the requesting node. The commonly used weak consistency mechanism is TTL-based (Time-To-Live) [12], in which a requesting node considers a cached copy up-to-date if its TTL has not expired. For strong cache consistency, *invalidation-based* and *polling-based* approaches are used. In the invalidation-based approach, the server keeps track of all nodes that cache the data, and sends invalidation messages to them when the data is changed. In the polling based approach, every time a node requests a data item and there is a cached copy, it first contacts the server to validate the cached copy. Since Polling-based approach may generate significant network traffic [15], the TTL-based approach is widely used for the weak cache consistency model and the invalidation-based approach is used for the strong cache consistency model.

Most previous schemes for index caching in peer-to-peer networks are based on the weak cache consistency model. When an index passes by a node, it is cached by that node. There is a *Time-To-Live* (TTL) timer associated with the index. The index will be removed from the cache after its TTL expires. Such index caching scheme is referred to as the *Path Caching with Expiration* (PCX) scheme. This scheme has two drawbacks. First, a cached index is not usable after the TTL expires, even if the index has not been updated. Second, an index may be updated before the TTL expires, but nodes caching the index may not know and still use the stale index.

The cache performance can be improved by actively maintaining cache consistency, and then a new query can always use the cached indices. However, the invalidation-based approach can not be directly applied to peer-to-peer networks due to scalability issues. Since there are huge number of nodes in peer-to-peer networks, it is impossible for the node that maintains the mapping information to keep track of all nodes caching the index. Therefore, the task of tracking caching nodes and pushing updates should be distributed in the network. Further, because the index size is very small, to do cache invalidation, the updated index should be sent so that caching nodes need not request for the updated index again. Following these ideas, we propose a *Dynamic-tree based Update Propagation* (DUP) scheme. In DUP, on top of the existing index search tree, a dynamic update propagation tree is constructed. This propagation tree contains only those nodes that are either interested in the index or essential for propagating the updates. By propagating the updates along the tree, the index query cost is reduced and the performance is improved.

The rest of the paper is organized as follows: Section II introduces the system model and analyzes existing schemes. In Section III, we present the technical details of DUP. Section IV evaluates the proposed scheme through extensive simulations. Section V discusses the related work, and Section VI concludes the paper.

II. PRELIMINARIES

A. System Model

In peer-to-peer networks, a data object can be searched by its name, usually called *Key*. In the structured peer-to-peer network, the network relies on a hash function to map the key to a virtual space. Each node in the network is responsible for part of this virtual space. It maintains the $(key, value)$ pair for all keys that fall into its responsible area. The $value$ in the pair indicates the nodes that host the data corresponding to the key. A node is the *Authority Node* of the (key, value) pairs it maintains.

Data is inserted or removed from nodes in the network from time to time, and nodes may join or leave the network at any time. When such a change happens, the node that hosts the data should inform the authority node. It also needs to send *keep-alive* messages periodically to the authority node to deal with node failures. The authority node needs to update the index, i.e., the (key, value) pair, whenever it receives update messages or considers the node hosting the data is dead because it did not receive the keep-alive message from the node for a specific amount of time.

B. The CUP Update Propagation Scheme

When the index is updated by the authority node, such update should be propagated to the nodes that cache the index to reduce the query latency. Roussopoulos and Baker [11] proposed a Controlled Update Propagation (CUP) scheme which actively pushes the updated indices to interested nodes along the index search tree. In this scheme, each node needs to record the interests of its neighboring nodes in the index search tree and push updated index to them when necessary. Based on the benefit and the overhead of pushing the updates, each node determines whether to push the index update further down the tree. As a result, an index is pushed hop-by-hop following the index search tree to reach interested nodes.

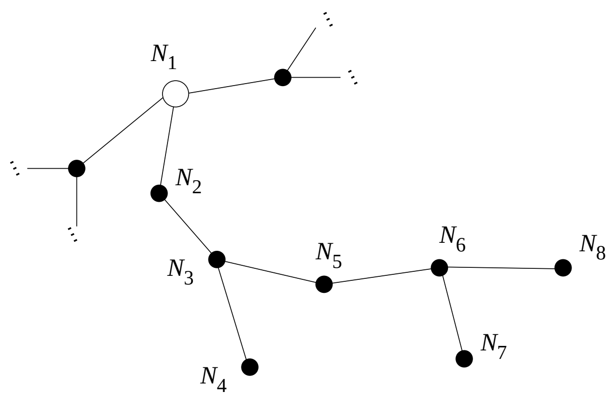


Figure 1. An index search tree for key K

Figure 1 can be used as an example to analyze the performance of CUP. As suggested in [3], [11], the number of hops that a packet needs to travel can be used to represent the cost. Node N_1 is the authority node for key K. Suppose the updated index of K has been pushed to node N_5. One the following three cases is possible when the index is pushed from N_5 to N_6:

1) N_6 does not access the index before the index's TTL expires. In this case, the index is not useful due to TTL expiration. Therefore, pushing the index increases the cost by one hop.
2) N_6 accesses the index exactly once before it expires. In this case, pushing the index to N_6 reduces the cost by 50% as it costs one hop to push the index to N_6 while if the index is not pushed to N_6, it costs two hops to send the query to N_5 and get the reply.

3) N_6 accesses the index more than once before it expires. Intuitively, pushing the index can further reduce the cost in this case. However, this may not be true if N_6 is able to cache the index. When the index is not pushed from N_5 to N_6, the first query from N_6 needs to spend two hops to send the query to N_5 and get the reply. Then the subsequent queries are served directly by N_6 as it now has the index in the cache. Therefore, pushing the index can at most reduce the cost by 50% when nodes cache their requested data.

CUP has performance limitations since it pushes the update along the query path. Intermediate nodes along the path receive the updated index even if they do not need it. For example, in Figure 1, if N_6 is the only node that is interested in the index update, the index is still pushed through the path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6$. If intermediate nodes decide to stop forwarding the index, N_6 is cut off from the update information. This incurs long delay and high cost when N_6 needs to access the index.

III. Dynamic-tree based Update Propagation

Although CUP performs better than PCX, its performance improvement has some limitations because updates are always propagated through the query path in CUP. To efficiently propagate the index updates, we propose a Dynamic-tree based Update Propagation (DUP) scheme.

A. Overview of DUP

The idea of DUP can be explained by Figure 2. Suppose only N_6 is interested in the index. When an update happens in N_1, N_1 pushes it directly to N_6. As the peer-to-peer network is an overlay network on top of the Internet, the physical distance between N_1 and N_6 is not necessarily much longer than that between N_1 and N_2. Such direct push can significantly improve the performance. It only costs one hop to push the update. If the update is not pushed to N_6, it costs eight hops for N_6 to send the request and get the index from N_1 in PCX. Therefore, the cost is reduced by 87.5%. This direct push is illustrated by the solid arrow in Figure 2 (a). The dynamic update propagation tree (DUP tree) contains only N_1 and N_6.

Later, if N_4 is also interested in the index (see Figure 2 (b)), N_1 pushes the index to N_3, the nearest common parent of N_4 and N_6. Then N_3 is in charge of pushing the index to N_4 and N_6. The new DUP tree, which is linked by the solid arrows, contains N_1, N_3, N_4, and N_6. Compared to PCX and CUP, this scheme only costs three hops while PCX costs ten hops and CUP costs five hops to serve N_4's and N_6's queries. Our scheme performs better because it takes short-cuts when pushing the updates. In the worst case when no short-cut is available, our scheme falls back to CUP and still performs well.

When a node needs to access an index that is not in the local cache, it sends a request to the root through the requester's index search path. Along the path, the first node that has a valid copy of the index serves the query by sending the index along the reverse path. When an update occurs at the root, it pushes the update to its downstream nodes in the DUP tree. Each node that receives the updated index refreshes its cache and repeats the pushing process. Finally, all nodes in the DUP tree receive the updated index and the update propagation ends.

B. Maintaining the DUP Tree

In this section, we present techniques to maintain the DUP tree. Note that the overhead of maintaining the DUP tree is at most equal to that of maintaining the CUP tree. Both schemes utilize the underlying index search tree, which is necessary for peer-to-peer networks even if CUP or DUP is not adopted. Compared to CUP, our scheme is more efficient since it reduces the number of nodes participating in the update propagation process.

Each node needs to do some book-keeping to support DUP. The following information needs to be maintained for the update propagation.

- **Access Tracking Information**: Each node maintains its access tracking information to determine whether it is interested in the index based on the interest measurement policy. In this paper, we adopt a simple policy: if the number of queries a node receives in the last TTL interval is greater than a threshold value c, the node is considered to be interested in the index.
- **Subscriber List**: In this list, each node records the node ids of the downstream nodes (including itself) that are interested in the index. It only records the nearest interested node from each of its downstream branches. When a node receives an updated index, it pushes the received index to nodes on the list.

The following messages are used for the DUP tree maintenance:

- **subscribe(**N_i**)**: the subscribe message for N_i.
- **unsubscribe(**N_i**)**: the unsubscribe message for N_i.
- **substitute(**N_i**,** N_j**)**: informs the upstream nodes to replace N_i with N_j in their subscriber lists.

Suppose no node is interested in the index initially. If a node, say N_6, finds that it is interested in the index, it adds itself to its subscriber list. Then it either sends out $subscribe(N_6)$ explicitly or piggybacks $subscribe(N_6)$ by setting the $interest$ bit in the request packet it sends out. This message is routed through the underlying index search tree until it reaches the root N_1. Intermediate nodes along the path add N_6 to their subscriber lists when they receive $subscribe(N_6)$. When the subscribe message reaches N_1, N_1 adds N_6 to its subscriber list

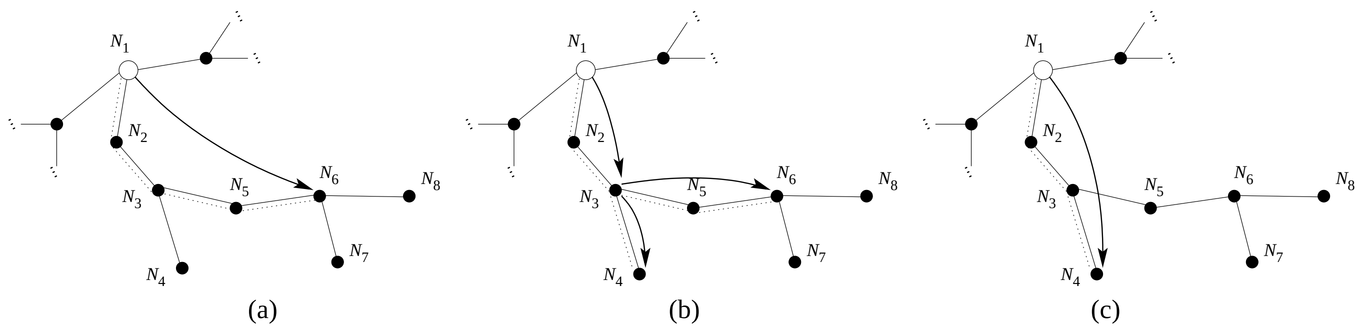


Figure 2. An evolving dynamic update propagation tree. Nodes linked by arrows are in the DUP tree. Nodes linked by dotted lines are in the virtual path.

and pushes the current and future updated index directly to N_6. Nodes that have at least one subscriber form a path, as shown by the dotted line in Figure 2 (a). This path is called the *virtual path*. Nodes N_2, N_3, and N_5 are in the virtual path but not in the DUP tree. Only N_1 and N_6 are in the DUP tree and involved in the update propagation, which reduces the cost of update propagation.

Later, if N_4 finds that it wants to be informed of the update, it sends out $subscribe(N_4)$ to set up a virtual path. When the virtual path reaches N_3, N_3 knows that two nodes from its two different branches are interested in the index[1]. N_3 then replaces the message by a $substitute(N_6, N_3)$ message to ask upstream nodes to replace N_6 with N_3 in their subscriber list. The first node in the DUP tree that receives this substitute message (e.g., N_1) replaces N_6 with N_3 in its subscriber list. After this, N_1 pushes future updates to N_3 and N_3 forwards the updates to N_4 and N_6.

If N_6 is no longer interested in the index after it joins the DUP tree, it sends $unsubscribe(N_6)$ to the upstream nodes in the index search tree. Upon receiving the unsubscribe message, nodes along the path remove N_6 from their subscriber list. This clears the virtual path for N_6. When the unsubscribe message reaches N_3 in the DUP tree, N_3 stops forwarding the index updates to N_6. If N_3 has only one child N_4 in its subscriber list, it sends a $substitute(N_3, N_4)$ to inform the upstream nodes that N_3 no longer needs the update. After the first upstream node in the DUP tree (N_1 in our example) catches this message, it pushes the updates directly to N_4 instead of N_3 (see Figure 2 (c).)

One nice property of DUP is the low overhead. The number of subscribers that each node needs to maintain is at most equal to the number of its direct children in the index search tree. For example, in Figure 2 (b), N_3 needs to maintain at most two subscribers. Suppose another descendant, N_5, N_7, or N_8, wants to join the DUP tree. For N_7 or N_8, N_6 takes care of them; for N_5, after it joins the DUP tree, it replaces N_6 as a subscriber of N_3 and N_5 lists N_6 as its subscriber. The formal description of the DUP algorithm is given in Figure 3.

[1]The request from N_5 branch is caught by N_5 or N_6 and never reaches N_3. This also explains why the subscriber list needs at most one entry for each downstream branch.

C. Node Arrival, Departure, and Failure

Since nodes may join or leave the network at any time, the topology of the index search tree may be changed and the update propagation may be affected. This section extends DUP to deal with such changes. Note that the underlining peer-to-peer network protocol takes care of topology changes of the index search tree. Detailed maintenance operations of the index search tree can be found in [2]. DUP does not interfere with these operations. It only makes necessary adjustments to the DUP tree when the topology changes. Most of these adjustments are kept local to the node that joins or leaves the network and the overhead is small.

After a new node joins the network, it is responsible for a subset of indices previously maintained by a neighboring node. In Figure 2 (a), suppose a new node N_3' is inserted between N_3 and N_5 and it takes care of indices that previously belong to N_3. After the insertion, N_3 notifies N_3' that N_6 is in its subscriber list. N_3' inserts N_6 to its subscriber list, and becomes an intermediate node in the virtual path. If the arriving node falls outside of any virtual path, such as between N_6 and N_8, nothing specific needs to be done.

A node may leave the network at will or due to node/link failure. If a node N_i leaves on its own, it informs its neighbors about its leaving. The neighboring node N_j that is chosen to take care of N_i's indices acts as N_i in the update propagation process after N_i left. The only exception is when the leaving node is the end node of a virtual path, e.g., N_6 in Figure 2 (a). In this case N_6 sends an $unsubscribe(N_6)$ upstream to clear the virtual path related to it. No specific action needs to be taken if a leaving node does not belong to any virtual path.

Dealing with node failure is more complicated. Different methods are used to deal with failed nodes with

```
Notations:
  • $S\_list_i$: the subscriber list of node $N_i$, initially empty.
  • $|S\_list_i|$: the length of $S\_list_i$.
  • $S\_list_i[0]$: the only member of $S\_list_i$ when $|S\_list_i| = 1$.

(A)  When a query for the index arrives at $N_i$:
     refresh the access tracking information;
     if ($N_i$ is interested in the index according to its policies and ($N_i \notin S\_list_i$)) then
         call $process\_subscribe(N_i, N_i)$;
     send out the request for the index if cache misses;

(B)  When the $subscribe(N_j)$ message arrives at $N_i$:
     call $process\_subscribe(N_j, N_i)$;

(C)  When the $substitute(N_j, N_k)$ message arrives at $N_i$:
     $S\_list_i = (S\_list_i - \{N_j\}) \bigcup \{N_k\}$
     if ($N_i$ is the root) then          return;
     if ($|S\_list_i| == 1$ ) then     /* $N_i$ is not in the DUP tree */
         forward $substitute(N_j, N_k)$ upstream;

(D)  When $N_i$ loses interest for the index: call $process\_unsubscribe(N_i, N_i)$;
(E)  When $unsubscribe(N_j)$ arrives at $N_i$: call $process\_unsubscribe(N_j, N_i)$;

  $process\_subscribe(N_j, N_i)$:
     if ($N_i$ is the root) then {          $S\_list_i$ = $S\_list_i \bigcup N_j$;          return; }
     if ($|S\_list_i| == 1$) then
         $N_k = S\_list_i[0]$; /* temporarily save the old subscriber id */
     $S\_list_i$ = $S\_list_i \bigcup N_j$;                        /* $|S\_list_i|$ is increased by one */
     if ($|S\_list_i| == 1$) then                        /* did not have a subscriber, now has one */
         send $subscribe(N_j)$ upstream;
     else if ($|S\_list_i| == 2$ ) then                 /* had one subscriber, now two */
         send $substitute(N_k, N_i)$ upstream;   /* replace the old subscriber $N_k$ with itself*/
     /* if $|S\_list_i| > 2$, already in the DUP tree before the message arrives, no extra action needed */

  $process\_unsubscribe(N_j, N_i)$:
     $S\_list_i = S\_list_i - \{N_j\}$;
     if ($N_i$ is the root) then          return;
     if ($|S\_list_i| == 0$) then                            /* does not have a subscriber */
         send $unsubscribe(N_i)$ upstream;
     else if ($S\_list_i == 1$) then                        /* has one subscriber */
         send $substitute(N_i, S\_list_i[0])$ upstream; /* replace itself with its subscriber */
     /* if $|S\_list_i| > 1$, it has two or more subscribers, remains in the DUP tree*/
```

Figure 3. The DUP algorithm

different roles. Figure 2 (b) is used as an example to illustrate how to handle node failures.

1) If the failed node does not belong to any virtual path, no specific action is needed.
2) The failed node is the last node of a virtual path (e.g., N_6). In this case, the virtual path from N_6 to N_3 is no longer necessary. The upstream node in the virtual path N_5 can detect this failure and send an $unsubscribe(N_6)$ upstream. The upstream nodes process this message according to the algorithm in Figure 3 (E).
3) The failed node is inside a virtual path (e.g, N_5) that has one subscriber. This node failure can be detected by its downstream neighbor node N_6 in the virtual path. N_6 deals with this node failure by sending a $subscribe(N_6)$ upstream. This message is caught by the node say N_i that replaces N_5[2]. If N_5's previous parent N_3 is also the parent of N_i, N_i needs to do nothing. Otherwise, N_i informs N_3 by sending an $unsubscribe(N_6)$. Then N_i takes care of the $subscribe(N_6)$ according to the algorithm in Figure 3 (B) and N_3 takes care of the $unsubscribe(N_6)$ message according to the

[2] N_i is not shown in Figure 2 (b) to keep its location general.

algorithm in Figure 3 (E).

4) The failed node is a node in the DUP tree that has multiple subscribers (e.g., N_3). This is similar to the above case except that both N_4 and N_5 send subscribe messages to node N_i which replaces N_3. N_i processes these messages according to the algorithm in Figure 3 (B).
5) The failed node is the root of the index search tree (e.g., N_1). In this case, all indices maintained by N_1 are lost. N_2 can still setup the virtual path and inform the new root N_i that it should push the index to N_3. N_i starts the update propagation process after receiving the refreshed index information.

IV. Performance Evaluations

Extensive simulations are carried out to evaluate the performance of the proposed scheme and compare it with PCX and CUP. The performance metrics used in this paper are the average query latency and the average query cost as they are widely adopted in previous studies [2], [3], [11], [16]. The average query latency is represented by the average number of hops that a request needs to travel before it reaches a valid index. The average query cost is defined as the total number of hops that the query related messages such as requests, replies and updates traveled in the network divided by the total number of queries. In CUP and DUP, the query cost also includes the messages used to propagate interests. For example, in CUP, extra messages are used to inform neighbors about their interests. In DUP, extra messages are used to maintain the DUP tree, such as the $subscribe$ and $unsubscribe$ messages.

In the simulation a peer-to-peer network with n nodes are studied. The maximum degree of the index search tree is D. The number of children for each node is uniformly selected from $[1, D]$. Different tree topologies are studied in our simulation and the results are similar. Therefore, we present the results based on one randomly generated topology for each D value.

The index is maintained at the root node. The TTL of the index is set to be 60 minutes, which is based on the measurement study of peer-to-peer networks [17]. If CUP or DUP is used, the root pushes the updated index to interested nodes exactly one minute before the previous index expires. The latency of message transfer between two nodes follows exponential distribution with mean value of 0.1 seconds.

Queries are generated with an arrival rate λ. The query inter-arrival time follows exponential distribution (default) or the heavy-tailed Pareto distribution. Pareto distribution is used in our simulation because recent studies show that some peer-to-peer networks exhibit Pareto query inter-arrival time patterns [10]. In Pareto distribution, the CDF of the inter-arrival time is $F(x) = 1 - (\frac{k}{x+k})^{\alpha}$, where usually $2 > \alpha > 0$. When $\alpha > 1$, the mean query arrival rate is $(\alpha - 1)/k$. Similar to [11], two α values 1.05 and 1.2 are used to test the effects of Pareto distribution. The scale parameter k of the Pareto distribution is set so that $(\alpha - 1)/k$ equals the query arrival rate λ used in each simulation.

The queries are distributed to nodes according to Zipf-like distribution, which has been frequently used to model non-uniform distribution. In the Zipf-like distribution, the query probability of node N_i ($1 \leq i \leq n$) is represented as $P_i = \frac{1}{i^{\theta} \sum_{k=1}^{n} \frac{1}{k^{\theta}}}$. This distribution represents the situation where a small number of nodes generate most of the queries while other nodes generate only a smaller number of queries. In the simulation, we vary θ in the range of $[0.5, 4]$ to show the effect of access skewness on the system performance.

Most system parameters are listed in Table I. The second column lists the default values of these parameters. In the simulation, we may change the parameters to study the impacts of these parameters. The ranges of the parameters are listed in the third column. Each simulation is kept running until at least the 95% confidence interval of the query latency is obtained. The total simulation time is at least $180,000$ seconds.

TABLE I
Simulation parameters

Parameter	Default value	Range
Number of nodes n	4096	128 to 65536
Maximum node degree D	4	2 to 10
Mean query arrival rate λ (queries per second)	10	0.1 to 100
Zipf parameter θ	2	0.5 to 4
Pareto parameter α	N/A	1.05, 1.20
Threshold value c	6	2 to 10

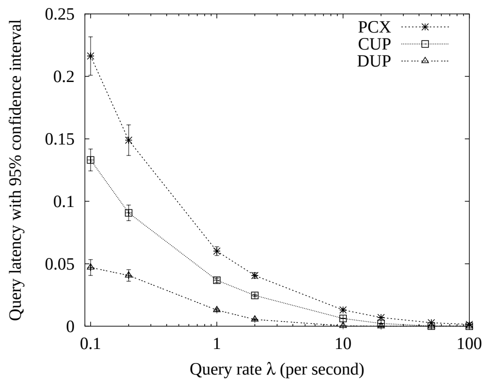
A. Simulation Results

1) Effects of Threshold Value c: In order to determine whether a node is interested in the index, the threshold value c is introduced in Section III-B. If c is large, few nodes are marked as interested nodes. As a result, few nodes can get the index updates, and the query latency is long. On the other hand, if c is too small, some nodes may still receive the index updates although they will never access the index, which increases the average query cost.

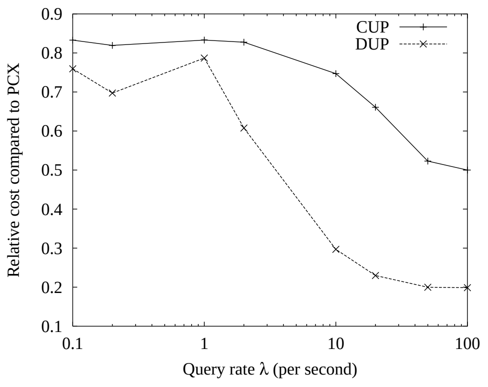
TABLE II

THE EFFECTS OF THE THRESHOLD VALUE c

c value	2	4	6	8	10
Average query cost ($\lambda = 0.1$)	0.47	0.36	0.33	0.31	0.30
Average query latency ($\lambda = 0.1$)	0.030	0.040	0.047	0.054	0.059
Average query cost ($\lambda = 1$)	0.12	0.10	0.095	0.087	0.083
Average query latency ($\lambda = 1$)	0.0051	0.0089	0.013	0.017	0.020
Average query cost ($\lambda = 10$)	0.083	0.078	0.077	0.080	0.083
Average query latency ($\lambda = 10$)	0.000032	0.00013	0.00030	0.00048	0.00066



(a)



(b)

Figure 4. The performance as a function of the mean query arrival rate λ

As shown in Table II, as c increases, the average query cost decreases because fewer nodes are considered interested nodes. One exception is when $\lambda = 10$ queries per seconds, the average query cost first decreases when c increases from 2 to 6 and then the cost increases as c increases from 6 to 10. This is because when the query rate is high, if the threshold value is too large, the nodes that should get the updated index cannot get it. Later queries have to send out requests to get the index, which increases the cost; on the other hand, if the threshold value is too small, more nodes receive the updates and the cost also increases. Overall, we found that a threshold value of 6 achieves a good balance between the query cost and query latency. Therefore, we will use this c value in the rest of our simulations.

2) Effects of Query Arrival Rate λ: In this section, we evaluate the effects of query arrival rate on the system performance, where the inter-arrival time follows an exponential distribution. Figure 4 (a) shows the average query latency with 95% confidence interval as the query arrival rate varies. As the query arrival rate increases, the probability that the index is cached by each node increases, and a cached copy can serve more queries before it expires. Thus, the average query latency decreases. Compared with PCX and CUP, DUP has the lowest query latency because the updates are proactively pushed to interested nodes, and the update propagation speed is faster than in CUP as the updates take short-cuts. Figure 4 (b) shows the relative average cost of CUP and DUP compared to PCX. When the query arrival rate is low, few queries are generated. For example, when $\lambda = 1$ query per second, only one query is generated per second in the whole network with a total of 4096 nodes. We can expect that pushing updates do not perform very well, but still better than PCX. As shown in Figure 4 (b), CUP and DUP reduce the cost by about 20%, and DUP performs better than CUP. As the query arrival rate increases, the performance of both schemes increases. However, the cost of CUP can at most be reduced to about 50% of that of PCX. This agrees with our analysis in Section II-B. Because DUP does not have such performance limitation, it performs much better than CUP. The cost of DUP can be reduced to 20% of PCX when the query arrival rate is high.

3) Effects of the Number of Nodes: In the simulation, we vary the number of nodes to study how the proposed scheme performs as the network size changes. Table III shows the query latency of different schemes under different node sizes and query arrival rates.

By checking each row of Table III, we can see that the query latency of all the schemes increases as the number of nodes increases. This is because the average distance

TABLE III

COMPARISON OF PCX, CUP, AND DUP WHEN THE NUMBER OF NODES CHANGES

Number of Nodes	256	1024	4096	16384	65536
PCX Latency ($\lambda = 0.1$)	0.17	0.18	0.22	0.22	0.21
CUP Latency ($\lambda = 0.1$)	0.098	0.11	0.13	0.13	0.12
DUP Latency ($\lambda = 0.1$)	0.028	0.043	0.047	0.049	0.049
PCX Latency ($\lambda = 1$)	0.039	0.053	0.060	0.062	0.061
CUP Latency ($\lambda = 1$)	0.021	0.033	0.037	0.039	0.040
DUP Latency ($\lambda = 1$)	0.0048	0.010	0.013	0.014	0.014
PCX Latency ($\lambda = 10$)	0.0063	0.011	0.013	0.014	0.014
CUP Latency ($\lambda = 10$)	0.0019	0.0049	0.0062	0.0069	0.0068
DUP Latency ($\lambda = 10$)	0.000091	0.00024	0.00030	0.00034	0.00034

from a node in the network to the root increases as the number of nodes increases. By checking each column of Table III, we can see that CUP performs better than PCX and DUP performs the best. In many cases, DUP performs an order of magnitude better than CUP, even though CUP already performs much better than PCX.

Figure 5 compares the average access cost of different schemes. It shows that CUP performs better than PCX, but the difference becomes smaller as the number of nodes increases. When the number of nodes increases, more nodes fall between an interested node and the authority node, which incurs larger pushing overhead in CUP. DUP is able to reduce the pushing overhead by skipping unnecessary nodes. Therefore its relative performance compared to PCX still increases when the number of nodes increases.

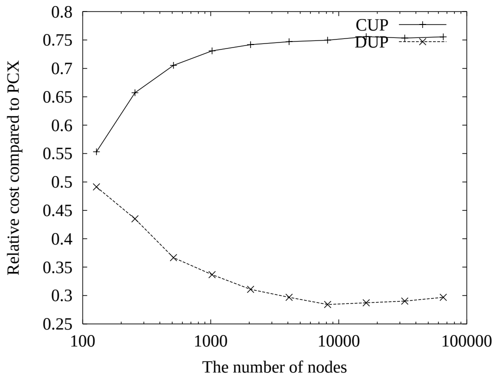


Figure 5. The performance as a function of the number of nodes

4) Effects of the Maximum Node Degree D: D determines the maximum number of children each node can have. If D is larger, each node can have more children. The average number of hops between a node in the network and the root decreases as D increases since the total number of nodes is fixed in this simulation. Therefore, the query latency decreases when D increases, as shown in Figure 6 (a). With a larger D, PCX performs better as the average number of hops between a node in the network and the root decreases, because a request needs to travel fewer number of hops when a cache miss occurs. However, DUP still has much lower cost than PCX and CUP, even when D is as large as ten.

5) Effects of Zipf Parameter θ: The Zipf parameter θ determines how the queries are distributed among the nodes. Small θ means that the query distribution is close to uniform. Large θ means that more queries are generated by fewer hot nodes, i.e., there are hot query spots in the peer-to-peer network. Figure 7 (a) shows that DUP has a very low query latency.

As shown in Figure 7 (b), the query cost of DUP is much smaller than that of PCX as θ increases, because DUP can deliver the update to hot spots with very low overhead. However, to push the index to interested nodes, CUP relies on many intermediate nodes. Since these nodes are less likely to access the index when θ is large, CUP does not perform well.

6) Effects of Pareto Arrival: In the previous simulations results, the query inter-arrival time follows an exponential distribution. In this section, we study the performance under the Pareto distribution, as some peer-to-peer networks exhibit such query arrival pattern. Pareto distribution has a parameter α that determines the burstyness of the queries. When α is small, the queries are more bursty, i.e., more queries arrive in a short time interval while there are longer idle time between such bursts. Two α values, 1.05 and 1.20 are used to study the effects of the burstyness of query arrivals.

As shown in Figure 8, in both cases, DUP performs much better than CUP. Generally speaking, all schemes perform better when α is 1.05, which means that the query burstyness improves the system performance. The reason is that more queries are generated in short intervals and the cached index can be used more times before it expires.

It is interesting to see when $\lambda > 30$ queries per sec-

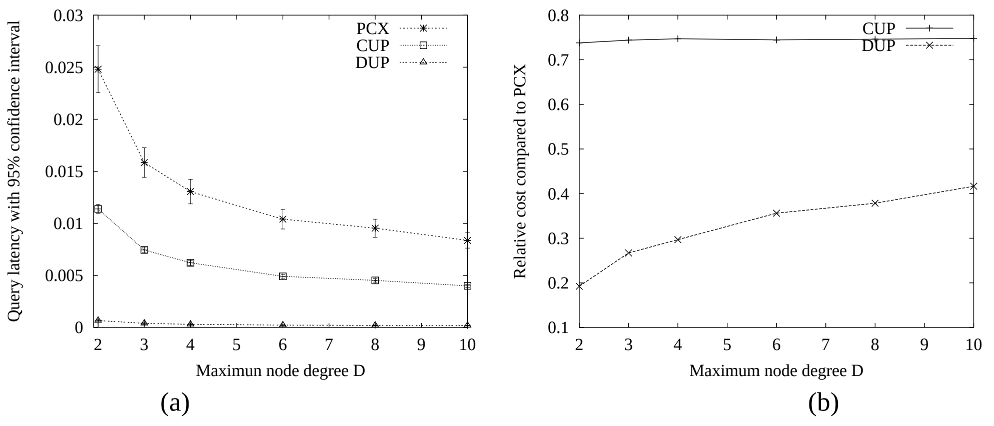


Figure 6. The effects of the maximum node degree

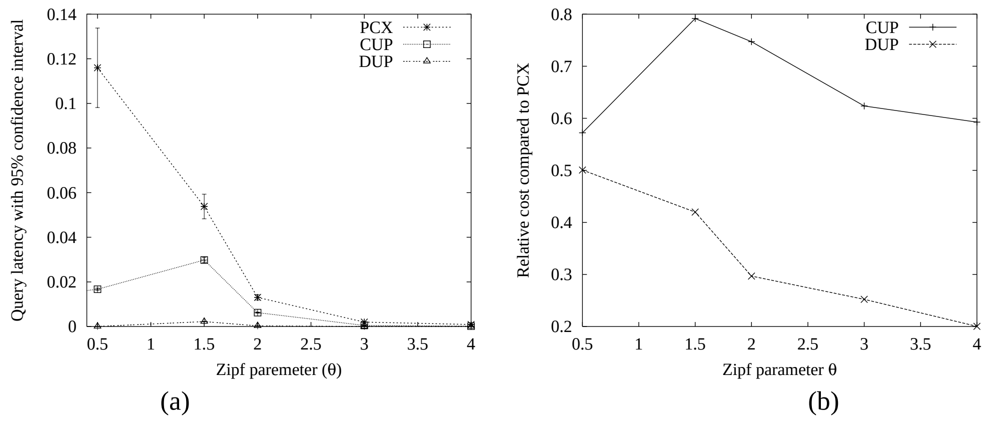


Figure 7. The effects of Zipf parameter θ

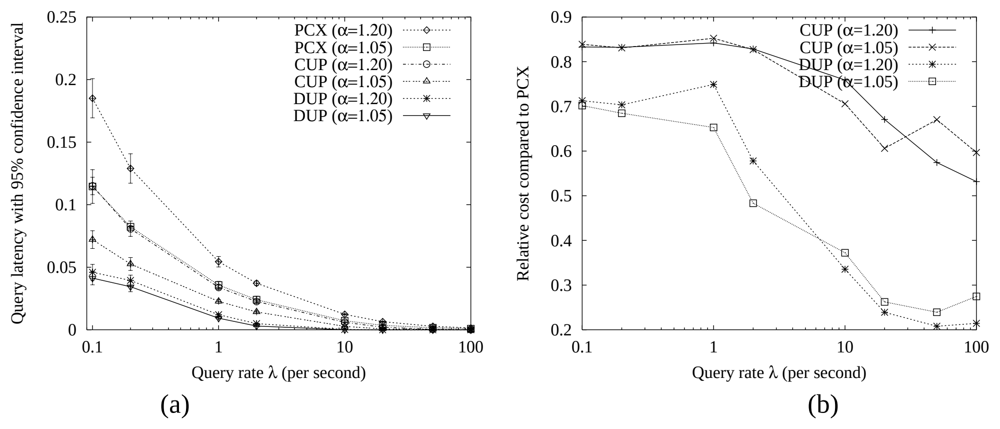


Figure 8. The Effects of Pareto arrival

ond, the relative query cost of CUP ($\alpha = 1.05$) and DUP ($\alpha = 1.05$) compared to PCX increases slightly. The reason is that when the query rate is high and the query is bursty, a node may be considered an interested node during the bursty time and becomes an uninterested node during the idle time. This affects the performance of the update propagation because some pushed updates are wasted. However, this effect is not significant and even with this effect, CUP and DUP still perform much better than PCX.

V. Related Work

The idea of caching indices to reduce query latency and network traffic has been studied in many researches [2], [3], [4], [9], [10], [18], [19]. In CAN [2], nodes cache the recently accessed indices to serve the queries from themselves or passing-by requests. Sripanidkulchai [18] and Markatos [10] proposed two similar schemes for Gnutella that cache the results of data queries (indices used in Gnutella) to reduce the query latency. These researches focus on caching the indices passively, but facing problems of cache consistency. Although TTL can be used for cache consistency, it increases the query latency when TTL expires. The CUP scheme proposed by Roussopoulos and Baker addresses this issues by using update propagation [11]. However, our performance analysis and simulation results show that even though CUP performs much better than pure index caching schemes, its performance is still limited.

DUP adopts the idea of application-level multicast [16], [20]. In Bayeux [20], each node joins a multicast group by sending a request all the way to the root, which then sends back the confirmation message to the requester through the underlying routing protocols. The root and all other nodes in Bayeux need to maintain the list of all their descendant nodes and process their descendants' join and leave requests. DUP is more scalable than Bayeux because each node only needs to maintain the information of its direct children in the DUP tree. SCRIBE [16] creates a multicast tree similar to the index search tree. The join and leave requests of a node are handled locally by its parent in the multicast tree. However, SCRIBE propagates data through the multicast tree similar to CUP. That is, intermediate nodes have to forward the data hop-by-hop to the subscriber. In DUP, intermediate nodes can be skipped to provide better performance.

VI. Conclusions

Index update propagation in peer-to-peer networks can significantly reduce the query latency and the query cost. In this paper, an update propagation scheme called DUP has been proposed. DUP builds a dynamic update propagation tree on top of the existing index searching structure with very low cost. Unlike the update propagation structure in some existing scheme, the DUP tree only involves the nodes that are essential for update propagation. By pushing updates along the DUP tree, both the query cost and the query latency can be reduced. Extensive simulation results showed that the proposed DUP scheme performs much better than existing update caching and propagation schemes.

DUP provides a low cost platform to propagate index updates in peer-to-peer networks. The idea of DUP may be applied to more general data dissemination scenarios. We plan to extend DUP to a general data dissemination platform in overlay networks.

References

[1] Napster, "http://www.napster.com," .

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

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

[4] B. Y. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," *Technical Report UCB/CSD-01-1141*, University of California at Berkeley, Computer Science Department, 2001.

[5] J. Xu, "On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks," *IEEE Infocom*, 2003.

[6] Gnutella, "http://www.gnutella.co.uk," .

[7] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A distributed anonymous information storage and retrieval system in designing privacy enhancing technologies," *LNCS 2009*, 2001.

[8] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," *ACM SIGCOMM*, 2002.

[9] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," *16th ACM International Conference on Supercomputing*, 2002.

[10] E. P. Markatos, "Tracing a large-scale peer to peer system: an hour in the life of gnutella," *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[11] M. Roussopoulos and M. Baker, "CUP: Controlled update propagation in peer-to-peer networks," *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

[12] J. Gwertzman and M. Seltzer, "World-Wide Web Cache Consistency," *USENIX 1996 Annual Technical Conf.*, Jan 1996.

[13] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, September/October 2003 (A preliminary version appeared in ACM MobiCom'00).

[14] O. Wolfson, S. Jajodia, and Y. Huang, "An Adaptive Data Replication Algorithm," *ACM Transactions on Database Systems*, vol. 22, no. 2, pp. 255–314, 1997.

[15] P. Cao and C. Liu, "Maintaining Strong Cache Consistency in the World-Wide Web," *IEEE Transactions on Computers*, pp. 445–457, April 1998.

[16] A. Rowstron, Anne-Marie Kermarrec, M. Castro, and P. Druschel, "SCRIBE: The design of a large-scale event notification infrastructure," *Networked Group Communication*, pp. 30–43, 2001.

[17] S. Saroiu, P. K. Gummadi and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," *Proceedings of Multimedia Computing and Networking (MMCN)*, January 2002.

[18] K. Sripanidkulchai, "The popularity of gnutella queries and its implications on scalability," *http://www-2.cs.cmu.edu/∼kunwadee/research/p2p/gnutella.html*, 2001.

[19] L. Xiao, X. Zhang, and Z. Xu, "On reliable and scalable peer-to-peer Web document sharing," *Proceedings of 2002 International Parallel and Distributed Processing Symposium*, 2002.

[20] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination," *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.