Lookup-ring: Building Efficient Lookups for High Dynamic Peer-to-peer Overlays[1]

Xuezheng Liu, Guangwen Yang, Jinfeng Hu, Ming Chen, Yongwei Wu

Department of Computer Science and Techonology
Tsinghua University, Beijing P.R.China
{liuxuezheng00,hujinfeng00,cm01}@mails.tsinghua.edu.cn, {ygw,wuyw}@tsinghua.edu.cn

Abstract. This paper is motivated by the problem of poor searching efficiency in decentralized peer-to-peer file-sharing systems. We solve the searching problem by considering and modeling the basic trade-off between forwarding queries among peers and maintaining lookup tables in peers, so that we can utilize optimized lookup table scale to minimize bandwidth consumption, and to greatly improve the searching performance under arbitrary system parameters and resource constraints (mainly the available bandwidth). Based on the model, we design a decentralized peer-to-peer searching strategy, namely the Lookup-ring, which provides very efficient keyword searching in high dynamic peer-to-peer environments. The simulation results show that Lookup-ring can easily support a large-scale system with more than 10^6 participating peers at a very small cost in each peer.

1. Introduction

The searching efficiency is a crucial factor for peer-to-peer (P2P) file-sharing systems (Napster [1], Gnutella [2], Kazaa [3]). Although centralized indexing is efficient (e.g. Napster, [1]), it has inherent defects [6] that research communities and internet users turn to decentralized systems, in which searching is performed cooperatively by forwarding queries among peers and use peers' lookup tables (containing replication of items' metadata) to find results (e.g. Gnutella [2], KaZaa [3]). Notable advancements [3, 4, 5, 6, 11, 13] have been made on decentralized searching to improve the performance, however, searching (especially searching by keywords) in decentralized P2P system still remains challenging.

Different from existing approaches which take into account either metadata replication [5, 6, 11, 13] or enhanced queries forwarding [4], in this paper we solve the problem of decentralized searching by simultaneously considering metadata replication and queries, and utilizing optimized lookup tables to minimize bandwidth consumption and greatly improve searching performance. Our concept is as follows: putting more metadata (e.g. file indices) in peers' lookup tables makes queries be resolved more quickly and reduces bandwidth costs on query forwarding; however, more indices imply that system variations (peers' joining or departure) will cause more corresponding updates for expired metadata and increase bandwidth costs on

[1] Supported by NSFC under Grant No. 60373004, No. 60373005, and 973 project numbered 2003CB3169007

metadata maintenance. So, there is a basic trade-off between queries and metadata maintenance, and we model this trade-off to find the optimized scales of peers' lookup tables, so as to minimize total bandwidth consumption or maximize searching performance under given environment parameters. In Section II, we propose the model to estimate optimized lookup table scales, and find that both bandwidth consumption and average searching hops can be reduced to $O(N^{1/2})$ (N is the number of peers) in comparison with the $O(N)$ complexity in conventional random walk strategy [5]. Based on the model, we propose a decentralized P2P file-sharing system, the *Lookup-ring*, which implements a general searching strategy with nearly optimal performance under arbitrary system parameters (system scale, magnitude of shared files and frequency for users issuing queries, etc) and resource constraints (mainly the bandwidth constraint in peers). In current Internet environment, Lookup-ring can easily afford a system with more than 10^6 peers where most searching queries are resolved within a few hops.

The rest of paper is organized as follows. Section II gives the model. Section III presents details of Lookup-ring design. Section IV presents performance evaluation. Section V discusses related works and Section VI concludes the paper.

2. Model for bandwidth and trade-off

In this section, we propose an analytic model to estimate bandwidth consumption and describe the trade-off between querying and metadata maintenance. We first define notations in the model (see Table.1). We consider a system consisting of N peers (N is around 10^6) and sharing U unique files (we don't count file replicas in U), denoted by $f_1, f_2, \ldots f_U$. Each unique file may have some replicas shared by users who download the file. We use r_i to denote the number of f_i's replicas, and TR to denotes the total replica number ($TR = r_1+r_2+\ldots+r_U$). For system variations, the peers' average session time is denoted by $T_{session}$. Based on measurement works [7, 8] we have referenced values of these system parameters, as listed in the Table.1 (these values are only used for reference in the model, not necessary).

Considering that there are totally k_i indices of file f_i in all peers' lookup tables (for i=1…U), we call k_i as f_i's "*indexing factor*". The search process is a sequence of probes: when a peer is probed, it attempts to match the query on its local file indices; we assume the searching is perfect and strict, i.e. a query for file f_i can always and only be resolved by a probe to peer containing an index to f_i. For random search process, the *search size* (number of probed peers) for resolving a query of f_i is a random variable, with the expectation equal to N / k_i [13, 4].

Now we present the model. First we estimate bandwidth costs for querying. Unique files have their respective popularities modeled by *query distribution*. Let $\boldsymbol{q} = < q_1, q_2, \ldots q_U >$ be a vector of probability that sum to 1, where q_i is the probability that a query is for file f_i. Therefore, $\boldsymbol{q}$ is the query distribution [13, 4]. Considering there are totally Q queries submitted per second, the totally bandwidth for querying is:

$$BW_{\text{query}} = \sum_{i=1}^{U} Q \cdot q_i \cdot (\frac{N}{k_i} - 1) \cdot m_q \qquad (1)$$

where m_q is the average size of querying message (bits), and (N / k_i-1) is the expectation of hops to resolve a query for f_i.

Table 1. Notation and Model Parameters.

Parameter	Meanings	Referenced value
N	Number of peers in the system	10^6
U	Number of unique files	$10 \cdot N = 10^7$
$f_1, f_2, \ldots f_U$	Unique files shared in the system	
r_i	Number of f_i's replicas	Zipf
TR	Total replica number. $TR = \sum_{i=1}^{U} r_i$	$200 \cdot N = 2 \cdot 10^8$
Q	Number of queries per second	$1/60 \cdot N$
$T_{session}$	Average peers' session time (on-line time)	1 hour
λ_{peer}	Poisson parameter for peer variations	1/3600
V_{peer}	Number of peer variatons per second for both join and departure	$\lambda_{peer} \cdot N$
V_{file}	Number of file varations per second for both adding and removing files	$1.74 \times 10^{-3} N$
k_i	Number of indices of f_i	
$\boldsymbol{q}=<q_1, \ldots q_U>$	Query rate distribution. $\sum_{i=1}^{U} q_i = 1$	$q_i \propto r_i$
m_q, m_p	Average message size for querying and updating expired index	0.5KByte
R_{msg}	For redundant messging: peer receives one updating message for R_{msg} times.	

Second, we estimate the maintenance costs. When replica variation occurs, we need to update the affected indices in lookup tables. So, the bandwidth costs is made up of the following parts: $BW_{\text{peer\_depart}}$ for updating expired indices pointing to a leaving peer; $BW_{\text{peer\_join}}$ for a joining peer downloading its lookup tables from others; and BW_{file} for updating lookup tables due to file variations (both sharing new files and removing shared files). We use V_{peer} and V_{file} to denote the variation frequency (time per second) for peer and file respectively. Peer variation are usually modeled with Poisson distribution with parameter $\lambda_{peer} = 1/T_{session}$ [14, 19], and we have $V_{peer} = \lambda_{peer} \cdot N$ for both joins and departures. For V_{file}, in [7] we know the largest number of successful downloaded files per peer per day is no more than 75 files (a very large number), and thus $V_{file} \approx 2 \cdot 75/(24 \times 3600) \cdot N$ for both new downloaded files and removed files. (The model describes stationary system behavior, so we assume number of new files to be approximate equal to deleted files in certain duration.)

Now we calculate maintenance costs. A failure of file replica invalidates all indices pointing to it. These "expired" indices should be updated sooner or later; otherwise the total number of valid indices will decrease. We suppose the indexing assignment has no preference for replicas with higher availability. Thus, for a unique file f_i with r_i replicas and totally k_i indices, one replica failure will averagely cause k_i/r_i expired indices. For $BW_{\text{peer\_depart}}$, seeing that departure of a peer P causes failures of all its replicas, for file f_i with r_i replicas the probability that a departing peer P contains f_i is r_i/N. So the expectation of expired indices caused by a peer departure is:

$$\text{expired number} = \sum_{i=1}^{U} \frac{r_i}{N} \cdot \frac{k_i}{r_i} = \sum_{i=1}^{U} \frac{k_i}{N} \tag{2}$$

So, bandwidth consumption for maintaining them is:

$$BW_{\text{peer\_depart}} = \sum_{i=1}^{U} \frac{k_i}{N} \cdot V_{peer} \cdot R_{msg} \cdot m_p \tag{3}$$

where m_p is the average size of updating message. In (3) we use R_{msg} to denote the *redundancy factor of messages*, which indicates that in order to updating a single index, a peer will averagely receive R_{msg} times of the corresponding updating message, each of which has complete updating information (this is defined for non-acknowledged messages. For acknowledged message transmission, R_{msg} is defined as the double value of non-acknowledged case). The R_{msg} is a system parameter to characterize updating algorithm in specific system. To make update tolerant to message lost, some algorithms utilize redundant messaging where peers may receive the same message more than once. In the model we use R_{msg} to reflect this manner.

A peer loses its lookup table after departure, and should download entire table in join time. The average number of indices in lookup table is $\sum_{i=1}^{U} k_i / N$, So:

$$BW_{\text{peer\_join}} = \sum_{i=1}^{U} \frac{k_i}{N} \cdot V_{peer} \cdot m_p \tag{4}$$

For file variations, from above analysis a variation of f_i may generate $k_i/r_i \cdot R_{msg} \cdot m_p$ bandwidth cost for updating indices, so totally bandwidth is:

$$BW_{\text{file}} = V_{file} \cdot R_{msg} \cdot m_p \cdot \sum_{i=1}^{U} \frac{k_i}{r_i} \cdot \frac{r_i}{TR} \tag{5}$$

where TR is total replica number (see Table.1), and the number of variations for f_i is assumed to be proportional to f_i's replicas' number r_i (i.e. number of peers containing f_i). Thus, summate all these cost and we will have the estimation of total bandwidth consumption, as follows:

$$BW_{\text{total}} = BW_{\text{query}} + BW_{\text{peer\_depart}} + BW_{\text{peer\_join}} + BW_{\text{file}} \tag{6}$$

Notice that $\{k_i\}$ are independent, and we can minimize each term of the summation in (6) by choosing the best k_i. The optimized choices of k_i for minimum BW_{total} is:

$$k_i^* = N \cdot \sqrt{\frac{Q}{V_{peer} + R_{msg} \cdot (V_{peer} + N/TR \cdot V_{file})} \cdot \frac{m_q}{m_p} \cdot q_i} = N \cdot \theta \sqrt{q_i} \tag{7}$$

subject to $k_i^* \leq N$ (recall that k_i is the indexing factor of file f_i), where θ is a system parameter independent to i. The minimum BW_{total} is:

$$\text{BW}_{\text{total}}^* = BW_{\text{total}}\big|_{k_i = k_i^*, i=1\cdots U} = \frac{2m_q Q}{\theta} \sum_{i=1}^{U} \sqrt{q_i} \leq \frac{2m_q Q}{\theta} \sqrt{U} \tag{8}$$

where we used Cauchy inequality to the summation. So, the average bandwidth cost in each peer based on k_i^* is:

$$\text{bw}_{\text{per peer}} = \frac{BW_{\text{total}}^*}{N} \leq \frac{2m_q}{\theta} \frac{Q}{N} \sqrt{\frac{U}{N}} \cdot \sqrt{N} \tag{9}$$

From (7) the optimized k_i is proportional to the querying rate $q_i^{1/2}$ for each file f_i. It is clear that k_i is a trade-off between querying and maintaining the system, since the

numerator $Q \cdot m_q \cdot q_i$ and denominator $(V_{peer}+R_{msg} \cdot (V_{peer}+N/TR \cdot V_{file})) \cdot m_p$ represent cost for queries and maintenance respectively. In (9) we have average bandwidth cost per peer. The Q/N is the number of queries each peer submits to system per second (e.g. 1/60). U/N is also a stationary environment parameter (e.g. 10~20 based on [8]). So, the bandwidth cost per peer is $O(N^{1/2})$ in optimization. Because of the square root, the scalability of system with optimized indexing factor is fairly good. Using practical parameter values in (9), we find the random searching strategy becomes surprisingly powerful under optimized indexing factors (i.e. optimized lookup table scale). For example, for $N=10^6$ peers with only 1 hour session time, using $R_{msg}=5$ (very redundant messaging) and other reference values in Table.1, the optimized strategy can support the heavy queries where each peer submit a query per minute, within only 15Kbps bandwidth per peer (both upstream and downstream). This is a very low bandwidth cost that modem connections can easily afford, and we can even reduce it with more efficient messaging (lower R_{msg}). For comparison, based on report in year 2001 [18], in Gnutella each peer consumes more than 150 Kbps bandwidth both upstream and downstream.

The model illustrates theoretical lower bound of peer consumptions for constructing a lookup system based on random searching process. It shows that with appropriate lookup scales and updating mechanism, a uniform system (i.e. no supernodes) with simple unbiased search is capable to support very large systems. In the following sections we give practical design derived from the model.

3. Design of Lookup-ring

This section presents design of Lookup-ring Lookup-ring is derived from the model to achieve optimized performance, in which indexing factors is calculated based on equation (7). Lookup-ring is built on top of most structured P2P infrastructures, e.g. Chord, Pastry [9] and SkipNet [20]. In this paper we illustrate how it works on top of Pastry and SkipNet as example. For details of their structures, please refer to [9, 20].

3.1 Indexing factor and file levels

Assuming we know the query rate distribution $<q_1,q_2,\ldots q_U>$ (due to limited space, we do not provide estimation of q_i, but only point out it is reasonable to assume q_i to be proportional to replica number r_i), we can obtain best indexing factor k_i^* with (7). We first quantify k_i^* into discrete levels, and files whose k_i^* belong to the same level have the same actual (quantified) indexing factor k_i. The indexing factor is quantified into m levels with radix 2, i.e. we use a set of m kinds of indexing factor values $\boldsymbol{M}=\{N, 2^{-1} \cdot N, 2^{-2} \cdot N, \ldots, 2^{-(m-1)} \cdot N\}$ for all indices. For f_i with k_i^*, the actual indexing factor k_i should be the closest $2^{-j} \cdot N$ in $\boldsymbol{M}$ to k_i^*, and we call f_i as a "j-level" file.

3.2 PeerId, fileId and peer groups

In Lookup-ring, each peer is assigned with a unique and uniformly distributed peerId. We also generate a uniformly distributed fileId for each *unique* file by hash functions. We use peerId to partition peers into groups, and fileId to match unique files with groups.

Peers are partitioned into hierarchical groups as follows. All peers with the same j-bit prefixes in peerIds are united into a j-level group, for $j=0,1,\ldots(m-1)$. A j-level

group is denoted by the *j*-bit common prefix of containing peers. The prefix of a group is also called the group's *groupId*. For example, 010-group is a 3-level group with groupId "010", which consists of all peers with the same 3-bit prefix "010". Due to uniformity of peerIds, a *j*-level group has approximately $2^{-j} \cdot N$ peers.

Each *j*-level unique file is matched to one *j*-level group whose groupId equals to *j*-bit prefix of the file's fileId. If a file is matched to a group, all peers belong to the group should contain the file's index in their lookup tables. Thus, a peer *P* with peerId id_P contains indices of all *j*-level files with fileIds sharing id_P's *j*-bit prefix, for j=0, 1, …, m, and a *j*-level file is indexed by approximately $2^{-j} \cdot N$ peers, as our original purpose. Consequently, any query for a *j*-level file can be resolved by traversing all *j*-level groups, i.e. forwarding query to 2^j peers with different *j*-bit peerId prefixes. If doing so, we also obtain all unique files with file level less than *j*. Therefore, we can resolve any query by traveling $2^{-(m-1)} \cdot N$ peers with different (m-1)-bit prefixes.

Since Lookup-ring is built on top of structured P2P infrastructure, the peer partitioning ought to be consonant with underlying peer organization, and peerId should have ability to partition peer into groups in DHT organization. If Lookup-ring is built on Pastry, we use Pastry's nodeId as peerId in Lookup-ring, because in Pastry's organization the nodeIds plays the role of partitioning nodes into prefex-based groups.[2] The peer groups and file levels are shown in Fig.1.

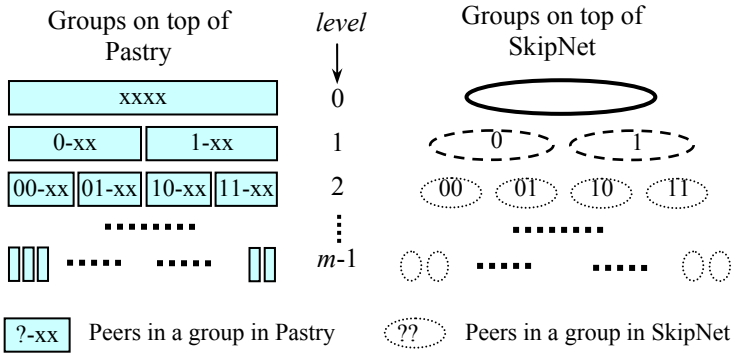

Fig. 1. Partition on top of Pastry and SkipNet.

3.3 Searching in Lookup-ring

Lookup-ring provides searching by keywords and substrings. Each unique file is associated with a "label" for searching, e.g. the filename. Each file index defines a match between a unique file's label and location of *one of* the file's replicas. Thus, a file index contains the file label, IP of location. We also store the file's fileId and location's peerId in the index. For 20~30-byte file label, we need only about 64-byte file index. Based on the model, each peer of a 10^6-peered system needs to contain about 10^4 indices, and the lookup table size is no more than 1Mbyte.

Lookup-ring has a "*prefix-traversing*" searching strategy. Consider a query *q* submitted in peer P_0. We first check P_0's lookup table to see whether *q* could be

[2] Other P2P infrastructures (e.g. chord) also have similar nodeId playing the partitioning role. For SkipNet, note that it is a "bi-id" system whose NameId indicates proximity between nodes while NumericId partition nodes into hierarchical rings. So, we use SkipNet's NumericId as the peerId in Lookup-ring.

resolved locally. Here we obtain all 0-level results of q. If q is satisfied (i.e. get enough results) we stop searching, otherwise q is forwarded to a peer P_1 which has a different 1-bit prefix with P_0. From both P_0 and P_1 we can find all 1-level results of q, because P_0 and P_1 represent all 1-level groups. If still unresolved, we keep up this prefix-traversing. In general, before the j-th step q has been forwarded to $2^{(j-1)}$ peers $\{P_0, P_1, \ldots P_{2\hat{}(j-1)-1}\}$ with the peerIds covering all (j-1)-bit prefixes. In the j-th step we forward q to another $2^{(j-1)}$ peers namely $P_{2\hat{}(j-1)},\ldots P_{2\hat{}j-1}$, so that j-bit peerId prefixes of all searched peers $\{P_0, P_1, \ldots P_{2\hat{}j-1}\}$ have covered all the j-bit prefixes. After that, we have traversed all groups with no more than j levels and found all results whose levels are no more than j. The searching process stops either query is resolved or we reach the last level ((m-1)-level) when all unique files' indices has been searched.

Due to the consistency of Lookup-ring' peerId with underlying DHT, it is very easy to perform prefix-traversing searching, because it is a natural property of most DHTs to perform such prefix-traversing [16]. Therefore, searching in Lookup-ring is very efficient without redundant query forwarding.

3.4 "Principle of logical locality" for location choices

Because a unique file usually has more than one replica, there is a problem for choosing location for file's indices. For each of the file's index we choose *only one* replica as the location. We propose our "principle of logical locality" for choosing locations of indices and for easy maintenance.

For a j-level unique file f_i, all f_i's indices are stored in a matched j-level group g. If one of f_i's replicas P fails, we need to efficiently update all affected indices in g, i.e. indices picking P as f_i 's location. To make maintenance easy and save bandwidth, peers in g whose indices of f pick the same location should to be situated in a *logical locality* in g (i.e. a continuous region is id-space), so that we can perform *locality-based update* in which messages are precisely spread to all peers in the affected region that exactly "need" the update, while other peers will not receive the message. For this purpose, we use "*principle of logical locality*" to choose location for each index, i.e., *when chooses the location for an index of a unique file, a peer should always pick the logically "closest" replica of that file.* In other words, location in peer's index should always be the living replica which is current the closest one to the peer in logical distance. The goal of maintenance is to keep this invariance after system variations.

Similar to the peerId, here the "locality" should also be consonant with underlying peer organization to facilitate updating algorithm. In Pastry, both locality and peer portioning is based on Pastry's nodeId. Thus, we ask each Lookup-ring peer to choose the replica whose peerId is currently the closest one. Obviously this design fits all fundamental considerations of our design, e.g., locality-based updating, since peers choosing the same location of a file are logically adjacent in DHT's id-space. However, from Fig.1 and Fig.2 the replica locations being chosen in a certain group are not uniformly distributed, since peerIds in a group have a common prefix and do not fill in peerId-space where replicas' peerIds scatter themselves. So, we can extend this approach to get better uniformity of choosing locations (this extension is not necessary to Lookup-rings). We first map peerIds of replicas into the group with linear transformation before using principle of locality. For a j-level file f matched to a j-level group g with j-bit groupId id_g, we map all peerIds of f's replicas into g by

right-shifting them by *j*-bits and add *j*-bit prefix id_g. After this linear transformation, replicas' mapped peerIds are uniformly distributed in *g* while also keep their primary order. Then, peer in *g* picks the replica of *f* whose mapped peerId is the closest one. Fig.2 shows this mapping, and in Fig.2 the I, II, and III are the three sections in the group (i.e. locality) which consist of peers choosing replica *a*, *b*, and *c* as the location of index, respectively. When variation occurs (e.g. *b* suddenly fails), peers in section II should be updated with new replica locations. Based on the principle, section II is then divided into I' and III' which should update their locations with *a* and *c*, and be merged into I and III, respectively. The boundaries of I' and III' can be determined only with peerId of *a*, *b* and *c* (the boundary between I' and III' has the equivalent distance to peerId(a) and peerId(c)). So, after *b*'s failure we have the following update strategy: *b*'s neighbor replica *a* and *c* find *b*'s failure (how they find the failure is explained in Section 3.5), send their locations and boundaries of I' and III' to two certain peers in I' and III' correspondingly (the dashed lines in Fig.2), and these peers spread received messages in I' and III' for updating all other peers in the either locality. When *b* joins there's a similar process: *b* calculates section II's boundaries from *a*'s and *c*'s locality and spread updating message in II.[3]

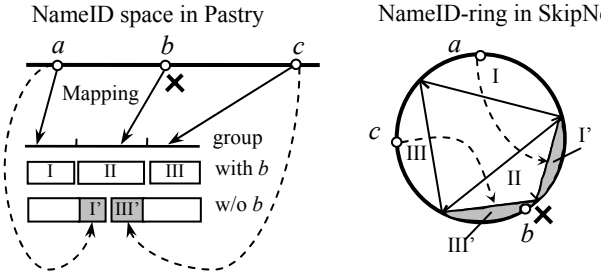

Fig. 2. Principle of locality in Lookup-ring, and the updates of indices after variation

3.5 Maintaining Lookup-ring

For maintenance, we should actively detect variations and update affected indices. We construct *file-ring* in Lookup-ring, where all replicas of a unique file connect to form a ring structure and keep connections with heart-beating messages, so that variations can be soon detected. After that, the detector generates an appropriate update immediately.

3.5.1 File-rings

A file-ring is shown in Fig.3. Consider a certain unique file f_i with *r* replicas stored in *r* peers. These peers are connected into a ring structure (file-ring), ordered with logical locality in DHT, namely $P_1,P_2,\ldots P_r$ (logical locality is defined in last Section). Peers participating in a file-ring should hold the links to its two neighbors (predecessor and successor in ring) and send "heart-beating" messages to them every T_{probe} of time to maintain connectivity. A peer may participate in many file-rings according to its shared unique files.

[3] In SkipNet the logical locality is defined by NameId rather than NumericId, because in each SkinNet-ring the sequence and neighborhood of peers are indicated by NameId. So, on top of SkipNet we use NameIds of replicas to determine the sections and guide location choosing. Fig.2 shows one group (i.e. a SkipNet-ring) and its sections based on replicas' NameIds.

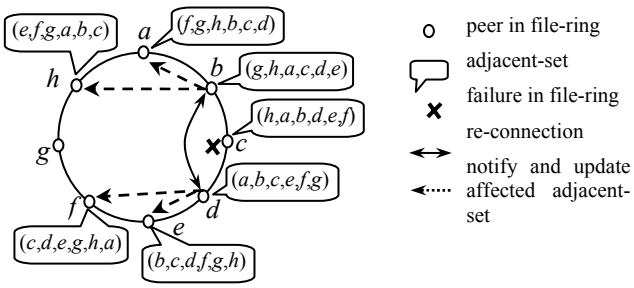

Fig. 3. Maintaining file-ring connectivity

3.5.2 Active variation detection and file-ring recovery

Peers periodically probe their file-ring neighbors. To reconnect broken file-ring, peers should be aware of not only its direct neighbor but also some nearby peers, namely "adjacent-set", similar to Pastry's leaf set [9]. When neighbor fails, a peer can reconnect file-ring with adjacent set. Then, it keeps heart-beating with its new neighbors, and also notify nearby peers for updating their adjacent-sets. Fig.3 shows a file-ring with 8 peers and adjacent-sets. When replica *c* fails (either peer failure or dropping replica), *b* and *d* will detect the failure after (T_{probe}+T_{out}) and begin to repair file-ring. Peer *b* and *d* first find each other from adjacent-sets and reconnect file-ring (*b* and *d* exchange adjacent-sets for verification and updating). Then, *b* sends its new adjacent-set to *a* and *h* for updating expired adjacent-sets in them, and *d* also updates *e*'s and *f*'s adjacent-sets. For replica *c* joining file-ring, a similar procedure is performed that *b* and *d* receive the joining request, break their interconnection and turn to keep the connection with *c*, and notify *a*, *h* and *e*, *f* for updating adjacent-sets.

We set the T_{probe} as 60 seconds and keep 16 peers in the adjacent-set. For variation, each detector notifies 7 peers in its side, with an acknowledged messaging.

3.5.3 Updating lookup tables after system variations

Replicas in file-ring are ranged by logical locality; therefore the two detectors of replica failure are exactly the two replicas whose locations should be used to update expired indices. Thus, in Fig.2 peer *a* and *c* will detect *b*'s failure within (T_{probe}+T_{out}) due to file-ring heartbeat messaging. After that, *a* and *c* calculate their respective updating sections (i.e. I' and III'), and each of them immediately sends an update message to *one peer* in corresponding section for maintaining lookup tables, via underlying P2P routing. The update message contains the new location of replica (*a* or *c*), the fileId of the unique file, and boundaries of section inside of which the message should be spread. The peer receiving the updating message then spreads it to entire updating section in the file's group, by way of message broadcasting algorithm in the underlying structured peer organization.

In detail, most DHTs can perform *locality-based message broadcasting* as a basic service, i.e. broadcasting messages to all peers in a consecutive section based on its logical locality in a partitioned group [16]. Lookup-ring utilizes DHT-based broadcasting for spreading its update messages, following the algorithm proposed in [16]. On top of Pastry (and Chord, etc), we can derive from the routing tables a spanning tree for an arbitrary nodeId section (rooted by any peer in the section). Via

broadcasting the first peer can spread the update information to all other M peers in the section through exactly (M-1) messages [16]. This broadcast has no message redundancy that each peer receives the needed message exactly once. So, the R_{msg} in the model (see Section.2) should be 1. To further guarantee the update, we use confirmed messaging that all updating messages should be acknowledged. If an acknowledgement is not received within a timeout period the message is retransmitted. Therefore, considering both acknowledgement and redundancy during broadcast, the message redundancy factor R_{msg} in our model should be 2 for Pastry.[4]

4. Performance evaluation

We perform our evaluation of Lookup-ring with simulations. We run our simulator on Linux running on Pentium IV CPU with 2G memory, which can support more 10^4 simulated peers. We construct and evaluate Lookup-ring on top of SkipNet. We implement SkipNet based on [20], using basic type of SkipNet with only R-table and density parameter k equal to 2. For shared files, the unique file number is 10 times of the peer number and the total file number is 200 times of peer number, derived from [8, 7]. The simulation has two aspects. First we examined feasibility and efficiency of Lookup-ring by simulating environments with different peer numbers and peer availabilities, in order to see bandwidth cost in each peer to support a heavy query load (one query per peer per minute). Second, we compared Lookup-ring with random walks searching [5] in order to see how much improvement we have gained. We are mostly concerned about the following two metrics: the average search size (in hop number), and the maximum query workload under a fixed bandwidth. The former indicates how quickly a query is resolved, and the latter shows system scalability.

Fig.4.a shows the messages and bandwidth consumptions for each peer in supporting one query per peer per minute, under different peer availability ($T_{session}$) and system scale (number of total peers). If m_q and m_q is both 1Kbit message on average, the values in y-axis of Fig.4.a is also the needed bandwidth for each peer. We can see the trend of bandwidth consumptions when enlarge system scale, which is roughly in proportion with the *square root* of peer number, e.g. when expand peer number for 10 times from 10^3 to 10^4, the bandwidth increase from 0.36 to 1.42 Kbps (i.e. 3.9 times, nearly $10^{1/2}$) for 1.5 hours online time ($T_{session}$=5400s). From this trend, we can deduce that for 10^6 peers, the bandwidth is nearly 10 times of the case with 10^4 peers, i.e. nearly 16Kbps for $T_{session}$=3600s and 12Kbps for $T_{session}$=7200s, in both upstream and downstream. This result shows very good scalability for large systems.

Fig.4.b is the average search size of Lookup-ring and random walk. The results demonstrate the improvement of search size in our strategy. In 10^4 peers, the search size is only 1/40 of random walk. This outperforming becomes more remarkable when peer number N grows, since we have $O(N^{1/2})$ search size while random walk is nearly $O(N)$.

Fig.4.c is the comparison of maximum supported query workload under different bandwidth. We compare Lookup-ring with random walk in system of 5000 and 10000 peers (T_{sesson}=3600s), with 1Kbit querying messages. From the results we also see that Lookup-ring greatly overcome the Gnutella-like system, esp. when system scale

[4] On top of SkipNet there's a slight difference that the derived spanning tree has some redundancy, where each peer will averagely receive an updating message for 1.5 times, and R_{msg} should be 3 for confirmed messaging. Here we omit the discussions and readers can refer to [12] for details.

grows. The reason is because by using adaptive indices, we significantly save query hops and simultaneously constrain the maintenance cost to a low level.

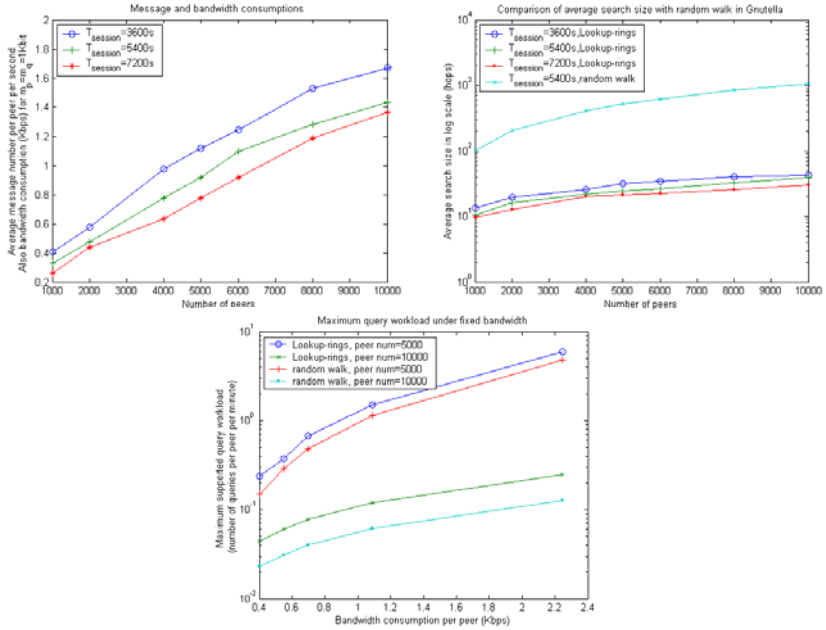

Fig. 4. Performance evaluation. a) up-left. b) up-right. c). bottom.

5. Related works

To improve searching efficiency, researches try to exploit all aspects of typical query-based decentralized searching. In strategy of forwarding queries, [5] propose to replace flooding-based query-forwarding with random walks, so that network traffic is reduced. [4] further exploit data correlations and user interests to guide forwarding directions and improve searching performance. Instead of [4], Lookup-ring doesn't need specific data correlations, and thus is suitable for more applications. In the aspect of local lookup tables, results caching [11] and supernode [3] are employed. [13] suggests replicating files in accordance with their query rates, so that the expectation of searching size is optimized. In comparison, Lookup-ring has fully controlled and optimized caching (the indices), and doesn't need supernode. Recently, researchers present to employ biased overlay topology towards peers with larger lookup tables, and Gia in [6] is an integrative design combining many above features. For DHT-based approaches, most DHTs support only precise search with precise resource ID, while the others have very limited capability in keyword search [6, 17]. Lookup-ring uses DHT as underlying organization for system maintenance, and the efficient keyword search is built on a higher level.

6. Conclusions

Our contribution is in the following aspects. First, we propose an analytic model to describe trade-off between query and maintenance, based on which the optimized lookup table scales can be estimated. Second, we design a efficient decentralized P2P searching strategy, where there are no supernodes and all peers are utilized uniformly. Third, we demonstrate the maximum query load and system scale that an unbiased decentralized P2P system can support. We show that unbiased decentralized P2P system can achieve a heave query load in a large-scale system, with low peer costs.

Reference

[1] Napster, the napster homepage. In http://www.napster.com/
[2] Gnutella, In http://www.gnutella.com
[3] KaZaA, file sharing network. In http://www.kazaa.com
[4] E. Cohen, A. Fiat, and H. Kaplan. Associative Search in Peer to Peer Networks: Harnessing Latent Semantics. In Proceedings of the IEEE INFOCOM'03 Conference. 2003
[5] Q, Lv, P. Cao, E. Cohen, K. Li, S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks . In Proceedings of 16th ACM International Conference on Supercomputing (ICS'02), 2002.
[6] Y. Chawathe, S. Ratnasamy, L. Breslau, N Lanham, S. Shenker. Making Gnutella-like P2P Systems Scalable, In Proceeding of ACM Sigcomm'03
[7] S. Saroiu, P. K. Gummadi, S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In Proceedings of Multimedia Computing and Networking 2002 (MMCN'02), CA, Jan. 2002.
[8] J. Chu, K. Labonte, and B. Levine. Availability and locality measurements of peer-to-peer file systems. In Proceedings of ITCom: Scalability and Traffic Control in IP Networks, July 2002.
[9] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In IFIP/ACM Middleware, Nov. 2001.
[10] J. Wang. Gnutella bandwidth usage. Nov. 2001. https://resnet.utexas.edu/trouble/p2p-gnutella.html.
[11] B. Bhattacharjee, et al. Efficient Peer-To-Peer Searches Using Result-Caching, In IPTPS'03
[12] X. Z Liu, J. F. Hu, D. X. Wang. Lookup-Rings: Building Efficient Lookups for High Dynamic Peer-to-peer Overlays. In http://166.111.68.162/granary/index.htm
[13] E. Cohen and S. Shenker. Replication strategies in unstructured Peer-to-Peer networks. In Proceedings of the ACM SIGCOMM'02 Conference. 2002
[14] A. Gupta, B. Liskov, R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. In HotOS IX, 2003
[15] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van R. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In IPTPS, 2003.
[16] S El-Ansary, L Alima, P. Brand, S. Haridi: Efficient broadcast in structured P2P networks. In IPTPS'03.
[17] M. Harren, J. M. Hellerstein, R. Huebsch. Complex queries in DHT-based P2P Networks. In IPTPS'01.
[18] J. Wang. Gnutella bandwidth usage. Nov. 2001. https://resnet.utexas.edu/trouble/p2p-gnutella.html.
[19] Z. Ge, D. R. Figueiredo, S. Jaiswal, J. Kurose, D. Towsley. Modeling Peer-Peer File Sharing Systems. In Proceedings of IEEE INFOCOM'03, 2003
[20] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In Proceedings of 4th USITS, Mar. 2003.