Supporting Multiple-Keyword Search in A Hybrid Structured Peer-to-Peer Network

Xing Jin W.-P. Ken Yiu S.-H. Gary Chan
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
Email: {csvenus, kenyiu, gchan}@cs.ust.hk

***Abstract*— Most existing techniques for keyword search in structured peer-to-peer (P2P) networks only support single-keyword exact-match lookups. In practice, however, users often have fuzzy information for identifying these items and tend to submit broad queries. The support of searching based on multiple keywords is hence desirable. In multiple-keyword search, a data item is associated with multiple keywords for storage. A query may also contain multiple keywords. The search result for a query should include all the data items whose storage keywords contain all the query keywords. Traditional DHT-based approaches achieve this by storing a data item (or its index) multiple times, each time with one of its keywords. A query is processed by searching each of the query keywords once. Therefore, the storage cost and search cost are both linear with the number of keywords. Clearly, it is not efficient in case of a large number of keywords.**

In this paper, we propose a hybrid structured network called *MKey* to address this problem. Its backbone is a structured network. Each node in the backbone is also the leader of a cluster formed by non-backbone nodes. Within a cluster, nodes form an unstructured network and cooperate to store data and answer queries. When inserting a data item, multiple copies of its index are stored in a few different clusters. A query is also mapped to multiple clusters, and a flooding search within these clusters is performed. The union of all the search results are returned to users as the final result.

As compared to traditional approaches, MKey has upper bounds for both the number of storage copies and the number of searching clusters for a query. Simulation results show that it can efficiently reduce storage cost and search cost, especially for a large number of keywords. Meanwhile, it can achieve good load balancing among nodes.

I. Introduction

In recent years there has been significant interest in P2P file sharing. Due to the distributed nature of content and dynamic membership of nodes, searching has become one of the core operations in most P2P networks. Current popular P2P systems can be generally classified as unstructured and structured, depending on their network structures. The first category refers to Gnutella-like systems that do not impose any structure on the overlay network [1], [2]. The dominant search mechanism is blind or informed flooding. The second category consists of solutions that impose a particular structure on the overlay network, which are commonly referred as Distributed Hash Tables (DHTs) [3]–[6]. These systems are efficient because data items can be located in a small number of hops. However, a major limitation of DHT systems is that they only support lookups with exact match, i.e., a search query can only contain a single keyword and the system can locate data items with exactly the same keyword. In practice, however, users often have only partial information for identifying these items. And it is hard to store an item with a universally accepted keyword. For example, if a user wants to find a Minnie Riperton's song *Loving you*, he may submit a query like "Minnie Riperton Loving you", or "Minnie Loving you", or "Loving you". However, only one of them can find the desired result in DHT systems. Therefore, we would like to develop fuzzy search functions in structured networks. For example, some websites providing BitTorrent indices (e.g. *http://bt.btchina.net*) have supported fuzzy search as follows. Each BitTorrent index (usually called a *.torrent* file) has a small description file with several hundred words. Users can submit a query with multiple keywords, and the *.torrent* files whose description files contain all the query keywords are returned. These websites use central servers to store description files and can easily conduct search. However, in structured peer-to-peer networks it becomes much more difficult.

In fact, researchers have considered many ways to improve search functions in DHTs. For example, [7] uses latent semantic indexing technique to build up full-text search in structured overlays; [8], [9] combine structured and unstructured overlays to reduce search cost. Different from these works, we consider multiple-keyword search in structured networks. We assume a data item shared by some peer can be represented by one or more keywords. These keywords are termed *storage keywords* since they determine the storage locations of the item. A query may also contain one or more keywords. The desired results are all the items whose storage keywords contains the keywords in the query.

Most proposed approaches supporting multiple-keyword search in DHTs use data replication method [10]–[13]. Generally, an item with multiple keywords is stored multiple times. Each time one of the keywords is used as the key to insert it into DHT nodes. When looking up some file, the query is resolved into several keywords, and each keyword is searched once. All the search results are sent back to the query initiator and their intersection is returned to the user as the final

This work was supported, in part, by Direct Allocation Grant (DAG05/06.EG10) and Competitive Earmarked Research Grant (HKUST6156/03E) of the Research Grant Council in Hong Kong.

1-4244-0355-3/06/$20.00 (c) 2006 IEEE
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE ICC 2006 proceedings.

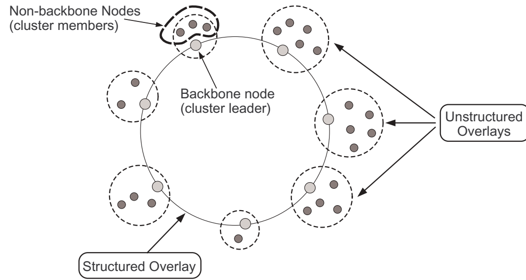


Fig. 1. The network structure of MKey

result. Observing that the search result of a single keyword may be of large size, [13] proposes to incrementally compute the intersection at different nodes and only return the final result to the user. Edge bandwidth to users is hence greatly saved. However, a problem in all the above approaches is that the storage cost and search cost are linear with the number of storage keywords and the number of query keywords, respectively. They are hence not efficient when a large number of keywords are used. On the other hand, to return a complete and accurate set of results to users, items should be stored with as many keywords as they can and a query also needs to contain lots of keywords. Clearly, this introduces high storage cost and search cost.

In this paper, we propose a hybrid structured network, *MKey*, to address this problem. MKey builds a hybrid overlay as Fig. 1 shows. Its backbone is a structured network. A backbone node further forms a cluster with some non-backbone nodes and serves as the cluster leader. Nodes within a cluster form an unstructured network and cooperate to share the storage loads and answer search queries assigned to their leader. Generally, the earlier joining nodes form the backbone; After the backbone network is completely built, a new node selects one cluster.

When storing a data item, its index is sent to the current cluster leader, who routes it to one or several other backbone nodes. A backbone node receiving the storage request either stores the index itself or continues routing the storage request to one of its cluster members. When looking up an item, the search request is also routed to some backbone nodes through the cluster leader. Each of these backbone nodes initiates a flooded search within its cluster and returns search results to the query initiator through its cluster leader. The union of all the search results serves as the final search result. In more details, we use a Bloom filter to summarize an item's storage keywords, which is a bit vector of certain length [14]. The Bloom filter is split into several bit vectors of the same length, each corresponding to a backbone node ID. Each of the nodes needs to store one copy of the item index within its clusters. In a lookup process, the multiple keywords in a query are similarly hashed to a bit vector of the same length. From this vector, a search range is determined and all the nodes within the range are searched. We carefully design the splitting/search mechanism to guarantee that the qualified items can be found.

As compared to previous approaches, MKey also stores multiple copies for an item index and searches multiple nodes to answer a query. However, it can provide upper bounds for both the storage cost and search cost, regardless of the number of keywords associated with items or in a query. Our simulation results show that MKey can efficiently store data items and answer user queries. The storage loads are well balanced among nodes.

The rest of the paper is organized as follows. In Section II we describe the design of MKey and its key components. In Section III we present analytic and simulation results for MKey. At last we conclude in Section IV.

II. System Design

A. System Architecture

As shown above, MKey consists of a structured network as its backbone. Any existing DHT systems such as Chord, Pastry and Tapestry can be modified to underlay MKey [3]–[6]. Each backbone node is also the leader of a cluster. Cluster members form an unstructured overlay and cooperate to share the loads (storage and search) assigned to their leader. There are two reasons to adopt this hybrid structure in MKey. First, different from traditional DHT systems where node IDs are computed by a hash function based on some node-specific information such as IP address, MKey has rigid restrictions for backbone node IDs (explained later). This has limited the size of useable ID space and hence the size of the backbone. With the hybrid overlay, the system can accommodate more nodes. Second, the sizes of clusters are adjustable according to the load distribution. A backbone node with large storage loads tends to form a large cluster. The whole system can hence achieve load balancing.

In MKey, a data item's keywords are represented by a *Bloom filter*, which is a m-bit vector [14]–[16]. This vector directs the storage locations of the item index. A query is also mapped to a m-bit vector according to its keywords, from which a search range is determined. We carefully control the mapping of these vectors and guarantee that the qualified items can be returned to users. To achieve this, each backbone node is assigned a m-bit ID. As mentioned, not all $O(2^m)$ IDs are useable. We hence design an ID management mechanism to scalably distribute and maintain legal MKey IDs.

In summary, MKey is most applicable to items/queries with a large number of keywords. If most items and queries only contain a few keywords, simple data replication methods such as [10]–[13] are enough. With the increase of keywords, however, MKey shows a significant improvement as compared to these approaches, in both storage cost and search cost. MKey can be extended to support full-text search as BitTorrent indices websites. If each data item has a description file, all the words in the description file can be hashed to a Bloom filter. Clearly, the length of Bloom filters should be set large enough.

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE ICC 2006 proceedings.

B. Bloom Filters

Bloom filters are compact data structures for probabilistic representation of a set in order to test whether or not an element is a member of a set. This compact representation is the payoff for allowing a small rate of false positives (that is, queries might incorrectly recognize an element as member of the set). However, false negative rate is always zero.

More formally, consider a set $A = \{a_1, a_2, ..., a_n\}$ of n elements. To describe it, bloom filters use a bit vector V of length m and k independent hash functions, $h_1, ..., h_k$, with range $\{1, ..., m\}$.

The following procedure builds a Bloom filter for set A:

Procedure $BloomFilter(set\ A,\ hash_functions,\ integer\ m)$
 $filter\ =\ allocate\ m\ bits\ initialized\ to\ 0$
 foreach $a_i\ in\ A$
  foreach $hash\ function\ h_j$
   $filter[h_j(a_i)]\ ==\ 1$
  end foreach
 end foreach
 return$filter$

Clearly, if an element a_x is member of set A, in the resulting Bloom filter V all the bits corresponding to the hashed values of a_x must be 1. Testing for membership of an element elm is then equivalent to testing whether all its corresponding bits of V are set 1. It works as follows:

Procedure $MembershipTest(elm,\ filter,\ hash_functions)$
 foreach $hash\ function\ h_j$
  if $filter[h_j(elm)]\ !=\ 1$ **return** NO
 end foreach
 return YES

Figure 2 shows an example of Bloom filter with $m = 8$ and $k = 3$. The filter begins as a vector with all bits zero. The target set A contains two keywords $Star$ and $Wars$. Each of them is hashed 3 times and the results correspond to 3 bit locations in V. These bits are set to 1. To check if a word is in set A, use the same method to hash it 3 times and check the corresponding bits. If any of the bits is 0, the word cannot be in the set; otherwise, either the word is in the set or the filter has yielded a false positive result.

C. Storage and Query in the Structured Backbone

In this subsection we focus on the design in the structured backbone network. In Section II-D we will discuss how to extend it to a hybrid overlay.

Storage: For a shared data item, we hash its storage keywords into a m-bit Bloom filter. We call this Bloom filter an *item descriptor*. In almost all existing DHT systems, node IDs are computed by a hash function based on node-specific information such as IP address and private key [3]–[6]. However, in our system, node IDs have strict formation rules. First of all, all the node IDs are m-bit. We define two classes of IDs: the first class consists of all the IDs that have only one bit of 1, and the second class consists of all the IDs that have two bits of 1. Clearly, the sizes of these two classes are m and $m(m-1)/2$, respectively.

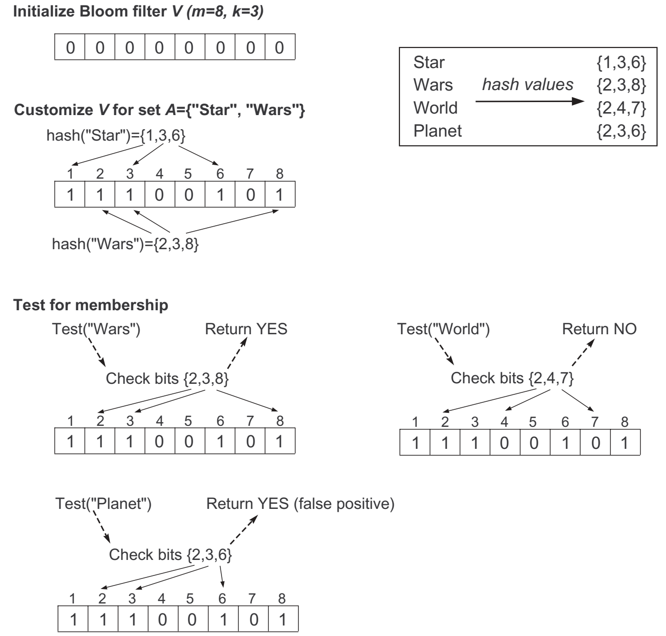


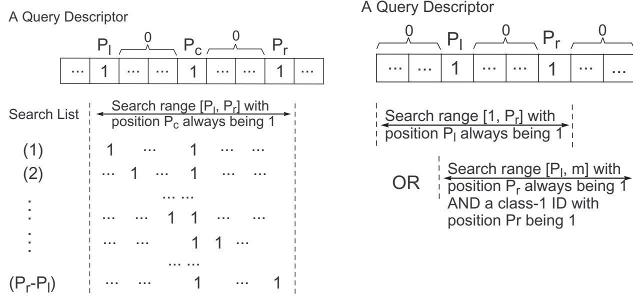
Fig. 2. An example of Bloom filter

Given an item descriptor, we split it into several class-2 or class-1 IDs (only possible for the last one) in a sequential order from left to right so that the bit-wise union of these IDs is equal to the item descriptor. For example, a 8-bit item descriptor 01010111 can be split into 01010000, 00000110 and 00000001. Each of these nodes needs to store one copy of the item index, which includes the item descriptor, all the keywords, a hash digest of the data content and a link to the item owner.

Search: In a lookup process, a query may contain one or more keywords. These keywords are hashed to a m-bit vector by the same method of computing item descriptors. We call this vector a *query descriptor*. Clearly, if all the query keywords appear in some item's storage keywords, the query descriptor must be a bit-wise subset of the item descriptor. In other words, the result of the bit-wise OR of the item descriptor and the query descriptor is equal to the item descriptor itself.

Given a query descriptor, the search method depends on the number of bits of 1 in it. We first consider the case that a query descriptor has at least three bits of 1. We select three consecutive bits of 1 (neglecting bits of 0 interleaving them) and denote their positions as P_l, P_c and P_r from left to right. We then search the following class-2 nodes: the bit at position P_c is 1 and a bit at some position between P_l and P_r (including P_l and P_r but excluding P_c) is 1, and all other bits are 0. For example, given a query descriptor 01010011, its second, fourth and seventh bits are three consecutive bits of 1. The nodes that we need to search then include 01010000, 00110000, 00011000, 00010100 and 00010010. Figure 3(a) shows the search process and summarizes it as a search within range $[P_l, P_r]$ with position P_c always being 1. Note that within

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE ICC 2006 proceedings.



a) A query descriptor with at least three bits of 1.

b) A query descriptor with only two bits of 1.

Fig. 3. Search mechanism in MKey.

the range only class-2 nodes are searched. The rationality is explained as follows. In a qualified item's descriptor, the bits at positions P_l, P_c and P_r must be 1. P_c then must belong to some class-2 ID when splitting the item descriptor for storage. The remaining bit of 1 in that ID must be between position P_l and P_r (including them) since we sequentially split item descriptors from left to right.

If a query descriptor has only two bits of 1, suppose the two bits are at position P_l and P_r. There are two ways to define the search range as shown in Fig. 3(b). Clearly, the one with smaller range is intelligently preferred. It is trivial to further extend this to the case that a query descriptor only contains one bit of 1.

The number of nodes that need to be searched N_s is computed as follows: (i) if a query descriptor has no less than tree bits of 1, $N_s = P_r$-P_l; (ii) if a query descriptor has only two bits of 1, if $P_l = 1$ or $P_r = m$, $N_s = P_r - P_l$; otherwise, $N_s = min\{P_r - 1, m - P_l + 1\} \leq (P_r - P_l + m)/2$; (iii) if a query descriptor only contains one bit of 1, $N_s = m$. Clearly, with more query keywords, less nodes need to be searched. Furthermore, in the usual case that a query descriptor contains no less than three bits of 1, we can select the three consecutive bits of 1 with the smallest left-to-right distance (i.e. $P_r - P_l$) so as to reduce the search cost.

On each of the nodes searched, the query keywords are straightforwardly compared with storage keywords in the indices. Qualified indices are returned to the user as a subset of the final results. With these returned indices, a user can browse their descriptions and select to download the desired items. In fact, the union of the search results from all the searched nodes is the final complete result. However, a user can also select to terminate a search if the partial results are enough (details neglected here).

Underlay Routing Method: With the splitting of item descriptors and query descriptors, the storage and search in MKey return to that in traditional DHT systems. Any existing DHT system such as Chord, Pastry and Tapestry may underlay MKey to provide these basic functions. We take Chord in our current implementations [3]. Chord has no restrictions on node IDs since it only orderly organizes nodes in a ring, while prefix routing in Pastry or postfix routing in Tapestry is more applicable to a large number of diverse node IDs that evenly distribute in ID space.

We neglect the details of Chord here. Interested readers can refer to [3]. With Chord properties, each backbone node in MKey needs to maintain information about $O(\log m)$ other nodes, and a lookup also requires $O(\log m)$ messages.

ID Assignment: An important feature of MKey is that backbone node IDs need to conform to strict regulations. To guarantee every node ID is legal and unique, a node cannot compute an ID only based on its local information. A straightforward approach to this problem is to adopt an authorized server to distribute IDs. However, this centralized approach is not scalable and may lead to a single point of failure. We hence propose to build a hierarchical tree for ID management.

We use a public rendezvous point (RP) to bootstrap ID assignment. Each new node is required to contact RP when joining. However, RP does not directly assign IDs for all joining nodes. Instead, it is only responsible for the assignment of class-1 IDs. And each class-1 node is responsible for a subset of class-2 IDs that have the same number of 0 at the head. Figure 4 shows an example of 8-bit ID management tree. In this two-level tree, each parent node distributes and maintains IDs in its child nodes. We call the parent nodes *ID managers* of their children.

Each class-2 node periodically sends $Alive$ message to its ID manager, a class-1 node. Correspondingly, a class-1 node needs to maintain information about the number of available IDs under its charge, including those that have not been allocated and those that have been assigned but the corresponding nodes have left the system. Moreover, class-1 nodes periodically report these information to RP.

Receiving a node's joining request, RP first checks whether all the class-1 IDs have been assigned. If not, this node will be given a class-1 ID. Otherwise, RP routes the request to the class-1 node with maximum available class-2 IDs. This new node will be given a class-2 ID by this class-1 ID. In Section II-D we will discuss how to handle new nodes when all the the class-1 and class-2 IDs have been assigned.

Clearly, RP and some class-1 nodes need to maintain up to $O(m)$ IDs. If m is very large, these nodes may be overloaded by ID management issues. In that case, the management tree can be further extended to multiple levels so that a node needn't manage too many IDs. The extension method can be diverse, and we do not discuss the details here.

D. Extending to a Hybrid Structure

Given the length of Bloom filters m (usually in the order of hundred in MKey), the number of useable node IDs in MKey is $O(m^2)$. This may not be enough for a large number of users. Furthermore, backbone nodes are likely to have different loads, and some of them may be quickly overloaded. To address these two problems, we build a hybrid overlay based on the structured backbone.

In general, each node in the backbone is extended to a cluster. Initially, a cluster only contains one backbone node,

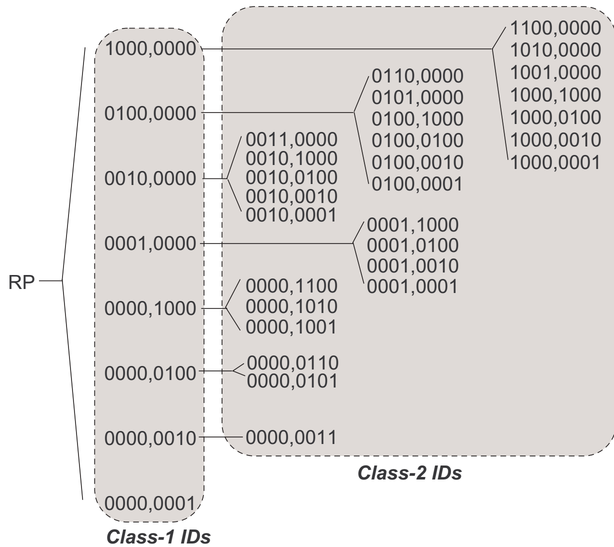


Fig. 4. An example of ID management tree

which serves as the cluster leader. After a complete backbone has been formed, new nodes are routed to join clusters. Nodes in a cluster form an unstructured overlay and cooperate to help their leader, by sharing the storage loads and answering search requests assigned to the leader. The storage and search requests from a cluster member are all sent to its cluster leader and relayed by the leader.

Joining a Cluster: Clusters are kept at small scales, say, consisting of at most several hundred members. The selection of a cluster for a new node to join depends on the storage loads (i.e. the total number of indices that have been stored in the cluster) and capacities (i.e. the number of current members) of clusters, as well as the distances from the new node to cluster leaders.

To monitor cluster loads, we extend the functionalities of the ID management tree. Each class-2 node needs to periodically report the load burden and size of its cluster to the ID manager. A class-1 node then reports a few clusters with the heaviest per-node load as well as its own cluster information to RP. RP can then roughly discover clusters with the heaviest per-node load in the system. RP then routes new nodes to these clusters, since storage loads are expected to be shared be all cluster members.

More specifically, a joining node first contacts RP as described above. If RP finds that no appropriate IDs can be assigned to it, RP selects a few clusters with the heaviest per-node loads. The new node can ping the leaders and select one with small distance to join.

Intra-cluster Management: Within a cluster, all members form an unstructured overlay. When joining a cluster, a new node gets a member list from the leader and sets up connections with them as neighbors. A node periodically exchanges $Alive$ messages with its neighbors to prevent unexpected node failure. More important, the leader's $Alive$ messages are periodically flooded within the cluster to prevent overlay partition.

Indices assigned to the leader for storage from the backbone network are distributedly stored at some cluster members. If the leader receives a search query from other backbone nodes, flooded search based on query keywords instead of a query descriptor is performed within the cluster. Note that multiple-keyword search are inherently supported in unstructured networks.

To achieve load balancing among cluster members, a member keeps monitoring loads on each of its neighbors. Receiving a storage request, the member compares its own load burden with that of its neighbors. If its own load burden is the minimal of the nodes, it stores the index locally. Otherwise, it forwards the storage request to the neighbor with the minimal load, which continues this process. In this way, storage loads can evenly distribute among all members.

III. Performance Evaluation

A. Performance Analysis

In the following analysis related to Bloom filters, we assume that hash functions are perfectly random. That is, the hash values of the elements in a set evenly distribute throughout the range of the function. The question of what hash function to use in practice remains an open problem; currently MD5 is a popular choice [15].

Suppose the Bloom filters are of m-bit length and use k hash functions. The target set A contains n elements. After all the elements of A are hashed into the Bloom filter, the probability that a specific bit is still 0 can be computed as [14]:

$$P_0(n) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

Similar to [16], we assume the bits in the Bloom filter are independently set to 0 with probability $P_0(n)$ and set to 1 with probability $(1 - P_0(n))$. Therefore, the average number of bit-1 in the Bloom filter is $m(1 - P_0(n))$. The number of storage copies for a data item with n keywords is hence $\left\lceil \frac{m(1-P_0(n))}{2} \right\rceil$.

Similarly, given a search query with t keywords, the number of bit-1 in the query descriptor is $m(1 - P_0(t))$. Consider the general case that $m(1 - P_0(t)) \geq 3$. We will select three consecutive bit-1 with the smallest left-to-right distance. In the worse case, all the bits of 1 evenly distribute throughout position 1 to position m. The backbone nodes searched is hence no more than $\frac{2(m-1)}{m(1-P_0(t))-1}$.

B. Simulation Results

We have conducted simulations to evaluate the performance of MKey. We build MKey on top of a public Chord implementation [17]. We set $m = 128$ for Bloom filters and take disjoint groups of bits from MD5 signature as independent hash functions [15]. The total number of nodes is $100,000$. We compare MKey with the simple data replication method used in [10]–[13].

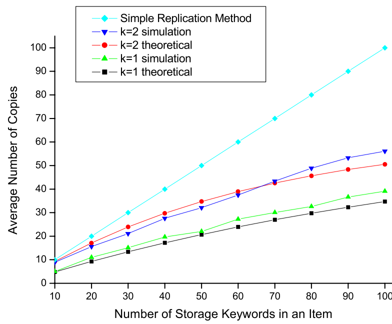


Fig. 5. Average number of copies in structured networks.

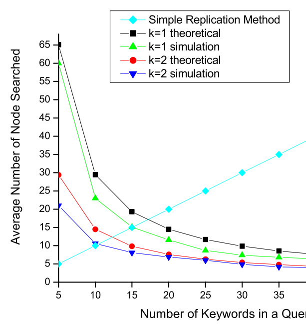


Fig. 6. Average number of nodes in structured networks that need to be searched.

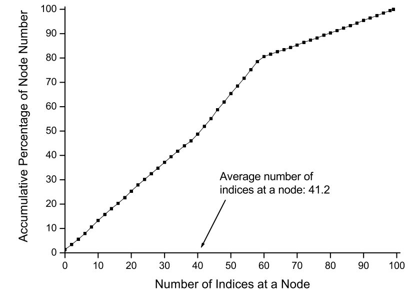


Fig. 7. Accumulative distribution of per-node storage load.

Figure 5 shows the average number of copies that needs to be stored at different nodes. It doesn't consider redundant replications for node leaving. As it shows, with the increase of storage keywords, MKey shows a much lower increase speed in the number of copies than simple replication method. In fact, for an item descriptor of length 128, the maximum number of copies is 64 in MKey. However, we do not suggest to fulfill an item descriptor, because many items would have the same descriptor in this case. Therefore, the width of Bloom filters should be properly enlarged if the average number of storage keywords is large.

Figure 6 shows the average number of nodes that need to be searched for a query in structured networks. It doesn't consider the forwarding nodes. With more keywords in a query, MKey achieves lower search cost. This is because the more bits of 1 a query descriptor contains, the smaller search range we can find. In Fig. 5 and Fig. 6, the simulation values and the analytic results match very well.

Figure 7 shows the accumulative distribution of storage load at a node. The average number of indices at a node is 41.2. As the figure shows, the maximum load at a node is less than three times of the average value. And more than 90% nodes have loads lower than two times of the average value. It shows that MKey achieves good load balancing among nodes through its cluster formation mechanism.

IV. Conclusion

Current structured P2P networks only support single-keyword exact-match lookups. In this paper, we propose MKey system to support multiple-keyword search. MKey forms a hybrid network with a structured backbone. Data items are stored multiple times and a query is answered by searching multiple nodes. Different from traditional approaches, MKey has upper bounds for storage cost and search cost, regardless of the number of keywords. Simulation results show that MKey can significantly reduce storage cost and search cost as compared to traditional data replication method. It can also achieve good load balancing through its cluster formation mechanism.

Acknowledgement

The authors would like to thank Jing Zhao, Weifeng Su and Gang Wang from the Database Group at the Hong Kong University of Science and Technology for their helpful discussions.

References

[1] "Gnutella." http://gnutella.wego.com.
[2] "Kazaa." http://www.kazaa.com.
[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proc. ACM SIGCOMM'01*, Sept. 2001.
[4] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *LNCS*, pp. 329–350, Nov. 2001.
[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM'01*, pp. 161–172, Aug. 2001.
[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE JSAC*, vol. 22, pp. 41–53, Jan. 2004.
[7] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in *Proc. ACM SIGCOMM'03*, Aug. 2003.
[8] E. Cohen, A. Fiat, and H. Kaplan, "Associative search in peer to peer networks: Harnessing latent semantics," in *Proc. IEEE INFOCOM'03*, April 2003.
[9] R. Zhang and Y. C. Hu, "Assisted peer-to-peer search with partial indexing," in *Proc. IEEE INFOCOM'05*, Mar. 2005.
[10] M. Balazinska, H. Balakrishnan, and D. Karger, "INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery," in *Proc. Pervasive'02*, Aug. 2002.
[11] M. Harren, J. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica, "Complex queries in DHT-based peer-to-peer networks," in *Proc. IPTPS'02*, March 2002.
[12] L. Garces-Erice, P. A. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross, "Data indexing in peer-to-peer DHT networks," in *Proc. ICDCS'04*, March 2004.
[13] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Proc. ACM/FIP/USENIX Middleware'03*, June 2003.
[14] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.
[15] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM TON*, vol. 8, pp. 281–293, June 2000.
[16] M. Mitzenmacher, "Compressed Bloom filters," *IEEE/ACM TON*, vol. 10, pp. 604–612, Oct. 2002.
[17] "p2psim: a simulator for peer-to-peer protocols." http://pdos.csail.mit.edu/p2psim/.