

# Distributed Suffix Tree for Peer-to-Peer Search

**Hai Zhuge and Liang Feng**

*China Knowledge Grid Research Group, Key Lab of Intelligent Information Processing  
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*

**Abstract:** Establishing an appropriate semantic overlay on Peer-to-Peer networks to obtain both semantic ability and scalability is a challenge. Current DHT-based P2P networks are limited in their ability to support semantic search. This paper proposes the DST (Distributed Suffix Tree) overlay as the intermediate layer between the DHT overlay and the semantic overlay. The DST overlay supports search of keyword sequences. Its time cost is sub-linear with the length of the keyword sequences. Using a common interface, the DST overlay is independent of the variation of the underlying DHT overlays. Analysis and experiments show that DST-based search is fast, load-balanced, and useful in realizing accurate content search on large networks.

**Key words:** DHT, Peer-to-Peer, Search, Semantics, Suffix Tree, Load Balance.

---

---

This work is supported by the National Basic Research Program of China (No. 2003CB317000).

# 1. Introduction

## 1.1 Motivation

With the development of the Internet and various communication networks, decentralized information sharing is showing its potential and advantages. To incorporate scalability and semantics into decentralized information sharing systems is a challenge.

Peer-to-Peer (P2P) networks support scalable decentralized information sharing. There are basically two types of P2P network: structured and unstructured. In unstructured P2P systems like Freenet [4], Gnutella [13] and Napster [23], there is no assumption about the assignment of data onto peers. Each peer manages its own data, and no global view of data exists in such networks. Many approaches to improve the efficiency of data lookup in these systems have been proposed [5, 33]. In structured P2P systems [17], data items (or indexes of data items) are assigned onto peers according to some rules. One type of structured P2P systems distributes data items onto peers via a Distributed Hash Table (DHT) [15, 16] like Can [24], Pastry [26], Chord [28] and Tapestry [34]. DHT-based systems largely solve the problem of scalability since each lookup of a data item can be resolved within  $O(\log n)$  (or  $O(n^{1/d})$ ) overlay routing hops for a network of  $n$  peers. But the major limitation of the early DHT-based systems is that they are based on exact identity matching. Most of such P2P systems are limited in their semantic ability to support intelligent applications, for example, the content search, which is based on the relationship between words rather than isolated words.

Establishing an appropriate semantic overlay on P2P networks is a way to acquire both scalability and semantic ability to support intelligent applications on large-scale networks. A first step towards such an overlay is to establish various distributed data structures.

Our approach is to establish a Distributed Suffix Tree overlay on the DHT-based systems to realize efficient keyword sequence search, which is needed in many applications, for example, search papers distributed on P2P e-science network according to the given title, abstract or keyword sequence representing the content.

## 1.2 Approach Overview, Features and Applications

In real P2P networks, each resource has a string descriptor. The proposed Distributed Suffix Tree (DST) approach uses the suffix tree algorithm [21] to reorganize string descriptors to form a global

suffix tree data structure, and distributes each edge of the global suffix tree onto the peers in P2P networks. When a lookup is initiated, the search process follows the path from the root of the global suffix tree to the next node satisfying this lookup, and then to the next node onwards until it reaches the nodes where the target resides.

The DST approach has the following features:

- (1) It is decentralized, scalable and load balanced.
- (2) It maintains the sequential relationship between indexed keywords, which offers more accurate meaning than isolated words. Its lookup time cost depends on the length of the input string and the type of the DHT-based lookup protocol. It is especially suitable for search with lengthy keyword sequences.

The DST approach can be used to implement the basic functions for many applications such as:

- (1) *P2P file sharing systems* [8]. Search is the basic function of P2P file storage systems. The DST approach enables users to find appropriate files by a series of words describing title or abstract.
- (2) *Peer Data Management Applications* [11, 17]. This type of applications mainly focuses on the XML-based complex queries and heterogeneous data integration. The DST approach can facilitate the processing of complex queries.
- (3) *Large-scale Text/String analysis and processing*. When the text/string to be analyzed is too large to be processed on a single computer, the DST approach can decompose the large text and distribute it onto peers for processing.
- (4) *Semantic Link Applications* [35, 36, 37]. By publishing the semantic objects as indexing nodes to form distributed indexing structures, queries can be forwarded along the chains of semantic object pointers to search for data objects indexed by their keys. The DST can be a substrate of the semantic overlay.

### **1.3 Related Work**

The system Brushwood [32] gives a general paradigm for distributing tree data structures onto P2P networks. It preserves the data locality by linearizing the tree node in pre-order and assigning each partitioned consecutive segment to a peer. This work is suitable for locality sensitive applications. SkipIndex [33] proposes a distributed high-dimensional index structure and routing scheme based on P2P overlay routing. It supports efficient similarity search and range queries for high-dimensional data by guaranteeing logarithmic lookup and maintenance cost even facing skewed datasets.

There are typically three index solutions to distributed keyword search: partitioning by document, partitioning by keyword, and hybrid indexing. In the first solution, each node is responsible for a number of documents. During search, the query is broadcast to all nodes which will complete the query locally. The search results will be aggregated and returned to the user. The second solution requires that each node maintains an inverted list of documents for each keyword it is responsible for [6, 12, 20, 25]. To answer a query with multiple terms, the query will be sent to the nodes responsible for those terms. The inverted list returned by the nodes will be joined and returned to the user. The third solution combines the former two and gives a hybrid indexing structure [29]. It gains the search efficiency and reduced bandwidth usage by sacrificing storage space. Unfortunately, although can take further operations, these works consider the keywords separately and do not take the relationship between keywords into account. And the relationship is needed in some contexts. For example, applications may want to search with a phrase. The keyword sequence search of the DST approach gives the user an ability to order which keywords should appear sequentially in the document. This ability is needed in many applications, for example, search papers distributed on P2P e-science network according to the given title, abstract or keyword sequence representing the content.

The exact-match mechanism of hash indexes is used as a substrate for textual search in Harren's work [14]. The approach splits each string into " $n$ -grams" (i.e., distinct length- $n$  substrings), and then indexes these  $n$ -grams on the DHT overlay. For each  $n$ -gram  $g_i$ , the pair  $(g_i, DHT\_key)$  is inserted into the hash index by the key  $g_i$ . The  $n$ -grams strategy has a good performance in substring search as it can decrease the number of irrelevant results. However, the " $n$ -grams" prevents search of substrings whose length is less than  $n$ .

The Prefix Hash Tree (PHT) [3] is designed to support range queries on an underlying DHT overlay. This approach hashes the prefix (i.e., the label of one node in the PHT) of a key in the DHT identifier space. Some corresponding algorithms are proposed to implement the lookup and range query operations. Also, multi-dimensional range query is supported by mapping multi-dimensional data onto a single dimension via linearization. However, while it gets the range query ability by assigning prefix onto DHT, it loses the properties to be extended to support keyword searches.

Some information retrieval techniques have been applied to the indexing method in structured P2P networks. In [27, 31], the content of documents is processed to compute the keys used in structured P2P systems. These techniques extract a vector of keywords from each document, and

typically, the keywords are selected according to the frequencies of their appearance. Then, the vector describing each document is used as the key to map the documents into the virtual multi-dimensional space of the network. However, the extracted vector has much higher dimension than the virtual space of the network. Thus, techniques for dimensionality reduction are required. Such a reduction should keep a minimal distortion, that is, the distance of the new vectors with the reduced dimensions should approximate the distance between the initial vectors.

The above approaches use the flat text description as the underlying data format, while others [2, 10] use XML to represent data. When using XML, both value and path indexes are needed for addressing the content as well as the structure of documents. And, these approaches focus on solving the problem of schema-based search. The keyword-based search techniques can work in cooperation with these approaches to realize more search types.

## 2. The System Model

The architecture of the DST-based system is shown in Figure 1. The Distributed Suffix Tree Overlay is the intermediate layer between the DHT overlay and the semantic overlay that supports high-level distributed intelligent applications. The bi-direction arrows represent information exchange between layers. A lower layer provides services for a higher layer.

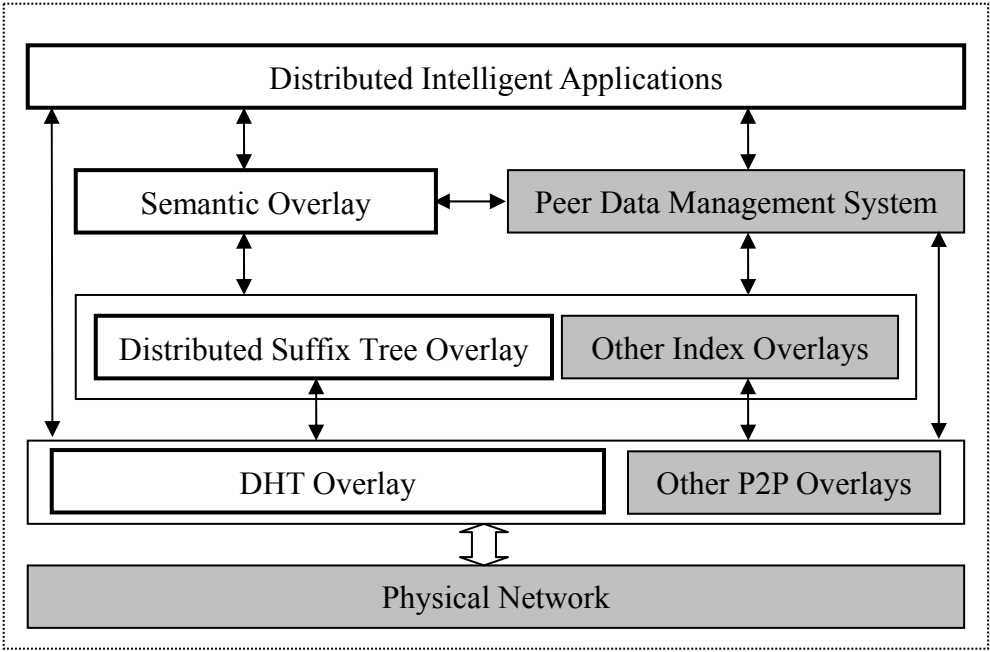


Figure 1. Architecture of the DST-based system.

The Distributed Suffix Tree Overlay supports the following keyword search services:

- *Single keyword search* — Given one keyword  $A$ , the DST approach returns a set of DHT keys associated with texts that contain  $A$ .
- *keyword sequences search* — Given keyword sequences like  $ABC$ , the DST approach returns a set of DHT keys associated with texts that contain  $ABC$ .
- *AND, OR, and NOT operations on the above two searches*, for example, a search can take the following forms: “ $A$  AND  $B$ ”, “ $A$  OR  $BC$ ”, and “NOT  $A$ ”.

The application interacts with the DST mainly in two ways. First, the application delivers the tuple  $(text, DHT\_key)$  to the DST for processing and indexing. The application is responsible for providing meaningful, summarized, and well-described *text* of the resource associated with the *DHT\_key*. Second, the DST provides *Search (keyword expression)* as the interface that yields the *DHT\_keys* of the *texts* satisfying the *keyword expression*.

The DST approach constructs a global virtual suffix tree overlay using the *texts* delivered by applications. It simplifies the design of applications by supporting a fast and reliable keyword search service with the following features:

- *Decentralization* — The DST approach does not need any superpeers. No peer is more important than any other. So it is able to deal with the situation that peers join and leave frequently.
- *Load Balance* — The DST approach never makes any superpeers. It scatters and distributes the virtual suffix tree evenly on each peer by using the DHT which is load balanced in nature.
- *Scalability* — The DST approach takes the DHT overlay as its substrate, which in turn makes it scalable even in large P2P networks. Additionally, its search cost only depends on the keyword expression and the DHT-based lookup protocol, not on the number of the involved *texts*. So it is suitable for even very large systems.
- *Speed* — The DST approach supports fast search. After optimization, its time cost is less than  $O(T + m)$  and has an upper bound of  $O(T + \lg n)$ , where  $T$  is the lookup cost of the underlying DHT overlay,  $m$  is the length of the keyword sequence and  $n$  is the maximum length of the indexed *texts*.

### 3. The Basic DST Approach

The DST approach consists of three basic parts: (1) the method for constructing the virtual

distributed suffix tree; (2) the search mechanism for results satisfying a given keyword expression; and, (3) the maintenance operations like inserting a tuple  $(text, DHT\_key)$  into the DST overlay and deleting a tuple from the overlay.

### 3.1 Suffix Tree

Suffix tree is an efficient string matching solution [21]. To search a string of length  $M$  in a text of length  $N$ , a suffix tree only needs  $M$  comparisons—this is clearly the minimum comparison to finish a search. The word ‘suffix’ here refers to the fact that suffix tree  $T$  contains all the suffixes of a given text  $S$ . Suffix tree  $T$  has the following attributes:

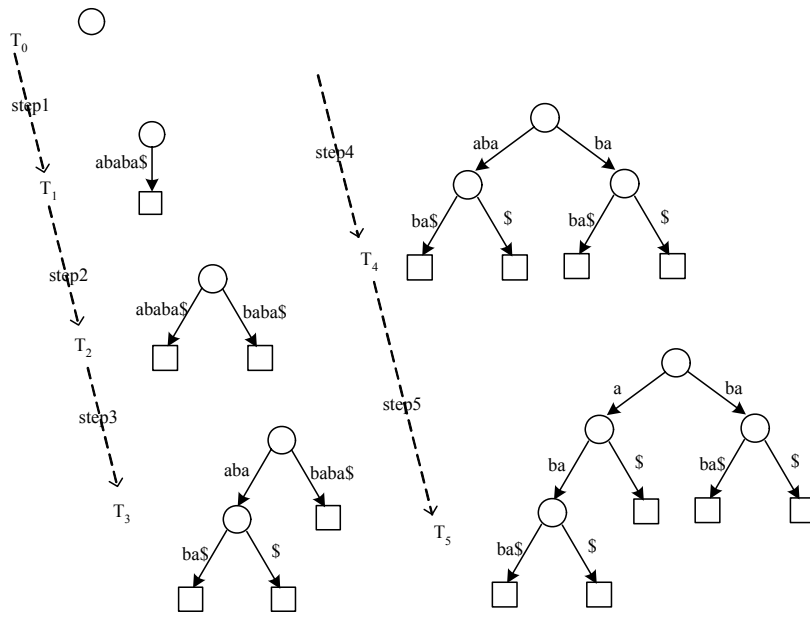
- (1) An arc may represent any nonempty substring of  $S$ .
- (2) Each nonterminal node, except the root, must have at least two offspring arcs.
- (3) The strings represented by sibling arcs must begin with different characters.

To construct a suffix tree, the given text  $S$  needs to satisfy that its final character should not appear elsewhere in  $S$ . This can be realized by simply appending a unique character to  $S$ .

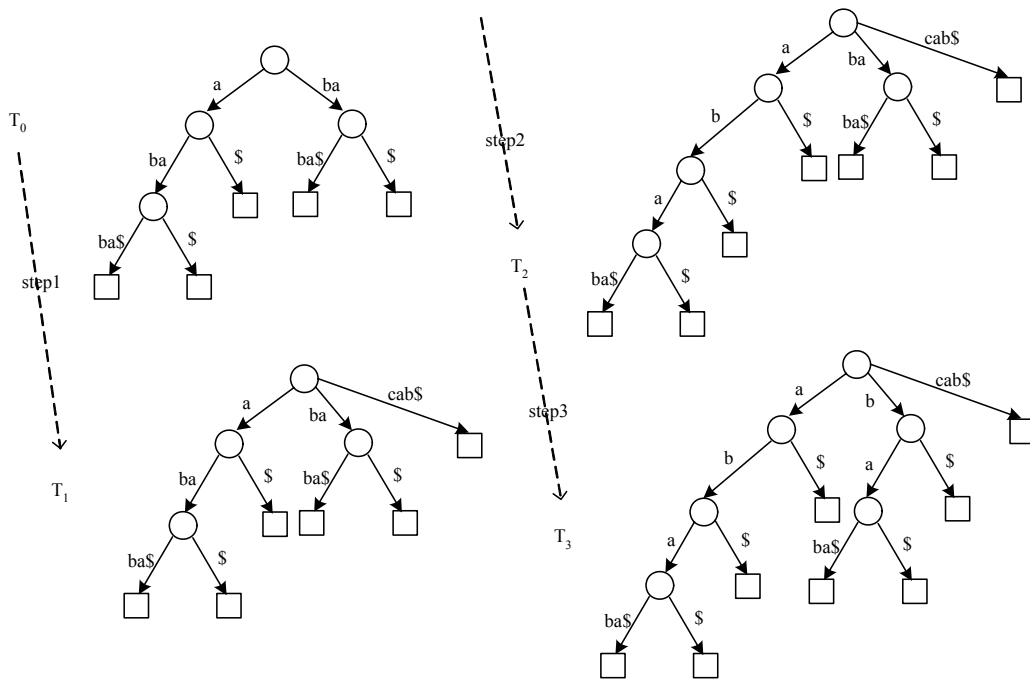
The construction of a suffix tree begins with an empty tree  $T_0$  and then enters paths corresponding to the suffixes of  $S$  one at a time, from the longest to the shortest. Take a string  $S$  (“*ababa*”) as an example. Because  $S$  does not fulfill the requirement of suffix trees, a unique character  $\$$  is appended to the end of  $S$ . The tree  $T$  corresponding to our example string  $S$  would be constructed by the algorithm in the steps shown in Figure 2, one step per suffix of  $S$ . Here, we omit the suffix “ $\$$ ” in the suffix tree, as it is just a placeholder without any meaning.

Let  $suffix_i$  be the suffix of  $S$  beginning at character position  $i$ . During step  $i$ , the algorithm inserts the path corresponding to the string  $suffix_i$  into the tree  $T_{i-1}$  to produce tree  $T_i$ . To insert  $suffix_i$ , a search for  $suffix_i$  is performed on tree  $T_{i-1}$  and eventually fails at some character position of some arc. We define  $head_i$  as the characters in  $suffix_i$  that have been matched during search, and define  $tail_i$  as  $suffix_i - head_i$ . In our example,  $suffix_3=aba\$$ ,  $head_3=aba$ , and  $tail_3=\$$ . A new nonterminal node is constructed to split the failing arc if necessary, and finally a new arc labeled  $tail_i$  is constructed from that nonterminal node to a new terminal node.

The construction of the generalized suffix tree for more than one string carries out in the same process, except that  $T_0$ s for the second and later strings are not empty tree and each arc may be marked to indicate to which strings it belongs. Figure 3 shows an example of the insertion process of string “*cab\\$*”, where  $T_0$  is not an empty tree but a suffix tree for “*ababa*”.



**Figure 2.** The suffix tree construction for “ababa”.



**Figure 3.** The suffix tree insertion for “cab\$”.

The deletion of a string  $S'$  from a generalized suffix tree is the reverse process of the insertion, so we can take the reverse order of figure 3 as the example of deletion. It processes paths corresponding to the suffixes of  $S'$  one at a time, from the shortest to the longest. The process involves two operations: (1) deleting arcs that are marked belonging solely to  $S'$ , and (2) linking in-arcs and out-arcs of nodes that have only one child.

### 3.2 Indexing

Before deploying a suffix tree onto the P2P network, we firstly extend the suffix tree to support keyword search by replacing the alphabet with a vocabulary. In other words, instead of sequences of letters, the DST approach uses suffix trees over sequences of words. And as a upper level overlay upon DHT, the DST approach need the  $key\_to\_peer(key)$  interface provided by the DHT overlay.

Every search process starts at the root edges of the DST. So, when a search request comes, the DST approach must know where to find the root edges. One way is to store all root edges at one peer. Then, the DST approach retrieves the root edges from that peer every time. This is simple, but sacrifices the load balance, and the peer storing all root edges becomes the bottleneck of the whole system.

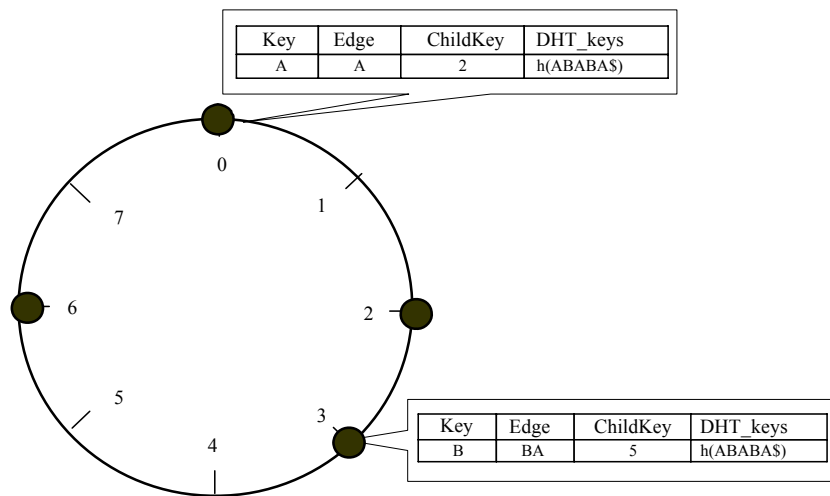
The DST approach takes the following way to store the root edges to solve the above problems: For each root edge,

- (1) draw out the initial word  $w$  of the root edge  $e$ , and take  $w$  as the  $key$  of  $e$ ; and,
- (2) use the DHT overlay's  $key\_to\_peer(key)$  interface to get the peer  $p$  responsible for  $w$ , and store the root edge  $e$  on  $p$ .

From the way of storing root edges, we can see that the DST approach works similarly to the inverted list approach [25]. The bottleneck never exists as the root edges are distributed to various peers. An example of storing the root edges of  $T_5$  in Figure 2 is shown in Figure 4. The Chord [28] is used as the underlying DHT overlay. And, notations such as 'a' and 'b' in suffix tree are replaced with their upper cases in order to reflect the changes from suffix tree to DST. The root edges of  $T_5$  in Figure 2 are  $A$  and  $BA$ . For  $BA$ , the initial word  $B$  is taken as the  $key$ . Suppose that the calling of  $key\_to\_peer(B)$  returns Peer3 according to Chord,  $BA$  will be stored at Peer3 responsible for key  $B$ .

From figure 4, we can also see that some other information has been stored with the edges. The information will be used to route search requests. They compose a routing entry (see Table 1) together in the following form: (Key, Edge, ChildKey, DHT\_keys). Key is used to identify a group of edges being the children of the same parent edge, but for the root edges, Key is the initial word. ChildKey is a globally unique DHT key used as the Key of its children edges. It is computed by the peer responsible for the edge. For the leaf edges, ChildKey is  $-1$ . The uniqueness of ChildKey can be ensured by using some techniques such as UUID — Universal Unique Identifier for uniquely identifying object or entity on the Internet. The computing mechanism of UUID relies on the network address of the host, a timestamp, and a randomly generated component for ensuring the uniqueness. Edge is the substring represented by the edge. The last field is DHT\_keys, a key set that

contains all the *DHT\_keys* of *texts* having one suffix that can match from the root to the current Edge

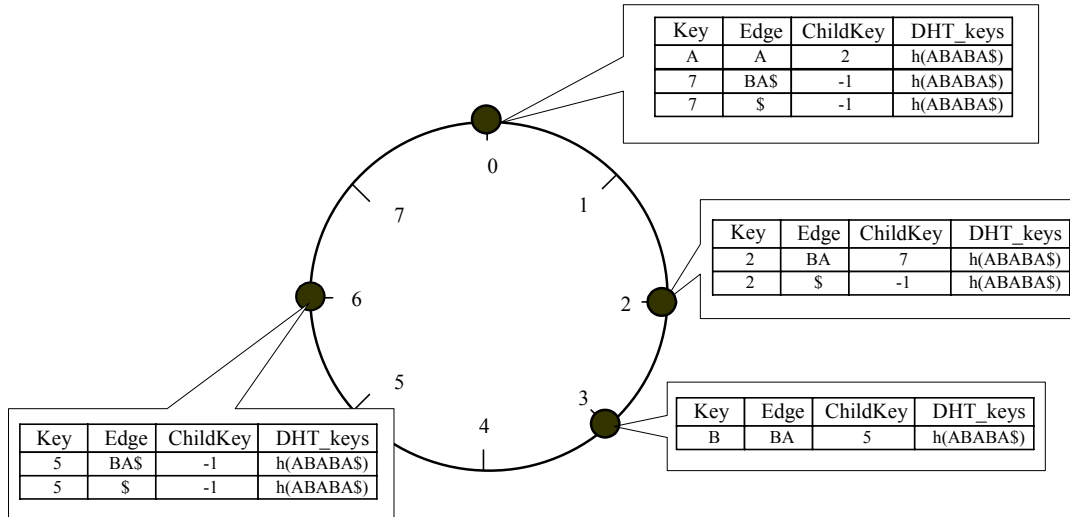


**Figure 4.** Root edges in DST using Chord as DHT overlay.

**Table 1.** Variables for routing entry.

Variable	Description
Key	The DHT key for the Edge
Edge	String representation of the edge
ChildKey	The DHT key for the Edge's Children Edges
DHT_keys	The DHT keys of <i>texts</i> having one suffix that can match from root to the Edge

After the root edges are distributed onto the DHT overlay, the Key fields of the edges in other levels can be obtained iteratively, as their Key fields are equal to the ChildKey fields of their parent edges. Each edge is stored at the peer responsible for its Key field. In this way, all edges in a suffix tree are distributed onto the DHT overlay. Figure 5 shows the DST of  $T_5$  in Figure 2 when all its edges are distributed onto the Chord overlay.



**Figure 5.** DST distribution of  $ABABA\$$  using Chord.

An indexing approach should guarantee that the search process can locate the targeting results correctly. The correctness is examined in the following.

**PROPERTY 1.** A Key field together with the initial word of an Edge field can exclusively determine one entry.

**PROOF.** This property can be proved by the following two aspects: (1) *Neither of them can exclusively determine one entry.* As described above, the edges having the same parent edge share the same Key, thus, Key cannot exclusively determine one entry. The edges having the same parent never have the same initial word of Edge according to the suffix tree algorithm, but those that do not have the same parent may have the same initial word of Edge, for example, there are four edges in  $T_5$  in Figure 2 having the same initial word  $B$ ; this means the initial word of Edge also cannot exclusively determine the entry. (2) *The Key and the initial word of Edge can determine the entry.* Each Key is a global unique DHT key, only the edges having the same parent have the same Key; this means Key can determine a group of edges that have the same parent. And, two arbitrary edges in this group must have different initial words of Edge according to the suffix tree algorithm; this means if in this group, the initial word of Edge can determine the edge. As they are combined together, the Key and the initial word of Edge can determine the edge, i.e., the entry. (End of Proof)

**PROPERTY 2.** The DST approach can compose a real suffix tree by iterating over the whole P2P network.

**PROOF.** Each entry in DST has a pointer called ChildKey pointing to its children entries, so given one entry, we can get all its children entries. All root entries satisfy that their Keys are equal to the initial words of their Edge fields. We can get all root entries by traversing the P2P network to find entries that conform to that characteristic. According to the above two inferences, the tree-structured entries of the DST can be retrieved by iterating from root entries to leaf entries and then compose a real suffix tree. (End of Proof)

Property 2 ensures that the DST has the same structure that a common suffix tree has, so it reserves some good properties from the suffix tree, such as the ability for searching and maintaining sequential relationships between words.

### 3.3 Search Process

The DST approach supports single keyword search, keyword sequences search, and the AND, OR, NOT operations of the above search types. Single keyword search is a special case of keyword sequences search, and the third type is based on the former two. Therefore, we first explain the keyword sequences search process, and then the others.

Consider a given keyword sequences:  $(A_1A_2A_3\dots A_n)$ . The search process begins with drawing out the initial word  $A_1$  and taking it as the Key  $K_1$ . Plus the initial word of root Edge,  $(K_1, A_1)$  can be obtained. According to Property 1, there is only one entry corresponding to  $(K_1, A_1)$ . The search process locates this entry by the following two steps: (1) the peer  $p$  storing this entry is located by calling the DHT overlay's interface  $key\text{-}to\text{-}peer(K_1)$ , and (2) on the peer  $p$  the entry is located by searching the entry whose Key field and the initial word are equal to  $K_1$  and  $A_1$  respectively. Suppose the Edge field of this entry is  $B_1B_2B_3\dots B_m$ , if  $A_1 = B_1$  and  $A_2 = B_2 \dots$  and  $A_m = B_m$ , then  $(ChildKey, A_{m+1})$  becomes the Key and the initial word of the Edge for the next entry. Continue these steps recursively until  $A_n$  is reached, then return the DHT\_keys fields of the ending entry. If any step or condition in the process fails, the whole search fails and ends with no matches found in the DST overlay.

The algorithm that implements the above search process is shown in Figure 6. The notation  $peer.foo()$  denotes that the function  $foo()$  is invoked and executed on  $peer$ . Remote calls and variable references are preceded by the remote peer identifier  $peer$ , while local variable references and procedure calls omit the local peer. Thus  $peer.foo()$  denotes a remote procedure call on  $peer$ , while  $peer.bar$  without parentheses is an RPC to lookup a variable  $bar$  on  $peer$ .

*seq\_multiple\_search* is the interface for keyword sequences search, it works by calling *process\_search*, which recursively calls itself to process the keywords.

*process\_search* is recursively called until the words in the *keywords* parameter are exhausted, and each calling will truncate at least one word from the *keywords*. If one remote procedure call is regarded as one step of the search process, the DST approach can finish one keyword sequences search within  $m$  steps, where  $m$  is the length of the keyword sequences. However, this does not take *key\_to\_peer* into account. *key\_to\_peer* is supported by the DHT overlay, thus the time requirement depends on which DHT overlay the application selects, for example, it is  $O(\log n)$  for Chord and  $O(n^{1/d})$  for CAN. Suppose  $T$  is the time *key\_to\_peer* uses, the time cost of the DST approach for a keyword sequences search is within  $m \times T$ . And, the time cost can be further reduced to  $T+m$ , which will be discussed in section 4.1.

The AND, OR and NOT operations of the keyword sequences search are other types of search supported by the DST approach. The basic idea for solving these problems is to get each result set of the keyword sequences search and then calculating the AND, OR and NOT operations on the result sets correspondingly. For example, suppose the keyword expression is  $A_1A_2\dots A_n$  AND  $B_1B_2\dots B_m$ , the *seq\_multiple\_search* should be called first to get the result set  $KeySet_1$  and  $KeySet_2$  for  $A_1A_2\dots A_n$  and  $B_1B_2\dots B_m$ . Then use the set intersection operation on  $KeySet_1$  and  $KeySet_2$  to get the final result set for the keyword expression.

Some techniques of query optimization can be used to reduce the bandwidth consuming and improve the search efficiency of the AND, OR, and NOT operations. When all the result sets are returned to the initial peer for further processing, the bandwidth consuming is the total size of all the result sets, i.e.,  $|KeySet_1| + |KeySet_2|$ . But if we transfer the smaller result set to the peer responsible for the larger result set for preprocessing, the bandwidth consumed can be reduced. For example, suppose  $|KeySet_1| < |KeySet_2|$  and the respective responsible peer is  $Peer_1$  and  $Peer_2$ ,  $Peer_1$  first transfers  $KeySet_1$  to  $Peer_2$ , then the intersection of  $KeySet_1$  and  $KeySet_2$  is finished on  $Peer_2$  and returned to the initial peer. In this way, the bandwidth consuming is reduced to  $|KeySet_1| + |KeySet_1 \text{ AND } KeySet_2|$ .

```

// ask peer to search results matching keyword expression
// key_to_peer is supported by DHT overlay
peer.seq_multiple_search( keywords )
    Key = keywords.initialword;
    peer = key_to_peer( Key );
    return peer.process_search( Key, keywords );

// process search in recursive way
peer.process_search( Key, keywords )
    targetEntry = NULL;
    foreach entry in peer
        if entry.Key == Key and entry.Edge.initialword == keywords.initialword
            targetEntry = entry;
            break;
    if targetEntry == NULL
        return NULL;
    length = min( targetEntry.Edge.length, keywords.length );
    for i= 1 to length
        if targetEntry.Edge[i] <> keywords[i]
            return NULL;
    if length == keywords.length
        return targetEntry.DHT_keys;
    else
        // truncate the front length words of keywords
        nextKeywords = keywords.truncate( length );
        nextKey = targetEntry.ChildKey;
        if nextKey == -1
            return NULL;
        else
            peer = key_to_peer( nextKey );
            return peer.process_search( nextKey, nextKeywords );

```

**Figure 6.** Search algorithm for matching a given keyword expression. Remote procedure calls and variable lookups are preceded by remote peer.

### 3.4 Tuple Insertion and Deletion

When a resource is published, the upper-level application is responsible for inserting the tuple (*text*, *DHT\_key*) into the DST overlay for indexing. Then the tuple insertion process is initiated. For each suffix of the *text*, the DST initiates a search request taking this suffix as the keyword sequences in order to add *DHT\_key* to *DHT\_keys* fields of the entries where the search passes, and to locate the

entry  $e$  where the search fails. Suppose  $e.Edge$  differs from the  $keywords$  parameter at position  $pos$ . If  $pos=1$ , then the algorithm composes a new entry  $(e.Key, keywords, -1, DHT\_key)$  and inserts it into the local repository of the peer that is responsible for  $e.Key$ ; otherwise, it splits  $e.Edge$  into  $head$  and  $tail$  at  $pos$ , creates a new global unique identifier  $Key$ , and thus composes three entries  $(e.Key, head, Key, e.DHT\_keys + DHT\_key)$ ,  $(Key, tail, e.ChildKey, e.DHT\_keys)$ ,  $(Key, keywords - head, -1, DHT\_key)$ , inserts them into the local repositories of the peers responsible for their  $Keys$  respectively and deletes  $e$ . All entry operations like splitting are atomic and have a lock on the entry. One insertion example of suffix tree is shown in Fig. 3. Here, we insert the text “CAB\$” into the DST overlay. Fig. 6 illustrates the above two cases of insertion. Fig. 6(a) shows the DST distribution after suffix  $CAB\$$  is indexed. Suppose  $key\_to\_peer(C)$  returns Peer6. On Peer6, the search will terminate at position 1 because there is no entry whose  $Key$  is  $C$ . Thus, a new entry  $(C, CAB\$, -1, \{h(CAB\$)\})$  is inserted into the DHT overlay. Fig. 6(b) shows the DST distribution after indexing suffix  $AB\$$ . The search passes entry  $(A, A, 2, \{h(ABABA\$)\})$  and terminates at position 2 at entry  $(2, BA, 7, \{h(ABABA\$)\})$ . Thus,  $(A, A, 2, \{h(ABABA\$)\})$  is changed to  $(A, A, 2, \{h(ABABA\$), h(CAB\$)\})$ ;  $(2, BA, 7, \{h(ABABA\$)\})$  is deleted; and, three entries  $(2, B, 1, \{h(ABABA\$), h(CAB\$)\})$ ,  $(1, A, 7, \{h(ABABA\$)\})$ , and  $(1, \$, -1, \{h(CAB\$)\})$  are created and inserted into the DHT overlay.

Tuple deletion occurs when a resource is removed. For each suffix of  $text$ , there exists one path from root to leaf in DST representing this suffix. The deletion algorithm deletes the  $DHT\_key$  from  $DHT\_keys$  fields of all entries in this path. If the  $DHT\_keys$  field of one entry is empty after deletion, this entry will be removed as it never denotes any resource. Entry deletion may make one edge have only one child edge or have no child edge. The former does not conform to the suffix tree algorithm, and therefore the concatenation of the two edges is needed. Suppose the two edges are  $parent$  and  $child$ , the concatenation algorithm firstly appends  $child.Edge$  to  $parent.Edge$  and replaces  $parent.ChildKey$  with  $child.ChildKey$ , and then deletes  $child$ . The latter makes a middle edge become a leaf edge; therefore, the  $ChildKey$  of the middle edge is set to  $-1$ .

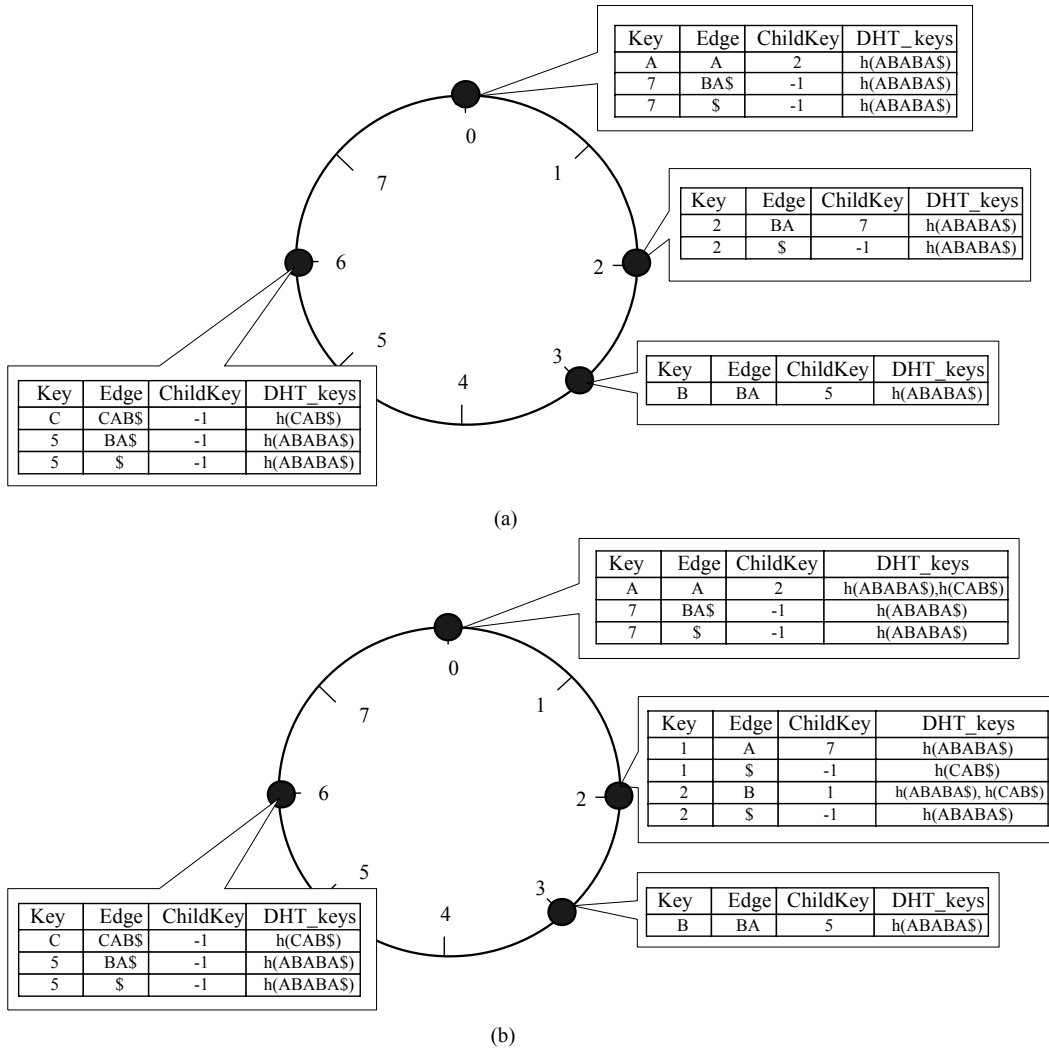


Figure 7. Examples of two cases of insertion.

## 4 Enhancement

### 4.1 Caching

The search process described in 3.3 is based on the DHT routing. This means each hop in the DST approach will produce a DHT *key\_to\_peer* mapping function call. This is time consuming in some cases. In order to avoid the frequent invoking of *key\_to\_peer*, each peer maintains a table to cache the IP addresses of the peers responsible for its *ChildKeys*. Also, considering the dynamicity of the DHT overlay, each peer periodically refreshes its cache table in *HeartBeat* messages to ensure the IP address and *ChildKey* pairs are up-to-date.

In this way, when the search process jumps from parent edge to child edge, the *key\_to\_peer(ChildKey)* is seldom needed to invoke. The time cost of search can be reduced to a magnitude like  $\lg n + m$ , where  $n$  is the number of peers and  $m$  is the length of keyword sequences.

## 4.2 Node Failure and Recovery

The DST approach relies on the DHT overlay to handle normal peer departures. However, in the case of node failure, the DHT overlay never guarantees the success of data taken-over. DST must take further work to ensure the integrity of the suffix tree structure in such cases. To ensure the integrity actually means to improve the entries availability. For this reason, one could maintain  $k$  different *Keys* for each entry to map it onto  $k$  points in the DHT overlay and accordingly replicate a single entry at  $k$  different peers. An entry is then unavailable only when all  $k$  replicas are simultaneously unavailable. One interesting thing is that one more *Key* can be computed by hashing an existing one, named seed *Key*, using secure hash functions like SHA-1 and also is globally unique. So theoretically, given one *Key*, we can take it as a seed to produce arbitrary number of *Keys*. At any time when the entry of the seed *Key* is unavailable, a recovery process is initiated. The successive *Keys* will be generated until an available up-to-date entry replica is detected. Then the entry of the seed *Key* is composed and reinserted into the DHT overlay. During the recovery period, any search request can be processed as normal, but the maintenance operations like tuple insertion and deletion which pass this entry will be suspended until the recovery is completed. The maintenance operations only affect the entry of the seed *Key*. During these operations, other entry replicas are only labeled as out-of-date, and search request can be answered as normal except that the results may be out of date. Fresh results can be got by a later re-request. After the maintenance operations at the entry of the seed *Key* are completed, the replicas will be updated gradually. If the entry of the seed *Key* is unavailable because of node failure when no replica is updated, the entry of the seed *Key* is first recovered in state of out-of-date, and then the maintenance operations will be taken again on this entry.

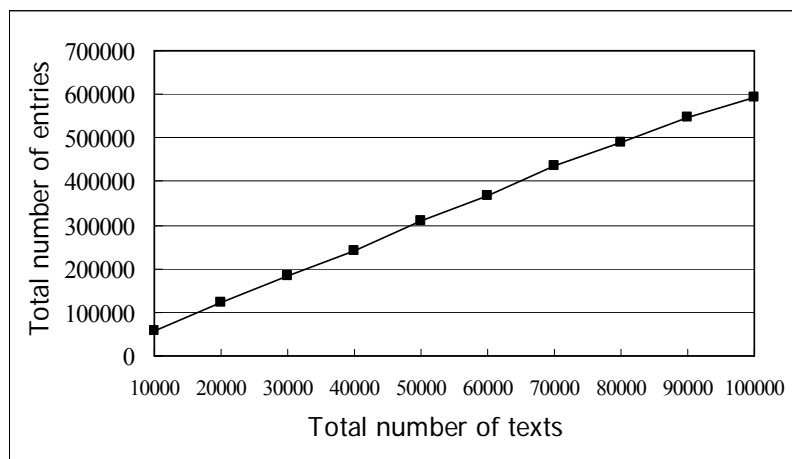
## 5 Simulation Results

This section presents simulation results demonstrating the benefits of the DST approach. We simulate a network consisting of  $10^3$  peers and taking Chord as its DHT overlay. The experimental data are collected from DBLP XML databases [7], which include XML metadata of 500,000 papers. The selected papers are firstly distributed into the DHT overlay, and then the titles of these papers are used as the description *texts* to construct the DST overlay. All experiments in this section are carried out upon this simulation infrastructure.

## 5.1 Load Balance

The DHT overlay performs load balancing well [15]. We hope to know whether the DST approach has the same ability of load balance, i.e. allocating entries to nodes evenly. So we carry out several experiments to examine the load balance property of the DST approach.

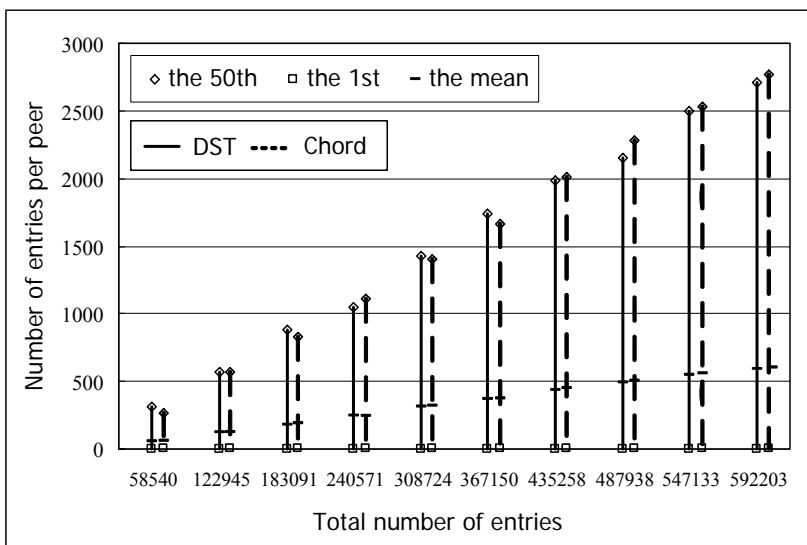
The first experiment is on how the number of entries created by the DST approach varies with the number of *texts* inserted into the DST. The reason for this experiment is that the load balance comparison between the DST and DHT overlay will use the number of entries in the DST. In this experiment, *texts* are randomly selected from 500,000 metadata and the total number of *texts* varies from  $10^4$  to  $10^5$  in increments of  $10^4$ . For each value, the experiment is repeated 10 times. Figure 8 shows the relationship between the total number of entries and the total number of *texts*. We can see that they almost conform to the linear relationship, i.e., the space cost of the DST approach is in linear with the number of inserted resources. This is a reasonably small usage of space for P2P networks.



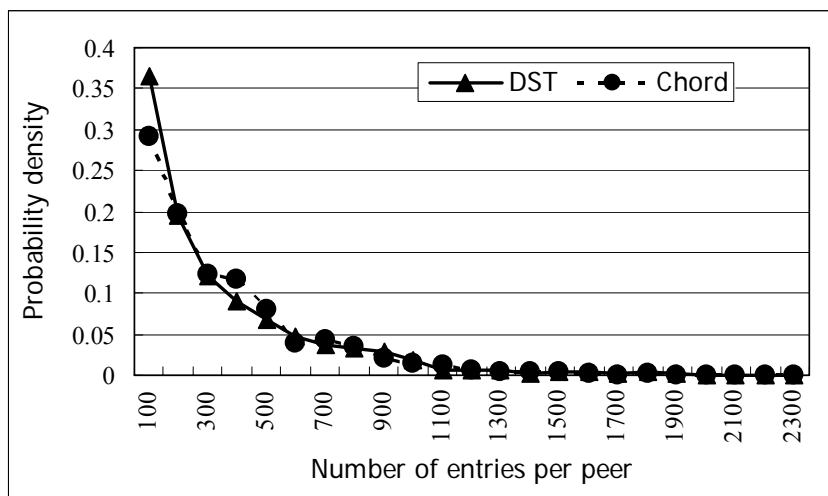
**Figure 8.** The total number of entries created by the DST varies with the total number of *texts*.

The load balance property of the DST approach is examined in comparison with the underlying DHT overlay, i.e. Chord in our simulation infrastructure. So, another experiment is carried out to test how the entries distributed on each peer in DST and how keys are distributed in Chord. Figure 9 plots the 1<sup>st</sup> percentile, the 50<sup>th</sup> percentile and the mean of the number of entries per peer in DST and the corresponding value of the number of keys per peer in Chord. The horizontal axis represents the total number of entries. As described above, these scale values are computed from the previous experiment. We can see that the number of entries per peer exhibits large variations that increase linearly with the total number of entries. The 50<sup>th</sup> percentile is about  $5.2 \times$  the mean value, while the

max value is about  $7.8 \times$  the mean value. Although with large variations, DST shows a similar pattern to the DHT overlay. To give more detailed information about the distribution of entries on peers, Figure 10 plots the probability density of the number of entries per peer when there are  $5 \times 10^4$  texts processed by the DST approach. This figure also contains the comparison with the DHT overlay, which shows that they exhibit the similar trend. From the two comparisons, we can see that the DST approach preserves the load balance of the DHT overlay, and has the same potential improvement as the DHT overlay. Load balance can be optimized by dividing one real peer into several virtual peers. More discussion on this can be found in [25, 28].



**Figure 9.** The 1<sup>st</sup> percentile, the 50<sup>th</sup> percentile and the mean of the number of entries per peer in DST and the number of keys per peer in Chord.



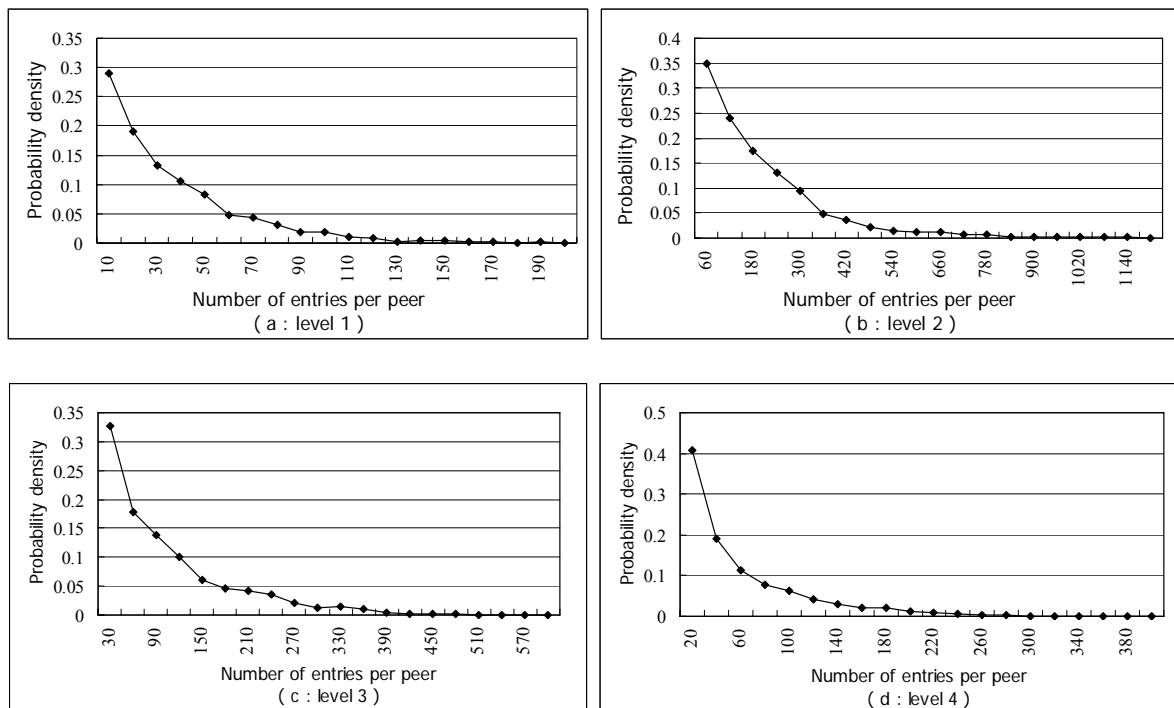
**Figure 10.** The probability density of the number of entries per peer. The total number of texts in DST is

$5 \times 10^4$ , while the total number of keys in Chord is equal to the total number of entries in DST.

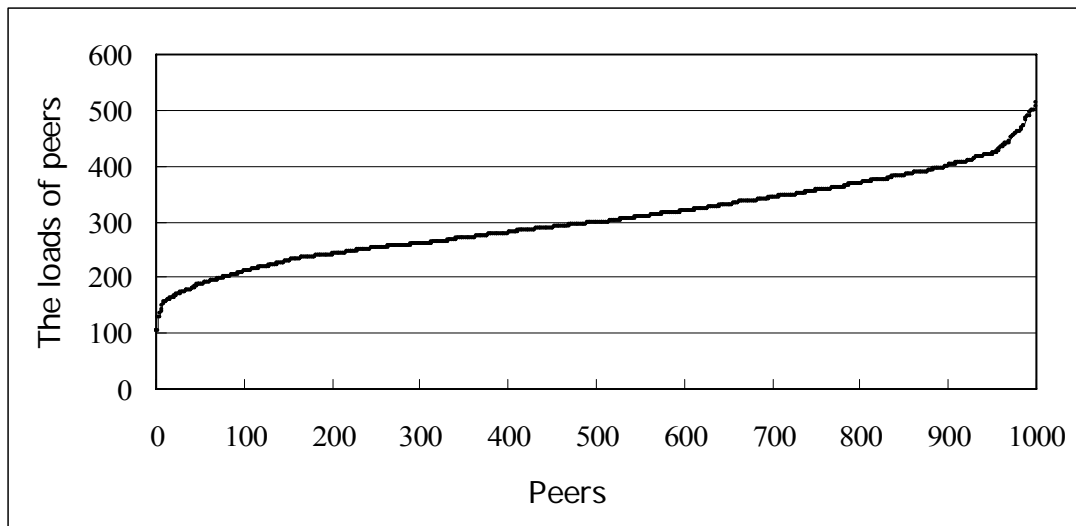
So far, we have validated the load balance property of the DST approach in a flat view but not considered the entries' level in DST. In the next experiment, we will evaluate the distribution of entries for different levels.

The DST approach balances the load in the tree structure by distributing edges in the same level evenly across the peers as described in section 3.2 about the generation of ChildKey. Figure 11 shows the distribution of the number of entries per peer for top 4 levels of the DST when there are  $5 \times 10^4$  texts processed by the DST approach. We can see that the distribution of edges in the same level is similar to the distribution of keys in Chord [28].

We carry out another experiment to evaluate the load of peers according to search requests. The keyword sequences for the search requests are randomly drawn from the texts, and their length varies from 1 to 10. We totally set up  $2 \times 10^4$  search requests and  $2 \times 10^3$  for each length. The load of one peer is calculated by the number of RPC call on it. Figure 12 shows the load of peers, where the horizontal axis represents the 1000 peers in ascending order of loads. We can see that the loads of most peers (more than 80%) are between 200 and 400, which are around the average load.



**Figure 11.** The probability density of the number of entries per peer for top 4 levels.



**Figure 12.** The loads of peers in case of  $2 \times 10^4$  search request.

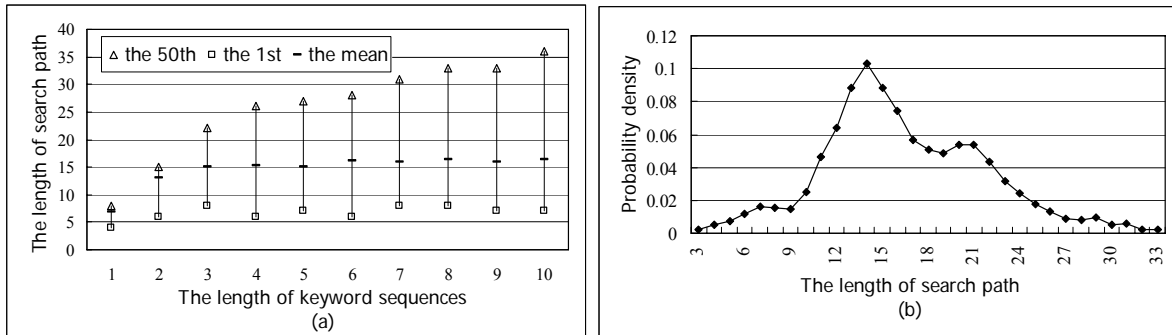
## 5.2 The Length of Search Path

The path length for a resource search is a key factor of the performance of any P2P systems. In the context of the DST approach, we define the length of search path as the number of peers traversed during a search process, and take it as the evaluation criterion of the search performance. As discussed in section 3.3 and section 4.1, the length of search path is determined by two factors, i.e., the length of the keyword sequences and the *key\_to\_peer* operation of the DHT overlay.

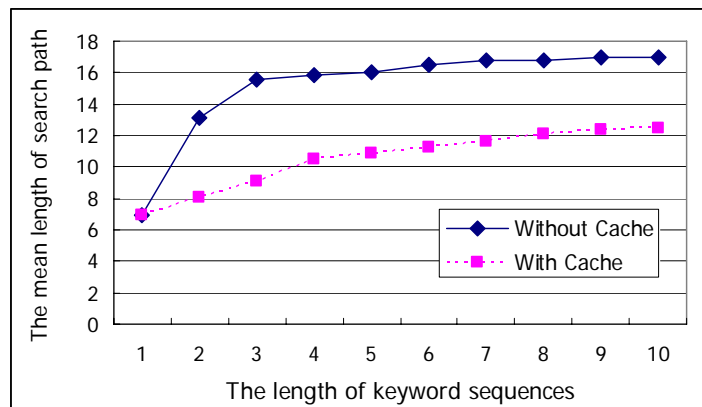
To understand DST's search performance in practice, we carry out an experiment to evaluate how the length of search path varies with the length of the keyword sequences. The keyword sequences for the search requests are randomly drawn from the texts storing in the simulation infrastructure, and their length varies from 1 to 10. For each length value, we repeat the evaluation 100 times to measure the length of search path. And as Chord was selected as the DHT overlay in the simulation infrastructure, the time cost of *key\_to\_peer* operation is  $O(\log n)$ , where  $n$  is the size of the P2P network.

Figure 13(a) plots the 1<sup>st</sup> percentile, the 50<sup>th</sup> percentile and the mean of the length of search path. The mean length of search path exhibits a sub-linear relationship with the length of keyword sequences, and so do the 1<sup>st</sup> and 50<sup>th</sup> percentiles. Besides, the mean length of search path tends to be a constant value. We have given a theoretical analysis about this interesting property in section 6. Figure 13(b) plots the distribution curve of the length of search path when  $m = 4$ . Most values are around the mean value and the big ones are very few. Figure 14 plots the effect of cache on the length of search path. Most *key\_to\_peer* operations can be saved by caching the calling result.

Besides the mean value is lower, the one with cache also shows a better probability density, and the length values are more concentrating on the mean value.



**Figure 13.** (a) The length of search path as the function of the length  $m$  of keyword sequences. (b) The probability density of the length of search path when  $m=4$ .



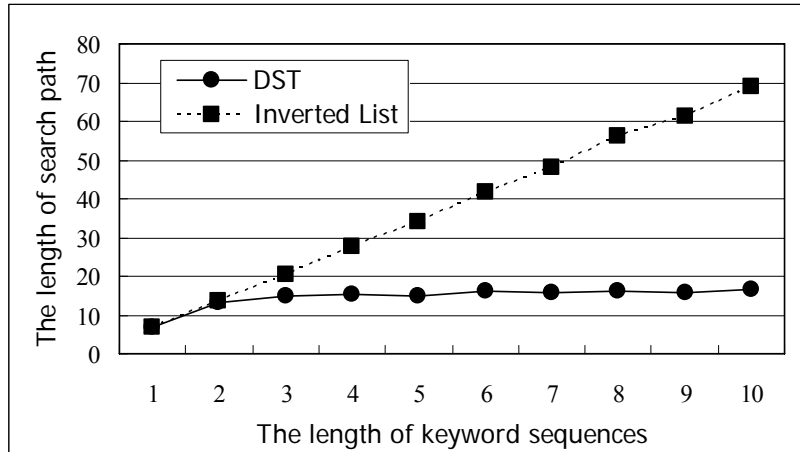
**Figure 14.** Effect of cache on the length of search path.

### 5.3 Comparison with the Inverted List Approach

The inverted list approach can be briefly described as follows [25]: Each keyword has a posting list of documents containing this keyword. Its data structure is distributed onto P2P networks by mapping keywords onto nodes and storing the posting list on this node.

The DST approach has a high performance in keyword sequence search, especially for lengthy sequences. Figure 15 plots the mean length of search path of the DST approach without cache and the inverted list approach. When the length of keywords  $> 2$ , the inverted list approach exhibits a linear tendency while the DST approach is tending towards a constant cost. And the DST approach maintains the sequential relationship of keywords naturally, while the inverted list approach must take further operations like intersections and position evaluations to make sure that the keywords

exist sequentially in the result *texts*. Although these operations could be processed locally, they are heavy tasks when the result datasets are large. However, tuple insertion and deletion operations in the DST approach require more messages than in the inverted list approach, as in this simulation, the DST approach generates about twice as many messages as the inverted list approach for each tuple insertion or deletion operation.



**Figure 15.** Average length of search path varies with the length of keyword sequences in the DST approach and in the inverted list approach.

## 6 Theoretical Analysis

We can see from the simulation that the length of search path is tending towards a constant value when the length of keyword sequences is bigger than a certain value. The underlying reason is that the length of search path has an upper bound (i.e., the depth of the whole virtual suffix tree). This section theoretically analyzes this upper bound.

The depth of the DST can be analyzed by a probabilistic model. A text can be regarded as a series of independent random variables taking its value randomly from the vocabulary.

We firstly define the repeated substring of a *text* as the substring that appears more than once within *text*. For example, for the *text* *ABABA*, substring *ABA* is one of the repeated substrings, since it appears twice within *ABABA*.

**LEMMA 1.** For a text, the depth of its corresponding suffix tree is less than or equal to the maximum length of its repeated substrings.

**PROOF.** In a suffix tree,  $group_i$  represents the group that leaf edges having the same parent can be gathered;  $prefix_i$  represents the substring corresponding to the path that the leaf edges in  $group_i$  share

from the root edge to their parent edge;  $depth_i$  represents the depth of leaf edges in  $group_i$ ; and  $depth$  represents the depth of the suffix tree.

Since each edge consumes at least one word, we could deduce the inequation:  $depth_i \leq strlen(prefix_i)$ .

Assume  $depth_k = \text{MAX}(depth_i \mid group_i \in group)$ , then  $depth = depth_k \leq strlen(prefix_k) \leq \text{MAX}(strlen(prefix_i) \mid group_i \in group)$ .

And, according to the attributes of suffix tree, each group should have at least two leaf edges. Thus  $prefix_i$  is shared by at least two leaf edges, and fulfills the definition of the repeated string. Therefore, the depth of a  $text$ 's corresponding suffix tree is less than or equal to the maximum length of the  $text$ 's repeated substrings. (End of Proof)

Now, the expected value of the depth of a suffix tree is available by analyzing the length of the repeated substrings in its corresponding text. Under the simple probabilistic model mentioned above, we hope to know what kind of probability distribution the length of repeated substring exhibits. To find the precise probability is difficult and not necessary in this research. The following lemma gives the up-bound.

**LEMMA 2.** Let  $P_l$  be the probability that the text contains repeated substrings of length  $\geq l$ . Suppose the length of the text is  $L$ , and the size of vocabulary is  $M$ , then

$$P_l \leq \frac{(L-l)^2}{M^l}$$

**PROOF.** We first investigate the number of ways to compose the text containing repeated substrings with length  $\geq l$  under the mentioned probabilistic model. The definition of the repeated substring implies an interesting property that any substring of a repeated substring is also a repeated substring. This means that if a text contains repeated substrings longer than  $l$ , it also contains repeated substrings of length  $l$ . Therefore, the way to compose the text containing repeated substrings of length  $\geq l$  is also the way to compose the text containing  $l$ -length repeated substrings. So we only need to compute the number of ways to compose the text containing  $l$ -length repeated substrings.

Let  $A_1A_2\dots A_L$  be the text and  $S$  be  $l$ -length repeated substring, suppose  $S$  appears twice at positions  $p$  and  $q$  ( $p < q$ ), and the remaining words in text are randomly selected from the vocabulary.  $S$  appears in text more than twice is a special case of the above composition method. According to different positions that  $S$  appears, the following two cases need to be examined:

(1) In *overlap case*, we have  $x = q - p < l$ . The two appearances should be equal to each other, so

$A_p A_{p+1} \dots A_{p+l-1} = A_q A_{q+1} \dots A_{q+l-1} = A_{p+x} A_{p+x+1} \dots A_{p+x+l-1}$ . Suppose  $k = l / x$ , then we have:  $A_p A_{p+1} \dots A_{p+x-1} = A_{p+x} A_{p+x+1} \dots A_{p+2x-1} = \dots = A_{p+kx} A_{p+kx+1} \dots A_{p+(k+1)x-1}$ , and  $A_p A_{p+1} \dots A_{p+l-kx-1} = A_{p+(k+1)x} A_{p+(k+1)x+1} \dots A_{p+x+l-1}$ . Therefore,  $A_p A_{p+1} \dots A_{p+x-1}$  can determine the way to compose the two appearances. As  $A_p A_{p+1} \dots A_{p+x-1}$  is also randomly selected from the vocabulary, it has  $M^x$  different cases. And the remaining words have  $M^{L-l-x}$  different cases. Thus, when the distance between two appearances  $x < l$ , we have the number of ways is less than  $C_{L-l-x+1}^1 M^x M^{L-l-x}$ , and  $C_{L-l-x+1}^1 M^x M^{L-l-x} = (L-l-x+1) M^{L-l}$ .

(2) In non-overlap case, we have  $x = q - p > l$ . The repeated substring has  $M^l$  different cases and the remaining words have  $M^{L-2l}$  different cases. Thus, when the distance between two appearances  $x > l$ , we have the number of ways is less than  $C_{L-l-x+1}^1 M^l M^{L-2l}$ , and  $C_{L-l-x+1}^1 M^l M^{L-2l} = (L-l-x+1) M^{L-l}$ .

The above analysis implies that the number of ways to compose the text containing  $l$ -length repeated substrings is less than

$$\sum_{x=1}^{L-l} (L-l-x+1) M^{L-l}.$$

And we have:

$$\sum_{x=1}^{L-l} (L-l-x+1) M^{L-l} < \sum_{x=1}^{L-l} (L-l) M^{L-l} = (L-l)^2 M^{L-l}.$$

Therefore, the number of ways to compose the text containing repeated substrings of length  $\geq l$  is less than  $(L-l)^2 M^{L-l}$ . As the number of ways to compose the  $L$ -length text is  $M^L$ , we have

$$P_l \leq \frac{(L-l)^2}{M^l} \text{ hold. (End of Proof).}$$

Lemma 2 tells us: the probability decreases with the increase of  $l$ . If  $M = 1000$  and  $L = 10000$ , the ratio of  $l > 20$  approaches 0. This means: for a text of length 10000 corresponding to vocabulary size 1000, the depth of its suffix tree is less than 20 with high probability.

As described in section 5.2, the time cost of the DST approach is measured by the length of search path. The worst case is the depth of the whole virtual suffix tree, in other words, the upper bound of the path length is the depth of the whole virtual suffix tree. Thus, the time cost of the DST approach is less than a reasonably small value in all probabilities.

Also from probabilistic analysis, the sum for  $l \geq 0$  of the probabilities that a random variable  $> l$  is the average value of that random variable, so the average length of the repeated strings is (it is also an up-bound)

$$\sum_{l=1}^{L-1} \frac{(L-l)^2}{M^l} < \sum_{l=1}^{L-1} \frac{L^2}{M^l}.$$

When  $l < \log_M^{L^2}$ , we have  $\frac{L^2}{M^l} > 1$ , but a probability value is never greater than 1. Therefore, the average length of the repeated strings can be precisely expressed as:

$$\sum_{l=1}^{\log_M^{L^2}} 1 + \sum_{l=\log_M^{L^2}}^{L-1} \frac{L^2}{M^l} < \sum_{l=1}^{\log_M^{L^2}} 1 + \sum_{l=\log_M^{L^2}}^{\infty} \frac{L^2}{M^l} = \log_M^{L^2} + \frac{M}{M-1}.$$

As  $\frac{M}{M-1}$  approaches 1, the average length is logarithmically small. Hence the search request could be answered in certain steps.

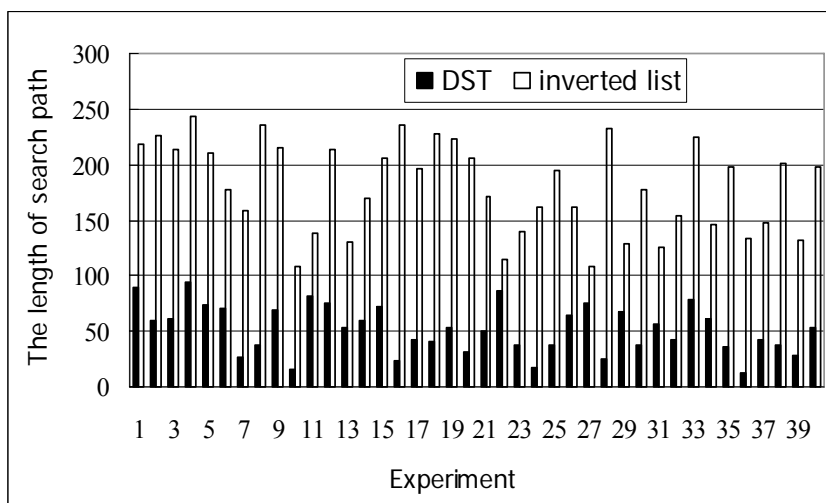
## 7 Application

The DST approach is very useful in deep document search due to the following two characteristics.

- (1) Semantic ability. The DST can express the sequential relationship of words and cluster the text blocks of the same sequential keywords. Sequential words reflect richer semantics than isolated words. Sequential words in a text reflect its main content. So it is useful in basic semantic representation and analysis in document systems.
- (2) High performance in keyword sequence search.

Especially, it is very useful in accurately retrieving scientific documents by keyword sequences, for example, in search by samples (text or sequential words extracted from text).

To examine the advantages of the DST approach in document retrieval, experimental data is collected from IEEE portal databases, which include XML metadata of published papers. A crawler randomly selects 40 papers and extracts their abstracts as the description *texts*. After filtering out the preposition, the article, the conjunction, and the punctuation, the number of keywords in these abstracts varies from 50 to 300. 40 selected search requests are sent to both the DST approach and the inverted list approach. Figure 16 shows the result of each search request.



**Figure 16.** The length of search path in exact retrieval of scientific articles by providing keywords extracted from abstract.

Figure 16 shows that the length of search path of the DST approach is less than that of the inverted list approach in processing one search request, in other words, the DST approach can finish one search in fewer routing hops. This means the reduced search latency and bandwidth consumption of the network. So the DST approach is suitable for distributed content-based document retrieval.

## 8 Evaluation

Here we discuss the applicability of the DST approach to get an insight into when the DST approach can improve the performance of an application.

- (1) *Keyword-based resource locating.* The DHT-based systems largely solve the problem of scalability as each lookup of a data item can be resolved within  $O(\log n)$  (or  $O(n^d)$ ) routing hops for a network of  $n$  peers. However, the DHT-based systems only support exact match lookups. The DST approach overcomes this shortcoming by supporting keyword sequence search.
- (2) *Time cost.* The DST works well with the keyword sequence search. As shown in the experiment and theoretically analysis, when the length of the keyword sequences is bigger than a certain value (like 3 in the experiment), the DST approach almost exhibits a constant time usage tendency. In other words, it can finish a keyword sequence search request by routing across certain number of peers. When the number of keywords is relatively large (e.g., 10), it can dramatically reduce the search latency and the bandwidth consumption of the network. However,

if the number of keywords is not very large or the sequential relationship between keywords is not needed, the DST approach behaves just like the inverted list approach.

- (3) *Supporting semantics.* The DST can express the sequential relationship between words that can represent the semantics of text better than isolated words, and can help discover the semantics between given words. So it can be a helpful tool for basic semantic representation and analysis, especially for large-scale text/string analysis applications.
- (4) *Maintenance cost.* The message transferring during resource publishing and removal occurs in two cases, one is search process, and the other is splitting or concatenation process. As the splitting and concatenation process only need a constant number of messages during the suffix insertion and deletion and the search cost exhibits a constant tendency, *the total number of messages* =  $\rho \times \text{the length of the text description of the resource}$ , where  $\rho$  is a nearly constant factor. However, the maintenance of the DST is more complicated than the inverted list approach as the DST must make sure that the structure of virtual suffix tree is maintained correctly during resource publishing and removal. More messages will be transferred than the inverted list approach to adjust the indexed entries to correctly reflect the changes.

## 9 Conclusion and Future Work

The DST reflects a kind of sequential semantic relationship between words so it supports efficient search in large-scale documents distributed on P2P networks. The major contribution of this paper includes two aspects. First, the proposed approach is scalable, fast and load balanced. Second, the time cost of keyword sequences search is in sub-linear with the length of the string to be searched. Besides, the DST overlay does not rely on any particular DHT overlay because the DST approach interacts with the DHT overlay only by a *key\_to\_peer* mapping function. That is, any underlying overlay is suitable for the DST approach as long as it provides the mapping function. This enables the DST approach to apply to wider applications. In addition, mechanisms of caching, replicas and ranking of results as well as the bloom filter technique [1] can be further incorporated with the DST approach to improve the search efficiency in applications.

However, this is just a step toward an ideal semantic overlay to support intelligent applications on large-scale P2P networks. Different semantic structures perform differently in different applications. Ongoing work is to incorporate the DST approach with other semantic structures [37] to construct a semantic-rich overlay on P2P networks.

## References

- [1] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, vol.13, no.7, 1970, pp.422–426.
- [2] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath Lookup Queries in P2P Networks. In *ACM CIKM International Workshop on Web Information and Data Management*, 2004.
- [3] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker. A Case Study in Building Layered DHT Applications. In *Proceedings of ACM SIGCOMM*, Aug. 2005, pp. 97–108
- [4] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Jul. 2000, pp. 46–66.
- [5] A. Crespo, and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, Jul. 2002, pp. 23–32.
- [6] F. M. Cuenca-Acuna, and T. D. Nguyen. Text-based Content Search and Retrieval in ad hoc P2P Communities. In *Proceedings of the International Workshop on Peer-to-Peer Computing*, May. 2002, pp. 220-234.
- [7] DBLP XML Database. <http://dblp.uni-trier.de/xml>
- [8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2001, pp. 202–215.
- [9] M. J. Freedman, and R. Vingralek. Efficient Peer-to-Peer Lookup Based on a Distributed Trie. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, Mar. 2002, pp. 66–75.
- [10] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating Data Sources in Large Distributed Systems. In *Proceedings of the International Conference on Very Large Data Bases*, Sep. 2003, pp. 874–885.
- [11] L. Garcés-Erice, P. A. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross. Data Indexing in Peer-to-Peer DHT Networks. In *Proceedings of International Conference on Distributed Computing Systems*, Mar. 2004, pp. 200–208.
- [12] O. D. Gnawali. A Keyword Set Search System for Peer-to-Peer Networks. Master's thesis, Massachusetts Institute of Technology, Jun. 2002.
- [13] Gnutella website. <http://www.gnutella.com>
- [14] M. Harren, et al. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, 2002, pp. 242–250.

- [15] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, May. 1997, pp. 654–663.
- [16] D. E. Knuth. *The Art of Computer Programming*, vol. 3: Sorting and Searching. Second Edition, Addison-Wesley, Mass, 1973.
- [17] G. Koloniari, and E. Pitoura. Peer to Peer Management of XML Data: Issues and Research Challenges. In *Proceedings of ACM SIGMOD*, Jun. 2005, pp. 6–17.
- [18] B. Kröll, and P. Widmayer. Distributing a Search Tree among a Growing Number of Processors. In *Proceedings of the ACM SIGMOD*, May. 1994, pp. 265–276.
- [19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 190–201.
- [20] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, Feb. 2003, pp. 207–215.
- [21] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, vol.23, 1976, pp.262-272.
- [22] R. Motwani, and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, 1995.
- [23] Napster. <http://www.napster.com>
- [24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Aug. 2001, pp. 161–172.
- [25] P. Reynolds and A. Vahdat. Efficient Peer-to-peer Keyword Searching. In *Proceedings of International Middleware Conference*, Jun. 2003, pp. 21–44.
- [26] A. Rowstron, and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *ACM/IFIP/USENIX International Middleware Conference*, Nov. 2001.
- [27] C. Schmidt, and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *the IEEE International Symposium on High-Performance Distributed Computing*, 2003.
- [28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the Conference on Applications, Technologies,*

- Architectures, and Protocols for Computer Communications*, Aug. 2001, pp. 149–160.
- [29] C. Tang and S. Dwarkadas. Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval. In *The First Symposium on Networked Systems Design and Implementation*, Mar. 2004.
- [30] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Aug. 2003, pp. 175–186.
- [31] B. Yang, and H. Garcia-Molina. Efficient Search in Peer-to-peer Networks. Technical Report 2001-47, Stanford University, Oct. 2001.
- [32] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Brushwood: Distributed Trees in Peer-to-Peer Systems. In *The 4th International Workshop on Peer-to-Peer Systems*, Feb. 2005.
- [33] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report: TR-703-04, University of Princeton, Computer Science Department, 2004, [http://www.cs.princeton.edu/\\_chizhang/skipindex.pdf](http://www.cs.princeton.edu/_chizhang/skipindex.pdf).
- [34] 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.
- [35] H. Zhuge. The Knowledge Grid. *World Scientific*, Singapore, 2004.
- [36] H. Zhuge, J. Liu, L. Feng, X. Sun, and C. He. Query Routing in a Peer-to-Peer Semantic Link Network. *Computational Intelligence*, vol. 21, no. 2, 2005, pp.197–216.
- [37] H. Zhuge, X. Sun, J. Liu, E. Yao, and X. Chen. A Scalable P2P Platform for the Knowledge Grid. *IEEE Transactions on Knowledge and Data Engineering*, vol.17, no.12, 2005, pp.1721–1736.

*Technical Report of Knowledge Grid Research Center, KGRC-2007-02, May, 2007.*

[www.knowledgegrid.net/TR](http://www.knowledgegrid.net/TR).