A Framework for Peer-To-Peer Lookup Services based on k-ary search

Sameh El-Ansary

Swedish Institute of Computer Science
Kista, Sweden

Luc Onana Alima

Department of Microelectronics and Information Technology
Royal Institute of Technology
Kista, Sweden

Per Brand

Swedish Institute of Computer Science
Kista, Sweden

Seif Haridi

Department of Microelectronics and Information Technology
Royal Institute of Technology
Kista, Sweden

SICS Technical Report T2002:06
ISSN 1100-3154
ISRN:SICS-T–2002/06-SE

Abstract

Locating entities in peer-to-peer environments is a fundamental operation. Recent studies show that the concept of distributed hash table can be used to design scalable lookup schemes with good performance (i.e. small routing table and lookup length). In this paper, we propose a simple framework for deriving decentralized lookup algorithms. The proposed framework is simple in that it is based on the well-known concept of k-ary search. To demonstrate the applicability of our framework, we show how it can be used to instantiate Chord. When deriving a generalized Chord from our framework, we obtain better performance in terms of the routing table size (38% smaller than the generalization suggested by the Chord authors).

Keywords: *Lookup, peer-to-peer, distributed hash table, k-ary search.*

1 Introduction

Peer-to-peer systems emerged as a special field of distributed systems where the lack of centralized control is a key requirement. Lookup services is one area in the peer-to-peer field that deserves a particular attention as a lookup service is a core requirement in peer-to-peer systems and applications. Given a certain key, the main task of a lookup service is to locate a network node that is responsible for that key.

The lookup problem in peer-to-peer systems has been approached in several ways. In our view, existing lookup services could be categorized based on two main properties: *i*) scalability, *ii*) hit guarantee, i.e., possibility of locating an entity in the system given that it is present. Depending on the application, other properties such as security and anonymity may be of interest.

In most of the early peer-to-peer systems such as Napster [3], Gnutella [2] and FreeNet [1], the hit guarantee and the scalability properties are either missing or not simultaneously satisfied. For example, the centralized directory in Napster offers the hit guarantee property while it renders the system unscalable. In Gnutella, the flooding approach prevents it from being scalable [5]. Furthermore, the hit guarantee is limited to the scope of the flooding. Similarly, in FreeNet the search scope is bounded and the use of caching can lead to inconsistent views of the network. The scalability of FreeNet is still to be evaluated.

Later approaches to the lookup problem are based on the concept of *Distributed Hash Table* (DHT). This approach is represented, for example, by systems such as Chord [6], Tapestry [8] and CAN [4]. The idea behind this approach is to let all the names of the different entities in the system be mapped to a single search space by using a certain hashing function and thus all the entities in the system have a consistent view of that mapping. Given that consistent view, various structures of the search space are used for locating entities. For example, in Chord, the search space is structured as a ring. In Tapestry, it is structured as a mesh. In CAN, it is structured as a d-dimensional coordinate space.

The hit guarantee property is well-addressed in the three above-mentioned systems as the whole search space is considered by the indexing structures in the three cases of ring, mesh and d-dimensional space and is no longer limited to the scope of a certain query. The different indexing structure are realized by means of routing tables. The hit guarantee is offered under normal failure conditions as the three algorithms provide fault-handling mechanisms to repair outdated routing tables. Scalability is also well-addressed because

	Lookup length	Routing entries	Comments
Chord	$log_2(N)$	$log_2(N)$	N, system size
Tapestry	$log_b(N)$	$blog_b(N)$	b, search space encoding base
CAN	$\frac{d}{4}n^{\frac{1}{d}}$	$2d$	d, some constant

Table 1: Lookup length and routing information required in three DHT-based lookup services

of the fact that a reasonable amount of routing information is required in order to offer an acceptable *lookup length* (i.e., number of hops to resolve a query). Table 1 shows that Chord and Tapestry both offer a lookup length and a number of routing table entries that grow logarithmically with the system size. CAN offers a lookup length that grows with the system size as a polynomial with order $1/d$, for some constant d and requires a constant amount of routing information.

1.1 Motivation and contribution

After exploration of some of the DHT-based lookup services, we were interested to answer the following question: *Is there a general abstraction that can be used to derive most of the existing DHT lookup services?*

By investigating the question, we observed that the idea of k-ary search seems to be general enough to derive several DHT-based lookup algorithms.

In this paper we show that:

- The lookup problem in peer-to-peer networks could be perceived as k-ary search.
- The DHT-based lookup service, Chord, is a special case of k-ary search where $k = 2$, i.e. performing binary search.
- This line of thinking can improve the lookup length of Chord and the number of routing table entries.

In general, DHT-based lookup services have three basic operations: Insertion, deletion and lookup. The scope of this paper will cover only the lookup operation. In a future paper, we will show how the k-ary search framework can simplify the insertion and deletion operations.

To present the suggested framework, in section 2, we introduce the Chord algorithm. In section 3, we show how the Chord algorithm can be perceived as an algorithm that mimics binary search. In section 4, we show how to

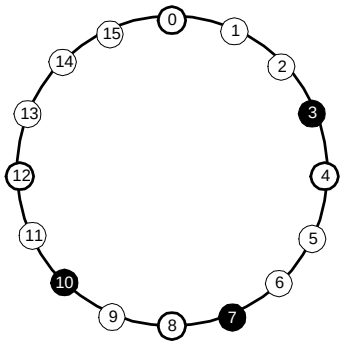

Figure 1: An example Chord network with 16 nodes.

perceive the lookup problem as k-ary search. Based on this result, in section 5, we show how the k-ary search framework can improve Chord lookup algorithm and the number of routing table entries. Finally, we conclude our work and present future directions in section 6.

2 The Chord lookup algorithm

In this section, we review the Chord system without considering the aspects of node joins and failures. We only focus on the lookup functionality.

Assuming a network of nodes where each node is assigned a number of keys, the Chord system provides a lookup service. That is, given a key K, a node running the chord algorithm will be able to determine the node to which K is assigned.

2.1 The Chord identifier/search space

The nodes' addresses and the keys of data items are both hashed to form a single identifier space. Where each identifier is encoded using m-bits. The identifiers are ordered in an identifier circle modulo 2^m.

2.2 Key assignment

Each identifier in the circle corresponds either to a node address or a key of a data item. Let *ID* be the function that maps nodes and keys to the identifier space. We say that a key K is assigned to node n iff

- $ID(K) = ID(n)$ or
- $ID(n)$ is the first identifier that corresponds to a node in the clockwise traversal of the identifier circle, starting from $ID(K)$.

When a key K is assigned to a node n, we say that node n is the successor of K. From now on, we do not make a distinction between a key and its identifier. The same applies for the nodes. Therefore, for an identifier k, we write $successor(k)$ to denote the node to which, the key that maps to k is assigned.

Using the system depicted in Figure 1, which has three nodes, namely node 3, 7 and 10, the idea of key assignment is as follows. All identifiers from 11 to 3 are assigned to node 3; all identifiers from 4 to 7 are assigned to node 7 and all identifiers from 8 to 10 are assigned to node 10.

2.3 The routing table

Each node in the Chord network maintains a routing table of m entries called the *finger table*. At a given node n, the i-th entry of this table, stores the node s such that s is the successor of $n \oplus_{2^m} 2^{i-1}$[1].

2.4 Key location

In this subsection, we briefly describe how to find the location of keys in a Chord network. When a node n receives a query for a key k, n will use its fingers as follows:

- If $k \in]n, successor(n)]$ then n returns successor of n and we say the query is resolved.
- If $k \notin]n, successor(n)]$ then, node n forwards the query to the node n', which is the closest preceding node of k according to n's finger table. When n' receives the forwarded query, it acts like node n.

[1]The notation $x \oplus_z y$ is used to denote $(x + y)\ mod\ z$.

2.5 Complexity

The m-th entry of each finger table contains the address of the node f_m, where $f_m = successor(n \oplus_{2^m} 2^{m-1})$. Thus, if a query cannot be resolved at a node n, the node n will forward the query to f_m, which is at least half way between n and the target. Using this argument, it is proven in [7] that $log_2(N)$ hops are sufficient to resolve a given query with a routing table of $log_2(N)$ entries.

3 Chord as binary-search

Although not explicitly stated in [6, 7], we can see that the Chord lookup algorithm mimics binary search. Seeing the Chord lookup as a binary search simplifies its understanding. In this section, we show how this is the case. Before explaining, we introduce the following definition:

Definition 3.1 *Let $I =]x, y]$ be an interval of identifiers. We call the node with identifier x, the responsible for I.*

The definition above deserves a comment. The responsible for a given interval I, is the node to which a query for a key k is forwarded once it is determined that k belongs to I.

To show how the Chord lookup algorithm can be perceived as binary search, we consider a fully populated Chord network with 16 nodes. We say that a Chord network is fully populated when there is a node at each identifier of the identifier space.

In order to determine the location of a key, a query is introduced to the Chord network. A query arrives at a node either as an *original* query or as a *forwarded* query. Therefore, a precise characterization of a query Q at an arbitrary node n can be given in terms of the number of hops that Q made in order to reach node n. Hence, an original query made zero hops while a forwarded query made one or more hops. We will denote a query that made i hops, $i \geq 0$, an i-hop query.

Let us see how the Chord lookup algorithm determines the location of key k assuming that the original query for k arrives at node 0.

When the original (or the 0-hop) query for k arrives at node 0, node 0 determines the search space for k, which for node 0, is the whole identifier space, denoted $]0, 0]$, traversing the ring clockwise. Then, node 0 performs the following steps:

1. Using its 4-th entry of the finger table, node 0 divides the search space into the two intervals $]0, 8]$ and $]8, 0]$.

2. Determines the interval to which k belongs.

3. Forwards the query to the node responsible for the interval to which k belongs. Given the two intervals above, the query is forwarded either to node 0 itself or to node 8.

At this point, two cases are to be considered depending on which node the query is forwarded to.

Case 1: the query was forwarded to node 0 itself. In this case, node 0 receives the query after *one* hop and performs the following steps:

1. Using its 3-rd entry of the finger table, node 0 divides the new search space (i.e. $]0, 8]$) into the two intervals $]0, 4]$ and $]4, 8]$.

2. Determines the interval to which k belongs.

3. Forwards the query to the node responsible for the interval to which k belongs. That is, the query is forwarded either to node 0 itself or to node 4.

Case 2: the query was forwarded to node 8. The characteristic of the forwarded query when it arrives at node 8 is that it made *one* hop. Thus, when node 8 receives this "one hop" query for k, node 8 determines that the search space for this query is $]8, 8 \oplus_{16} \frac{16}{2^1}]$ and 8 performs the following steps:

1. Using its 3-rd entry of the finger table, node 8 divides the search space for k into the two intervals $]8, 12]$ and $]12, 0]$.

2. Determines the interval to which k belongs.

3. Forwards the query to the node responsible for the interval to which k belongs. At this point, the query is forwarded to either node 8 itself or to node 12.

By continuing the above startegy of processing forwarded queries, we can observe that each node that receives an x-hop query, $0 \leq x \leq 3$, has only two forwarding alternatives, which means that the search process follows a path of a binary search tree. Figure 2 illustrates this behavior.

As illustrated in figure 2, after each hop, the search space is halved into two intervals. Therefore, any other node in the network is reachable from the originating node of the query, within 4 hops.

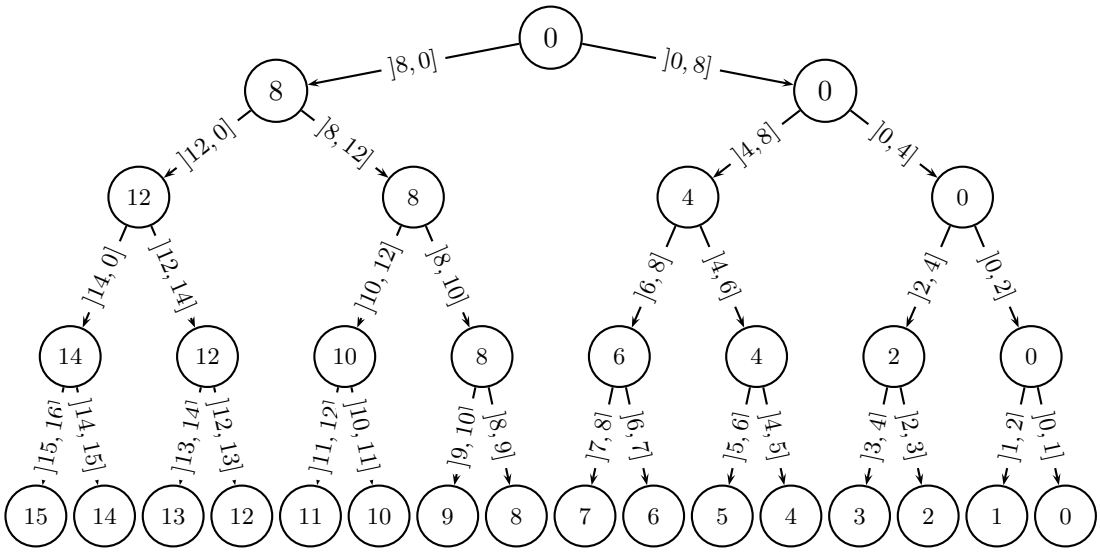

Figure 2: Decision tree for a query originating at node 0 in a 16-node network applying binary search

In general, if N and H are respectively the system size and the maximum number of hops, then when a node n receives a i-hop query, $0 \leq i \leq H-1$, the node n does the following:

1. Determines the search space for the query. This search space is given by the interval
$$]n, n \oplus_N \frac{N}{2^i}] \tag{1}$$
2. Using the $(H-i)$-th entry of its finger table, node n divides the search space for k into two intervals: $]n, n \oplus_N \frac{N}{2^{i+1}}]$ and $]n \oplus_N \frac{N}{2^{i+1}}, n \oplus_N \frac{N}{2^i}]$
3. Determines the interval to which k belongs.
4. Forwards the query to the node responsible for the interval to which k belongs. More precisely, node n forwards the query to either node n itself or to node $n \oplus_N \frac{N}{2^{i+1}}$.

3.1 Complexity

Given that, for each query, the Chord lookup algorithm follows a path of a binary search tree rooted at the node where the query originated, the following results follow.

Theorem 3.1 (Lookup length) *The maximum number of hops for any query to be resolved, is* $log_2(N)$.

Proof : Follows from the fact that the height of a binary tree of N nodes is $log_2(N)$. ∎

Theorem 3.2 (Routing Table Entries) *The maximum number of routing table entries at each node is* $log_2(N)$.

Proof : Let n be an arbitrary node. From the above algorithm, node n must be able to forward any x-hop query, $0 \leq x \leq H-1$, where H is the maximum number of hops required to resolve any query.

In order for the node n to route an x-hop query, the node n must select *exactly one* destination between *two* possible forwarding alternatives. But, as the node n does not need an entry for routing to itself, only one entry is needed.

Overall, since x varies from 0 to $H-1$, and one entry is needed for each x-hop query, therefore, H entries in the routing table are needed.

Since n is an arbitrary node, the theorem follows. ∎

4 Lookup services as k-ary search

Having observed that the Chord algorithm mimics binary search, we generalize this idea to develop a modified algorithm that rather mimics k-ary search , $k \geq 2$. We consider a fully populated system that consists of N nodes and assume that the maximum number of hops required to resolve any query is H. In addition, we assume that the identifier space is organized as a circle modulo N.

When a node n receives a i-hop query for key y, $0 \leq i \leq H-1$, the node n does the following:

1. Determines the search space for the for key y. This search space is given by the interval
$$]n, n \oplus_N \frac{N}{k^i}] \tag{2}$$
2. Using the $(H-i)$-th entry of its finger table, node n divides the search space for y into k intervals:
$]n \oplus_N \frac{N}{k^{i+1}}j, n \oplus_N \frac{N}{k^{i+1}}(j+1)]$, $0 \leq i \leq H-1$, $0 \leq j \leq k-1$.
3. Determines the interval to which y belongs.

4. Forwards the query to the node responsible for the interval to which y belongs. More precisely, node n forwards the query to one of the nodes:
   $n \oplus_N \frac{N}{k^{i+1}} j$, $0 \leq j \leq k-1$.

Figure 3 illustrates the behavior of this algorithm in the case of $k = 4$ in a 16 node system.

4.1 Complexity

Given that for each query, the general algorithm presented in the above section, follows a path of a k-ary search tree rooted at the node where the query originated, the following results follow.

Theorem 4.1 (Lookup length) *The maximum number of hops for any query to be resolved, is $log_k(N)$.*

Proof : Follows from the fact that the height of a k-ary tree of N nodes is $log_k(N)$. ∎

Theorem 4.2 (Routing Table Entries) *The maximum number of routing table entries at each node is (k-1)$log_k(N)$.*

Proof : Let n be an arbitrary node. From the above algorithm, node n must be able to forward any x-hop query, $0 \leq x \leq H-1$, where H is the maximum number of hops required to resolve any query.

In order for the node n to route an x-hop query, the node n must select *exactly one* destination between k possible forwarding alternatives. One of these destinations is the node n itself and as the node n does not need an entry for routing to itself, only $k-1$ entries are needed.

Overall, since x varies from 0 to $H-1$, and $k-1$ entries are needed for each x-hop query. Therefore, $(k-1)H$ entries in the routing table are needed.

Since n is an arbitrary node, the theorem follows. ∎

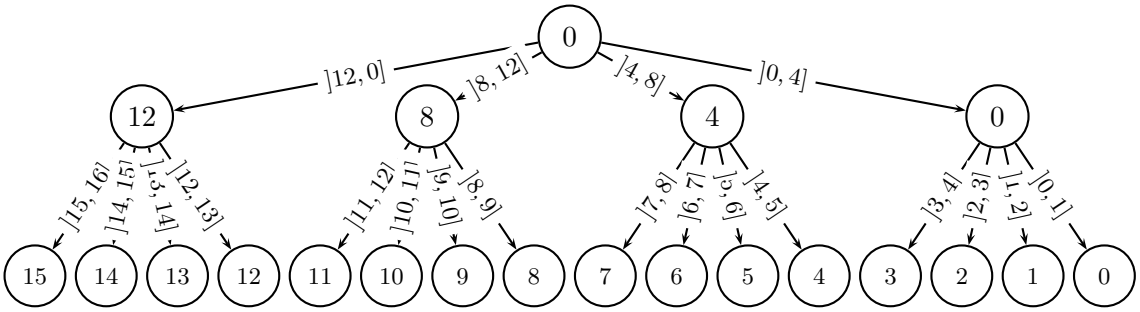

Figure 3: Decision tree for a query originating at node 0 in a 16-node network applying 4-ary search.

5 k-ary search for improving Chord

Having perceived chord as a special case of k-ary search, where $k = 2$, we can observe that if we need to improve the lookup length of Chord to a desired value H, we can choose a suitable k to achieve that value based on the following formula:

$$H = log_k(N)$$

The number of routing table entries, R will be:

$$R = (k-1)log_k(N)$$

We refer to our generalization of Chord by k-ary Chord.

The authors of the Chord system suggested as a future work in [7], a modification of the Chord lookup algorithm if a certain number of hops was desired. The modification suggested the placement of the fingers at intervals that are integer powers of $(1 + \frac{1}{d})$ instead of powers of 2, for some constant d. In that case, the lookup length is $log_{1+d}(N)$. The cost of the modification is an increase in the number of routing table entries to $\frac{log(N)}{log(1+\frac{1}{d})}$. We refer to this generalization by Chord(d).

In order to compare our result with the suggested generalization of the Chord authors, we take $x = 1 + d$ and we obtain table 2.

If we let $k = x = 4$, we can see that the number of routing table entries of the k-ary Chord is 38% smaller than Chord(d) as shown in table 3.

A more elaborate analysis of the size of the routing table as a function of the system size is shown in Figure 4.

	Chord(d)	k-ary Chord
H	$log_x(N)$	$log_k(N)$
R	$\frac{log(N)}{log(\frac{x}{x-1})}$	$(k-1)log_k(N)$

Table 2: Chord(d) vs. k-ary Chord

N	$R_{\text{Chord}(d)}$	$R_{k-\text{ary Chord}}$
2^4	9.637683359	6
2^8	19.27536672	12
2^{16}	38.55073343	24
2^{32}	77.10146687	48
2^{64}	154.2029337	96

Table 3: Number of routing entries for different system sizes with $k = x = 4$

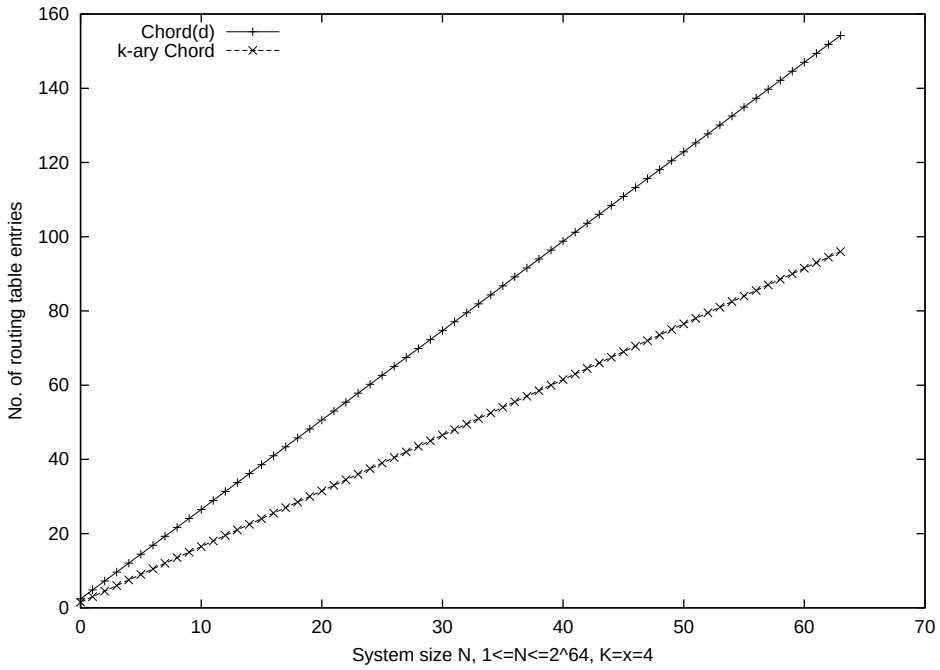

Figure 4: Evolution of routing table entries as a function of the system size.

6 Conclusion and future work

In this paper, we have presented a simple framework for designing distributed hash table based lookup services. The proposed framework is simple in that it is based on a well-known technique, that of k-ary search.

The paper shows how the idea of k-ary search can be used to derive the Chord lookup algorithm. More importantly, the generalization of the Chord lookup algorithm based on the k-ary search requires, for the same system size and the same lookup length, a routing table which is 38% smaller than the one required in the generalization suggested by the Chord authors.

As future work, we plan to show how this framework can be used to instantiate other distributed hash table based lookup algorithms. In addition, we show in a future paper how our framework simplifies the handling of node joins and failures.

References

[1] FreeNet, *http://freenet.sourceforge.net*.

[2] Gnutella, *http://www.gnutella.com*.

[3] Napster, *http://freenet.sourceforge.net*.

[4] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *A scalable content addressable network*, Tech. Report TR-00-010, Berkeley, CA, 2000.

[5] M. Ripeanu, I. Foster, and A. Iamnitchi, *Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design*, 2002.

[6] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, ACM SIGCOMM 2001 (San Deigo, CA), August 2001, pp. 149–160.

[7] ———, *Chord: A scalable peer-to-peer lookup service for internet applications*, Tech. Report TR-819, MIT, January 2002.

[8] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph., *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*, U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.