S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord *

Valentin A. Mesaros
Dep. of Computer Science
Univ. catholique de Louvain
Louvain-la-Neuve, Belgium
valentin@info.ucl.ac.be

Bruno Carton
Distributed Systems
CETIC
Gosselies, Belgium
bc@cetic.be

Peter Van Roy
Dep. of Computer Science
Univ. catholique de Louvain
Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be

Abstract

Chord is one of the simplest peer-to-peer systems that addresses the issue of efficient data location. Despite its simplicity, one of its main limitations remains the asymmetric organization of its routing. This leads to problems like inability to make in-place notifications of routing entry changes, and incapacity to support symmetric applications and to efficiently exploit network proximity. As a solution to this limitation, we propose S-Chord, an extension to Chord. In S-Chord the routing is organized in a symmetric manner, and the circular search space can be walked through bidirectionally. This results, for the worst-case, in an improvement of lookup efficiency of 25%, compared to Chord with the same size routing table. Furthermore, on average, assuming a uniform distribution of queries, S-Chord results in a 10% improvement. To test our theoretical results we implemented the S-Chord lookup algorithm and applied it to different networks.

Keywords: peer-to-peer and distributed system, Chord, S-Chord, routing symmetry, lookup efficiency

1. Introduction

With the advent of popular applications like Gnutella and Napster, it was observed that the content location and routing through the peer-to-peer (p2p) system can lead to serious scalability problems that had to be addressed. So they have been; examples of "well" structured systems providing such solutions are: CAN [4], Chord [6], Pastry [5], and Tapestry [7]. They all increase scalability and improve routing through the system by employing a distributed hash table (DHT).

Chord is one of the simplest p2p system employing a straightforward routing algorithm. Despite its simplicity, Chord is limited by its asymmetric organization of the routing. This results in three main drawbacks. First, unlike other p2p systems, in Chord the lookup is asymmetric, making very likely that the lookups from a node n to another node p take a different number of hops than the lookups from p towards n. Second, the lookup failure rate is quite high during node departures. As shown in [2], the asymmetric routing entries of Chord result in inability to perform in-place notification of routing entry changes. Third, as discussed in [1], in Chord the underlying network proximity is both awkward and costly to exploit.

To overcome these drawbacks, we propose S-Chord, an extension to the Chord system, providing a symmetric p2p lookup protocol. As will be shown in Section 3.1, the symmetry in S-Chord is threefold: "routing entry symmetry", "routing cost symmetry", and "finger table symmetry". Related to our work is the research done in Hyperchord [2], where a certain degree of symmetry is introduced in order to improve the node join and leave mechanism. Their solution is based on hypercube routing providing routing entry symmetry and routing cost symmetry. S-Chord has the same symmetric properties as Hyperchord. In addition, in S-Chord the fingers are organized symmetrically, and the way the routing is managed results in an improvement of the lookup efficiency; i.e., with a routing table of the same size as Chord (and Hyperchord), S-Chord resolves keys in up to 25% less hops.

We begin our presentation by an overview of the Chord system. We continue with the introduction of S-Chord by defining its symmetry, explaining the mechanism for constructing its finger table, and presenting the lookup protocol. We conclude our work after presenting some simulation results.

*This work was partially funded at UCL by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under IST-2001-33234 PEPITO, and at CETIC (http://www.cetic.be) by the Walloon Region (DGTRE) and the European Union (ERDF and ESF).

2. Chord overview

In order to describe the way the routing table is formed in S-Chord, we first recall how Chord is organized. We describe the finger table and the node join/leave mechanism.

2.1. Finger table in Chord

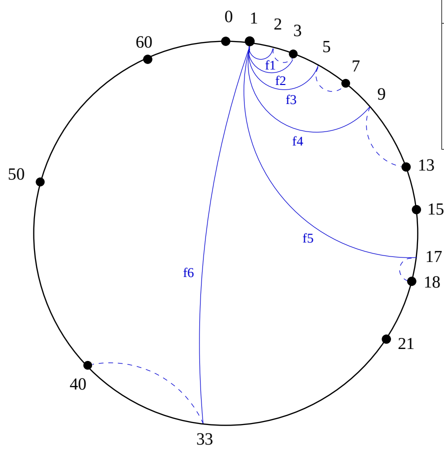
In Chord, the search space is organized as a virtual ring within which hashed node and data item key *identifiers* are spread by using a consistent hashing. For a search space of size $N = 2^k$ the identifiers can be situated on a circle of numbers ranging from 0 to $2^k - 1$. A base hash function is used to assign each node and data item key a k-bit identifier (id). We will use the term "node" to refer to both the node and its identifier under the hash function, as the meaning will be clear from the context.

Each node has a *predecessor* and a *successor* representing references to the previous and, respectively, the subsequent node in the search space. A key is stored at the node succeeding the id of that key on the circular search space. Thus, the naive lookup procedure for a certain key reduces to looking for the first node whose id is greater than, or equal to the id of that key along the search space, going clockwise.

To speed up the lookup process, each node maintains supplementary references (called fingers) about some other nodes inside a *finger table*. Given a search space of size $N = 2^k$, besides the references to its predecessor and successor, each node in the Chord system stores k fingers. There is a distinction between *finger_start* and *finger_node*. The finger_start represents the value a finger should have, whereas the corresponding finger_node represents the value the finger actually has.

We denote the i^{th} finger_start by $\widehat{f}[i]$ and the i^{th} finger_node by $f[i]$. Now we can define the i^{th} finger_start at node n in Chord as $n.\widehat{f}[i] := n \oplus 2^{i-1}$ (if not stated otherwise, the modulo arithmetics are positive and computed with respect to the search space size N). Further, the i^{th} finger_node at node n is the first node succeeding n by at least 2^{i-1} going clockwise. That is, $n.f[i] := successor(n \oplus 2^{i-1})$, where $successor(u)$ is a function that returns the first present node that follows u along the circular search space. At lookup, a node forwards the query to the closest finger to that key, making the distance to the node storing the key be at least halved at each hop. Thus, Chord guarantees key resolution in a maximum $log_2 N$ hops.

In Figure 1 we illustrate an example finger table in a Chord system with 11 nodes chosen from a search space of size $N = 64$. We want to determine the routing information that node 1 stores. The first finger_node points to 3, as node 3 is the first node that succeeds finger_start $1 \oplus 2^0 = 2$. The second finger_node also points to 3, as node 3 is the first node that, going clockwise, succeeds node 1 with at least $2^1 = 2$. The remaining finger_nodes are 7, 13, 18, and respectively 40. Note that, at a node, it is the finger_nodes that actually constitute the finger table.



finger no.	$\widehat{f}$	f
1	2	3
2	3	3
3	5	7
4	9	13
5	17	18
6	33	40

Figure 1: The fingers at node 1 in a poorly populated Chord network of size 64.

2.2. Node join and leave in Chord

When joining the system, a node n has to determine its successor and predecessor, and to populate its fingers. In order to guarantee a successful join, each node runs periodically a so called "stabilization protocol". The departure of a node is treated in Chord in the same way as a node failure. By periodically running the stabilization protocol, and looking up the fingers at each node, the system correctness is ensured. Note that node departures in Chord can lead to temporary finger table inconsistency (i.e., finger references to dead nodes) allowing lookup failures to happen.

3. The base S-Chord protocol

The S-Chord lookup protocol is based on the Chord protocol. It mainly differs from Chord by the symmetric organization of its routing table, and its routing policy. In this section we present the symmetry of S-Chord and show how its symmetric organization of the routing improves the lookup efficiency.

3.1. Symmetry in S-Chord

In S-Chord the symmetry is threefold. We have "routing entry symmetry", "routing cost symmetry", and "finger table symmetry".

Routing entry symmetry is that for any two different nodes, n and p, if p has a finger to n, then n has a finger to p. This symmetry provides a node with the ability to announce its arrival and departure to the interested nodes; i.e., the nodes that should refer to it. In a poorly populated network the routing entry symmetry is not achieved per se. However, as described in [3], when doing in-place notifications this problem can be taken into account and solved. Furthermore, note that unlike Chord where the virtual ring can be walked through in only one direction (i.e., clockwise), the routing entry symmetry of S-Chord provides the ability to walk through the virtual ring in both directions.

The routing entry symmetry and the associated lookup protocol provide the S-Chord system with another characteristic that we call: the "routing cost symmetry". That is, it is very likely that the lookup path lengths between any two nodes in the system are equal (see Figure 6), thus supporting symmetric applications. Nevertheless, the two paths may differ; i.e., we don't support "routing symmetry"[1].

In S-Chord, the routing entries in the finger table of any node n are organized symmetrically with respect to the axis between n and $n \oplus \frac{N}{2}$ (i.e., half the search space of node n). This symmetry that we call "finger table symmetry" provides a fast access to the whole search space.

3.2. Finger table in S-Chord

As described in Section 2, in Chord, for a given search space of size N, the size of the finger table at each node is $\lceil log_2 N \rceil$. In S-Chord we keep the same size of the finger table as in Chord.

To support symmetry, we organize the finger table in two symmetric sides. Thus, each node maintains a finger table with at most $2 * m$ entries, where $m = \lceil log_4 N \rceil$ (hereinafter we use m to denote $\lceil log_4 N \rceil$, with N representing the search space size). We refer to the set of the fingers in the interval $[1, m]$ as the right side of the finger table. Similarly, we refer to the the set of fingers in the interval $[m+1, 2m]$ as the left side of the finger table. Equation 1 defines the finger_starts at node n in S-Chord, for both sides of the finger table. They are located at positive and negative distances of powers of four from n in both directions.

$$n.\widehat{f}[i] := \begin{cases} n \oplus 4^{i-1} & i \in [1, m] \\ n \ominus 4^{2m-i} & i \in [m+1, 2m] \end{cases} \quad (1)$$

Equation 2 defines the finger_node at node n. For i found in the right side of the finger table, the i^{th} finger_node at node n will contain the *id* of the first node succeeding n by at least 4^{i-1} going clockwise (i.e., *successor*$^+$). For i found in the left side, the i^{th} finger_node at node n will contain the *id* of the first node succeeding n by at least 4^{2m-i} going counterclockwise (i.e., *successor*$^-$). Note that $n.f[1]$ is the same as the successor of n, whereas $n.f[2m]$ is the same as the predecessor of n.

$$n.f[i] := \begin{cases} successor^+(n \oplus 4^{i-1}) & i \in [1, m] \\ successor^-(n \ominus 4^{2m-i}) & i \in [m+1, 2m] \end{cases} \quad (2)$$

In Figure 2 we illustrate an example of an S-Chord system with 11 nodes chosen from a search space of size $N = 64$ (i.e., $m = 3$). We want to determine the routing information that node 1 stores. The first finger_node points to 3, as node 3 is the first node that succeeds finger_start $1 \oplus 4^0 = 2$. Furthermore, the second and the third finger_nodes are 7 and 18, respectively. The fourth finger_node points to 40, as node 40 is the first node that, going counterclockwise, succeeds finger_start $1 \ominus 4^2 = 49$. The fifth and the sixth finger_nodes point to 60 and 0, respectively.

Two main reasons motivated us to choose the fingers in the left side of the finger table using the operation *successor*$^-$ rather than the operation *successor*$^+$.

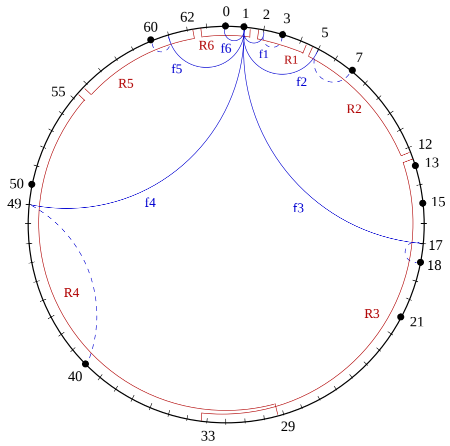


Figure 2: The fingers and their responsibilities at node 1 in a poorly populated S-Chord network of size 64.

[1] We use the term "routing symmetry" as defined in the networking literature, meaning that the paths (in both directions) between two nodes in the network are exactly the same.

First, since a finger is not situated in the middle of the search space partition it is responsible for (as will be described Section 3.3), it is better to locate the finger going along the longer branch of its responsibility instead of going along the smaller one. This is because, if the finger_start is not present, it is more likely that the corresponding finger_node be well positioned to address the corresponding responsibility. Second, by using the operation *successor*$^{-}$, the routing entry symmetry is better supported. That is, if the finger i of a node p points to a node n, then the finger $2m-(i-1)$ of node n will point to node p, or very close to it.

As in Chord, at any node there may be situations where the i^{th} finger gets close, and sometimes even equal to the $i+1^{th}$ finger. For instance, such a scenario would appear at node 1 in Figure 2 if node 3 were not present; thus the first finger at node 1 would be 7 instead of 3 (i.e., $f[1]=f[2]$).

Since the finger_nodes of a node are chosen with respect to Equation 2, there are chances that fingers from the right side and the left side of the finger table of a node overlap. This situation happens for any two fingers $i \in [1,m]$ and $j \in [m+1,2m]$ of the same node with the condition that there are no nodes in the interval $\left[\widehat{f}[i] \rightarrow \widehat{f}[j]\right]$. An example of two fingers from different sides of the finger table getting close to each other would appear in the system in Figure 2 if node 40 were not present; thus the fourth finger at node 1 would be 21 instead of 40.

Nevertheless, as will be described in the next section, by well choosing the finger responsibilities of a node, the finger overlapping does not affect the lookup efficiency or the lookup algorithm.

3.3. Finger responsibility in S-Chord

Each finger of a node n has a well determined responsibility. The node responsibility has the form of an interval and defines the range of keys expected to be found in a minimum number of hops via that finger, going from node n.

Since the search space at node n is split among its fingers, the finger responsibilities are used when routing. Thus, the request for a key k is sent to the finger whose responsibility includes k.

Whereas in Chord a finger is situated at the beginning of the search space partition it is responsible for, in S-Chord, in order to support symmetry, a finger is located inside it.

Equations 3 and 4 define the responsibility intervals at node n, $R_n(i)$ and $R_n(j)$, of fingers $i \in [1,m]$ and $j \in [m+1,2m]$, respectively. For simplicity and clarity, we consider that $n.f[0]=n.f[2m+1]=n$ for any node $n \in]0,N[$. Similarly, for $n=0$ we consider $n.f[0]=0$ and $n.f[2m+1]=N$.

$$\left]n.f[i-1] \oplus \left\lfloor \frac{n.\widehat{f}[i] \ominus n.f[i-1]}{2} \right\rfloor \rightarrow n.f[i] \oplus \left\lfloor \frac{n.\widehat{f}[i+1] \ominus n.f[i]}{2} \right\rfloor\right] \tag{3}$$

$$\left]n.f[j] \ominus \left\lfloor \frac{n.f[j] \ominus n.\widehat{f}[j-1]}{2} \right\rfloor \rightarrow n.f[j+1] \ominus \left\lfloor \frac{n.f[j+1] \ominus n.\widehat{f}[j]}{2} \right\rfloor\right] \tag{4}$$

Note that for computing the lower and the upper bounds of the responsibility interval we considered the floor of the ratios. The reason is that this results in a smaller number of hops. Indeed, the number of hops to reach the item found at equal distance between two successive fingers i and $i+1$ of the same node will be lesser if finger i is chosen, instead of finger $i+1$, since via finger $i+1$ the query will do an additional hop.

Here is an example of setting the finger responsibilities of node 1 in the network shown in Figure 2. One can see that, for instance, the finger responsibility for fingers 1, 2, and 4 are $R_1(1) =]2 \rightarrow 4]$, $R_1(2) =]4 \rightarrow 12]$, and $R_1(4) =]29 \rightarrow 55]$, respectively.

The finger responsibility as defined in Equations 3 and 4 can be applied correctly only to monotonically increasing values of the fingers modulo the network size. Since there are chances that fingers of the same node overlap, the fingers of any node have to be ordered before computing their responsibilities. Thus, at a node n, the fingers have to be ordered by the distance between themselves and the node n, going clockwise. Once the fingers ordered, changing the value of a finger i at a node n will only engage the change of its responsibility and those of the neighboring fingers $i-1$ and $i+1$ of node n. Furthermore, since the finger responsibility is computed with respect to the finger_nodes, finger overlapping does not affect the lookup efficiency, considering that the fingers of a node are ordered before computing their responsibilities.

It is interesting to mention that the Equations 3 and 4 represent a refined definition of the finger responsibility described in [3]. The latter does not ensure good lookup performances for particular scenarios of poorly populated networks. Indeed by taking into account the fact that there is no node between a finger_node and its finger_start, Equations 3 and 4 define a smarter responsibility interval.

3.4. Lookup in S-Chord

In the S-Chord system, a key is stored at the first node equal to, or greater than the *id* of that key on the circular search space. Thus, like in Chord, the lookup

```
n.find_successor+(k)
  if k ∈]n, successor] then
    return successor;
  elseif k ∈]predecessor, n] then
    return n;
  else
    n' = closest_node(k);
    return n'.find_successor+(k);
  fi
```

```
n.closest_node(k)
  for i = 1 upto 2m
    if k ∈ R_n(i) then
      return n.f[i];
    fi
  return n;
```

Figure 3: Key lookup using the finger table and finger responsibilities in S-Chord.

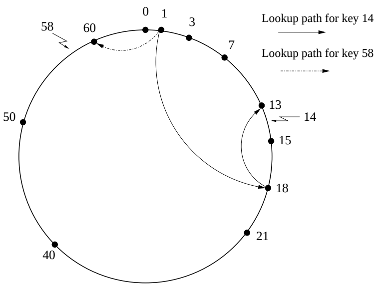


Figure 4: Queries for keys 14 and 58, starting at node 1, in a poorly populated S-Chord network of size 64.

for a certain key reduces to looking for the first node whose *id* is greater than, or equal to the *id* of that key.

In Figure 3, the pseudo-code [2] for the operations *find_successor*$^+$ and the *closest_node* are presented. The operation *find_successor*$^+$ is executed at node n to look for the successor of k in the circular search space going clockwise. Firstly, it is checked whether the key falls in the range between n and its successor, or its predecessor. In both cases the direct responsible node is returned. Otherwise, if the key is found farther in the ring, look for the closest node to k (i.e., n') and forward the request to it.

Given a key k at node n, the operation *closest_node* returns the closest node to k. At node n, when looking for the closest node to a certain key k, the responsibility of the fingers of n is checked. Hence, the closest node known by n is to be the node referred by the finger of n whose responsibility includes k.

In [3] we prove the correctness of the lookup algorithm of S-Chord, and that the maximum number of hops necessary to reach the responsible node of a given key is $\lceil \frac{3}{4} log_2 N \rceil$. That is 25% smaller than $log_2 N$ in Chord and Hyperchord). The intuition behind is that at each suite of three hops the distance to the node storing the key is divided by 16, instead of 8 in Chord.

Two examples of lookup paths starting node 1, for keys 14 and 58, are illustrated in Figure 4. First, consider the lookup for key 14. Since, at node 1, 14 is included in the responsibility of $f[3]$, the request is forwarded to node 18. From node 18, the request is forwarded to node 15, since 14 is included in the responsibility of $f[5]$ of node 18. Note that the query was forwarded counterclockwise in the circular search space. Finally, node 13 finds out that its successor, node 15, is directly responsible for key 14, and thus returns 15 to node 1. Consider now the lookup for key 58. Since, at node 1, 58 is included in the responsibility of $f[5]$, the request is forwarded to node 60. Node 60 finds out that it is directly responsible of key 58, and thus returns 60 to node 1.

[2]The remote calls and variables are preceded by the remote node *id*, while the local procedure calls and variables omit it.

3.5. Node join and leave in S-Chord

As in Hyperchord [2], in S-Chord, due to the routing entry symmetry, we can introduce in-place notification of routing entry changes. That is, a node joining the system is able to announce its arrival to nodes in the system interested in pointing to it by their fingers. Similarly, a node leaving the system is able to announce its departure to nodes in the system pointing to it, thus dramatically reducing the time period during which a finger table remains inconsistent. In [3] we provide the algorithms for node join and leave.

4. Simulation results

We implemented the Chord and S-Chord lookup algorithms, and simulated lookup scenarios in different networks. For our simulations we considered the query distribution to be uniform over the search space.

For the first test suite we focused on the lookup path length. We measured the maximum and the average path length for both systems, for fully populated networks of sizes ranging from 2^0 to 2^{16}. The measurements (see Figure 5) confirmed our expectations. That is, in the worst case, the number of hops a lookup can take in S-Chord is 25% less than in Chord. We observed that on average lookups take around 10% less hops in S-Chord than they do in Chord.

The second test suite analyses the routing cost symmetry for both systems. We measured the percentage $P(x)$ of any pair of two nodes n and p such that the absolute difference between the distance (n, p), in num-

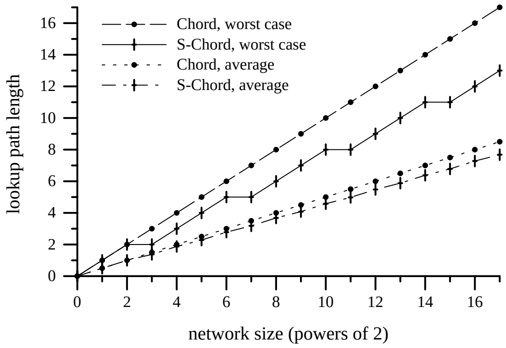


Figure 5: Worst case and average path length function the network size for Chord and S-Chord fully populated networks.

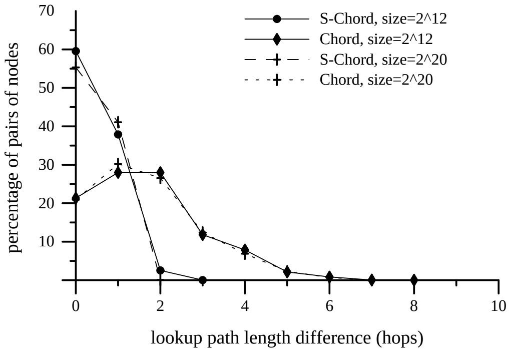


Figure 6: Distance variation between pairs of nodes in Chord and S-Chord, for two poorly populated networks (sizes 2^{12}, 2^{20}).

ber of hops, and the distance (p, n), equals x:

$$|path\_len(n \rightarrow p) - path\_len(p \rightarrow n)| = x \ .$$

For this test, we considered two networks (of sizes 2^{12}, and 2^{20}) partially populated (1024 nodes randomly chosen). As illustrated in Figure 6, in S-Chord, for 60% of pairs the difference was 0, and for 90% it was less than or equal to 1. In contrast, in Chord there were only 20% of pairs with difference 0, and 50% with difference less than or equal to 1. This shows us that, whereas in Chord we have pairs characterized by a strong asymmetry of the routing cost, in S-Chord the routing cost is highly symmetric.

5. Conclusions and further work

In this paper we have introduced S-Chord, with its threefold symmetry, as a candidate solution to the asymmetry drawbacks of Chord. S-Chord is based on Chord and provides the same correctness guarantees. In addition, for steady scenarios (low rate of nodes joining/leaving) it improves lookup efficiency up to 25%. Moreover, since S-Chord is also using notifications for the node leave procedure, we are confident that for dynamic scenarios the average hop length in S-Chord is lower than the one in Chord and comparable with the one in Hyperchord.

We have promising ongoing research focused on the generalization of S-Chord. We also investigate what we believe to be an open question; i.e., achieve "routing symmetry" through proximity neighbor selection, and exploit the underlying network proximity by using the system's symmetry.

Acknowledgments

We are grateful to Kevin Glynn for helping us with the formalization of the finger responsibility. We also thank our colleagues Raphaël Collet and Luc Onana for their constructive comments.

References

[1] M. Castro, Y. Hu P. Druschel, and A. Rowstron. Exploiting network proximity in distributed hash tables. In *Proc. of FuDiCo*, June 2002.

[2] K. Lakshminarayanan, A. Rao, S. Surana, R. Karp, and I. Stoica. Hyperchord: A peer-to-peer data location architecture. Technical Report CS-021208, U.C. Berkeley, December 2001.

[3] V. Mesaros, B. Carton, and P. Van Roy. S-Chord: Using symmetry to improve lookup efficiency in Chord. Technical Report 2002-08, UC-Louvain, December 2002.

[4] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, August 2001.

[5] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of the 18th International Conference on Distributed Systems Platforms*, November 2001.

[6] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. of the 2001 ACM SIGCOMM*, August 2001.

[7] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-011141, U.C. Berkeley, April 2001.