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


Valentin A. Mesaros
Univ. catholique de Louvain
2 place Sainte Barbe
B-1348 Louvain-la-Neuve
*valentin@info.ucl.ac.be*

Bruno Carton
CETIC
8 rue Clément Ader
B-6041 Gosselies
*bc@cetic.be*

Peter Van Roy
Univ. catholique de Louvain
2 place Sainte Barbe
B-1348 Louvain-la-Neuve
*pvr@info.ucl.ac.be*



**Report No. UCL/INFO-2002-08**
December, 2002



**Abstract**

In recent years, efficient data location in peer-to-peer systems has become the subject of many research theses. Chord is one of the simplest peer-to-peer systems that addresses this issue. 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. We prove that the way S-Chord organizes its routing 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 networks of different sizes.


## 1 Introduction

Recently, researchers from various fields (e.g., distributed computing, networking) have given paramount attention to peer-to-peer (p2p) systems [6], considered as possible solution treasure to problems like scalability, fault tolerance, availability, and load balancing.


*This research was funded at UCL by the PEPITO project within the fifth framework programme of the European Union, and at CETIC (`http://www.cetic.be`) by the ORAGE project.

With the advent of popular applications like Gnutella [1] and Napster [2], it was observed that the content location and routing through the p*2*p 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 [9], Chord [12], Tapestry [13], and Pastry [11]. They all increase scalability and improve routing through the system by employing a distributed hash table (DHT), where keys are associated with data objects, and peers are responsible for storing a certain range of keys.

Chord is one of the simplest p*2*p system, employing a straightforward routing algorithm, and its correctness can be easily proved. The algorithm used for routing through the system is based on binary search. Chord structures its search space of size $N = 2^k$ as a virtual ring where each participating node maintains routing information (called *fingers*) about $k$ other nodes. The system uses a scalable lookup protocol to guarantee a bounded number of $log_2 N$ hops.

Despite its simplicity, Chord is limited by its asymmetric organization of the routing. This results in three main drawbacks. First, unlike other p*2*p systems, in Chord the lookup is asymmetric. That is, the circular search space is walked through only clockwise, making it 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 [7], the asymmetric routing entries of Chord result in inability to perform in-place notification of routing entry changes. Third, in Chord the underlying network proximity is both awkward and costly to exploit. Indeed, as discussed in [4], in Chord "proximity routing" is quite difficult and not very effective, and "proximity neighbor selection" remains an open question.

To overcome these drawbacks, we propose S-Chord, an extension to the Chord system, providing a symmetric p*2*p lookup protocol. As will be shown in Section 3.1, the symmetry in S-Chord is threefold; we have "routing entry symmetry", "routing cost symmetry", and "finger table symmetry". Related to our work is the research done in Hyperchord [7], where the authors focus on introducing a certain degree of symmetry in order to improve the node join and leave mechanism. Their solution is based on hypercube routing, where a node has as fingers other nodes found at Hamming distance 1 from itself. Hyperchord provides routing entry symmetry and routing cost symmetry.

S-Chord has the same symmetric properties as Hyperchord. In addition, S-Chord organizes the fingers symmetrically, and the way the routing is managed results in an improvement of the lookup efficiency. That is, with a routing table of the same size as Chord[1] (and Hyperchord), S-Chord resolves keys in up to 25% less hops.

We begin our presentation by a brief overview of the Chord system. We continue with introducing S-Chord by defining its symmetry and explaining the mechanism for constructing its finger table. Then, we present the lookup protocol together with its correctness proof, and the join and leave protocol. We conclude with some simulation results, and discussions on possible extensions

[1]If not stated otherwise, by Chord we mean the basic Chord [12] system.

and improvements in S-Chord.

# 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 join/departure mechanism.

## 2.1 Finger table in Chord

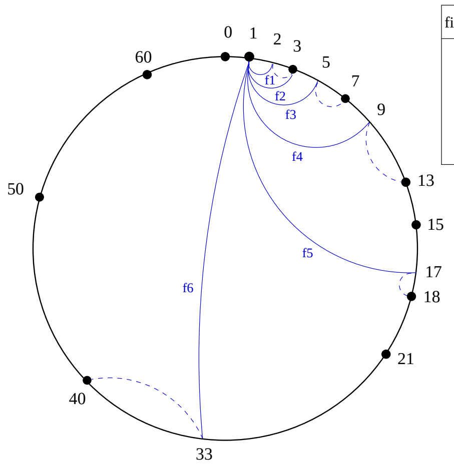
In Chord, the search space is organized as a virtual ring within which hashed node and data item key *identifiers* are spread by using a consistent hashing. For a search space of size $N = 2^k$ the identifiers can be situated on a circle of numbers from 0 to $2^k - 1$. A base hash function is used to 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 circular search space. In the system 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 fingers about some other nodes inside a *finger table*. At a node $n$, a finger represents an additional reference that $n$ has to a node in the network. 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. Indeed, if the node whose $id$ is equal with the value of the finger_start is not present in the system, the first present succeeding node along the virtual ring is the finger. This node is called the finger_node.

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.

The supplementary routing information of $k = \lceil log_2 N \rceil$ fingers allows Chord to guarantee key resolution in a maximum $log_2 N$ hops. At lookup, a node will forward 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.




| 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.

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 constitutes the finger table.

## 2.2 Arrival and departure of nodes in Chord

In order to guarantee the correctness of the lookup protocol in Chord, it must be ensured that the successor pointer at each node is up to date. Thus, the node join operation is very important in Chord.

Depending on its $id$, each node has a well determined place in the circular search space. When joining the system, a node $n$ has to determine its successor and predecessor, and to populate its fingers. To this end, it starts by looking for its successor from which it determines the predecessor. Then, via its successor it looks for its fingers. Furthermore, it notifies its successor about its presence. It must be mentioned that in order to guarantee a successful join, each node runs periodically a so called "stabilization protocol". The stabilization protocol is to guarantee that concurrent node arrivals preserve reachability of the existing nodes. It is when running the stabilization protocol that the predecessor of $n$ notices the presence of $n$ in the system, and thus completes the join procedure.

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 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 our *view* of symmetry over Chord and show how the symmetric organization of the routing in S-Chord 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. As will be described in Section 4, in a poorly populated network the routing entry symmetry is not achieved per se. However, when doing the in-place notifications this problem is taken into account and solved (see Figure 7). 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: the routing cost symmetry. That is, it is very likely that the lookup path lengths between any two nodes in the system are equal, thus supporting symmetric applications. Nevertheless, the two paths may differ; i.e., we don't support "routing symmetry"[2].

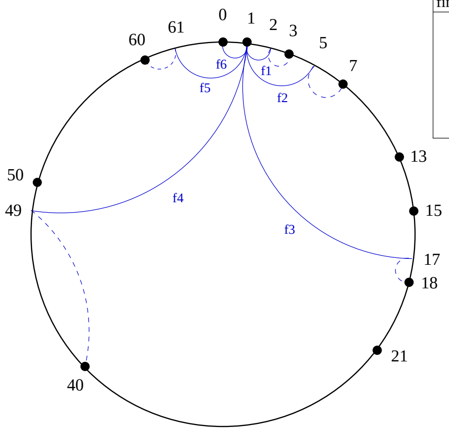
In S-Chord, the routing entries in the finger table of any node $n$ are organized symmetrically with respect to the axis between $n$ and $n \oplus \frac{N}{2}$ (i.e., half the search space of node $n$). This finger table symmetry provides a fast access to the whole search space.

## 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. The reason for this is to make S-Chord comparable in performance with Chord.

To support symmetric, we organize the finger table in two symmetric sides. Thus, each node maintains a finger table with at most $2 * m$ entries, where

[2]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.




| finger no. | $\widehat{f}$ | f |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 5 | 7 |
| 3 | 17 | 18 |
| 4 | 49 | 40 |
| 5 | 61 | 60 |
| 6 | 0 | 0 |

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

$m = \lceil log_4 N \rceil$ (hereinafter we use $m$ to denote $\lceil log_4 N \rceil$, where $N$ represents 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.

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

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., $successor^-$). We note that $n.f[1]$ is the same as the successor of $n$, and $n.f[2m]$ is the same as the predecessor of $n$.

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

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.

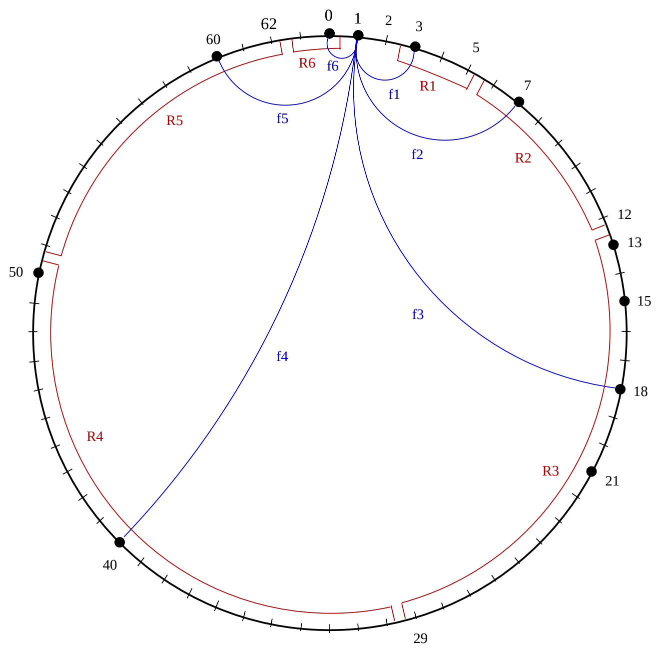



Figure 3: The fingers and their responsibilities at node 1 in a poorly populated S-Chord network of size 64.

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 $successor^+$. 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., $f[1] = f[2]$).

Moreover, in S-Chord, there may be situations where fingers $i \in [1, m]$ and $j \in [m+1, 2m]$ of the same node get close to each other, or even overlap. An example of two fingers from different sides of the finger table getting close to each other would appear in Figure 2 if node 40 were not present; thus the fourth finger at node 1 would be 21 instead of 40.

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 $\left[\widehat{f}[i] \rightarrow \widehat{f}[j]\right]$. Nevertheless, as will be described in the next section, by well choosing the responsibilities of the fingers 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 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 (i.e., determine which finger to forward the request to). Thus, the request for a key $k$ is sent to the finger whose responsibility includes $k$. As will be described in Section 3.4.1, when resolving a key at node $n$ the responsibility of its fingers is considered only if the key does not fall in the interval ranging between the predecessor and the successor of $n$. Indeed, for a key falling in the interval between node $n$ and its predecessor, the direct responsible for that key is $n$. On the other hand, if the key is found between node $n$ and its successor, the direct responsible for the key is the successor of $n$.

Whereas in Chord a finger is situated at the beginning of the search space partition it is responsible for, in S-Chord a finger is located inside it. Indeed, the responsibility of the $i^{th}$ finger of a node starts from the half way point between it and the $i-1^{th}$ finger, and ends at the half way point between it and the $i+1^{th}$ finger.

Equation 3 defines the finger responsibilities at node $n$. For simplicity and clarity, we consider that $n.f[0] = n.f[2m+1] = n$ for any node $0 < n < N$, whereas for $n = 0$ we consider $n.f[0] = 0$ and $n.f[2m+1] = N$.

$$R_n(i) := \left]n.f[i-1] \oplus \left\lfloor \frac{n.f[i] \ominus n.f[i-1]}{2} \right\rfloor \longrightarrow n.f[i] \oplus \left\lfloor \frac{n.f[i+1] \ominus n.f[i]}{2} \right\rfloor\right],\ i \in [1, 2m] \quad (3)$$

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 less 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 3. One can see that, for instance, the finger responsibility for fingers 1, 2, and 4 are $R_1(1) = ]2 \rightarrow 5]$, $R_1(2) = ]5 \rightarrow 12]$, and $R_1(4) = ]29 \rightarrow 50]$, respectively.

The finger responsibility as defined in Equation 3 can be applied correctly only to monotonically increasing values of the fingers modulo the network size.

```
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 4: Key lookup using the finger table and finger responsibilities in S-Chord.

Since, as described in Section 3.2, there are chances that fingers of the same node overlap, the fingers of a node have to be ordered before computing their responsibilities. At a node $n$, the fingers have to be ordered by the values corresponding to the distance between themselves and node $n$, going clockwise.

Note that 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 do not affect the lookup efficiency, considering that the fingers of a node are ordered before computing their responsibilities.

## 3.4 Lookup in S-Chord

In this section we describe the S-Chord lookup algorithm, and then we prove its correctness. We also show that the number of hops needed for resolving a key is bounded, and actually up to 25% smaller than in Chord.

### 3.4.1 The algorithm

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 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.

In Figure 4, the pseudo-code for the *find_successor*$^+$ and the *closest_node* operations 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. Note that the remote calls and variables are preceded by the remote node *id*, while the local procedure calls and variables omit the local node *id*.

In operation *find_successor*$^+$ we first check whether the key falls in the range between $n$ and its successor, or its predecessor. In both cases the direct responsible node is returned. That is, return the successor of $n$ in the first case, and $n$ itself, in the second case. Otherwise, if the key is found farther in the ring,

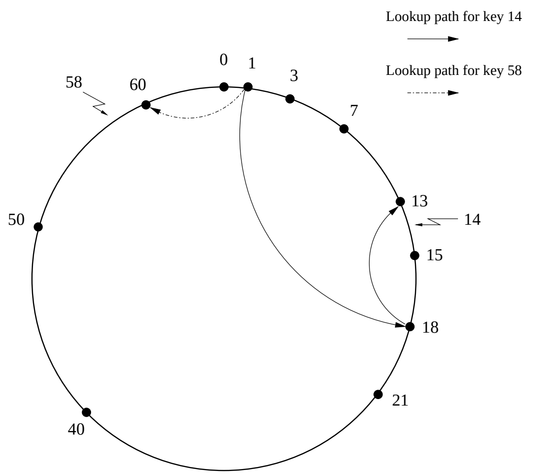



Figure 5: The path queries for keys 14 and 58, starting at node 1, in a poorly populated S-Chord network of size 64.

look for the closest node (known by $n$) to $k$, that is $n'$, and forward the request to it (i.e., apply *find_successor*$^+$ on $n'$).

Choosing the appropriate finger to forward the request to constitutes the core of the lookup algorithm. Given a key $k$ at node $n$, the operation *closest_node* in Figure 4 returns the closest node (known by $n$) to $k$. To this end, the finger responsibility described in Section 3.3 is employed.

As shown in Section 3.3, in S-Chord a finger is found inside the domain it is responsible for. This is the case since a node can route back and forth within the search space, depending on whether the *id* of the key is less, or greater than the node *id*, respectively.

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 $k$.

Two examples of lookup paths starting node 1, for keys 14 and 58, are illustrated in Figure 5. This is the same network as in Figure 1.

First, consider the lookup for key 14. Since, at node 1, 14 is included in the responsibility of finger 3 (i.e., $f[3]$), 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 will find out that its successor, node 15, is directly responsible for key 14, and thus return 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 will find out that it is directly responsible of key 58, and thus will return 60 to node 1.

### 3.4.2 The correctness

In this section we prove the correctness of the lookup algorithm in S-Chord, and that the maximum number of hops necessary to reach the responsible node of a given key is $\lceil \frac{3}{4} log_2 N \rceil$. We will prove that this holds under the following assumptions:

**(A1)** since the denser the network, the longer the lookup path length, we assume the network to be fully populated;

**(A2)** the network is stable (i.e., nodes do not join or leave);

**(A3)** the network size is $N = 4^{2*l}$.

Before proving the correctness of the algorithm, let us define some concepts. The lookup of a key $k$ at step $i$ consists in finding the closest finger, $n_k^{i+1}$, of the closest node, $n_k^i$ (determined at step $i-1$) to $k$, and in delegating the lookup to that finger. Thus, the node responsible for the lookup on key $k$ at step $i$ is $n_k^i$. We refer to the origin of the query on key $k$ by $n_k^0$, and without loss of generality we consider that for all $k$, $n_k^0 = 0$.

The smallest distance between a node $u$ and a node $v$ in a network of size $N$ is $\Delta(u,v) := Min(u \ominus v, v \ominus u)$. The set of keys at distance less than or equal to $d$ from a node $n$ is $K(n,d) = \{k \mid \Delta(n,k) \leq d\}$. At step $i$, the maximum distance between the closest node known and the desired key $k$ in a network of size $N$ is:

$$MaxDist(i,N) := Max\{\Delta(n_k^i, k) \mid k \in K(n_k^0, \frac{N}{2})\}. \tag{4}$$

Since the network is fully populated, for $1 \leq i \leq l$, there are two fingers at distance $4^{i-1}$ for each node; one on its left hand side and the other on its right hand side. Hence, the number of steps needed to perform a lookup on a key localized at distance $d$ from a node $n$ is equal to the number of steps needed to perform a lookup on a key localized at distance $d$ from any other node $u$ in the network. For instance, the lookup path length on key 36 started at node 30 is equivalent to the one on key 10 started at node 4. Hence, in a fully populated network, the number of steps of a lookup on key $k$ started at node 0 is equal to 1 + the number of steps of the lookup on a key localized at distance $\Delta(n_k^1, k)$ from node 0; $n_k^1$ being the finger to which 0 delegates the lookup on key $k$. By taking this property into account, we can redefine $MaxDist(i,N)$ as follows:

$$MaxDist(i,N) := \begin{cases} \frac{N}{2}, \ i = 0 \\ Max\{\Delta(n_k^1, k) \mid k \in K(n_k^0, MaxDist(i-1,N))\}, \ i > 0. \end{cases} \tag{5}$$

We know that it takes one hop to evolve from $MaxDist(i,N)$ to $MaxDist(i+1,N)$. Since the responsible of each key is reached when $MaxDist(j,N) = 0$, we observe that $j$ denotes the upper bound of the number of hops taken by a lookup for a given key. We compute $MaxDist(i,N)$ for different values of $i$:

after 0 hops : $MaxDist(0, N) = \frac{N}{2} = \frac{1}{2} * 4^{2*l}$
after 1 hops : $MaxDist(1, N) = \frac{1}{2} * 4^{2*l} - 4^{2*l-1} = 4^{2*l-1}$
after 2 hops : $MaxDist(2, N) = \frac{1}{2} * (4^{2*l-1} - 4^{2*l-2}) = \frac{3}{2} * 4^{2*l-2}$
after 3 hops : $MaxDist(3, N) = \frac{3}{2} * 4^{2*l-2} - 4^{2*l-2} = \frac{1}{2} * 4^{2*l-2}$
after 4 hops : $MaxDist(4, N) = \frac{1}{2} * 4^{2*l-2} - 4^{2*l-3} = 4^{2*l-3}$
...

We can observe that $MaxDist(3, N) = \frac{1}{2} * 4^{2*(l-1)}$, is equivalent to $MaxDist(0, 4^{2*(l-1)})$ for a fully populated network of size $\frac{N}{16}$. We can also observe that there are three steps between $MaxDist(0, N)$, and $MaxDist(3, N)$. Thus, by defining $MaxHops(N)$ to be the maximum number of hops needed to reach the responsible of any key in a network of size $N$, we can state that $MaxHops(4^0) = 0$, and:

$$MaxHops(4^{2*l}) = 3 + MaxHops(4^{2*(l-1)}). \tag{6}$$

By resolving the Equation 6 we have:

$$MaxHops(4^{2*l}) = 3 * l = \frac{3}{4} log_2 4^{2*l}. \tag{7}$$

Similarly, we can prove that for different values of the network size we have:

$$MaxHops(4^{2*l+\frac{1}{2}}) = \frac{3}{4} log_2 4^{2*l+\frac{1}{2}} + \frac{1}{4}, \tag{8}$$

$$MaxHops(4^{2*l+1}) = \frac{3}{4} log_2 4^{2*l+1} + \frac{1}{2}, \tag{9}$$

$$MaxHops(4^{2*l+\frac{3}{2}}) = \frac{3}{4} log_2 4^{2*l+\frac{3}{2}} - \frac{1}{4}. \tag{10}$$

One can note that equations 9 and 10 lead to the same number of hops, which means that doubling a network of size $4^{2*l+1}$ does not increase the maximum lookup path length.

Thus, we can conclude that in S-Chord, the maximum number of hops necessary to reach the direct responsible of a given key in a network of size $N$ is $\lceil \frac{3}{4} log_2 N \rceil$; that is 25% smaller than $log_2 N$ in Chord and Hyperchord.

# 4 Arrival and departure of nodes in S-Chord

The join operation is very important in a Chord based system; the correctness of the whole system depends on its success. As in Hyperchord [7], 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.

In Figure 6 we present the pseudo-code for node *join* and *leave*, together with two associated operations, namely *build_fingers* and *find_successor*$^-$. When joining the system, a node starts by populating its finger table, then it informs

```
n.join(n')
  successor = n'.find_successor+(n);
  predecessor = successor.predecessor;
  build_fingers();
  inform_fingers(join);

n.build_fingers()
  f[1] = successor;
  f[2m] = predecessor;
  for i = 2 upto m
    f[i] = f[1].find_successor+(f̂[i]);
  for i = m + 2 upto 2m
    f[i] = f[2m].find_successor−(f̂[i]);

n.leave()
  inform_fingers(leave);

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

Figure 6: Node join and leave in S_Chord.

the interested nodes about its arrival. When leaving, a node informs the interested nodes about its departure. Note that operation *find_successor*$^-$ represents the left hand correspondent of the operation *find_successor*$^+$ defined in Figure 4. Moreover, note that we have not described the stabilization protocol; it remains the same as the one in Chord [12], ensuring the system correctness.

When a node $n$ announces its intention (i.e., join or leave), it does that for all the nodes interested. In a fully populated network, the nodes interested in the intentions of $n$ correspond to the fingers of $n$. In a poorly populated network, besides the fingers of $n$, other nodes may be interested in the intentions of $n$. These nodes are found at distances $\delta_R$, and $\delta_L$ from the targeted finger, depending on whether we are referring to the fingers in the right hand side, or to those in the left hand side, respectively.

$$\delta_R := (successor \ominus n) - 1 \tag{11}$$

$$\delta_L := (n \ominus predecessor) - 1 \tag{12}$$

In Equations 13 and 14 we define the distance between a node $n$ about to advertise and the last node to be informed. This distance will be used when forwarding the intentions of node $n$ to all interested nodes. Note that the first nodes to be informed are the actual fingers of $n$.

$$D_R(i) := \delta_R + (\widehat{f}[i] \ominus n), \quad i \in [1, m] \tag{13}$$

$$D_L(i) := \delta_L + (n \ominus \widehat{f}[i]), \quad i \in [m+1, 2m] \tag{14}$$

As shown in operation *inform_fingers*, Figure 7, the intention of a node $n$ is sent separately to fingers $i$ in the right side, and in the left side, together with its *id*, and the corresponding distance for which the message $M$ should be forwarded. Note that we use *successor* interchanged with $f[1]$, likewise for *predecessor* and $f[2m]$. When executing the operation *inform*, a node $n$ processes

```
n.inform_fingers(M)
  for i = 1 upto m
    f[i].inform(n, i + m, D_R(i), M);
  for i = m + 1 upto 2m
    f[i].inform(n, i − m, D_L(i), M);

n.inform(n', i, D, M)
  process_msg(n', i, M);
  if i ≤ m then
    if (n' ⊖ f[2m]) ≤ D then
      predecessor.inform(n', i, D, M);
    fi
  else
    if (f[1] ⊖ n') ≤ D then
      successor.inform(n', i, D, M);
    fi
  fi
```

```
n.process_msg(n', i, M)
  if M == join then
    if i ≤ m then
      if (f[i] ⊖ f̂[i]) > (n' ⊖ f̂[i]) then
        f[i] = n';
      fi
    else
      if (f̂[i] ⊖ f[i]) > (f̂[i] ⊖ n') then
        f[i] = n';
      fi
    fi
  elseif M == leave then
    if i == 1 then
      f[1] = successor_list[1];
    elseif i ≤ m then
      f[i] = find_successor⁺(f̂[i]);
    else
      f[i] = find_successor⁻(f̂[i]);
    fi
  fi
```

Figure 7: In-place notification of routing entries for node join and leave in S-Chord.

the message $M$ corresponding to finger $i$, and forwards it until the node found at distance $D$ from the node $n'$ (i.e., the node that made the announcement). If the message is join and $n'$ represents a better finger than the one found in $f[i]$ then update $f[i]$. If the message is leave, look for a new finger. For $i$ equal to 1, take the first node into the *successor_list*. As an optimization for tolerating sequences of up to $r-1$ node failures, a node $n$ stores a *successor_list* denoting the list of the first $r$ successors of $n$.

By being able to announce node departures, the period of time during which a finger table remains inconsistent can be reduced dramatically. Moreover, we can pass from a "pull" model, where the nodes periodically look for the correct fingers, to a "push" model, where the nodes announce their intentions. Note that, in this case, the "push" model is less costly (in number of messages) than the "pull" model.

# 5 Simulation results

We implemented and simulated the lookup algorithms of Chord and S-Chord. The algorithms were simulated in software as recursive functions using the Mozart [3] programming platform. The measurements we made concerned only the lookup path length, and the routing cost symmetry. For all the simulations we considered the distribution of the queries to be uniform over the search space.

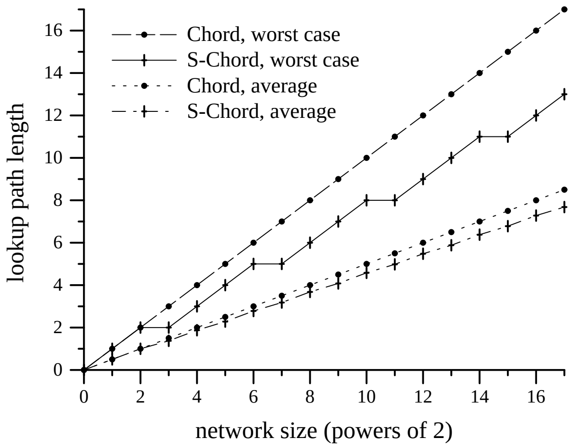



Figure 8: Worst case and average path length function the network size for Chord and S-Chord fully populated networks.

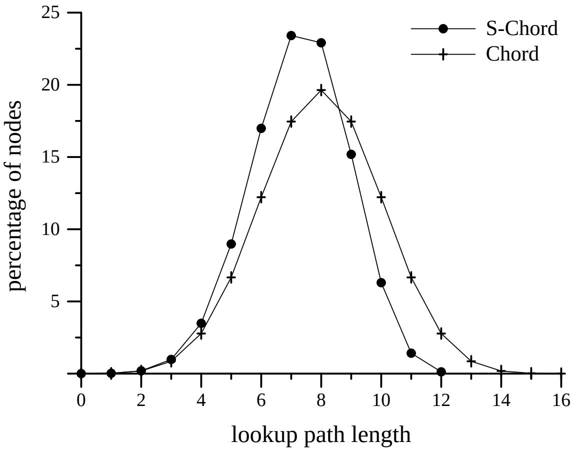



Figure 9: Distribution of the path length in Chord and S-Chord for $N = 2^{16}$.

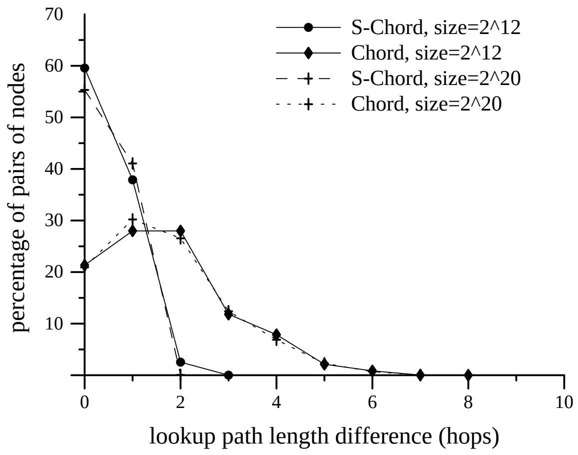



Figure 10: Distance variation between pairs of nodes in Chord and S-Chord, measured for two poorly populated networks of sizes $2^{12}$ and $2^{20}$, respectively.

For the first test 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 confirmed our expectations (see Figure 8). That is, in the worst case, the number of hops a lookup can take in S-Chord is 25% less than in Chord. Note that for certain values of the network size S-Chord does even better. This is the case represented by Equation 10 for networks of size $N = 2^l$, where ($l$ mod $N$)= 3. We observed that on average lookups take around 10% less hops in S-Chord than they do in Chord.

In Figure 9 we plot the percentage of nodes that can be reached within different lookup path lengths, for a network of size $N = 2^{16}$. Note that the distribution corresponding to S-Chord is closer to the origin than the distribution corresponding to Chord. This results in a larger number of lookups of smaller lengths, thus providing an explanation for the 10% improvement of the average path length in S-Chord.

The second test analyses the routing cost symmetry for both systems. To this end, we measured the percentage $p(x)$ of any pair of two nodes $n$ and $n'$ such that the absolute difference between the distance ($n$, $n'$), in number of hops, and the distance ($n'$, $n$), equals $x$:

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

For this test, we considered two networks (of sizes $2^{12}$, and $2^{20}$), partially populated (1024 nodes randomly chosen; the nodes are uniformly distributed over the search space). As illustrated in Figure 10, 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.

# 6 Discussions

In this section we discuss several ideas that we have about extending and improving S-Chord.

## 6.1 Neighbor proximity

As was first mentioned in [10], mapping the overlay network to the underlying physical network represents an important issue in current p*2*p systems. Ignoring the topology of the underlying network when building p*2*p networks can lead to extra communication costs between peers. In Chord, a previous attempt was made in [5] in order to provide proximity routing, but as described in [4], the solution is not very effective.

The symmetry of S-Chord opens new perspectives for exploiting the network proximity. We have the idea of applying NetProber [8] over S-Chord, thus providing proximity neighbor selection. NetProber is a simple, distributed, and scalable component that can be combined with any connected overlay network in order to allow the latter to adapt within a finite amount of time. This results in running the NetProber algorithm at each node $n$ in the system, and measuring the physical distance between $n$ and each node within its neighborhood in order to perform a better mapping between the logical network and the physical one.

We observe that, due to the routing cost symmetry, and the routing entry symmetry of S-Chord, the mapping operations can evolve to a stable point. That is, for any two nodes $u$ and $v$, if $u$ gets closer (in number of physical hops) to $v$, then $v$ gets closer to $u$ too.

## 6.2 Generalization

Given a network of size $N$, S-Chord employs a finger table with $log_2 N$ entries to resolve keys in maximum $\lceil \frac{3}{4} log_2 N \rceil$ number of hops. In order to allow more flexibility, it should be able to parameterize the system by the finger table size that a node has to store, and the upper bound of the lookup path length for a network of a given size.

We are currently working on a function that given the network size $N$, and the desired degree of lookup efficiency, returns a finger table which respects the symmetry of S-Chord. The preliminary results that we have obtained are comparable and even better than the generalization proposed for Chord [12], encouraging us to further explore this idea.

### 6.3 Node failure

In order to cope with the failure of sequences of $r$ nodes, as in Chord, each S-Chord node has to store a list of $r+1$ successor nodes (called *successor_list*). This is less than in Hyperchord where $r+1$ successors and $r+1$ predecessors are needed.

## 7 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, we are confident that for dynamic scenarios (relative high rate of nodes joining/leaving) the average hop length in S-Chord is lower than the one in Chord and comparable with the one in Hyperchord, since S-Chord is also using notifications for the node leave procedure. We will address this issue in details in a further study.

The symmetry in a Chord based system such as S-Chord is useful for improving the node join and leave operations. Indeed, the nodes joining/leaving the system can announce their intentions to the interested nodes, thus employing a "push", rather than a "pull" model. Further on, the routing entry symmetry and the routing cost symmetry in S-Chord is useful for exploiting the underlying network proximity.

By the end of the present work we were well surprised to discover that there are also other possibilities for symmetrically choosing the fingers of a node resulting in even better lookup efficiency. We will address this issue in a further study together with the generalization of S-Chord which is ongoing research. The preliminary results encourage us to follow the work in this direction.

We would also like to investigate what we believe to be an open question; i.e., achieving "routing symmetry" through proximity neighbor selection based on a previous work, namely NetProber.

## Acknowledgments

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

## References

[1] Gnutella. `http://gnutella.wego.com`.

[2] Napster. `http://www.napster.com`.

[3] Mozart system. `http://www.mozart-oz.org`, December 2002.

[4] M. Castro, Y. Hu P. Druschel, and A. Rowstron. Exploiting network proximity in distributed hash tables. In *International Workshop on Future Directions in Distributed Computing (FuDiCo 2002)*, June 2002.

[5] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[6] E. Oram et al. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, 2001.

[7] 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.

[8] L. Onana, V. Mesaros, P. Van Roy, and S. Haridi. NetProber: a component for enhancing efficiency of overlay networks in p2p systems. In *Proc. of the 2nd IEEE International Conference on Peer-to-Peer Computing (P2P 2002)*, Spetember 2002.

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

[10] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proc. of the 1st IEEE International Conference on Peer-to-Peer Computing (P2P 2001).*

[11] 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.

[12] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.

[13] 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.