Chord: A scalable peer-to-peer look-up protocol for internet applications

Robert Morris, M. Frans Kaashoek, David Karger, Hari Balakrishnan, Ion Stoica, David Liben-Nowell, Frank Dabek

Write-up by Christine Kiefer

Saarland University, Department of Computer Science

Saarbrücken, 30 November 2003

Content

1 Motivation..... 3

2 Introduction to Chord..... 4

2.1 Properties of Chord..... 4

2.2 Chord software..... 4

3 The Chord Protocol..... 5

3.1 The Chord Ring..... 5

3.2 Key Location 6

3.2.1 Simple key location..... 6

3.2.2 Scalable key location..... 7

4 Joining of Nodes..... 8

4.1.1 Impact of node joins on correctness..... 10

4.1.2 Impact of node joins on performance..... 10

5 Failure of Nodes..... 11

6 Applications..... 12

7 Summary..... 13

8 References..... 13

This paper was produced as a write-up to a presentation I held as part of the seminar „Peer-to-peer Systems“ given by Prof. Dr. Gerhard Weikum, Chair for Database Systems and Information Retrieval, in the winter semester 2003/2004. Topic of the presentation was Chord, a peer-to-peer protocol developed at the MIT by Robert Morris, M. Frans Kaashoek, David Karger, Hari Balakrishnan, Ion Stoica, David Liben-Nowell and Frank Dabek. Their paper “Chord: A scalable lookup peer-to-peer protocol for Internet Applications” [1] served as basis literature for my presentation and for this write-up.

1 Motivation

Peer to Peer Systems are loosley organized systems without any centralized control or any hierarchical system. Each node runs a software with equivalent functionality. Peer to peer systems provide users with features such as availability, permanence, redundant storage, selection of nearby servers, anonymity.

Peer to peer applications have become more and more popular during the last years. Well-known examples for such applications are Napster and Gnutella, which implement file-sharing for mp3s. Newer applications like eDonkey implement file-sharing even for large files, such as movies.

The core problem of peer-to-peer applications is the efficient location of nodes. How does a node know which node in the system contains the data it is searching for?

Several solutions have been found to address this problem:

- Napster uses a central server where all queries are directed to first. The server informs the user where he can download the data he is looking for.
  Problem: The central server yields a single point of failure.

- Gnutella uses message flooding to locate data objects. With message flooding, a message is forwarded to every node in the system until the corresponding node containing the data object has been found. Most often, a TTL time is set to limited the number of messages.
  Problem: The number of messages increases linearly to the number of nodes.

- Newer applications use super-peers, which store indexed information about the nodes of the system and therefore work in the same way as central server. But by spreading the indexed information over several super-peers, the risk of total failure of the system is limited.
  Problem: Even though the risk of failure and the problem of availability is minimized, they can still occur.

Since a mp3 which is temporarily unavailable does not cause tremendous consequences for anyone, the applications mentioned above match the needs of their users very well despite of their deficiencies.
In order to use peer-to-peer systems for more sophisticated applications such as business applications, a protocol with higher standards is needed.

2 Introduction to Chord

Chord is a peer-to-peer protocol which presents a new approach to the problem of efficient location. Chord uses routed queries to locate a key with a small number of hops, which stays small even if the system contains a large number of nodes.
What distinguishes Chord from other applications is its simplicity, its provable performance and provable correctness.
Basically, Chord supports just one operation: given a key, it maps the key onto a node. Data localization can be implemented by associating each key with a data item.

2.1 Properties of Chord

- **Decentralization**
  In a peer-to-peer system using Chord, there exists no central server or superpeer. Each node is of the same importance as any other node. Therefore, the system is very robust, since it does not have a single point of failure.

- **Availability**
  The protocol functions very well even if the system is in a continuous state of change: Despite major failures of the underlying network and despite the joining of large number of nodes, the node responsible for a key can always be found.

- **Scalability**
  The cost of a Chord lookup grows only logarithmically in the number of nodes in the system, so Chord can be used for very large systems.

- **Load balance**
  Chord uses a consistent hash function to assign keys to nodes. Therefore, the keys are spread evenly over the nodes.

- **Flexible naming**
  Chord imposes no constraints on the key structure, so the user is granted a large amount of flexibility in the data can be named.

2.2 Chord software

The Chord software consists of 3000 lines of C++ code in form of a library to be linked with the application. The software interacts with the application in two ways:

- It provides a lookup(key) – function, which yields the IP address of the node responsible for the key.
- It notifies the node of changes in the set of keys the node is responsible for.

3 The Chord Protocol

The following section describes the Chord protocol, including some examples for pseudocode of the functions. The protocol contains functions to locate nodes and to deal with joins and failures of nodes.

3.1 The Chord Ring

The Chord protocol uses SHA-1 [2] as consistent hash function to assign a m-bit identifier to each node and each key.
Consistent hash functions are hash functions with some additional advantageous properties, i.e. they let nodes join and leave the system with minimal disruption [3][4].
The m is an integer which should be chosen big enough to make the probability that two nodes or two keys receive the same identifier negligible.
The hash function calculates the key identifier by hashing the key, and the node identifier by hashing the IP address of the node.

The key and the node identifiers are arranged on an identifier circle of size 2^m called the Chord ring. The identifiers on the Chord ring are numbered from 0 to 2^{m-1}. A key is assigned to a node whose identifier is equal to or greater than the identifier of the key. This node is called the successor node of k, denoted by successor(k), and is the first node clockwise from k on the circle.

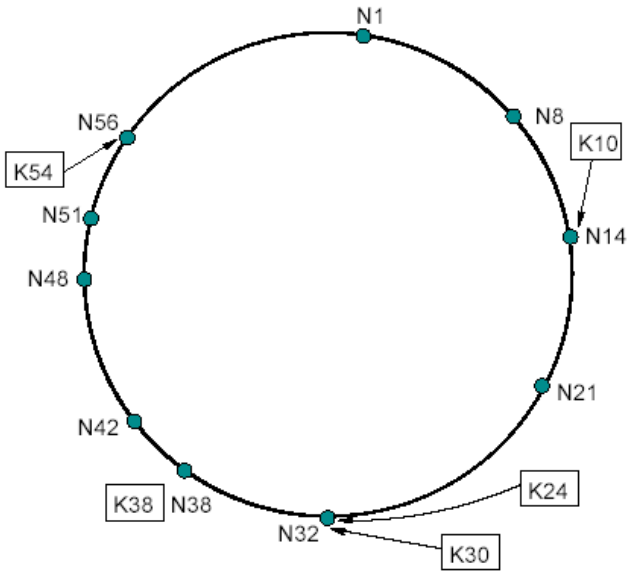

Fig. 3-1 shows a Chord ring with m = 6, 10 nodes and 5 keys. Since the successor of K10 is N14, K10 is located at N14.

3.2 Key Location

The core function of the Chord protocol is the key location function. For a better understanding, a simple key location function is introduced first. Next, the scalable key location function will be demonstrated.

3.2.1 Simple Key Location

In order to let a node n find a certain key k, we call n.lookup(k). To execute the lookup, the protocol will call the function find_successor (Fig. 3-2a), which will return the successor node from node n if k lies between n and its successor or forward the query around the circle otherwise (Fig 3-2b).

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ϵ (n; successor])
     return successor;
  else
     // forward the query around the
        circle
    return successor.find_successor(id);
```

Fig. 3-2a: Pseudocode for simple key location

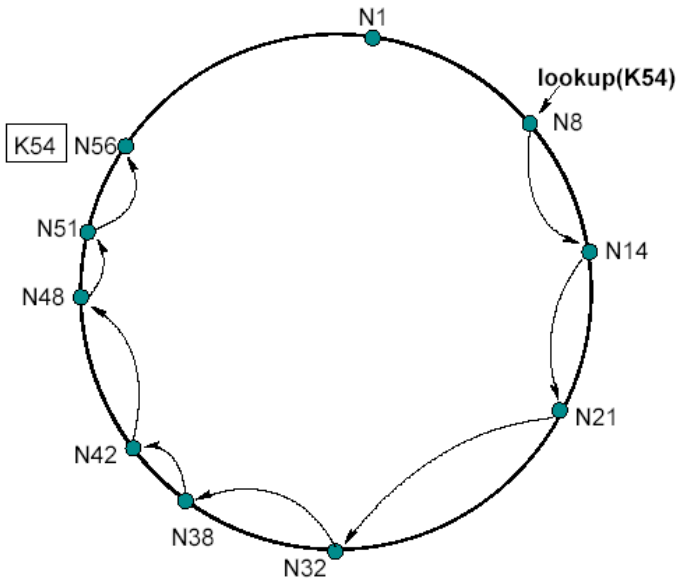

Fig. 3-2b: In the simple lookup function, the query is forwarded around the cirle

In the worst case, the query needs to be forwarded N times in a circle with N nodes, so the cost of a lookup is linear in the number of nodes.
In systems with a large number of nodes, lookups would be too slow. Therefore, Chord uses a scalable key location function which will provide more efficient lookups.

3.2.2 Scalable Key Location

In order to provide more efficient lookups, additional routing information is stored to accelerate lookups. Each node n maintains a routing table with up to m entries (where m is the number of bits of the identifiers) which is called the finger table.
The i^{th} entry in the table at node n contains the first node s that succeds n by at least 2^{i-1} This node s is called the i^{th} finger of node n. Fig. 3-2d shows the routing table for node N8.

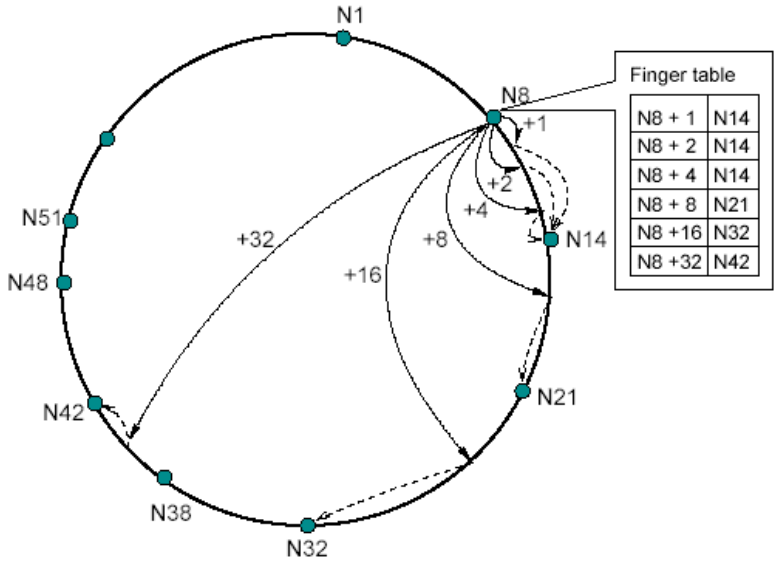

Fig. 3-2d: Finger table entries are calculated by the formula finger[i] = successor (n + 2^{i-1})

Important characteristics of this scheme are:

- Each node stores information about only a small number of nodes (m).
- Each nodes knows more about nodes closely following it than about nodes farer away.
- A finger table generally does not contain enough information to directly determine the successor of an arbitrary key k. A node has to contact other nodes in order to resolve the hash table.

When a node is asked to find a certain key, it will determine the highest predecessor of this key in its routing table and forward the key to that node. This procedure will recursivley determine the node responsible for the key. The lookup time is O(log N), since the query is forwarded at least half the remaining distance around the circle in each step (see Fig. 3-2e).

```
// ask node n to find the successor of id
n.find_successor(id)
if (id ϵ (n; successor])
return successor;
else n0 = closest_preceding_node(id);
return n0.find successor(id);

// search the local table for the highest
// predecessor of id
n.closest_preceding_node(id)
for i = m downto 1
if (finger[i] ϵ (n; id))
return finger[i];
return n;
```

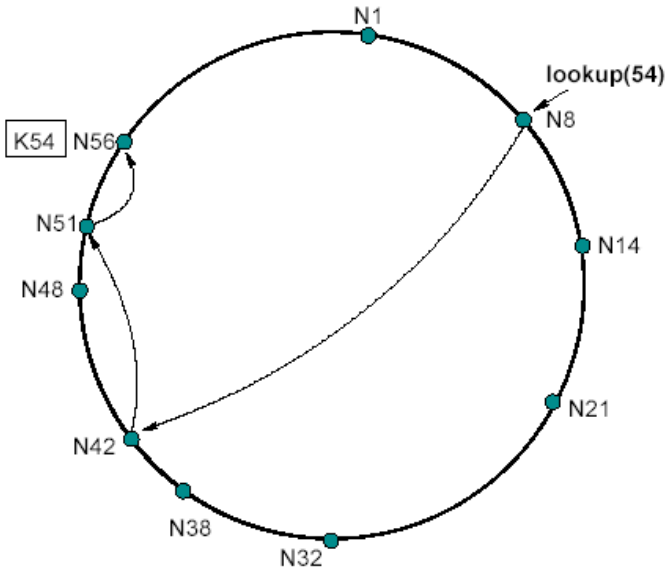

Fig. 3-2e: Pseudocode and illustration for the scalable lookup key function. Since queries are forward at least half the remaining distance around the circle, lookup speed is O(log N).

4 Joining of Nodes

When a node joins the system, the successor pointers of some nodes will have to change. It is important that the successor pointers are up to date at any time because the correctness of lookups is not guaranteed otherwise. The Chord protocol uses a stabilization protocol running periodically in the background to update the successor pointers and the entries in the finger table. The following pseudocode will explain the functioning of the stabilization protocol in detail. Fig. 4-1 illustrates an example.

```
// create a new Chord ring.
n.create()
predecessor = nil;
successor = n;
```

Node n is the first node to start a new Chord ring. It does not have a predecessor and it is its own successor.

```
// join a Chord ring containing node n0.
n.join(n0)
predecessor = nil;
successor = n0.find successor(n);
```

Node n wants to join the Chord ring. Knowing that n0 is part of the ring, it will ask n0 to find the successor of n

```
// verifies n's immediate successor, and
tells the successor about n.
n.stabilize()
x = successor.predecessor;
if (x ϵ (n; successor))
successor = x;
successor.notify(n);
```

stabilize() is run periodically by each node. x is set to the predecessor of the successor of n, which will be n except for the case that a new node has joined recently between x and its successor. In this case, n will set its successor to x and notifies x of its own existence.

// n0 *thinks it might be our predecessor.*
n.notify(n0)
if (*predecessor* **is nil**
or n0 ϵ (*predecessor*; n))
predecessor = n0;
	n0 notifies n of its existence. If n does not yet have a predecessor if n0 is closer to n than its current predecessor, n sets its predecessor pointer to n0.

// *refreshes finger table entries.* n.fix _fingers() next = next + 1 ; **if (**next > m) next = 1 ; *finger*[next] = *find successor*(n + 2^{next-1});	each node runs fix_fingers periodically. This is how nodes update their finger tables and how new nodes initialize their finger table
// *checks whether predecessor has failed.* n.check_predecessor() **if** (*predecessor* **has failed**) *predecessor* = nil;	Each node checks periodically whether its predecessor has failed, so it can clear its pointer and accept a new predecessor.

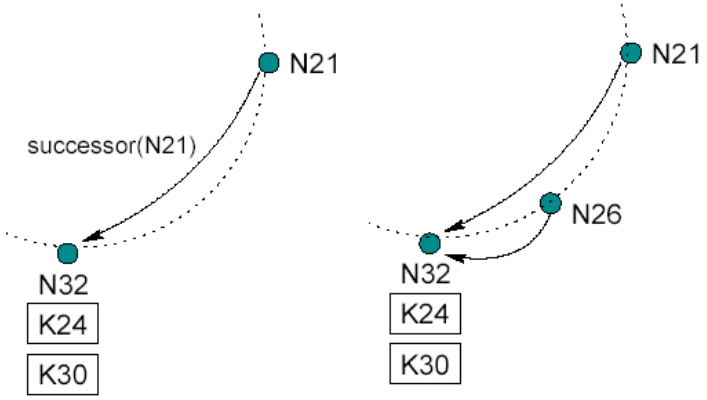

Fig. 4-1a. N26 joins the Chord ring. It sets its successor pointer to N32 and notifies N32, so N32 sets its predecessor pointer to N26.

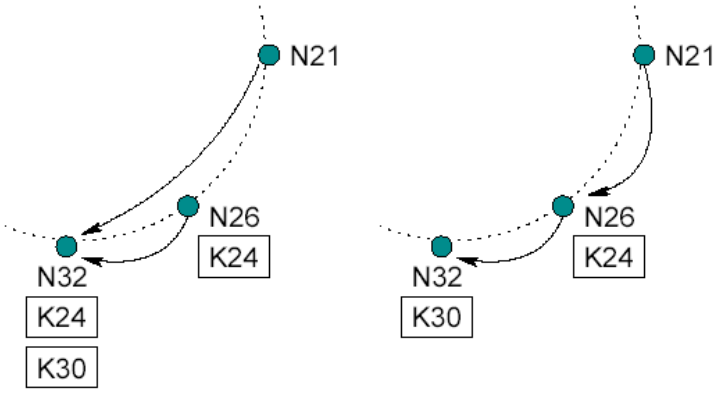

Fig. 4-1b. N26 copies K24. Next time N21 runs stabilize(), it detects N26 as its new successor. N21 changes its successor pointer to N26 and notifies N26, so N26 sets its predecessor pointer to N21.

4.1 Impact of node joins on correctness

When nodes have joint recently and a lookup occurs before stabilization has finished, the system finds itself in one of these three states:

- **All finger table entries and successor pointers are correct** at the time of the lookup: No impact on correctness, the lookup will be successfull in time O(log N).

- **The successor pointers are correct, but the finger table entries are not:** The lookup will still be correct, but might be a little slower: in case a larger number of nodes has joined between the target and the target's predecessor, the find_successor function will initially undershoot and some of the hops will be in linear time (see Fig. 4-1).

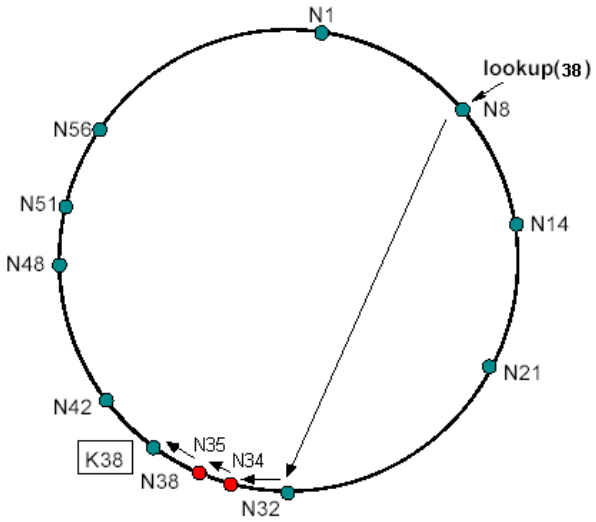

Fig. 4-1: When a large number of nodes join between the target and the target's predecessor, lookups might be slightly slower, but still correct.

- **Neither finger table entries nor successor pointers are correct**: In this case, the lookup will fail. The higher-layer software using Chord will notice that the data was not found and will retry after a short pause.

This leads to the conclusion that node joins have no impact on correctness.

4.2 Impact of node joins on performance

When stabilization has been completed, there is no impact on performance beyond increasing N (total number of nodes) in the O(log N) lookup time.
When stabilization has not been completed, the lookup speed might be affected if nodes join between the target and the target's successor, as mentioned in case two of section 4.1. But this is only the case if the number of joining nodes is very large. In general, it can be stated that lookups take O(log N) hops as long as the time it takes to adjust finger tables is less than the time it takes the network to double in size.

5 Failure of Nodes

The correctness of the Chord protocol relies on the fact that each node knows its successor. When nodes fail, it is possible that a node does not know its new successor, and that it has no chance to learn about it. Figure 5-1 demonstrates an example.

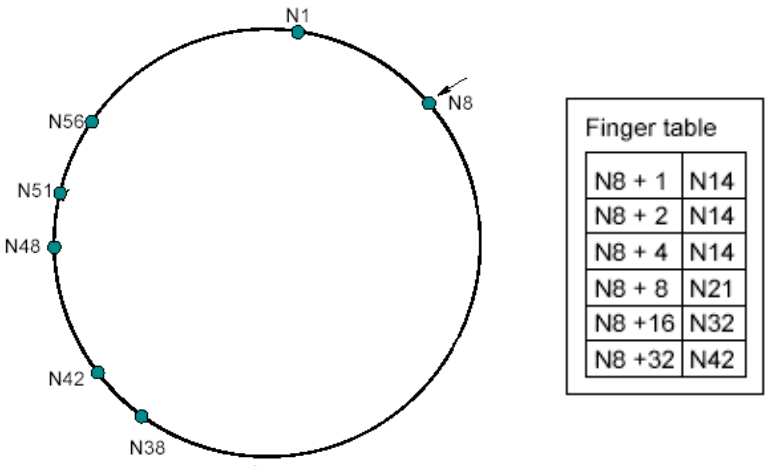

Fig. 5-1: When nodes N14, N21 and N32 fail simultaneously, N8 has no chance to learn about its new successor N38, since it does not show up in the finger table of N8.

To avoid this situation, each node maintains a successor list of size r, containing the node's first r successors. When the successor node does not respond, the node simply contacts the next node on its successor list.
Assuming that each node fails with a probability p, the probability that every node on the successor list fails is p^r. Increasing r makes the system more robust. With this parameter tuning, any degree of robustness can be achieved.
It can be proven that under the assumption that the network is initially stable, and every node fails with probability ½, find_successor still finds the closest living successor to the query key and the expected time to execute find_succesor is O(log N) [1]. Simulation results have shown that even massive failures have little impact on robustness (Figure 5-2).

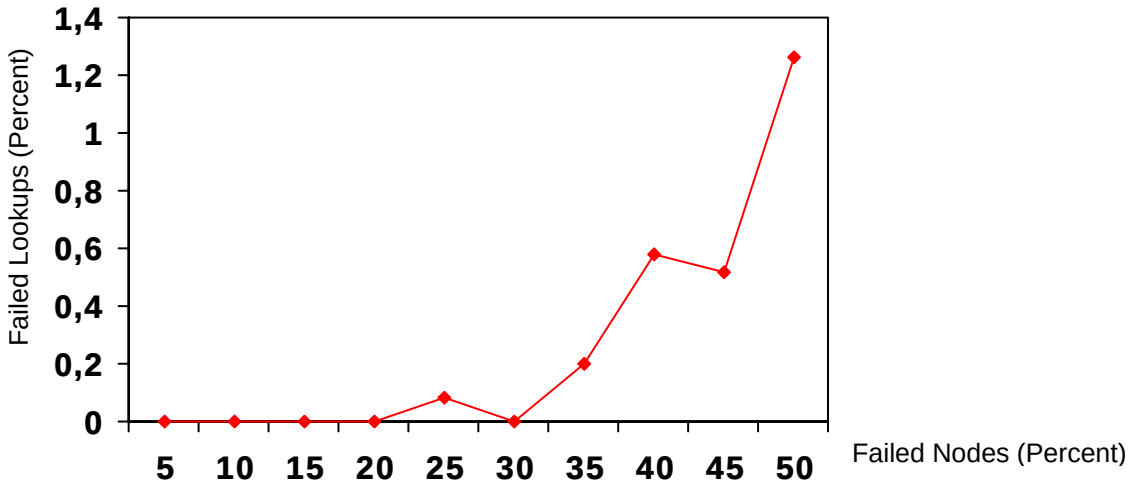

Fig. 5-2: If 50% of the nodes fail, only about 1,3% of the lookups fail.

6 Applications

The following applications are examples Chord could be used for:

- **Cooperative mirroring,** in which multiple providers of content cooperate to store and serve each others' data. Spreading the total load evenly over all participants' hosts lowers the total cost of the system, since each participant Needs to provide capacity only for the average load, not for that participant's peak load.

- **Time-shared storage** for nodes with intermittent connectivity. If someone wishes their data to be always available, but their server is only occasionally available, they can offer to store others' data while they are connected, in return for having their data stored elsewhere when they are disconnected. The data's name can serve as a key to identify the (live) Chord node responsible for storing the data item at any given time.

- **Chord-based DNS**
  DNS provides a lookup service, with host names as keys and IP addresses (and other host information) as values. Chord could provide the same service by hashing each host name to a key [7].
  Chord-based DNS would require no special servers, while ordinary DNS relies on a set of special root servers. DNS requires manual management of the routing information (NS records) that allows clients to navigate the name server hierarchy; Chord automatically maintains the correctness of the analogous routing information. DNS only works well when host names are structured to reflect administrative boundaries; Chord imposes no naming structure. DNS is specialized to the task of finding named hosts or services, while Chord can also be used to find data objects that are not tied to particular machines.

7 Summary

Chord is a simple but powerful protocol which solves the problem of efficient data location. Its only operation is to map a key to the responsible node.
Each node maintains routing information about O(log N) other nodes, and lookups are feasible via O(log N) messages. Therefore, Chord scales well with number of nodes what makes it an interesting application for larger systems.
Chord continues to function correctly even if the system undergoes major changes and if the routing information is only partially correct.

8 References

[1] MORRIS, R., KAASHOEK, M. F., KARGER, D., BALAKRISHNAN, H., STOICA, I., LIBEN-NOWELL, D., DABEK, F.: Chord: A scalable peer-to-peer lookup protocol for internet applications
To Appear in IEEE/ACM Transactions on Networking.
http://www.pdos.lcs.mit.edu/chord/
http://www.pdos.lcs.mit.edu/chord/papers/paper-ton.pdf

[2] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/ NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[3] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663.

[4] LEWIN, D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT, 1998. Available at the MIT Library, http://thesis.mit.edu/.

[5] MORRIS, R.,: Slides for sigcomm 2001
http://www.pdos.lcs.mit.edu/~rtm/slides/sigcomm01.ppt

[6] WEIKUM, G.: Slides of first lesson of the seminar „Peer-to-Peer Systems"
http://www.mpi-sb.mpg.de/units/ag5/teaching/ws03_04/p2p-data/10-21-intro.pdf

[7] COX, R., MUTHITACHAROEN, A., AND MORRIS, R. Serving DNS using Chord. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).