Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica[*], Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan[†]
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Abstract

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis, simulations, and experiments show that Chord is scalable, with communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes.

1. Introduction

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is efficient location of data items. The contribution of this paper is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

The *Chord protocol* supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses a variant of consistent hashing [11] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and involves relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumed that nodes were aware of most other nodes in the system, making it impractical to scale to large number of nodes. In contrast, each Chord node needs "routing" information about only a few other nodes. Because the routing table is distributed, a node resolves the hash function by communicating with a few other nodes. In the steady state, in an N-node system, each node maintains information only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Chord maintains its routing information as nodes join and leave the system; with high probability each such event results in no more than $O(\log^2 N)$ messages.

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance. Chord is simple, routing a key through a sequence of $O(\log N)$ other nodes toward the destination. A Chord node requires information about $O(\log N)$ other nodes for *efficient* routing, but performance degrades gracefully when that information is out of date. This is important in practice because nodes will join and leave arbitrarily, and consistency of even $O(\log N)$ state may be hard to maintain. Only one piece information per node need be correct in order for Chord to guarantee correct (though slow) routing of queries; Chord has a simple algorithm for maintaining this information in a dynamic environment.

The rest of this paper is structured as follows. Section 2 compares Chord to related work. Section 3 presents the system model that motivates the Chord protocol. Section 4 presents the base Chord protocol and proves several of its properties, while Section 5 presents extensions to handle concurrent joins and failures. Section 6 demonstrates our claims about Chord's performance through simulation and experiments on a deployed prototype. Finally, we outline items for future work in Section 7 and summarize our contributions in Section 8.

2. Related Work

While Chord maps keys onto nodes, traditional name and location services provide a *direct* mapping between keys and values. A value can be an address, a document, or an arbitrary data item. Chord can easily implement this functionality by storing each key/value pair at the node to which that key maps. For this reason and to make the comparison clearer, the rest of this section assumes a Chord-based service that maps keys onto values.

DNS provides a host name to IP address mapping [15]. Chord can provide the same service with the name representing the key and the associated IP address representing the value. Chord requires no special servers, while DNS relies on a set of special root

[*] University of California, Berkeley. istoica@cs.berkeley.edu
[†] Authors in reverse alphabetical order.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.

SIGCOMM'01, August 27-31, 2001, San Diego, California, USA.

servers. DNS 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.

The Freenet peer-to-peer storage system [4, 5], like Chord, is decentralized and symmetric and automatically adapts when hosts leave and join. Freenet does not assign responsibility for documents to specific servers; instead, its lookups take the form of searches for cached copies. This allows Freenet to provide a degree of anonymity, but prevents it from guaranteeing retrieval of existing documents or from providing low bounds on retrieval costs. Chord does not provide anonymity, but its lookup operation runs in predictable time and always results in success or definitive failure.

The Ohaha system uses a consistent hashing-like algorithm for mapping documents to nodes, and Freenet-style query routing [18]. As a result, it shares some of the weaknesses of Freenet. Archival Intermemory uses an off-line computed tree to map logical addresses to machines that store the data [3].

The Globe system [2] has a wide-area location service to map object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search short cuts [22]. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

The distributed data location protocol developed by Plaxton *et al.* [19], a variant of which is used in OceanStore [12], is perhaps the closest algorithm to the Chord protocol. It provides stronger guarantees than Chord: like Chord it guarantees that queries make a logarithmic number hops and that keys are well balanced, but the Plaxton protocol also ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. The advantage of Chord is that it is substantially less complicated and handles concurrent node joins and failures well. The Chord protocol is also similar to Pastry, the location algorithm used in PAST [8]. However, Pastry is a prefix-based routing protocol, and differs in other details from Chord.

CAN uses a d-dimensional Cartesian coordinate space (for some fixed d) to implement a distributed hash table that maps keys onto values [20]. Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$. Thus, in contrast to Chord, the state maintained by a CAN node does not depend on the network size N, but the lookup cost increases faster than $\log N$. If $d = \log N$, CAN lookup times and storage needs match Chord's. However, CAN is not designed to vary d as N (and thus $\log N$) varies, so this match will only occur for the "right" N corresponding to the fixed d. CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes. Chord also has the advantage that its correctness is robust in the face of partially incorrect routing information.

Chord's routing procedure may be thought of as a one-dimensional analogue of the Grid location system [14]. Grid relies on real-world geographic location information to route its queries; Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to Grid's.

Chord can be used as a lookup service to implement a variety of systems, as discussed in Section 3. In particular, it can help avoid single points of failure or control that systems like Napster possess [17], and the lack of scalability that systems like Gnutella display because of their widespread use of broadcasts [10].

3. System Model

Chord simplifies the design of peer-to-peer systems and applications based on it by addressing these difficult problems:

- **Load balance:** Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.
- **Decentralization:** Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.
- **Scalability:** The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.
- **Availability:** Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.
- **Flexible naming:** Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

The Chord software takes the form of a library to be linked with the client and server applications that use it. The application interacts with Chord in two main ways. First, Chord provides a `lookup(key)` algorithm that yields the IP address of the node responsible for the key. Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for. This allows the application software to, for example, move corresponding values to their new homes when a new node joins.

The application using Chord is responsible for providing any desired authentication, caching, replication, and user-friendly naming of data. Chord's flat key space eases the implementation of these features. For example, an application could authenticate data by storing it under a Chord key derived from a cryptographic hash of the data. Similarly, an application could replicate data by storing it under two distinct Chord keys derived from the data's application-level identifier.

The following are examples of applications for which Chord would provide a good foundation:

Cooperative Mirroring, as outlined in a recent proposal [6]. Imagine a set of software developers, each of whom wishes to publish a distribution. Demand for each distribution might vary dramatically, from very popular just after a new release to relatively unpopular between releases. An efficient approach for this would be for the developers to cooperatively mirror each others' distributions. Ideally, the mirroring system would balance the load across all servers, replicate and cache the data, and ensure authenticity. Such a system should be fully decentralized in the interests of reliability, and because there is no natural central administration.

Time-Shared Storage for nodes with intermittent connectivity. If a person wishes some data to be always available, but their

FS: API

Block Store: path parsing n use chord to look up files

Chord: distributed routing to locate files

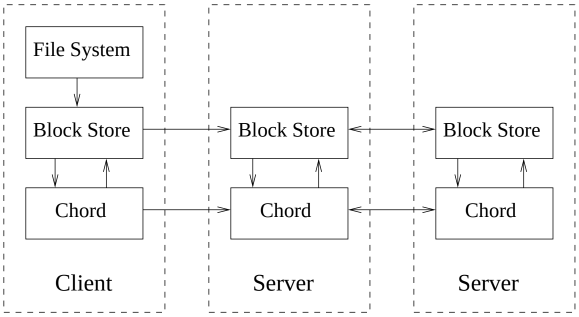


Figure 1: Structure of an example Chord-based distributed storage system.

machine is only occasionally available, they can offer to store others' data while they are up, in return for having their data stored elsewhere when they are down. 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. Many of the same issues arise as in the Cooperative Mirroring application, though the focus here is on availability rather than load balance.

Distributed Indexes to support Gnutella- or Napster-like keyword search. A key in this application could be derived from the desired keywords, while values could be lists of machines offering documents with those keywords.

Large-Scale Combinatorial Search, such as code breaking. In this case keys are candidate solutions to the problem (such as cryptographic keys); Chord maps these keys to the machines responsible for testing them as solutions.

Figure 1 shows a possible three-layered software structure for a cooperative mirror system. The highest layer would provide a file-like interface to users, including user-friendly naming and authentication. This "file system" layer might implement named directories and files, mapping operations on them to lower-level block operations. The next layer, a "block storage" layer, would implement the block operations. It would take care of storage, caching, and replication of blocks. The block storage layer would use Chord to identify the node responsible for storing a block, and then talk to the block storage server on that node to read or write the block.

4. The Base Chord Protocol

The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes. This section describes a simplified version of the protocol that does not handle concurrent joins or failures. Section 5 describes enhancements to the base protocol to handle concurrent joins and failures.

4.1 Overview

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. It uses *consistent hashing* [11, 13], which has several good properties. With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an N^{th} node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location—this is clearly the minimum necessary to maintain a balanced load.

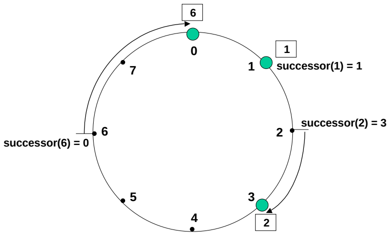


Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In an N-node network, each node maintains information only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages.

Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(\log^2 N)$ messages.

4.2 Consistent Hashing

The consistent hash function assigns each node and key an m-bit *identifier* using a base hash function such as SHA-1 [9]. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term "node" will refer to both the node and its identifier under the hash function. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered in an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by *successor*(k). If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then $successor(k)$ is the first node clockwise from k.

Figure 2 shows an identifier circle with $m = 3$. The circle has three nodes: 0, 1, and 3. The successor of identifier 1 is node 1, so key 1 would be located at node 1. Similarly, key 2 would be located at node 3, and key 6 at node 0.

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. No other changes in assignment of keys to nodes need occur. In the example above, if a node were to join with identifier 7, it would capture the key with identifier 6 from the node with identifier 0.

The following results are proven in the papers that introduced consistent hashing [11, 13]:

THEOREM 1. *For any set of N nodes and K keys, with high probability:*

1. *Each node is responsible for at most $(1 + \epsilon)K/N$ keys*

2. *When an $(N+1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).*

When consistent hashing is implemented as described above, the theorem proves a bound of $\epsilon = O(\log N)$. The consistent hashing paper shows that ϵ can be reduced to an arbitrarily small constant by having each node run $O(\log N)$ "virtual nodes" each with its own identifier.

The phrase "with high probability" bears some discussion. A simple interpretation is that the nodes and keys are randomly chosen, which is plausible in a non-adversarial model of the world. The probability distribution is then over random choices of keys and nodes, and says that such a random choice is unlikely to produce an unbalanced distribution. One might worry, however, about an adversary who intentionally chooses keys to all hash to the same identifier, destroying the load balancing property. The consistent hashing paper uses "k-universal hash functions" to provide certain guarantees even in the case of nonrandom keys.

Rather than using a k-universal hash function, we chose to use the standard SHA-1 function as our base hash function. This makes our protocol deterministic, so that the claims of "high probability" no longer make sense. However, producing a set of keys that collide under SHA-1 can be seen, in some sense, as inverting, or "decrypting" the SHA-1 function. This is believed to be hard to do. Thus, instead of stating that our theorems hold with high probability, we can claim that they hold "based on standard hardness assumptions."

For simplicity (primarily of presentation), we dispense with the use of virtual nodes. In this case, the load on a node may exceed the average by (at most) an $O(\log N)$ factor with high probability (or in our case, based on standard hardness assumptions). One reason to avoid virtual nodes is that the number needed is determined by the number of nodes in the system, which may be difficult to determine. Of course, one may choose to use an a priori upper bound on the number of nodes in the system; for example, we could postulate at most one Chord server per IPv4 address. In this case running 32 virtual nodes per physical node would provide good load balance.

4.3 Scalable Key Location

A very small amount of routing information suffices to implement consistent hashing in a distributed environment. Each node need only be aware of its successor node on the circle. Queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. However, this resolution scheme is inefficient: it may require traversing all N nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as the successor information is maintained correctly.

As before, let m be the number of bits in the key/node identifiers. Each node, n, maintains a routing table with (at most) m entries, called the *finger table*. The i^{th} entry in the table at node n contains the identity of the *first* node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$ (and all arithmetic is modulo 2^m). We call node s the i^{th} *finger* of node n, and denote it by n.*finger*$[i]$.*node* (see Table 1). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of n is its immediate successor on the circle; for convenience we often refer to it as the *successor* rather than the first finger.

In the example shown in Figure 3(b), the finger table of node 1 points to the successor nodes of identifiers $(1 + 2^0) \bmod 2^3 = 2$, $(1 + 2^1) \bmod 2^3 = 3$, and $(1 + 2^2) \bmod 2^3 = 5$, respectively. The successor of identifier 2 is node 3, as this is the first node that follows 2, the successor of identifier 3 is (trivially) node 3, and the successor of 5 is node 0.

Notation	Definition
finger$[k].start$	$(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
.*interval*	[*finger*$[k]$.*start*, *finger*$[k+1]$.*start*)
.*node*	first node $\geq n$.*finger*$[k]$.*start*
successor	the next node on the identifier circle; *finger*$[1]$.*node*
predecessor	the previous node on the identifier circle

Table 1: Definition of variables for node n, using m-bit identifiers.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to determine the successor of an arbitrary key k. For example, node 3 in Figure 3 does not know the successor of 1, as 1's successor (node 1) does not appear in node 3's finger table.

What happens when a node n does not know the successor of a key k? If n can find a node whose ID is closer than its own to k, that node will know more about the identifier circle in the region of k than n does. Thus n searches its finger table for the node j whose ID most immediately precedes k, and asks j for the node it knows whose ID is closest to k. By repeating this process, n learns about nodes with IDs closer and closer to k.

The pseudocode that implements the search process is shown in Figure 4. The notation *n.foo()* stands for the function *foo()* being invoked at and executed on node n. Remote calls and variable references are preceded by the remote node identifier, while local variable references and procedure calls omit the local node. Thus *n.foo()* denotes a remote procedure call on node n, while *n.bar*, without parentheses, is an RPC to lookup a variable *bar* on node n.

find_successor works by finding the immediate predecessor node of the desired identifier; the successor of that node must be the successor of the identifier. We implement *find_predecessor* explicitly, because it is used later to implement the join operation (Section 4.4).

When node n executes *find_predecessor*, it contacts a series of nodes moving forward around the Chord circle towards id. If node n contacts a node n' such that id falls between n' and the successor of n', *find_predecessor* is done and returns n'. Otherwise node n asks n' for the node n' knows about that most closely precedes id. Thus the algorithm always makes progress towards the precedessor of id.

As an example, consider the Chord ring in Figure 3(b). Suppose node 3 wants to find the successor of identifier 1. Since 1 belongs to the circular interval $[7, 3)$, it belongs to 3.*finger*$[3]$.*interval*; node 3 therefore checks the third entry in its finger table, which is 0. Because 0 precedes 1, node 3 will ask node 0 to find the successor of 1. In turn, node 0 will infer from its finger table that 1's successor is the node 1 itself, and return node 1 to node 3.

The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in *find_predecessor* to halve the distance to the target identifier. From this intuition follows a theorem:

each node == 1 immediate successor == must be right at all time, to ensure correctness

each row of finger table of each node == 1 successor == eventual consisteny, to speed up search

finger table == distributed routing table == not 1 node has all the node's locations

finger table's start n int. columns are fixed (if node id is fixed), only successor could be any num

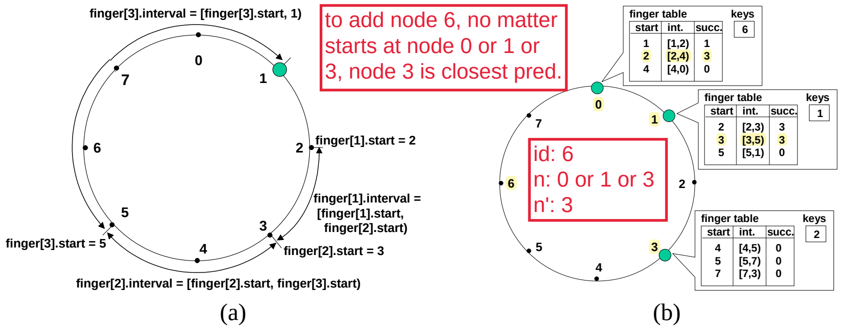
all 3 finger tables present the same consistent route !!!!!!!! only 3 nodes thats why keys are matched as such

key == hashed filename

start == mod eqn (depends on node id and row of entry)

int == mod eqn diff btwn 1st n 2nd row of table. range of keys match to succ.

[== including
) == not including
[3,5) == 3 from mod eqn on node 1 2nd row, 5 from mod eqn on 3rd row == key between 3 - 4.9999, go to successor node 3 == key 5 goes to node 0



to add node 6, no matter starts at node 0 or 1 or 3, node 3 is closest pred.

id: 6
n: 0 or 1 or 3
n': 3

Figure 3: (a) The finger intervals associated with node 1. (b) Finger tables and key locations for a net with nodes 0, 1, and 3, and keys 1, 2, and 6.

THEOREM 2. *With high probability (or under standard hardness assumptions), the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$.*

PROOF. Suppose that node n wishes to resolve a query for the successor of k. Let p be the node that immediately precedes k. We analyze the number of query steps to reach p.

Recall that if $n \neq p$, then n forwards its query to the closest predecessor of k in its finger table. Suppose that node p is in the i^{th} finger interval of node n. Then since this interval is not empty, node n will finger some node f in this interval. The distance (number of identifiers) between n and f is at least 2^{i-1}. But f and p are both in n's i^{th} finger interval, which means the distance between them is at most 2^{i-1}. This means f is closer to p than to n, or equivalently, that the distance from f to p is at most half the distance from n to p.

If the distance between the node handling the query and the predecessor p halves in each step, and is at most 2^m initially, then within m steps the distance will be one, meaning we have arrived at p.

In fact, as discussed above, we assume that node and key identifiers are random. In this case, the number of forwardings necessary will be $O(\log N)$ with high probability. After $\log N$ forwardings, the distance between the current query node and the key k will be reduced to at most $2^m/N$. The expected number of node identifiers landing in a range of this size is 1, and it is $O(\log N)$ with high probability. Thus, even if the remaining steps advance by only one node at a time, they will cross the entire remaining interval and reach key k within another $O(\log N)$ steps. □

In the section reporting our experimental results (Section 6), we will observe (and justify) that the average lookup time is $\frac{1}{2}\log N$.

4.4 Node Joins

In a dynamic network, nodes can join (and leave) at any time. The main challenge in implementing these operations is preserving the ability to locate every key in the network. To achieve this goal, Chord needs to preserve two invariants:

1. Each node's successor is correctly maintained.

2. For every key k, node $successor(k)$ is responsible for k.

In order for lookups to be fast, it is also desirable for the finger tables to be correct.

This section shows how to maintain these invariants when a single node joins. We defer the discussion of multiple nodes joining simultaneously to Section 5, which also discusses how to handle a node failure. Before describing the join operation, we summarize its performance (the proof of this theorem is in the companion technical report [21]):

```
// ask node $n$ to find $id$'s successor
$n$.find_successor($id$)
  $n' =$ find_predecessor($id$);
  return $n'$.successor;

// ask node $n$ to find $id$'s predecessor
$n$.find_predecessor($id$)
  $n' = n$;
  while ($id \notin (n', n'.successor]$)
    $n' = n'$.closest_preceding_finger($id$);
  return $n'$;

// return closest finger preceding $id$
$n$.closest_preceding_finger($id$)
  for $i = m$ downto 1
    if (finger[$i$].node $\in (n, id)$)
      return finger[$i$].node;
  return $n$;
```

Figure 4: The pseudocode to find the successor node of an identifier id. Remote procedure calls and variable lookups are preceded by the remote node.

Q: why finding predecessor == recursively hopping successors ??
A: new node's successor == new node's predecessor's successor.

Q: hop anti-clockwise node by node thru predecessor's pointer ?? why not use immediate successor of node / finger table successors ?? why predecessor ?? why successor ?? successor == clockwise next existing node in the ring (green dot, not no dot)

n == any random node (unchanged)
id == new node (unchanged)
n' == finger table successor (keeps changing while hopping)
e.g. node 0 finger table -> n: 0, id:6, n': 3 -> since successor 3 is closest id's predecessor -> node 3 is indeed 6's pred. -> hop sucesrs. til in btwn node 0 n 3

THEOREM 3. *With high probability, any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables.*

To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*. A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node, and can be used to walk counterclockwise around the identifier circle.

To preserve the invariants stated above, Chord must perform three tasks when a node n joins the network:

1. Initialize the predecessor and fingers of node n.

2. Update the fingers and predecessors of existing nodes to reflect the addition of n.

3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for.

We assume that the new node learns the identity of an existing Chord node n' by some external mechanism. Node n uses n' to

so far when we talked about search, there is only successors. predecessor is for finger table freshness.

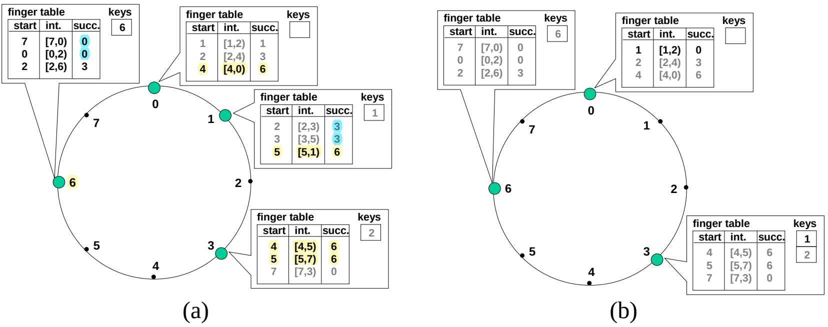
each node 1 immediate predecessor of node, 1 immediate successor of node, but many successors of key, 1 for each key if your previous entries' successor == current entry start -> then 2 entries same successor

sucessors updated from 0 -> 6 on all finger tables

after node 6 joins

all 4 tables need to be updated for a new globally consistent route == on all other 3 tables, only column successor was updated !!!

grey == last round entry, unchanged
black == new entry after node 6
m == max no. of nodes in ring ??



(a) node 6 — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

(a) node 0 — keys: (none)

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

(a) node 1 — keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

(a) node 3 — keys: 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

(b) node 6 — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

(b) node 0 — keys: (none)

start	int.	succ.
1	[1,2)	0
2	[2,4)	3
4	[4,0)	6

(b) node 3 — keys: 1, 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

Figure 5: (a) Finger tables and key locations after node 6 joins. (b) Finger tables and key locations after node 3 leaves. Changed entries are shown in black, and unchanged in gray.

Q: i still dont get hows finger table seeded by help of any node ??
A: treat "start" of each row as a new node joining looking for successor
seeding finger table == a sequence of new nodes joining

initialize its state and add itself to the existing Chord network, as follows.

Initializing fingers and predecessor: Node n learns its predecessor and fingers by asking n' to look them up, using the *init_finger_table* pseudocode in Figure 6. Naively performing *find_successor* for each of the m finger entries would give a runtime of $O(m \log N)$. To reduce this, n checks whether the i^{th} finger is also the correct $(i+1)^{th}$ finger, for each i. This happens when *finger*$[i]$.*interval* does not contain any node, and thus *finger*$[i]$.*node* $\geq$ *finger*$[i+1]$.*start*. It can be shown that the change reduces the expected (and high probability) number of finger entries that must be looked up to $O(\log N)$, which reduces the overall time to $O(\log^2 N)$.

As a practical optimization, a newly joined node n can ask an immediate neighbor for a copy of its complete finger table and its predecessor. n can use the contents of these tables as hints to help it find the correct values for its own tables, since n's tables will be similar to its neighbors'. This can be shown to reduce the time to fill the finger table to $O(\log N)$.

Updating fingers of existing nodes: Node n will need to be entered into the finger tables of some existing nodes. For example, in Figure 5(a), node 6 becomes the third finger of nodes 0 and 1, and the first and the second finger of node 3.

Figure 6 shows the pseudocode of the *update_finger_table* function that updates existing finger tables. Node n will become the i^{th} finger of node p if and only if (1) p precedes n by at least 2^{i-1}, and (2) the i^{th} finger of node p succeeds n. The first node, p, that can meet these two conditions is the immediate predecessor of $n - 2^{i-1}$. Thus, for a given n, the algorithm starts with the i^{th} finger of node n, and then continues to walk in the counter-clock-wise direction on the identifier circle until it encounters a node whose i^{th} finger precedes n.

We show in the technical report [21] that the number of nodes that need to be updated when a node joins the network is $O(\log N)$ with high probability. Finding and updating these nodes takes $O(\log^2 N)$ time. A more sophisticated scheme can reduce this time to $O(\log N)$; however, we do not present it as we expect implementations to use the algorithm of the following section.

Transferring keys: The last operation that has to be performed when a node n joins the network is to move responsibility for all the keys for which node n is now the successor. Exactly what this entails depends on the higher-layer software using Chord, but typically it would involve moving the data associated with each key to the new node. Node n can become the successor only for keys that were previously the responsibility of the node immediately follow-

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);


// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

update_others() == thru reverse reference ??

m == no. entries in finger table
n == current node to update its finger table
i == each finger table entry
s == new node 6
p == predecessor node (keeps updating)
finger[i].node == old successor

s in_set_of([n, finger[i].node]) == if node 6 is closer than old successor node 0

Figure 6: Pseudocode for the node join operation.

problem of concurrent join == each node needs global update, tho there is 1 true globally route, its slow as many overwrites happen. if look up happens in the midst of updates where theres an inconsistent route, it will route to wrong file node
successor of node alone can guarantee correctness == each hop 1 node each time, slow
successor of keys is to speed up

ing n, so n only needs to contact that one node to transfer responsibility for all relevant keys.

5. Concurrent Operations and Failures

In practice Chord needs to deal with nodes joining the system concurrently and with nodes that fail or leave voluntarily. This section describes modifications to the basic Chord algorithms described in Section 4 to handle these situations.

5.1 Stabilization

The join algorithm in Section 4 aggressively maintains the finger tables of all nodes as the network evolves. Since this invariant is difficult to maintain in the face of concurrent joins in a large network, we separate our correctness and performance goals. A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allows these lookups to be fast as well as correct.

If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors. The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in $O(\log N)$ steps. The second case is where successor pointers are correct, but fingers are inaccurate. This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail. The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause. This pause can be short, since stabilization fixes successor pointers quickly.

Our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages. Stabilization by itself won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space. These pathological cases cannot be produced by any sequence of ordinary node joins. It is unclear whether they can be produced by network partitions and recoveries or intermittent failures. If produced, these cases could be detected and repaired by periodic sampling of the ring topology.

Figure 7 shows the pseudo-code for joins and stabilization; this code replaces Figure 6 to handle concurrent joins. When node n first starts, it calls $n.join(n')$, where n' is any known Chord node. The $join$ function asks n' to find the immediate successor of n. By itself, $join$ does not make the rest of the network aware of n.

but why not make it aware ??

Every node runs *stabilize* periodically (this is how newly joined nodes are noticed by the network). When node n runs *stabilize*, it asks n's successor for the successor's predecessor p, and decides whether p should be n's successor instead. This would be the case if node p recently joined the system. *stabilize* also notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n.

As a simple example, suppose node n joins the system, and its ID lies between nodes n_p and n_s. n would acquire n_s as its successor. Node n_s, when notified by n, would acquire n as its predecessor. When n_p next runs *stabilize*, it will ask n_s for its predecessor (which is now n); n_p would then acquire n as its successor. Finally, n_p will notify n, and n will acquire n_p as its predecessor. At this point, all predecessor and successor pointers are correct.

since only need successor of node to guarantee correctness, "stabilize" !!
"stabilize" == regular ping to immediate successor node to see if any new node get in between themselves.

predecessor node was not notified during new node join, find out about new node from successor during regular ping

```
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
    successor = x;
  successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';

// periodically refresh finger table entries.
n.fix_fingers()
  i = random index > 1 into finger[];
  finger[i].node = find_successor(finger[i].start);
```

Figure 7: Pseudocode for stabilization.

n == new node
n' == any chord node
x == newer than new node, potential between new node n its successor
m == total no. of finger entries

As soon as the successor pointers are correct, calls to *find_predecessor* (and thus *find_successor*) will work. Newly joined nodes that have not yet been fingered may cause *find_predecessor* to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor (*finger*[1]) pointers through the newly joined nodes until the correct predecessor is reached. Eventually *fix_fingers* will adjust finger table entries, eliminating the need for these linear scans.

The following theorems (proved in the technical report [21]) show that all problems caused by concurrent joins are transient. The theorems assume that any two nodes trying to communicate will eventually succeed.

THEOREM 4. *Once a node can successfully resolve a given query, it will always be able to do so in the future.*

THEOREM 5. *At some time after the last join all successor pointers will be correct.*

The proofs of these theorems rely on an invariant and a termination argument. The invariant states that once node n can reach node r via successor pointers, it always can. To argue termination, we consider the case where two nodes both think they have the same successor s. In this case, each will attempt to notify s, and s will eventually choose the closer of the two (or some other, closer node) as its predecessor. At this point the farther of the two will, by contacting s, learn of a better successor than s. It follows that every node progresses towards a better and better successor over time. This progress must eventually halt in a state where every node is considered the successor of exactly one other node; this defines a cycle (or set of them, but the invariant ensures that there will be at most one).

We have not discussed the adjustment of fingers when nodes join because it turns out that joins don't substantially damage the performance of fingers. If a node has a finger into each interval, then these fingers can still be used even after joins. The distance halving argument is essentially unchanged, showing that $O(\log N)$ hops suffice to reach a node "close" to a query's target. New joins influence the lookup only by getting in between the old predecessor and successor of a target query. These new nodes may need to be scanned linearly (if their fingers are not yet accurate). But unless a

new node notify successor by jumping through hops in finger table

why didnt new node just ask successor for predecessor as theres reverse reference ?? or successor ping predecessor after new node's notification ?? it would be much quicker than wait for predecessor to ping ??

tremendous number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible. Formally, we can state the following:

THEOREM 6. *If we take a stable network with N nodes, and another set of up to N nodes joins the network with no finger pointers (but with correct successor pointers), then lookups will still take $O(\log N)$ time with high probability.*

More generally, so long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups should continue to take $O(\log N)$ hops.

5.2 Failures and Replication

When a node n fails, nodes whose finger tables include n must find n's successor. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is re-stabilizing.

The key step in failure recovery is maintaining correct successor pointers, since in the worst case *find_predecessor* can make progress using only successors. To help achieve this, each Chord node maintains a "successor-list" of its r nearest successors on the Chord ring. In ordinary operation, a modified version of the *stabilize* routine in Figure 7 maintains the successor-list. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list. At that point, n can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, *stabilize* will correct finger table entries and successor-list entries pointing to the failed node.

After a node failure, but before stabilization has completed, other nodes may attempt to send requests through the failed node as part of a *find_successor* lookup. Ideally the lookups would be able to proceed, after a timeout, by another path despite the failure. In many cases this is possible. All that is needed is a list of alternate nodes, easily found in the finger table entries preceding that of the failed node. If the failed node had a very low finger table index, nodes in the successor-list are also available as alternates.

The technical report proves the following two theorems that show that the successor-list allows lookups to succeed, and be efficient, even during stabilization [21]:

THEOREM 7. *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key.*

THEOREM 8. *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then the expected time to execute find_successor in the failed network is $O(\log N)$.*

The intuition behind these proofs is straightforward: a node's r successors all fail with probability $2^{-r} = 1/N$, so with high probability a node will be aware of, so able to forward messages to, its closest living successor.

The successor-list mechanism also helps higher layer software replicate data. A typical application using Chord might store replicas of the data associated with a key at the k nodes succeeding the key. The fact that a Chord node keeps track of its r successors means that it can inform the higher layer software when successors come and go, and thus when the software should propagate new replicas.

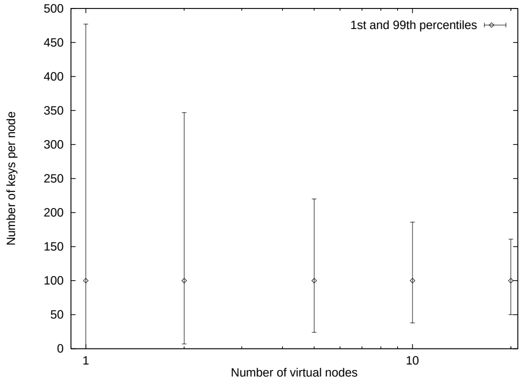


Figure 9: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

6. Simulation and Experimental Results

In this section, we evaluate the Chord protocol by simulation. The simulator uses the lookup algorithm in Figure 4 and a slightly older version of the stabilization algorithms described in Section 5. We also report on some preliminary experimental results from an operational Chord-based system running on Internet hosts.

6.1 Protocol Simulator

The Chord protocol can be implemented in an *iterative* or *recursive* style. In the iterative style, a node resolving a lookup initiates all communication: it asks a series of nodes for information from their finger tables, each time moving closer on the Chord ring to the desired successor. In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor. The simulator implements the protocols in an iterative style.

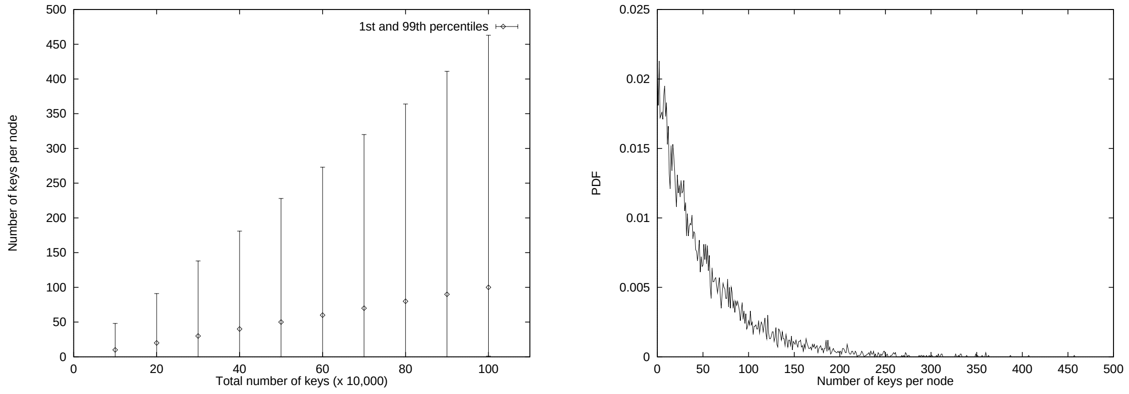
6.2 Load Balance

We first consider the ability of consistent hashing to allocate keys to nodes evenly. In a network with N nodes and K keys we would like the distribution of keys to nodes to be tight around N/K.

We consider a network consisting of 10^4 nodes, and vary the total number of keys from 10^5 to 10^6 in increments of 10^5. For each value, we repeat the experiment 20 times. Figure 8(a) plots the mean and the 1st and 99th percentiles of the number of keys per node. The number of keys per node exhibits large variations that increase linearly with the number of keys. For example, in all cases some nodes store no keys. To clarify this, Figure 8(b) plots the probability density function (PDF) of the number of keys per node when there are 5×10^5 keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or $9.1\times$ the mean value. For comparison, the 99th percentile is $4.6\times$ the mean value.

One reason for these variations is that node identifiers do not uniformly cover the entire identifier space. If we divide the identifier space in N equal-sized bins, where N is the number of nodes, then we might hope to see one node in each bin. But in fact, the probability that a particular bin does not contain any node is $(1-1/N)^N$. For large values of N this approaches $e^{-1} = 0.368$.

As we discussed earlier, the consistent hashing paper solves this problem by associating keys with virtual nodes, and mapping multiple virtual nodes (with unrelated identifiers) to each real node. Intuitively, this will provide a more uniform coverage of the identifier space. For example, if we allocate $\log N$ randomly chosen virtual nodes to each real node, with high probability each of the



(a)

(b)

Figure 8: (a) The mean and 1st and 99th percentiles of the number of keys stored per node in a 10^4 node network. (b) The probability density function (PDF) of the number of keys per node. The total number of keys is 5×10^5.

N bins will contain $O(\log N)$ nodes [16]. We note that this does not affect the worst-case query path length, which now becomes $O(\log(N \log N)) = O(\log N)$.

To verify this hypothesis, we perform an experiment in which we allocate r virtual nodes to each real node. In this case keys are associated to virtual nodes instead of real nodes. We consider again a network with 10^4 real nodes and 10^6 keys. Figure 9 shows the 1st and 99th percentiles for $r = 1, 2, 5, 10$, and 20, respectively. As expected, the 99th percentile decreases, while the 1st percentile increases with the number of virtual nodes, r. In particular, the 99th percentile decreases from $4.8\times$ to $1.6\times$ the mean value, while the 1st percentile increases from 0 to $0.5\times$ the mean value. Thus, adding virtual nodes as an indirection layer can significantly improve load balance. The tradeoff is that routing table space usage will increase as each actual node now needs r times as much space to store the finger tables for its virtual nodes. However, we believe that this increase can be easily accommodated in practice. For example, assuming a network with $N = 10^6$ nodes, and assuming $r = \log N$, each node has to maintain a table with only $\log^2 N \simeq 400$ entries.

6.3 Path Length

The performance of any routing protocol depends heavily on the length of the path between two arbitrary nodes in the network. In the context of Chord, we define the path length as the number of nodes traversed during a lookup operation. From Theorem 2, with high probability, the length of the path to resolve a query is $O(\log N)$, where N is the total number of nodes in the network.

To understand Chord's routing performance in practice, we simulated a network with $N = 2^k$ nodes, storing 100×2^k keys in all. We varied k from 3 to 14 and conducted a separate experiment for each value. Each node in an experiment picked a random set of keys to query from the system, and we measured the path length required to resolve each query.

Figure 10(a) plots the mean, and the 1st and 99th percentiles of path length as a function of k. As expected, the mean path length increases logarithmically with the number of nodes, as do the 1st and 99th percentiles. Figure 10(b) plots the PDF of the path length for a network with 2^{12} nodes ($k = 12$).

Figure 10(a) shows that the path length is about $\frac{1}{2}\log_2 N$. The reason for the $\frac{1}{2}$ is as follows. Consider some random node and a random query. Let the distance in identifier space be considered in binary representation. The most significant (say i^{th}) bit of this distance can be corrected to 0 by following the node's i^{th} finger. If the next significant bit of the distance is 1, it too needs to be corrected by following a finger, but if it is 0, then no $i - 1^{st}$ finger is followed—instead, we move on the the $i - 2^{nd}$ bit. In general, the number of fingers we need to follow will be the number of ones in the binary representation of the distance from node to query. Since the distance is random, we expect half the $\log N$ bits to be ones.

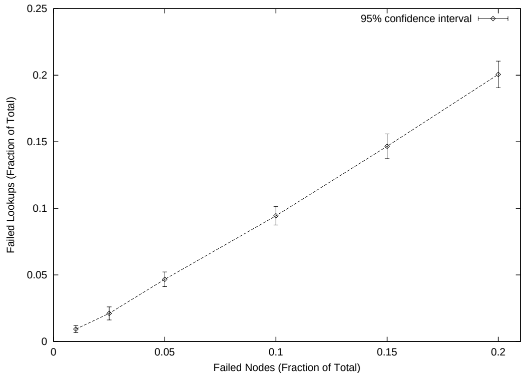
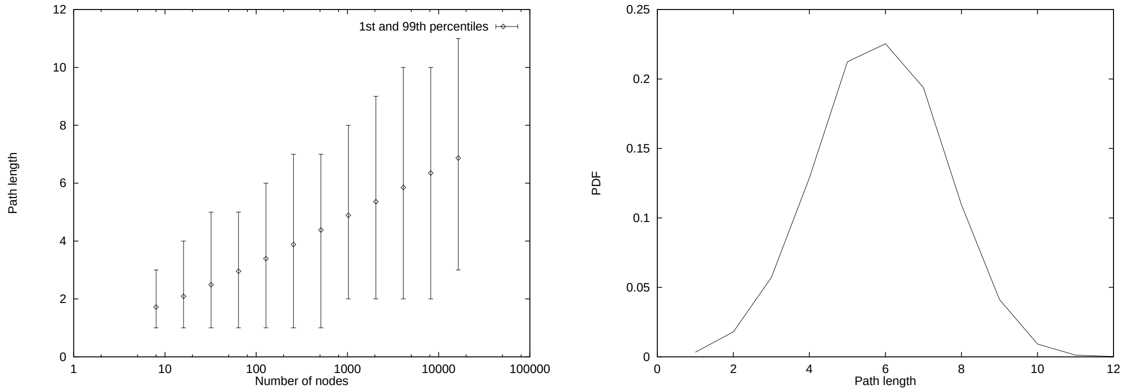


Figure 11: The fraction of lookups that fail as a function of the fraction of nodes that fail.

6.4 Simultaneous Node Failures

In this experiment, we evaluate the ability of Chord to regain consistency after a large percentage of nodes fail simultaneously. We consider again a 10^4 node network that stores 10^6 keys, and randomly select a fraction p of nodes that fail. After the failures occur, we wait for the network to finish stabilizing, and then measure the fraction of keys that could not be looked up correctly. A correct lookup of a key is one that finds the node that was originally responsible for the key, before the failures; this corresponds to a system that stores values with keys but does not replicate the values or recover them after failures.

Figure 11 plots the mean lookup failure rate and the 95% confidence interval as a function of p. The lookup failure rate is almost exactly p. Since this is just the fraction of keys expected to be lost due to the failure of the responsible nodes, we conclude that there is no significant lookup failure in the Chord network. For example, if the Chord network had partitioned in two equal-sized halves, we



(a)

(b)

Figure 10: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

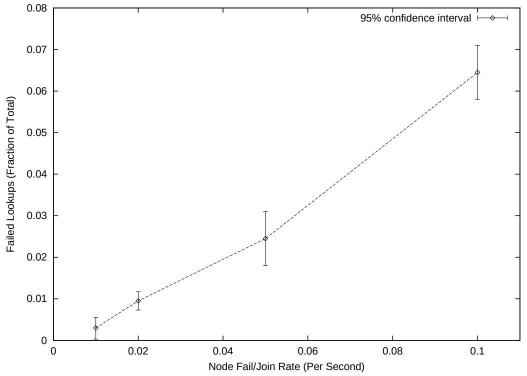


Figure 12: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.

would expect one-half of the requests to fail because the querier and target would be in different partitions half the time. Our results do not show this, suggesting that Chord is robust in the face of multiple simultaneous node failures.

6.5 Lookups During Stabilization

A lookup issued after some failures but before stabilization has completed may fail for two reasons. First, the node responsible for the key may have failed. Second, some nodes' finger tables and predecessor pointers may be inconsistent due to concurrent joins and node failures. This section evaluates the impact of continuous joins and failures on lookups.

In this experiment, a lookup is considered to have succeeded if it reaches the current successor of the desired key. This is slightly optimistic: in a real system, there might be periods of time in which the real successor of a key has not yet acquired the data associated with the key from the previous successor. However, this method allows us to focus on Chord's ability to perform lookups, rather than on the higher-layer software's ability to maintain consistency of its own data. Any query failure will be the result of inconsistencies in Chord. In addition, the simulator does not retry queries: if a query is forwarded to a node that is down, the query simply fails. Thus, the results given in this section can be viewed as the worst-case scenario for the query failures induced by state inconsistency.

Because the primary source of inconsistencies is nodes joining and leaving, and because the main mechanism to resolve these inconsistencies is the stabilize protocol, Chord's performance will be sensitive to the frequency of node joins and leaves versus the frequency at which the stabilization protocol is invoked.

In this experiment, key lookups are generated according to a Poisson process at a rate of one per second. Joins and failures are modeled by a Poisson process with the mean arrival rate of R. Each node runs the stabilization routines at randomized intervals averaging 30 seconds; unlike the routines in Figure 7, the simulator updates all finger table entries on every invocation. The network starts with 500 nodes.

Figure 12 plots the average failure rates and confidence intervals. A node failure rate of 0.01 corresponds to one node joining and leaving every 100 seconds on average. For comparison, recall that each node invokes the stabilize protocol once every 30 seconds. In other words, the graph x axis ranges from a rate of 1 failure per 3 stabilization steps to a rate of 3 failures per one stabilization step. The results presented in Figure 12 are averaged over approximately two hours of simulated time. The confidence intervals are computed over 10 independent runs.

The results of figure 12 can be explained roughly as follows. The simulation has 500 nodes, meaning lookup path lengths average around 5. A lookup fails if its finger path encounters a failed node. If k nodes fail, the probability that one of them is on the finger path is roughly $5k/500$, or $k/100$. This would suggest a failure rate of about 3% if we have 3 failures between stabilizations. The graph shows results in this ball-park, but slightly worse since it might take more than one stabilization to completely clear out a failed node.

6.6 Experimental Results

This section presents latency measurements obtained from a prototype implementation of Chord deployed on the Internet. The Chord nodes are at ten sites on a subset of the RON test-bed in the United States [1], in California, Colorado, Massachusetts, New York, North Carolina, and Pennsylvania. The Chord software runs on UNIX, uses 160-bit keys obtained from the SHA-1 cryptographic hash function, and uses TCP to communicate between nodes. Chord runs in the iterative style. These Chord nodes are part of an experimental distributed file system [7], though this section considers only the Chord component of the system.

Figure 13 shows the measured latency of Chord lookups over a range of numbers of nodes. Experiments with a number of nodes larger than ten are conducted by running multiple independent

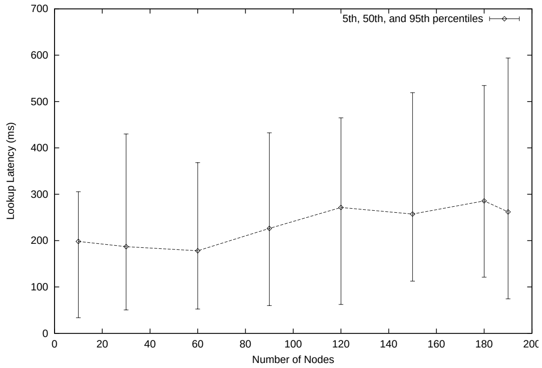


Figure 13: Lookup latency on the Internet prototype, as a function of the total number of nodes. Each of the ten physical sites runs multiple independent copies of the Chord node software.

copies of the Chord software at each site. This is different from running $O(\log N)$ virtual nodes at each site to provide good load balance; rather, the intention is to measure how well our implementation scales even though we do not have more than a small number of deployed nodes.

For each number of nodes shown in Figure 13, each physical site issues 16 Chord lookups for randomly chosen keys one-by-one. The graph plots the median, the 5th, and the 95th percentile of lookup latency. The median latency ranges from 180 to 285 ms, depending on number of nodes. For the case of 180 nodes, a typical lookup involves five two-way message exchanges: four for the Chord lookup, and a final message to the successor node. Typical round-trip delays between sites are 60 milliseconds (as measured by `ping`). Thus the expected lookup time for 180 nodes is about 300 milliseconds, which is close to the measured median of 285. The low 5th percentile latencies are caused by lookups for keys close (in ID space) to the querying node and by query hops that remain local to the physical site. The high 95th percentiles are caused by lookups whose hops follow high delay paths.

The lesson from Figure 13 is that lookup latency grows slowly with the total number of nodes, confirming the simulation results that demonstrate Chord's scalability.

7. Future Work

Based on our experience with the prototype mentioned in Section 6.6, we would like to improve the Chord design in the following areas.

Chord currently has no specific mechanism to heal partitioned rings; such rings could appear locally consistent to the stabilization procedure. One way to check global consistency is for each node n to periodically ask other nodes to do a Chord lookup for n; if the lookup does not yield node n, there may be a partition. This will only detect partitions whose nodes know of each other. One way to obtain this knowledge is for every node to know of the same small set of initial nodes. Another approach might be for nodes to maintain long-term memory of a random set of nodes they have encountered in the past; if a partition forms, the random sets in one partition are likely to include nodes from the other partition.

A malicious or buggy set of Chord participants could present an incorrect view of the Chord ring. Assuming that the data Chord is being used to locate is cryptographically authenticated, this is a threat to availability of data rather than to authenticity. The same approach used above to detect partitions could help victims realize that they are not seeing a globally consistent view of the Chord ring.

An attacker could target a particular data item by inserting a node into the Chord ring with an ID immediately following the item's key, and having the node return errors when asked to retrieve the data. Requiring (and checking) that nodes use IDs derived from the SHA-1 hash of their IP addresses makes this attack harder.

Even $\log N$ messages per lookup may be too many for some applications of Chord, especially if each message must be sent to a random Internet host. Instead of placing its fingers at distances that are all powers of 2, Chord could easily be changed to place its fingers at distances that are all integer powers of $1 + 1/d$. Under such a scheme, a single routing hop could decrease the distance to a query to $1/(1+d)$ of the original distance, meaning that $\log_{1+d} N$ hops would suffice. However, the number of fingers needed would increase to $\log N/(\log(1 + 1/d) \approx O(d \log N)$.

A different approach to improving lookup latency might be to use server selection. Each finger table entry could point to the first k nodes in that entry's interval on the ID ring, and a node could measure the network delay to each of the k nodes. The k nodes are generally equivalent for purposes of lookup, so a node could forward lookups to the one with lowest delay. This approach would be most effective with recursive Chord lookups, in which the node measuring the delays is also the node forwarding the lookup.

8. Conclusion

Many distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. It offers a powerful primitive: given a key, it determines the node responsible for storing the key's value, and does so efficiently. In the steady state, in an N-node network, each node maintains routing information for only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Updates to the routing information for nodes leaving and joining require only $O(\log^2 N)$ messages.

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct. Our theoretical analysis, simulations, and experimental results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

We believe that Chord will be a valuable component for peer-to-peer, large-scale distributed applications such as cooperative file sharing, time-shared available storage systems, distributed indices for document and service discovery, and large-scale distributed computing platforms.

Acknowledgments

We thank Frank Dabek for the measurements of the Chord prototype described in Section 6.6, and David Andersen for setting up the testbed used in those measurements.

9. References

[1] ANDERSEN, D. Resilient overlay networks. Master's thesis, Department of EECS, MIT, May 2001. `http://nms.lcs.mit.edu/projects/ron/`.

[2] BAKKER, A., AMADE, E., BALLINTIJN, G., KUZ, I., VERKAIK, P., VAN DER WIJK, I., VAN STEEN, M., AND TANENBAUM., A.

The Globe distribution network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)* (San Diego, CA, June 2000), pp. 141–152.

[3] Chen, Y., Edler, J., Goldberg, A., Gottlieb, A., Sobti, S., and Yianilos, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.

[4] Clarke, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[5] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). `http://freenet.sourceforge.net`.

[6] Dabek, F., Brunskill, E., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I., and Balakrishnan, H. Building peer-to-peer systems with Chord, a distributed location service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Elmau/Oberbayern, Germany, May 2001), pp. 71–76.

[7] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (To appear; Banff, Canada, Oct. 2001).

[8] Druschel, P., and Rowstron, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)* (Elmau/Oberbayern, Germany, May 2001), pp. 65–70.

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

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

[11] 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.

[12] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.

[13] 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/`.

[14] Li, J., Jannotti, J., De Couto, D., Karger, D., and Morris, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking* (Boston, Massachusetts, August 2000), pp. 120–130.

[15] Mockapetris, P., and Dunlap, K. J. Development of the Domain Name System. In *Proc. ACM SIGCOMM* (Stanford, CA, 1988), pp. 123–133.

[16] Motwani, R., and Raghavan, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

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

[18] Ohaha, Smart decentralized peer-to-peer sharing. `http://www.ohaha.com/design.html`.

[19] Plaxton, C., Rajaraman, R., and Richa, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.

[20] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001).

[21] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. Tech. Rep. TR-819, MIT LCS, March 2001. `http://www.pdos.lcs.mit.edu/chord/papers/`.

[22] van Steen, M., Hauck, F., Ballintijn, G., and Tanenbaum, A. Algorithmic design of the Globe wide-area location service. *The Computer Journal 41*, 5 (1998), 297–310.