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

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

January 10, 2002

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 and simulations 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

*University of California, Berkeley. istoica@cs.berkeley.edu

†MIT Laboratory for Computer Science, {rtm, dln, karger, kaashoek, fdabek, hari}@lcs.mit.edu.

Authors in reverse alphabetical order.

<chord@lcs.mit.edu>; <http://pdos.lcs.mit.edu/chord/>.

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.

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. Section 6 presents simulations supporting our claims about Chord's performance. 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 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 [21]. 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 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

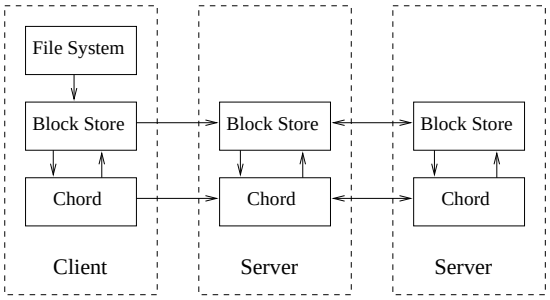


Figure 1: Structure of an example Chord-based distributed storage system.

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 Chord Protocol

This section describes the 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.

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 desirable 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.

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.

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

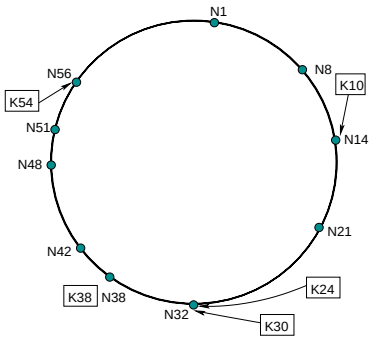


Figure 2: An identifier circle consisting of 10 nodes storing five keys.

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 $m = 6$. The identifier circle has 10 nodes and stores five keys. The successor of identifier 10 is node 14, so key 10 would be located at node 1. Similarly, keys 24 and 30 would be located at node 32, key 38 at node 38, and key 54 at node 56.

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 26, it would capture the key with identifier 24 from the node with identifier 32.

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

Theorem 4.1. *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 non-random 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 Simple Key Location

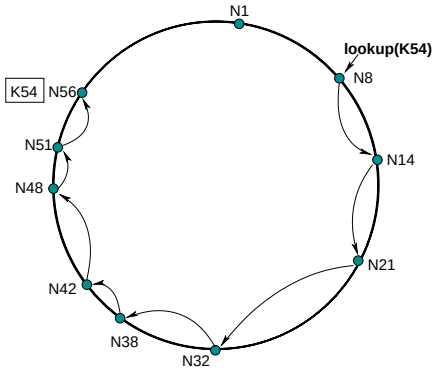
This section describes a simple but slow Chord lookup algorithm. Succeeding sections will describe how to extend the basic algorithm to increase efficiency, and how to maintain the correctness of Chord's routing information.

Lookups could be implemented on a Chord ring with little per-node state. Each node need only know how to contact its current successor node on the identifier circle. Queries for a given identifier could be passed around the circle via these successor pointers until they encounter a pair of nodes that straddle the desired identifier; the second in the pair is the node the query maps to.

The pseudo-code that implements the query process in this case is shown in Figure 3(a). 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 to node n, while *n.bar*, without parentheses, is an RPC to lookup a variable *bar* on node n.

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

(a)



(b)

Figure 3: (a) Pseudo-code to find the successor node of an identifier *id*. Remote procedure calls and variable lookups are preceded by the remote node. (b) The path taken by a query from node 8 for key 54, using the pseudo-code in Figure 3(a).

Figure 3(b) shows an example in which node 8 performs a lookup for key 54. Node 8 invokes $find\_successor$ for key 54 which eventually returns the successor of that key, node 56. The query visits every node on the circle between nodes 8 and 56. The result returns along the reverse of the path followed by the query.

4.4 Scalable Key Location

The lookup scheme presented in the previous section uses a number of messages linear in the number of nodes. To accelerate lookups, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as each node knows its correct successor.

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 $n.finger[i]$ (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 the immediate successor of n on the circle; for convenience we often refer to the first finger as the *successor*.

The example in Figure 4(a) shows the finger table of node 8. The first finger of node 8 points to node 14, as node 14 is the first node that succeeds $(8 + 2^0) \bmod 2^6 = 9$. Similarly, the last finger of node 8 points to node 42, as node 42 is the first node that succeeds $(8 + 2^5) \bmod 2^6 = 40$.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about

Notation	Definition
finger[k]	first node on circle that succeeds ($n+2^{k-1}$) mod 2^m, $1 \leq k \leq m$
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.

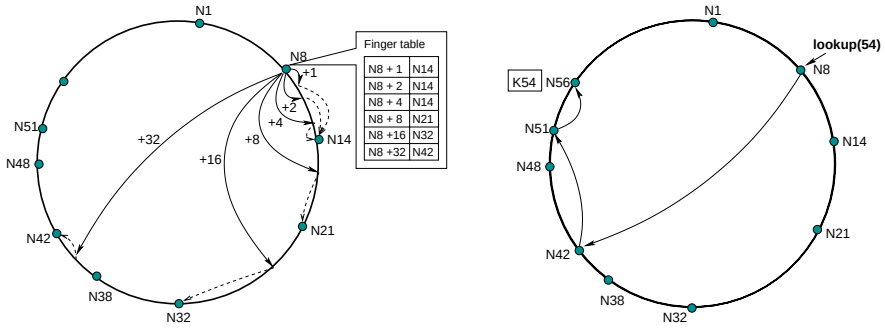


Figure 4: (a) The finger table entries for node 8. (b) The path a query for key 54 starting at node 8, using the algorithm in Figure 5.

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 directly determine the successor of an arbitrary key k. For example, node 8 in Figure 4(a) cannot determine the successor of key 34 by itself, as this successor (node 38) does not appear in node 8's finger table.

Figure 5 shows the pseudo-code of the *find_successor* operation, extended to use finger tables. If id falls between n and n's successor, *find_successor* is done and node n returns its successor. Otherwise, n searches its finger table for the node n' whose ID most immediately precedes id, and then invokes *find_successor* at n'. The reason behind this choice of n' is that the closer n' is to id, the more it will know about the identifier circle in the region of id.

As an example, consider the Chord circle in Figure 4(b), and suppose node 8 wants to find the successor of key 54. Since the largest finger of node 8 that precedes 54 is node 42, node 8 will ask node 42 to resolve the query. In turn, node 42 will determine the largest finger in its finger table that precedes 54, i.e., node 51. Finally, node 51 will find out that its own successor, node 56, succeeds key 54, and thus will return node 56 to node 8.

Since each node has finger entries at power-of-two intervals around the iden-

```
// ask node n to find the successor of id
n.find_successor(id)
    if (key ∈ (n, n.successor])
        return n.successor;
    else
        n' = closest_preceding_node(id);
        return n'.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] ∈ (n, id))
            return finger[i];
    return n;
```

Figure 5: Scalable key lookup using the finger table.

tifier circle, each node can forward a query at least half way along the remaining distance between the node and the target identifier. From this intuition follows a theorem:

Theorem 4.2. *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 $\frac{1}{2}\log N$.

4.5 Dynamic Operations and Failures

In practice, Chord needs to deal with nodes joining the system and with nodes that fail or leave voluntarily. This section describes how Chord handles these situations.

4.5.1 Node Joins and Stabilization

In order to ensure that lookups execute correctly as the set of participating nodes changes, Chord must ensure that each node's successor pointer is up to date. It does this using a basic "stabilization" protocol. Chord verifies and updates finger table entries using a combination of existing (and possibly out-of-date) fingers and corrected successor pointers.

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. This stabilization protocol 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. We discuss the latter case in Section 5.3. 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.

Figure 6 shows the pseudo-code for joins and stabilization. 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.

Every node runs $stab()$ periodically. This is how nodes in the system learn about newly joined nodes. When node n runs $stab()$, it asks its 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. Also $stab()$ 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. In its call to $join()$, n acquires n_s as its successor. In addition, n copies all keys with IDs larger or equal to its ID from n_s. Node n_s, when

```
// ask n' to build n's finger table.
n.build_fingers(n')
   i_0 := ⌊log(successor − n)⌋ + 1;   // first non-trivial finger.
  for each i ≥ i_0 index into finger[];
    finger[i] = n'.find_successor(n + 2^(i−1));

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

```
// 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';
```

Figure 6: Pseudocode for stabilization.

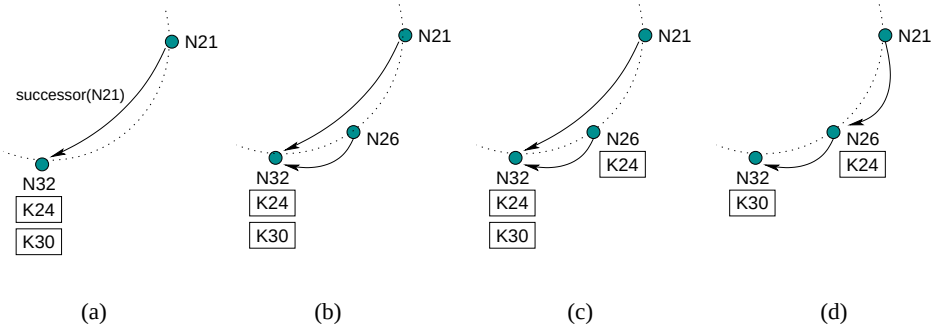


Figure 7: Example illustrating the join operation. Node 26 joins the system between nodes 21 and 32. The arcs represent the successor relationship. (a) Initial state: node 21 points to node 32; (b) node 26 finds its successor (i.e., node 32) and points to it; (c) node 26 copies all keys between 26 and 31 from node 32; (d) stabilize procedure updates the successor of node 21 to node 26.

notified by n, would acquire n as its predecessor. When n_p next runs *stab*(), 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. Figure 7 illustrates the join procedure, when n's ID is 26, and the IDs of n_s and n_p are 21 and 32, respectively.

As soon as the successor pointers are correct, calls to *find_predecessor*() 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 $ffs()$ will adjust finger table entries, eliminating the need for these linear scans.

The following result, proved below, shows that the inconsistent state caused

by concurrent joins is transient.

Theorem 4.3. *If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network.*

In other words, after some time each node is able to reach any other node in the network by following successor pointers.

4.5.2 Impact of Node Joins on Lookup Performance

In this section, we consider the impact of node joins on lookup performance. Once stabilization has completed, the new nodes will have no effect beyond increasing the N in the $O(\log N)$ lookup time. If stabilization has not yet completed, existing nodes' finger table entries may not reflect the new nodes. The ability of finger entries to carry queries long distances around the identifier ring does not depend on exactly which nodes the entries point to; the distance halving argument depends only on ID-space distance. Thus the fact that finger table entries may not reflect new nodes does not significantly affect lookup speed. The main way in which newly joined nodes can influence a lookup's speed is if the new nodes' IDs are between the target's predecessor and the target. In that case the lookup will have to be forwarded through the intervening nodes, one at a time. But unless a 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 result:

Theorem 4.4. *If we take a stable network with N nodes, and another set of up to N nodes joins the network, and all successor pointers (but perhaps not all finger pointers) are correct, then lookups will still take $O(\log N)$ time with high probability.*

Proof. The original set of fingers will, in $O(\log N)$ time, bring the query to the old predecessor of the correct node. With high probability, at most $O(\log N)$ new nodes will land between any two old nodes. So only $O(\log N)$ new nodes will need to be traversed along successor pointers to get from the old predecessor to the new predecessor. □

More generally, as long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups will continue to take $O(\log N)$ hops. We can achieve such adjustment by repeatedly carrying out lookups to update our fingers. It follows that lookups perform well so long as $\log^2 N$ rounds of stabilization happen between any N node joins.

4.5.3 Failure and Replication

The correctness of the Chord protocol relies on the fact that each node knows its successor. However, this invariant can be compromised if nodes fail. For example, in Figure 4, if nodes 14, 21, and 32 fail simultaneously, node 8 will not

know that node 38 is now its successor, since it has no finger pointing to 38. An incorrect successor will lead to incorrect lookups. Consider a query for key 30 addressed to node 8. Node 8 will return node 42, the first node it knows about from its finger table, instead of the correct successor, node 38.

To increase robustness, each Chord node maintains a *successor list* of size r, containing the node's first r successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r. An implementation should use a fixed r, chosen to be $2 \log_2 N$ for the foreseeable maximum number of nodes N.

Handling the successor list requires minor changes in the pseudo-code in Figures 5 and 6. A modified version of the *stabilize* procedure in Figure 6 maintains the successor list. Successor lists are stabilized as follows: node u reconciles its list with its successor s by copying s's list ℓ, adding s the front of ℓ, and deleting the last element. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor. 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.

A modified version of the *closest_preceding_node* procedure in Figure 5 searches not only the finger table but also the successor list for the most immediate predecessor of *id*. In addition, the pseudo-code needs to be enhanced to handle node failures. If a node fails during the *find_successor* procedure, the lookup proceeds, after a timeout, by trying the next best predecessor among the nodes in the finger table and the successor list.

The following results quantify the robustness of the Chord protocol, by showing that neither the success nor the performance of Chord lookups is likely to be affected even by massive simultaneous failures. Both theorems assume that the successor list has length $r = O(\log N)$. A Chord ring is *stable* if every node's successor list is correct.

Theorem 4.5. *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.*

Proof. Before the failures, each node was aware of its r immediate successors. The probability that all of these successors fail is $(1/2)^r$, so with high probability every node is aware of its immediate living successor. As was argued in the previous section, if the invariant that every node is aware of its immediate successor holds, then all queries are routed properly, since every node except the immediate predecessor of the query has at least one better node to which it will forward the query. □

Theorem 4.6. *In a network that is initially stable, if every node then fails with*

probability 1/2, then the expected time to execute find_successor is $O(\log N)$.

Proof. We consider the expected time for a query to move from a node that has the key in its i^{th} finger interval to a node that has the key in its $(i-1)^{st}$ finger interval. We show that this expectation is $O(1)$. Summing these expectations over all i, we find that the time to drop from the m^{th} finger interval to the $(m - \log N)^{th}$ finger interval is $O(\log N)$. At this point, as was argued before, only $O(\log N)$ nodes stand between the query node and the true successor, so $O(\log N)$ additional forwarding steps arrive at the successor node.

To see that the expectation is $O(\log N)$ consider the current node n that has the key in its i^{th} finger interval. If n's i^{th} finger s is up, then in one forwarding step we accomplish our goal: the key is in the $(i-1)^{st}$ finger interval of node s. If s is down then, as argued in the previous theorem, n is still able to forward (at least) to *some* node. More precisely, n was aware of z immediate successors; assume $z \geq 2 \log N$. If we consider the $(\log N)^{th}$ through $(2 \log N)^{th}$ successors, the probability that they all fail is $1/N$. So with high probability, node n can forward the query past at least $\log N$ successors. As was implied by Lemma **??**, it is unlikely that all $\log N$ of these skipped nodes had the same i^{th} finger. In other words, the node to which n forwards the query has a different i^{th} finger than n did. Thus, independent of the fact that n's i^{th} finger failed, there is a probablity 1/2 that the next node's i^{th} finger is up.

Thus, the query passes through a series of nodes, where each node has a distinct i^{th} finger (before the failures) each of which is up independently with probability 1/2 after the failures. Thus, the expected number of times we need to forward the query before finding an i^{th} finger that is up is therefore 2. This proves the claim. □

Under some circumstances the preceding theorems may apply to malicious node failures as well as accidental failures. An adversary may be able to make some set of nodes fail, but have no control over the choice of of the set. For example, the adversary may be able to affect only the nodes in a particular geographical region, or all the nodes that use a particular access link, or all the nodes that have a certain IP address prefix. Because Chord node IDs are generated by hashing IP addresses, the IDs of these failed nodes will be effectively random, just as in the failure case analyzed above.

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.

4.5.4 Voluntary Node Departures

Since Chord is robust in the face of failures, a node voluntarily leaving the system could be treated as a node failure. However, two enhancements can improve Chord performance when nodes leave voluntarily. First, a node n that

is about to leave may transfer its keys to its successor before it departs. Second, n may notify its predecessor p and successor s before leaving. In turn, node p will remove n from its successor list, and add the last node in n's successor list to its own list. Similarly, node s will replace its predecessor with n's predecessor. Here we assume that n sends its predecessor to s, and the last node in its successor list to p.

5 Chord Protocol Analysis

The previous section described the (major part of the) chord protocol, but analyzed it only in certain simple models. In particular, we gave theorems regarding the eventual stabilization of the chord ring after nodes stopped joining, and we gave theorems regarding the robustness of a stable chord ring in the presence of failures. In this section, we delve deeper and prove that the chord protocol is robust in more realistic models of system usage. We consider a model in which nodes are continuously joining and departing, and show that (i) the system remains stable and (ii) lookups continue to work, and work quickly.

5.1 Lookups eventually succeed

The following theorems show that all lookup problems caused by concurrent joins are transient. The theorems assume that any two nodes trying to communicate will eventually succeed.

Theorem 5.1. *Once a node can successfully resolve a given query, it will always be able to do so in the future.*

Theorem 5.2. *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 now formalize this argument.

Definition 5.3. Node s is *reachable* from node p if, by starting at p and following successor pointers, one eventually reaches s. We also say node p can *reach* node s.

Definition 5.4. An *arc path* from p to s is a path of successor pointers, starting at p and ending at s, that only goes through nodes between p and s.

Lemma 5.5. *If at some time t there is an arc path from p to s, then at all future $t' > t$ there is an arc path from p to s.*

Proof. By induction on time, which in this case can be considered as the number of changes to the system (successor or predecessor pointers).

When a node joins it sets up a successor pointer, which lets it reach nodes it couldn't reach before, but clearly doesn't destroy any existing arc path.

Now consider stabilization. Consider a time when node p changes its successor from s to a. It does so only because p contacted s and heard about a, and because $p < a < s$. This means that at some earlier time node s learned about node a. This can only have happened because a told s about itself, which could only happen if a's successor was s at some earlier time. At this time, there was arc path from a to s (namely, the successor link). It follows by induction that just before p changes its pointer to a, there is still an arc path from a to s. Since $p < a < s$, the arc path from a to s cannot include p. Thus, when p changes its pointer the arc path from a to s is undisturbed. But the concatenation of the edge (p, a) with the arc path from a to s forms an arc path from p to s (since all nodes on the path are either a, which is between p and s, or on the arc path from a to s, and thus between a and s, and thus between p and s).

Now consider any arc path from x to y that used the successor edge from p to s (so might be disrupted by the change in p's sucessor). Since it is an arc path, both p and s must be between x and y. We have just argued that all the nodes on the new path from p to s are between p and s; it follows that they are between x and y as well. Thus, the path from x to y remains an arc path. □

Corollary 5.6. *If at time t there is a path of successor arcs from a to b, then at all $t' > t$ there is still a path of successor arcs from a to b.*

Proof. By the previous lemma, each successor arc on the path from a to b can only be replaced by a path; it cannot be disconnected. The concatenation of all these replacement paths forms a path from a to b. □

Corollary 5.7. *Suppose that a is the first node in the Chord network. Then at any time, every node can reach a via successor pointers.*

Proof. By induction on joins. When a node joins, its successor pointer points to a node that can reach a; thus the new node can reach a as well. The previous claim shows that since the new node can initially reach a, it can always reach a. □

Theorem 5.8. *If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor arcs will form a cycle on all the node in the network.*

Proof. Notice that if two nodes share a successor, one of them will eventually change successor pointers. Its new successor will be closer on the circle than the old one, so there can be at most n changes in its successor pointer. Thus after n^2 steps, we must be in a stable state in which every node is the successor of at

most (and thus exactly) one node. Of course we also know that every node has exactly one successor. The only graphical structure satisfying this constraint (indegree one and outdegree one) is a set of cycles. But by our invariant, every node can reach the very first node ever in the network, so the set must consist of exactly one cycle. □

5.2 Effect on Lookup Performance

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 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 5.9. *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.*

Proof. The original set of fingers will, in $O(\log N)$ time, bring the query to the old predecessor of the correct node. With high probability, at most $O(\log N)$ new nodes will land between any two old nodes. So only $O(\log N)$ new nodes will need to be traversed along successor pointers to get from the old predecessor to the new predecessor. □

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 will continue to take $O(\log N)$ hops. We can achieve such adjustment by repeatedly carrying out searches to update our fingers. It follows that lookups perform well so long as $\log^2 N$ rounds of stabilization happen between any N node joins.

5.3 Strong Stabilization

The *stabilize*() protocol described in Figure 6 aims to guarantee that, for any node u, the predecessor of the successor of node u is the node u itself. This is a local consistency condition that is necessary, but not sufficient, for proper behavior in a Chord network. For example, the Chord network shown in Figure 8 is stable under this protocol. However, this network is globally inconsistent — in fact, there is no node u so that *successor*(u) is the first node to follow u on the identifier circle.

Definition 5.10. We say that a Chord network is (1) *weakly stable* if, for all nodes u, we have *predecessor*(*successor*(u)) = u; (2) *strongly stable* if, in

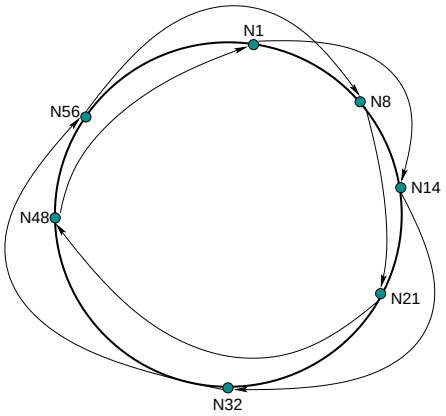


Figure 8: A weakly stable loopy network. The arrows represent successor pointers. The predecessor of a node n's successor is n itself.

addition, for each node u, there is no node v in u's component so that $u < v < successor(u)$; and (3) *loopy* if it is weakly but not strongly stable.

The protocols in Figure 6 maintain strong stability in a strongly stable network. Thus, so long as all nodes operate according the this protocol, it would seem that our network will be strongly stable, so that our lookups will be correct. But we now wish to take a more cautious look. It is conceivable that a bug in an implementation of the protocol might lead to a loopy state. Alternatively, the model might break down—for example, a node might be out of contact for so long that some nodes believe it to have failed, while it remains convinced that it is alive. Such inconsistent opinions could lead the system to a strange state. We therefore aim in this section to develop a protocol that will stabilize the network from an *arbitrary* state, even one not reachable by correct operation of the protocol.

A Chord network is weakly stable iff it is stable under the *stabilize* protocol of Figure 6. Since this protocol guarantees that all nodes have indegree and outdegree one, a weakly stable network consists of a collection of cycles. For a node u, we will say that *u's loop* consists of all nodes found by following successor pointers starting from u and continuing until we reach a node w so that $successor(w) \geq u$. In a loopy network, there is a node u so that u's loop is a strict subset of u's component.

In this section, we present a stabilization protocol (replacing that in Figure 6) to produce a strongly stable network. Note that this protocol does not attempt to reconnect a disconnected network; we rely on some external means to do so. The fundamental stabilization operation by which we unfurl a loopy cycle is based upon *self-search*, wherein a node u searches for itself in the network. For simplicity, we will assume for the moment that this self-search uses only successor pointers and does not make use of larger fingers. If the network is

```
n.join(n')
  on_cycle = false;
  predecessor = nil;
  s = n'.find_successor(n);
  while (not s.on_cycle) do
    s := s.find_successor(n');
  successor[0] = s;
  successor[1] = s;
```

```
n.update_and_notify(i)
  s = successor[i]
  x = s.predecessor;
  if (x ∈ (n, s))
    successor[i] = x;
  s.notify(n);
```

```
n.stabilize()
  u = successor[0].find_successor(n);
  on_cycle = (u = n);
  if (successor[0] = successor[1]
          and u ∈ (n, successor[1]))
    successor[1] = u;
  for (i = 0, 1)
    update_and_notify(i);
```

Figure 9: Pseudocode for strong stabilization.

loopy, then a self-search from u traverses the circle once and then finds the first node on the loop succeeding u — i.e., the first node w found by following successor pointers so that $predecessor(w) < u < w$.

To strongly stabilize a loopy Chord network, we extend the weak stabilization protocol by allowing each node u to maintain a second successor pointer. This second successor is generated by self-search, and improved in exactly the same way as in the previous protocol. The pseudocode is given in Figure 9.

Theorem 5.11. *Any connected Chord network becomes strongly stable within $O(N^2)$ rounds of strong stabilization.*

Although $O(N^2)$ is a slow running time, the situation of a loopy cycle is an extremely low probability event. Over the infinite life of the system, the amount of time that we spend recovering from a loopy state is neglible.

There are two key intuitions behind the correctness of this algorithm. Combined, they show that the only stable configuration of the network is the desired one.

First, we show that if the network is weakly stable but not strongly stable, then at least one node will find a improved second successor when it performs a search for itself in the stabilization algorithm.

Having ruled out the "wrong" weak stabilization, we consider non-loops—i.e, situations in which some nodes have more than one successor pointer. Every node has at least one successor pointer, meaning there are at least N successor pointers in the system. If even one node has two distinct pointers (with $successor[0] \neq successor[1]$) then in total there are *more* than N distinct successor pointers. If this happens, then some node s has two distinct other nodes pointing at it as a successor. As we saw in the previous stabilization algorithm, this is not a stable situation: the closer predecessor p will eventually notify s, and then the farther predecessor will hear about and switch to p.

It follows that the only stable situation is when every node has exactly one successor pointer, which points to that nodes true successor in the network.

Lemma 5.12. *If the network contains a loopy cycle, then there is some node u whose self-search reveals a node v so that $u < v < successor(u)$.*

Proof. If there is a loopy cycle $\mathcal{C}$, then, by definition, there are $u \in \mathcal{C}$ and $s \in \mathcal{C}$ so that $u < s < successor(u)$. Since $\mathcal{C}$ is a cycle, repeatedly following

successor pointers from u eventually leads to s; because $u < s < \mathit{successor}(u)$, we cannot find s on the first traversal of the identifier circle. More generally, then, there must exist $u, w \in \mathcal{C}$ so that w is not on u's loop. Let v be the first node reached following successor pointers from u so that v is not on u's loop. Then $\mathit{predecessor}(v) < u < v$, and $\mathit{predecessor}(v)$ is on u's loop.

Denote the nodes of u's loop as $s^0(u), s^1(u), \ldots, s^\ell(u)$ where (1) $s^0(u) = u$, (2) $\mathit{successor}(s^i(u)) = s^{i+1}(u)$, and (3) $s^\ell(u) = \mathit{predecessor}(v)$. For some $0 \leq i < \ell$, we must have $s^i(u) < v < s^{i+1}(u)$. Note that v cannot fall in the range $(s^\ell(u), u)$, since otherwise v is actually on u's loop.

If $i = 0$, the node u's self-search yields an improved successor v — we have $u = s^0(u) < v < s^1(u) = \mathit{successor}(u)$. For $i \geq 1$, we have $s^i(u) < v < s^{i+1}(u) = \mathit{successor}(s^i(u))$. The self-search by $s^i(u)$ also yields v — an improvement over $s^{i+1}(u) = \mathit{successor}(s^i(u))$ — since $s^i(u)$'s self-search path is a subpath of u's self-search path. □

Lemma 5.13. *If at some time t there is an arc path from p to s, then at all future times $t' \geq t$, there is an arc path from p to s.*

Proof. We proceed by induction on changes to the system, exactly as in the proof for weak stabilization. Joins and self-searches only add an edge and cannot destroy an arc path, nor can replacing a duplicated edge. Updates via predecessors maintain arc paths just as in Lemma 5.5. □

Corollary 5.14. *If at time t there is a path of successor arcs from a to b, then at all times $t' \geq t$ there is still a path of successor arcs from a to b.* □

Claim 5.15. *If the Chord network is connected but not strongly stable, then, after $O(1)$ rounds of stabilization, some successor pointer improves.*

Proof. If there is a node u so that two distinct nodes, say $p_1 < p_2$, both have u as a successor, then after p_2 stabilizes, node u will have a predecessor p so that $p_1 < p_2 \leq p < u$. When p_1 subsequently stabilizes, p_1 will replace its pointer to u by one to p.

If there is a node with two distinct successor pointers, i.e., $u.\mathit{successor}[0] \neq u.\mathit{successor}[1]$, then there are $n+1$ distinct successor pointers, and thus, by the pigeonhole principle, for some node u, there must be two distinct nodes that have successor pointers to u and the previous case applies.

Otherwise, every node points to a single node and is pointed to by a single node, so the network is a collection of cycles. Since we are connected by assumption, we have a single cycle; since we are not strongly stable by assumption, this cycle is loopy. Then by Lemma 5.12, one round of self-search finds an improved successor for some node. □

Proof of Theorem 5.11. By Corollary 5.14, connectivity is maintained throughout strong stabilization. By Claim 5.15, until we are strongly stable, we can always improve a successor pointer in $O(1)$ rounds of stabilization.

Note that any stabilization operation that alters one of node u's successor pointers improves it, in the sense that the new successor is closer to u on the

identifier circle than the old successor is. There are $2N$ pointers (two per node), and each pointer can only improve N times (since there are only n choices of nodes at which it can point). Thus after $O(N^2)$ improvements, each node must have both successor pointers directed at its true successor on the circle. □

Observe that a loopy Chord network will never permit any new nodes to join until its loops merge — in a loopy network, for all u, we have $u.on\_cycle = \textbf{false}$, since u's self-search never returns u in a loopy network. Thus, if the network somehow finds its way into a loopy state, it will heal itself within $O(N^2)$ rounds, unaffected by nodes attempting to join.

We have stated this algorithm so that each stabilization round may takes $O(N)$ time for the self-search. We can reduce this time to $O(\log N)$ time, whp, using fingers. The fingers can be built up using pointer doubling or having each node u invoke $u.find\_successor(u + 2^{i-1})$ for increasing i. Inductively, it is straightforward that $u.finger[i]$ will be in u's loop, and therefore that the finger-based search will give the same result as the successor-only search.

Strong stabilization in the presence of failures. Maintaining a successor list of length $O(\log N)$ will, as before, ensure that our graph, whp, stays connected as long as $\Omega(\log N)$ rounds pass before $N/2$ nodes fail. (This successor list can be formed by following either successor pointer from each node.) Recall, though, only N failures can occur before we are strongly stable, since, as discussed above, no nodes can join a loopy network. (Of course, failures at roughly this rate will cause the ring to disappear rapidly.)

However, if one of u's successors fails, then there may be a large number of nodes between the failed successor and the first live entry in $u.successor\_list$. So we may slip backwards using the sense of "progress" from above. But there are at most N failures before the network empties. If $O(N^2)$ improvements occur after any of the N failures, then we are strongly stable, so we have the following:

Theorem 5.16. *Start from an arbitrary connected state with successor lists of length $O(\log N)$. Allow failures at a rate of at most $N/2$ nodes in $\Omega(\log N)$ steps. Then, whp, in $O(N^3)$ rounds, the network is strongly stable.*

5.4 Fast Strong Stabilization for Two Well-Interleaved Loops

While the previous section shows that we can stabilize even a loopy graph, the time bounds for such stabilization are high. The most likely imaginable scenario that could lead to the creation of a loopy graph is a *network partition* that completely disconnects some of the nodes in the network from others. To model the problem, we start with a weakly stable network consisting of two loops — i.e., starting from a node u, following successor pointers, one returns to u after traversing the identifier circle exactly twice.

We modify $u.stabilize()$ to allow the u to move large distances when it is far from accurate, by allowing u to move to any node which fingers $u.successor$,

rather than just the predecessor. This allows u to find its true successor on the cycle in time $O(\log^2 N)$ (rather than $\Omega(N)$), regardless of where it enters the cycle.

- Each node u maintains a list of *backwards fingers* — for every i, node u stores the closest node with identifier at most $u - 2^{i-1}$ that fingers u.
- In $u.stabilize()$, node u contacts its current successor s, and, if s is on the cycle, changes its successor to the node $v > u$ in s's list of backwards fingers minimizing $v - u$.

This optimization may result in a brief period in which node u previously pointed to a node on the cycle, and then doesn't because its current successor is fingered by a node not on the cycle, and u "backs up" to point to that node instead. When that node joins the cycle, though, node u will continue its march towards its true successor.

Lemma 5.17. *Whp, in $O(\log^2 N)$ rounds of following backwards fingers, a node arrives at its cycle successor.*

Proof. Suppose node u's current successor is v, and that $2^{-i-1} < v - u \leq 2^{-i}$. We consider the number of backward finger links that need to be taken before u sets its successor to a node w so that $w - u \leq 2^{-i}$. Suppose that $u.successor - u > 2^{-i}$ for $O(\log N)$ rounds. Note that, although the probability of each node in $(u + 2^{-i}, u + 2^{-i-1}]$ being fingered by a node within 2^{-i+1} of u is not independent, it is *more* likely that the next node we see will be fingered by such a node if the current one is not.

Each node in this range is, in expectation, the ith finger for one other node. So after $O(\log N)$ steps through nodes between distance 2^{-i} and 2^{-i-1} of node u, whp, we step to a node v that is fingered by a node within less than distance 2^{-i} from u, by the Chernoff bound.

Then, in $O(\log^2 N)$ time, whp, u is forwarded to its true cycle successor — after $O(\log N)$ distance halvings, there are, whp, at most $O(\log N)$ nodes between $u.successor$ and u. □

In the 2-loop case, self-search is powerful: any node u whose true successor v is not in u's loop will find v via its self-search. If all nodes simultaneously complete their self-search, then, we can stabilize quickly:

Lemma 5.18. *If, synchronously, all nodes in a weakly-stable 2-loop network complete a self-search, and then run the strong stabilization protocol, then, with high probability, we are strongly stable in $O(\log N)$ rounds.*

Proof. A *chain* is a consecutive sequence of nodes on the identifier circle, all of which fall on the same loop. The *tail* of a chain is its last node, and is the one node in the chain whose true successor lies in the other loop. With high probability, there are no more than $O(\log N)$ nodes in a chain.

Once all the self-search successor pointers are added, every node has a pointer to its true successor; we must only correct the pointer from each chain's tail u

to the next node in u's loop. This pointer will be moved backwards one node at a time across the intervening chain (from the other loop) to point at u's true successor. This chain has $O(\log N)$ nodes with high probability, so in $O(\log N)$ rounds of stabilization we are done. □

In the asychronous model, however, the first nodes that complete their self-searches may break the search algorithm, possibly causing other nodes attempting to self-search to fail. (By Theorem 5.11, we will eventually stabilize, but there is no guarantee of efficiency.) We can handle this problem by simple patience:

- when a a node u's self-search reveals that the network is loopy, u waits $\Theta(\log N)$ rounds (for all other nodes to complete their self-searches) and only then adds the new successor pointer.

We do not know exactly how long each self-search will take — some nodes' self-search path may take a number of long geographic network hops, and therefore be substantially slower than others. One can easily verify that waiting until only $O(1)$ self-searches are unfinished yields strong stabilization in $O(\log^2 N)$ time with high probability. Under the assumption that search time is independent of position on the identifier circle, waiting until at most $N/2$ self-searches are unfinished allows us to strongly stabilize in $O(\log^3 N)$ time with high probability. (With high probability, there are no more than $O(\log N)$ consecutive chains whose head has not completed its self-search; each chain requires only $O(\log^2 N)$ rounds to traverse, whp.)

Another possible result of the healing of a network partition is two completely disconnected Chord rings that can reach each other after the partition heals, but every node has dropped all pointers to the other loop. To combat this event for short-duration network partitions, we can do the following, for each node u:

- Remember the closest node v that u has ever pointed to, regardless of whether it currently is up or down, in the last, say, $O(\log^2 N)$ rounds.
- Every $O(1)$ rounds, ping v to see if it has been resurrected. If so, add a second successor pointer to it.

Typically, a node will come back to life when a network partition heals (though if it does not heal rapidly, a large number of failures will mean that for most nodes u, their closest ever neighbor will actually have failed while the network was partitioned).

For the purposes of stabilization, the situation after the network partition heals is exactly that of the above 2-loop case after the completion of the self-searches. In time $O(\log^2 N)$, then, we will re-stabilize these two loops into one, as long as at least a constant fraction of the nodes alive at the time of the partition survive until the healing of it.

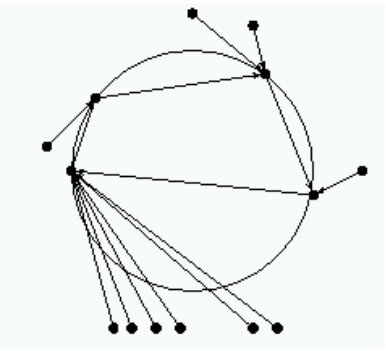


Figure 10: A pseudostar.

5.5 A Dynamic Model

Until now we have assumed that the initial state of the system is a ring. In practice, we cannot assume that we ever actually start with a ring, because there will always be some recently joined nodes that have not yet fit into the ring. Thus, in this section we try to prove a more powerful result: that our stabilization algorithm continuously keeps the system in a ring-like state.

In this section, for simplicity of presentation, we limit ourselves to a synchronous model of stabilization. With mild complications on the definitions that follow, we can handle (without an increase in running time) a network with a reasonable degree of asynchrony, where most machines are operating at roughly the same rate, and messages take roughly consistent times to reach their destinations. We refer to a *round* of stabilization as the $O(1)$ time required for each node to run *stabilize*(), disregarding any time required for the transfer of keys.

Each node has exactly one successor, so the graph defined by successor pointers is a *pseudoforest*, a graph in which all components are directed trees pointing towards a root cycle (instead of a root node). We will limit our consideration to connected networks, where the graph is a *pseudotree*. The network is (weakly) stable when all nodes are in the cycle. For each cycle node u, then, there is a tree rooted at u which we call u's *appendage*, and denote $\mathcal{A}_u$.

We insist that a node u joining the system invoke $u.join(n)$ for an existing node n that is already on the cycle. We can use an external infrastructure to enforce this, or we can use the more complicated *join*() protocol in Section 5.3.

Definition 5.19. A *pseudostar* is a Chord network in which:

(i) The cycle is non-loopy;

(ii) $\mathcal{A}_u \subseteq [p, u]$, where p is the predecessor of u on the cycle;

(iii) for every node $v \in \mathcal{A}_u$, we have $u = v.successor$.

See Figure 10.

Lemma 5.20. *Starting from a psuedostar, execute an arbitrary sequence of joins while running stabilize*(). *Then the resulting network is still a psuedostar.*

Proof. We proceed by induction over changes to the system.

By *(iii)*, the $n.join(v)$ protocol finds a node s on the cycle so that $s.predecessor < n < s$. Thus every search ends with $n.successor$ on the cycle and $(n.successor).predecessor < n < n.successor$. The correctness of *find_successor* on the cycle is guaranteed inductively by property *(i)*. This immediately yields properties *(ii,iii)*. Property *(i)* is maintained since joins cannot cause a non-loopy cycle to become loopy, by Lemma 5.5.

The $u.stabilize()$ protocol cannot create a loop by Lemma 5.5, and incorporates a node into the cycle by having u's cycle predecessor point at one of u's other predecessors, say p. All of u's predecessors that are in the range $[p_c, p]$ shift to point to p instead of u, where p_c is u's cycle predecessor, yielding *(ii,iii)*. □

Note that this holds even for completely arbitrary joins to the system — e.g., if the identifiers of joining nodes were chosen maliciously and not randomly.

Definition 5.21 (Ring-Like State). A Chord network is in the *c-ring-like state* iff for some constant c,

(i) The network is a pseudostar;

(ii) Nodes that joined the network at least $8c^2 \log^2 N$ rounds ago:

- *(a)* are all on the cycle;
- *(b)* comprise at least half of the nodes in the network;
- *(c)* are independently and uniformly distributed around the identifier circle;
- *(d)* never fall in the range $[u + 2^{i-1}, u.finger[i]]$.

(iii) For the nodes that joined the network in the last $8c^2 \log^2 N$ rounds:

- *(a)* the nodes are independently and uniformly distributed around the identifier circle;
- *(b)* for any consecutive nodes $u_1, u_2, \ldots, u_{\log N}$ on the cycle, we have $\sum_{i=1}^{\log N} |\mathcal{A}_{u_i}| \leq c \log N$;

Increasing the constant c increases the exponent of the $1 - 1/n^{O(1)}$ probability of success in the results of this section.

Note that there may be bias in the order in which nodes that joined the network recently are incorporated into the cycle — e.g., there is a bias towards joining the cycle sooner for nodes close to nodes already on the cycle and nodes falling between two nodes on the cycle which are close together — so we cannot say that the distribution of all nodes on the cycle is uniform and independent.

For technical reasons, we will consider a strongly stable network with correct fingers, and all nodes' identifiers independently and uniformly chosen to be in the ring-like state. (This allows the creation of a ring-like network.)

Properties *(ii.c,iii.a)* are immediate from the random join model, so these are trivially maintained.

Lemma 5.22. *Start in the c-ring-like state with N nodes, and allow up to N random joins at arbitrary times over at least $8c^2 \log^2 N$ rounds. Then, whp, we end up in the c-ring-like state. (To improve the probability of success, adjust c upwards.)*

Intuitively, we distinguish between "old" nodes which have been present for longer than $8c^2 \log^2 N$ rounds, "middle-aged" nodes which have been present for less time, and the at most N "new" nodes which join during the current $8c^2 \log N$ rounds of stabilization. By definition of the ring-like state old nodes are in the cycle (i.e., are reachable by successor pointers from all nodes on the cycle). From the fact that identifiers are random, we know that $O(\log N)$ nodes join between any two old nodes.

We also have that finger pointers are correct with respect to old nodes — that is, that no finger pointer bypasses an old node — by *(ii.d)*. This essentially places us back into the analysis of the previous section, with the old nodes playing the role of the initial cycle, before other nodes joined. As was argued above, since only N nodes join, the fingers pointing at old nodes suffice to route lookups quickly for new nodes. This implies that no node ends up with too many nodes in its appendage during the time period being analyzed.

Since no appendage is large, and since at least one node falls off each appendage per round, it follows that any node present at the beginning of the analysis (i.e., the middle-aged nodes) will fall off the appendage and onto the cycle during the rounds of stabilization. All these middle-aged nodes will therefore enter the cycle before they become old, thus preserving the invariant of the ring-like state definition. After all the middle-aged nodes enter the cycle, we require an additional $O(\log^2 N)$ rounds of stabilization to ensure that all of the fingers are correct with respect to the middle-aged nodes, since *fix_fingers*() is quick by the above.

Proof of Lemma 5.22. Suppose that our process begins running at time t_0. Call the *old nodes* those which entered the network at time $t_0 - 8c^2 \log^2 N$ or earlier, the *middle-aged nodes* those that entered in the time range $(t_0 - 8c^2 \log^2 N, t_0)$, and the *new nodes* those that enter at time t_0 or later. If the initial network is in the ring-like state in virtue of being strongly stable with correct fingers, we proceed as if all nodes in the network at t_0 were old. Properties *(i,ii,iii)* still hold at time t_0, the last vacuously.

Call an *old gap* the interval between two consecutive old nodes on the cycle, and a *gap* the interval between each adjacent pair of nodes on the cycle at time t_0, regardless of whether the nodes are old or middle-aged. By *(ii)*, there are at least $N/2$ old nodes, all on the cycle, so there are at least $N/2$ old gaps, and, since gaps refine old gaps, we have

($*$) whp, at most $c \log N$ new nodes will fall into any sequence of $\log N$ consecutive gaps.

since the joins are random and the old nodes are independently and uniformly spread over the identifier circle by *(ii.c)*.

We claim that, whp, the conditions *(i,ii,iii)* still hold at time $t_0 + 8c^2 \log^2 N$:

(i) Immediate from Lemma 5.20.

(ii.a) Consider the (middle-aged) nodes S originally found in a particular gap, say in $\mathcal{A}_u$. We claim that within $4c \log N$ rounds, all of the nodes of S

will be incorporated into the cycle. Since no node on the cycle ever leaves it, this implies the desired condition.

Let J_t be the set of nodes that have joined in this gap by time $t_0 + t$, and let $J = J_{8c^2 \log^2 N}$ be the set of all nodes that join in this gap throughout the entire process. Note that by $(*)$, we have, whp, $|J| \leq c \log N$, and by property *(iii.b)* we have $|S| \leq c \log N$. Therefore $|J_t + S| \leq 2c \log N$, whp.

At time t, some of the nodes of $S \cup J_t$ have joined the cycle, and the remaining nodes of $S \cup J_t$ are divided among the appendages of u and those nodes of $S \cup J_t$ now on the cycle. If $S \cup J_t$ is not entirely incorporated into the cycle, then note that within two rounds of stabilization, at least one node from $S \cup J_t$ is incorporated into the cycle. In the first round, some node $v \in \{u\} \cup S \cup J_t$ on the cycle has at least two predecessors (one on the cycle, say p_c, and its closest appendage precedessor, say p_a), and stores p_a as its predecessor. (We have $v > p_a > p_c$ since the network is a pseudostar.) In the second round, p_c acquires p_a as its successor, and now p_a is on the cycle.

Thus in $4c \log N$ rounds of stabilization, at least $2c \log N$ nodes from $S \cup J_{4c \log N}$ join the cycle, unless at some time $t < 4c \log N$, all of $S \cup J_t$ is already on the cycle. But $|S \cup J_{4c \log N}| \leq 2c \log N$, whp. So in either case, whp after $4c \log N$ rounds, all of $S \cup J_t \supseteq S$ has been incorporated into the cycle, for some $t \leq 4c \log N$.

(ii.b) There were N nodes at time t_0, and at most N joined over the next $8c \log^2 N$ rounds.

(ii.c) Immediate from the join model — each node has its identifier uniformly and independently selected.

(ii.d) After all middle-aged nodes have joined the network, the procedure *u.fix_fingers*() will fix each *u.finger*[i] to point to the first node following $u + 2^{i-1}$ on the cycle at the time. Whp, *u.find_successor*($u + 2^{i-1}$) requires $2c \log N$ time: whp, the total number of times that the search path from u will land on a middle-aged node (which may not have a finger to halve the distance) is $c \log N$, and the distance, whp, halves only $\log N$ times before we reach the target node.

Whp, node u has only $c \log N$ fingers, so fixing all of them requires $2c^2 \log^2 N$ time after the $4c \log N$ rounds that incorporate the middle-aged nodes in the cycle, whp. So in $8c^2 \log^2 N > 4c \log N + 2c^2 \log^2 N$ rounds, we are done.

(iii.a) Immediate from the join model.

(iii.b) By $(*)$, whp, at most $c \log N$ of the new nodes will fall into any sequence of $\log N$ gaps. By *(ii.a)*, all non-new nodes in this range are incorporated into the cycle, so only these $c \log N$ new nodes are in appendages in these gaps. This range may now span more than $\log N$ cycle nodes, but this only decreases the size of appendages.

□

In the static case, where "eventually" all successor pointers are correct, lookups will eventually produce the correct result, so lookups will eventually be correct as well. At this point, searches for data associated with a particular key will consistently return the same node, so that data can be stored and retrieved. In our more complicated dynamic case, this is no longer obvious—as nodes join the network, data must be moved to the new nodes; searches for the data might arrive at a node that does not have the data yet. However, our analysis above claiming a near steady state can be adapted to show that when a lookup on a particular key will always find a node that either has the data or is about to have the data, so that by waiting for a brief (logarithmic) amount of time, the data can be retrieved.

5.6 Analysis of Joins with Failures

We now generalize our model to include failures. For intuition, we begin with a simple model of a ring involving only failures (of course, such a ring will not last for very long). Then we give an analysis involving both joins and failures.

5.6.1 The pure failure model

As was described above, a node maintains its successor list by copying its successor's successor list and prepending its successor to it. In this section, we ask how often (relative to the failure rate of nodes) this copying needs to take place.

Lemma 5.23. *Suppose that each node in an N-node Chord ring has a successor list of length $2c \log N$ containing at least the $c \log N$ next living successors in the ring. Suppose $N/4$ arbitrarily-chosen nodes (though independent of their identifiers) fail at any time during the execution of $2c \log N$ successor-list copyings by each node. Then at the end, every node will still have a successor list with the $c \log N$ next living successors on the ring.*

Proof. Since the failures are independent of the identifier, the failures are random in the identifier space.

With high probability, for every u, at least one of the $c \log N$ live nodes in $u.successor\_list$ will remain alive throughout this batch of failures. Thus the ring will remain connected. Inductively, it is straightforward to show that any node which appears in the ith position of $u.successor\_list$ was alive i rounds ago. So after $2c \log N$ rounds, no node that was originally down (before these $N/4$ failures) will appear in any successor list.

Thus, after the $2c \log N$ rounds, $u.successor\_list$ contains the first $2c \log N$ nodes following u on the cycle that were alive at the start of this process (or, if some of these have subsequently failed, replacements from farther along the cycle). But, whp, fewer than half of these $2c \log N$ nodes fail, so $u.successor\_list$ contains at least the next $c \log N$ living successors on the ring. □

5.6.2 Allowing Failures in the Ring-Like State

Realistically, we need to incorporate a notion of failures into the definition of the ring-like state. We continue to assume a synchronous model, which can again be weakened with additional complication (but without a significant running time increase). As before, $\mathcal{A}_u$ denotes the nodes in the tree rooted at the cycle node u. Denote by $\mathsf{last}(\ell)$ the last element of a list ℓ.

Definition 5.24. A *robust strongly non-loopy pseudotree* is a pseudotree with successor lists in which:

(i) the cycle is non-loopy;

(ii) $\mathcal{A}_u \subseteq [p, u]$, where p is the predecessor of u on the cycle;

(iii) for every node $v \in \mathcal{A}_u$, we have $v < v.successor < u$.

(iv) if the successor list of $s = u.successor$ skips over a live node $v \in [s, \mathsf{last}(s.successor\_list)]$, then v is not in $u.successor\_list$.

Lemma 5.25. *Starting from a robust strongly non-loopy psuedotree, execute an arbitrary sequence of joins and failures while running stabilize(). Then, if the resulting network is still connected, it is still a robust strongly non-loopy psuedotree.*

Proof. We proceed by induction over changes to the system. The *join*() and *stabilize*() operations maintain properties *(i,ii,iii)* just as in Lemma 5.20, and property *(iv)* since they set $u.successor\_list$ to contain only $u.successor$ and nodes from $(u.successor).successor\_list$.

For node failures, we assume that the network remains connected throughout the failures. Property *(i)* is maintained since property *(iv)* guarantees that we do not destroy an arcpath from u to any node that remains on the cycle after the failures. By properties *(i,iii)*, every entry w in a node v's successor list has $w > v$. So, in any failure that causes $v \in \mathcal{A}_u$ to set its successor to any entry in its successor list, we still have $v < v.successor < s$, where s is the first living cycle node after u. This yields *(ii,iii)*. Property *(iv)* only refers to live nodes, so failures cannot falsify it. □

Note that this holds even for completely arbitrary joins and failures to the system — e.g., if the selection of failing nodes and the identifiers of joining nodes were chosen maliciously and not randomly — so long as the network remains connected.

In a network with successor lists of length $2c \log N$, we will say that a node u is *fully incorporated into the cycle* iff it has been in the cycle for at least $2c^2 \log N$ consecutive rounds. Note that merely having a cycle node u point to node v is insufficient for v to be robustly on the cycle, since if u fails immediately after setting $u.successor = v$, then v will fall off the cycle.

Definition 5.26 (Ring-Like State with Successor Lists). A Chord network is in the *ring-like state with successor lists* if it has successor lists of length $2c \log N$ and

(i) The graph defined by the first live successor of every node is a robust strongly non-loopy pseudotree;

(ii) Nodes that joined the network at least $16c^5 \log^2 N$ rounds ago:

- *(a)* are all fully incorporated into the cycle;
- *(b)* comprise at least half of the nodes in the network;
- *(c)* are independently and uniformly distributed around the identifier circle;
- *(d)* never fall in the range $[u + 2^{i-1}, u.\mathit{finger}[i]]$.

(iii) For the nodes that joined the network in the last $16c^5 \log^2 N$ rounds:

- *(a)* the nodes are independently and uniformly distributed around the identifier circle;
- *(b)* for any consecutive nodes $u_1, u_2, \ldots, u_{\log N}$ on the cycle, we have $\sum_{i=1}^{\log N} |\mathcal{A}_{u_i}| \leq c^2 \log N$;

(iv) The successor list for every node u:

- *(a)* contains no nodes that failed more than $16c^5 \log^2 N$ rounds ago;
- *(b)* contains at most $c \log N$ nodes that failed within the last $16c^5 \log^2 N$ rounds;
- *(c)* contains every live node in $[u, \mathsf{last}(u.\mathit{successor\_list})]$ that successfully entered the cycle at least $16c^5 \log^2 N$ rounds ago;

(v) Node failures are independent and uniform among all nodes that exist at the time of the failure.

As before, we will consider as ring-like a strongly stable network with correct fingers, successor lists of length $2c \log N$ satisfying *(iv)*, nodes' identifiers that are independently and uniformly chosen, and failures are independently and uniformly chosen from nodes alive at the time of the failure, and pretend that all nodes are old.

Lemma 5.27. *Consider a Chord network with successor lists, and suppose that failures are independent and uniformly chosen from all live nodes at the time of the failure. For any k, running stabilize() can only decrease the probability that more than k nodes in $u.\mathit{successor\_list}$ will fail, for any node u.*

Proof. For any node v, consider the ith entry w_i in $v.\mathit{successor\_list}$. We know inductively that w_i was alive i stabilization rounds ago, since it was then placed into w_{i-1}'s successor list. We have no information as to whether w_i failed in the last $i - 1$ rounds.

At time t, the $u.\mathit{stabilize}()$ procedure adjusts $u.\mathit{successor}$ to a live node s, and invokes $u.\mathit{fix\_successor\_list}()$ to replace the tail of $u.\mathit{successor\_list}$ by $s.\mathit{successor\_list}$. By the above, the ith entry of $s.\mathit{successor\_list}$ at time $t - 1$ — which is the $(i+1)$st entry of $u.\mathit{successor\_list}$ at time t — was alive in round $t - 1 - i$. The previous $(i + 1)$st entry in $u.\mathit{successor\_list}$ was alive in round $t - 1 - (i + 1)$, and could have failed in round $t - 1 - i$. By the assumption of random failures, there is a decrease in the probability that the $(i+1)$st entry of $u.\mathit{successor\_list}$ has already failed. If both nodes are alive at time t, then, again by random failures, the probability that one fails at any time $t' > t$ is identical to that probability for the other. □

When a cycle node u fails, the nodes in $\mathcal{A}_u$ are incorporated into $\mathcal{A}_{u.successor}$; we say that $\mathcal{A}_u$ and $\mathcal{A}_{u.successor}$ have *merged*.

Lemma 5.28. *Start with a network of N nodes in the ring-like state with successor lists of length $2c \log N$, and allow up to $3N/4$ random joins and $N/4$ random failures at arbitrary times over at least $16c^5 \log^2 N$ rounds. Then, whp, we end up in the ring-like state with successor lists of length $2c \log N$. (To improve the probability of success, adjust c upwards.)*

The intuition is as follows. As in Lemma 5.22, we distinguish old, middle-aged, and new nodes; by assumption, the $\Theta(N)$ old nodes are randomly distributed on the cycle, and so $O(\log N)$ new nodes join between any two old nodes. Search is, as before, $O(\log N)$, since fingers are correct with respect to old nodes. Cycle nodes can fail, causing their appendages to merge together, but whp only $O(\log N)$ consecutive cycle nodes fail, so the size of an appendage is $O(\log N)$ by *(iii.b)*, whp, including middle-aged and new nodes.

Unlike in Lemma 5.22, it is not true that a node from appendage (fully) enters the cycle in each round, since the cycle node that points to it may fail immediately. However, whp, within $O(\log N)$ rounds, some node from each appendage will become fully incorporated into the cycle — in $O(\log N)$, attempts, it will join the cycle with a predecessor that does not fail in this entire process. Once a node is fully incorporated into the cycle, whp, it never leaves, so within $O(\log^2 N)$ rounds, all middle-aged nodes in an appendage will join the cycle. As before, we require another $O(\log^2 N)$ rounds of stabilization after this point to fix all of the fingers.

Proof of Lemma 5.28. As in Lemma 5.22, call the *old*, *middle-aged*, and *new nodes* those which entered the network before time $t_0 - 16c^5 \log^2 N$, between then and t_0, and after t_0, respectively. If the initial network is in the ring-like state in virtue of being strongly stable with correct fingers, we proceed as if all nodes in the network at t_0 were old — properties *(i,ii,iii,iv,v)* still hold at time t_0. Define *gaps* and *old gaps* as before; we have

(∗) whp, at most $c \log N$ new nodes will fall into any sequence of $\log N$ consecutive gaps.

Whp, $c \log N$ successive original cycle nodes will not fail, so each appendage will merge with fewer than $c \log N$ successive original appendages:

(†) whp, at any time during this process, nodes originally found in $\mathcal{A}_u$ will merge only with nodes originally found in fewer than $c \log N$ consecutive appendages adjacent to u.

Whp, the $2c \log N$ nodes of the original *u.successor_list* will not all fail for any u — since only $c \log N$ can have failed already, by *(iv.a,b)*, and the probability that the remaining $c \log N$ nodes fail is small, by the Chernoff bound — which by Lemma 5.27 implies that, whp, *u.successor_list* will never consist entirely of nodes that will fail:

(‡) whp, the graph remains connected throughout this process.

If a node has been on the cycle for $2c^2 \log N$ rounds, then, by the Chernoff bound, whp it is contained in at least $c \log N$ successor lists. Similarly, if a node has been off of the cycle for $2c^2 \log N$ rounds, then no node on the cycle will store it in its successor list, whp. Therefore:

(§) whp, once a node has been in the cycle for at least $2c^2 \log N$ rounds, it is present in at least $c \log N$ successor lists; if it has been out of the cycle for at least $2c \log N$ rounds, it is present in no successor lists on the cycle.

Now, we claim that properties *(i–v)* continue to hold, whp:

(i) By (‡), immediate from Lemma 5.25.

(ii.a) Consider the (middle-aged) nodes S originally found in a particular gap, say in $\mathcal{A}_u$. We claim that within $12c^5 \log^2 N$ rounds, all live nodes from S will be fully incorporated into the cycle. By (§), whp, these nodes never subsequently leave the cycle, since whp not all of the $c \log N$ nodes storing one of these nodes in their successor lists fail. This implies the desired condition.

By (†), nodes of S will merge with the nodes contained in at most $c \log N$ adjacent gaps. By (∗), at most $c^2 \log N$ new nodes will fall into these other gaps; by property *(iii.b)*, there are at most $c^3 \log N$ middle-aged nodes in these gaps. So the nodes of S can merge with at most $2c^3 \log N$ other nodes. By property *(iii.b)* we have $|S| \leq c^2 \log N$. Whp, then, the total number of nodes that will ever appear in appendages with nodes of S is at most $2c^3 \log n + c^2 \log N \leq 3c^3 \log N$.

We claim that within $4c^2 \log N$ rounds, whp, some node v in an appendage containing nodes from S will become fully incorporated into the cycle. In two rounds of stabilization, as in Lemma 5.22, some such node v will begin to become incorporated into the cycle. (During these rounds of stabilization, the node v may fall out of the cycle, and some other v may begin to join, but for our purposes this is irrelevant.) The key observation is that it is only harder for node v to become fully incorporated into the cycle if, for any node u that will fail at some time during this process, the node u fails as soon as v first appears in $u.\mathit{successor\_list}$. (If it fails later, then u can propagate information about v backwards along the cycle.) If v enters $2c^2 \log N$ successor lists, whp, at least $2c \log N$ of those nodes will never fail. Thus, since v enters another successor list in at most two rounds, after at most $4c^2 \log N$ rounds v will be fully incorporated into the cycle.

Now, there are at most $3c^3 \log N$ nodes in S's appendages, and at least one is fully incorporated into the cycle every $4c^2 \log N$ rounds. Therefore, in $12c^5 \log^2 N$ rounds, all nodes of S will have been fully incorporated into the cycle, whp.

(ii.b) There were N nodes at time t_0, and up to $3N/4$ joins. Even if all $N/4$ failures were all of old nodes, at time $t_0 + 16c^5 \log^2 N$ at least half of the nodes existed at time t_0.

(ii.c,iii.a,v) Immediate from the join and failure models.

(ii.d) After all middle-aged nodes have joined the network, the procedure u.*fix_fingers*() will fix each u.*finger*[i] to point to the first node following $u + 2^{i-1}$ on the cycle at the time. Whp, u.*find_successor*($u + 2^{i-1}$) requires $3c \log N$ time: whp, the total number of times that the search path from u will land on a middle-aged node (which may not have a finger to halve the distance) is $c \log N$, the total number of times that the search path will attempt to move to a failed node is $c \log N$, and the distance, whp, halves only $\log N$ times before we reach the target node. (Whp, the events of not being able to follow the distance-halving finger from two consecutive nodes on the search path are independent, since, whp, the step from u passes to at least the last $c \log N$ nodes in its successor list. Whp, this next node thus has a different ith finger than u.)

Whp, node u has only $c \log N$ fingers, so fixing all of them requires $3c^2 \log^2 N$ time after the $12c^5 \log^2 N$ rounds that incorporate the middle-aged nodes in the cycle, whp. So in $16c^5 \log^2 N$ rounds, we are done.

(iii.b) By ($*$), whp, at most $c \log N$ of the new nodes will fall into any sequence of $\log N$ gaps. By (†), at most $c \log N$ appendages will join the appendages in these gaps because of failures. So the total number of appendage nodes in these gaps is at most $c^2 \log N$. By *(ii.a)*, all non-new nodes in this range are incorporated into the cycle, so only these $c^2 \log N$ new nodes are in appendages in these gaps. This range may now span more than $\log N$ cycle nodes, but this only decreases the size of appendages.

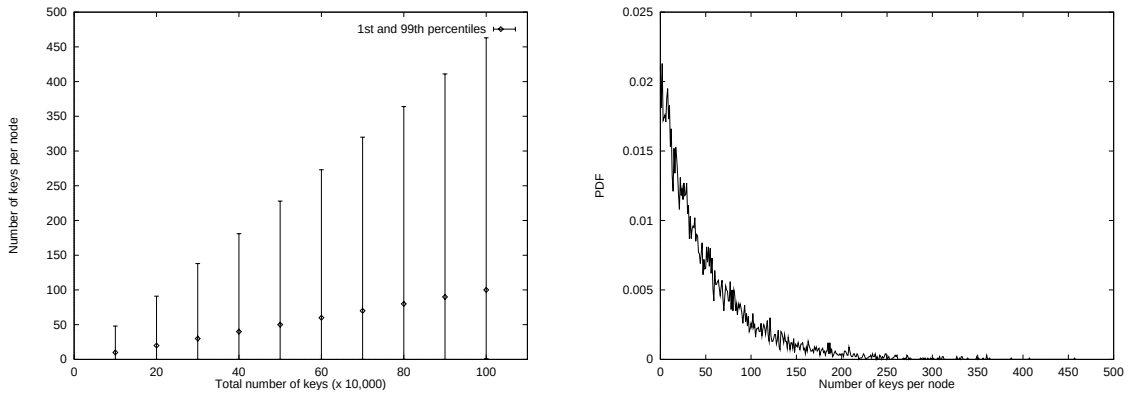
(iv.a,c) Immediate from (§).

(iv.b) From Lemma 5.27, the probability of having $c \log N$ failed nodes in a successor list only decreases via *stabilize*(). Whp, at most $c \log N$ of the $2c \log N$ nodes in any original u.*successor_list* fail during the whole process, so, whp, only $c \log N$ of the entries in any successor list have failed.

□

6 Simulation and Experimental Results

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



(a) (b)

Figure 11: (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.

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 11(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 11(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× the mean value. For comparison, the 99th percentile is 4.6× 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

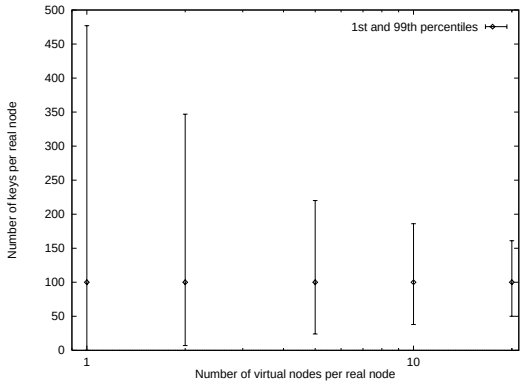


Figure 12: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

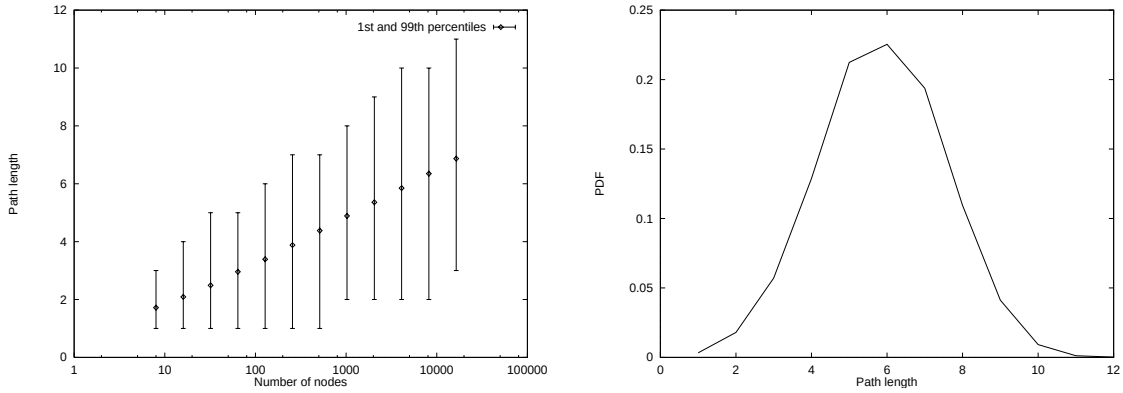
bin. But in fact, the probability that a particular bin does not contain any node is $(1 - 1/N)^N$. For large values of N this approaches $e^{-1} = 0.368$.

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 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 12 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× to 1.6× the mean value, while the 1st percentile increases from 0 to 0.5× 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,



(a) (b)

Figure 13: (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.

we define the path length as the number of nodes traversed during a lookup operation. From Theorem 4.2, 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 13(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 13(b) plots the PDF of the path length for a network with 2^{12} nodes ($k = 12$).

Figure 13(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.

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

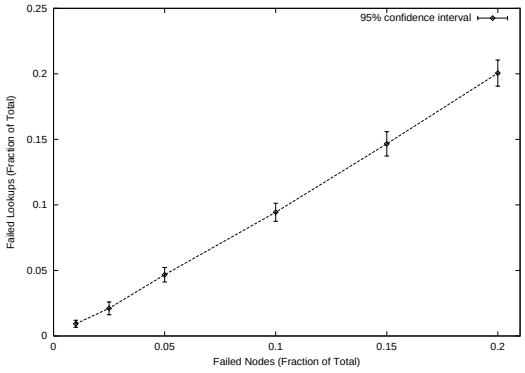


Figure 14: The fraction of lookups that fail as a function of the fraction of nodes that fail.

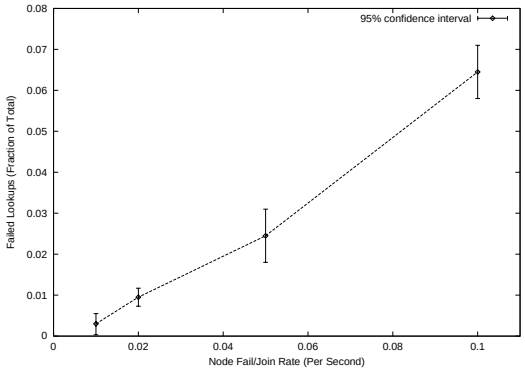


Figure 15: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.

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 14 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 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 6, the simulator updates all finger table entries on every invocation. The network starts with 500 nodes.

Figure 15 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 15 are averaged over approximately two hours of simulated time. The confidence intervals are computed over 10 independent runs.

The results of figure 15 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.

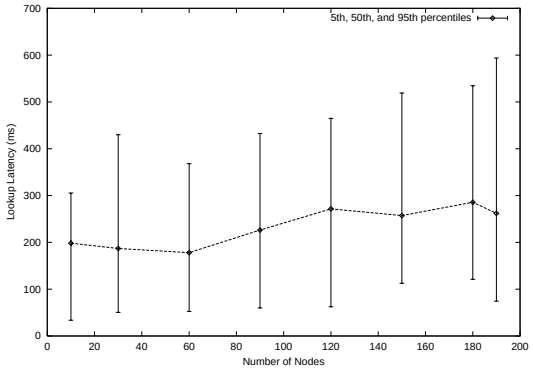


Figure 16: Lookup latency on the Internet prototype, as a function of the total number of nodes. Each of the ten physical sites runs multiple independent copies of the Chord node software.

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 16 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 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 16, 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 16 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.

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 David Andersen for setting up the testbed used in the measurements of the Chord prototype described in Section 6.6.

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), pp. 161–172.

[21] 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.