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

Ion Stoica[†], Robert Morris[‡], David Liben-Nowell[‡], David R. Karger[‡], M. Frans Kaashoek[‡], Frank Dabek[‡], Hari Balakrishnan[‡]

Abstract—

A fundamental problem that confronts peer-to-peer applications is the efficient location of the node that stores a desired 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 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: communication cost and the state maintained by each node scale logarithmically with the number of Chord nodes.

I. Introduction

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

The *Chord protocol* supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses consistent hashing [12] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and requires relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumes that each node is aware of most of the other nodes in the system, an approach that does not scale well to large numbers of nodes. In contrast, each Chord node needs "routing" information about only a few other nodes. Because the routing table is distributed, a Chord node communicates with other nodes in order to perform a lookup. In the steady state, in an N-node system, each node maintains information about only $O(\log N)$ 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.

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 of information per node need be correct in order for Chord to guarantee correct (though possibly slow) routing of queries; Chord has a simple algorithm for maintaining this information in a dynamic environment.

The contributions of this paper are the Chord algorithm, the proof of its correctness, and simulation results demonstrating the strength of the algorithm. We also report some initial results on how the Chord routing protocol can be extended to take into account the physical network topology. Readers interested in an application of Chord and how Chord behaves on a small Internet testbed are referred to Dabek *et al.* [9]. The results reported by Dabek *et al.* are consistent with the simulation results presented in this paper.

The rest of this paper is structured as follows. Section II compares Chord to related work. Section III presents the system model that motivates the Chord protocol. Section IV presents the Chord protocol and proves several of its properties. Section V presents simulations supporting our claims about Chord's performance. Finally, we summarize our contributions in Section VII.

II. Related Work

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

To clarify comparisons with related work, we will assume in this section a Chord-based application that maps keys onto values. A value can be an address, a document, or an arbitrary data item. A Chord-based application would store and find each value at the node to which the value's key maps.

DNS provides a lookup service, with host names as keys and IP addresses (and other host information) as values. Chord could provide the same service by hashing each host name to a key [7]. Chord-based DNS would require no special servers, while ordinary DNS relies on a set of special root servers. DNS requires manual management of the routing information (`NS` records) that allows clients to navigate the name server hierarchy; Chord automatically maintains the correctness of the analogous routing information. DNS only works well when host names are structured to reflect administrative boundaries; Chord imposes no naming structure. DNS is specialized to the task of finding named hosts or services, while Chord can also be used to find

[†]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.

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.

data objects that are not tied to particular machines.

The Freenet peer-to-peer storage system [5], [6], 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 map documents to nodes, and Freenet-style query routing [20]. 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 [4].

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 shortcuts [25]. 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.* [21] is perhaps the closest algorithm to the Chord protocol. The Tapestry lookup protocol [26], used in OceanStore [13], is a variant of the Plaxton algorithm. Like Chord, it guarantees that queries make no more than a logarithmic number of hops and that keys are well-balanced. The Plaxton protocol's main advantage over Chord is that it ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. Chord, on the other hand, is substantially less complicated and handles concurrent node joins and failures well. Pastry [23] is a prefix-based lookup protocol that has properties similar to Chord. Like Tapestry, Pastry takes into account network topology to reduce the routing latency. However, Pastry achieves this at the cost of a more elaborated join protocol which initializes the routing table of the new node by using the information from nodes along the path traversed by the join message.

CAN uses a d-dimensional Cartesian coordinate space (for some fixed d) to implement a distributed hash table that maps keys onto values [22]. 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 (GLS) [15]. GLS 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.

Napster [18] and Gnutella [11] provide a lookup operation to find data in a distributed set of peers. They search based on user-supplied keywords, while Chord looks up data with unique identifiers. Use of keyword search presents difficulties in both systems. Napster uses a central index, resulting in a single point of failure. Gnutella floods each query over the whole system, so its communication and processing costs are high in large systems.

Chord has been used as a basis for a number of subsequent research projects. The *Chord File System (CFS)* stores files and meta-data in a peer-to-peer system, using Chord to locate storage blocks [9]. New analysis techniques have shown that Chord's stabilization algorithms (with minor modifications) maintain good lookup performance despite continuous failure and joining of nodes [16]. Chord has been evaluated as a tool to serve DNS [7] and to maintain a distributed public key database for secure name resolution [1].

III. 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 applications that use it. The application interacts with Chord in two main ways. First, the Chord library provides a `lookup(key)` function 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

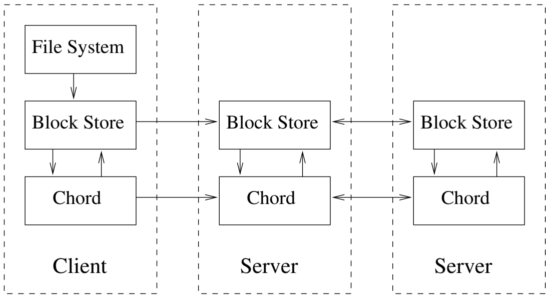


Fig. 1. Structure of an example Chord-based distributed storage system.

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 can provide a good foundation:

Cooperative mirroring, in which multiple providers of content cooperate to store and serve each others' data. The participants might, for example, be a set of software development projects, each of which makes periodic releases. Spreading the total load evenly over all participants' hosts lowers the total cost of the system, since each participant need provide capacity only for the average load, not for that participant's peak load. Dabek *et al.* describe a realization of this idea that uses Chord to map data blocks onto servers; the application interacts with Chord achieve load balance, data replication, and latency-based server selection [9].

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

We have built several peer-to-peer applications using Chord. The structure of a typical application is shown in Figure 1. The highest layer implements application-specific functions such as file-system meta-data. The next layer implements a general-purpose distributed hash table that multiple applications use to insert and retrieve data blocks identified with unique keys. The distributed hash table takes care of storing, caching, and replication of blocks. The distributed hash table uses Chord to identify the node responsible for storing a block, and then communicates with the block storage server on that node to read or write the block.

IV. 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. In this paper we assume that communication in the underlying network is both symmetric (if A can route to B, then B can route to A), and transitive (if A can route to B and B can route to C, then A can route to C).

A. Overview

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. Chord assigns keys to nodes with *consistent hashing* [12], [14], 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 a $O(1/N)$ 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 other nodes. In an N-node network, each node maintains information about only $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages.

B. Consistent Hashing

The consistent hash function assigns each node and key an m-bit *identifier* using SHA-1 [10] as a base hash function. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term "node" will refer to both the node and its identifier under the hash function. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered on 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. In the remainder of this paper, we will also refer to the identifier circle as the *Chord ring*.

Figure 2 shows a Chord ring with $m = 6$. The Chord ring has 10 nodes and stores five keys. The successor of identifier 10 is node 14, so key 10 would be located at node 14. Similarly, keys 24 and 30 would be located at node 32, key 38 at node 38, and key 54 at node 56.

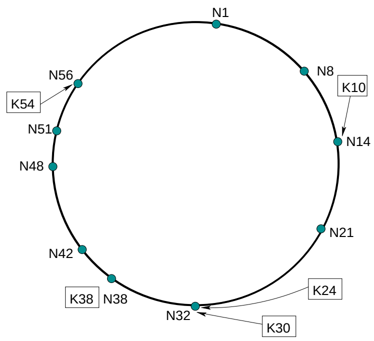


Fig. 2. An identifier circle (ring) consisting of 10 nodes storing five keys.

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 [12], [14]:

Theorem IV.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 $\Omega(\log N)$ *virtual nodes*, each with its own identifier. In the remainder of this paper, we will analyze all bounds in terms of work *per virtual node*. Thus, if each real node runs v virtual nodes, all bounds should be multiplied by v.

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. A similar model is applied to analyze standard hashing. Standard hash functions distribute data well when the set of keys being hashed is random. When keys are not random, such a result cannot be guaranteed—indeed, for any hash function, there exists some key set that is terribly distributed by the hash function (e.g., the set of keys that all map to a single hash bucket). In practice, such potential bad sets are considered unlikely to arise. Techniques have also been developed [3] to introduce randomness in the hash function; given any set of keys, we can choose a hash function at random so that the keys are well distributed with high probability *over the choice of hash function.* A similar technique can be applied to consistent hashing; thus the "high probability' claim in the theorem above. Rather than select a random hash function, we make use of the SHA-1 hash which is expected to have good distributional properties.

Of course, once the random hash function has been chosen, an adversary can select a badly distributed set of keys for that hash function. In our application, an adversary can generate a large set of keys and insert into the Chord ring only those keys that map to a particular node, thus creating a badly distributed set of keys. As with standard hashing, however, we expect that a non-adversarial set of keys can be analyzed as if it were random. Using this assumption, we state many of our results below as "high probability" results.

C. 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.

Figure 3(a) shows pseudocode that implements simple key lookup. 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 of procedure **foo** on node n, while n.*bar*, without parentheses, is an RPC to fetch a variable *bar* from node n. The notation $(a, b]$ denotes the segment of the Chord ring obtained by moving clockwise from (but not including) a until reaching (and including) b.

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.

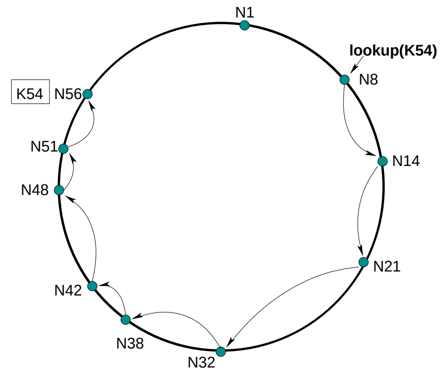
D. Scalable Key Location

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 up to m entries (we will see that in fact only $O(\log n)$ are distinct), 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., $s = \textit{successor}(n+2^{i-1})$, 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 $n.\textit{finger}[i]$ (see Table I). 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*.

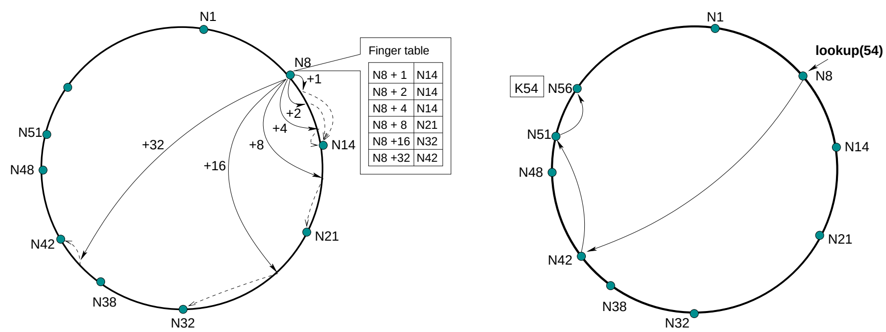
```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the circle
    return successor.find_successor(id);
```

(a)



(b)

Fig. 3. (a) Simple (but slow) pseudocode 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 pseudocode in Figure 3(a).



(a) (b)

Fig. 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.

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

TABLE I

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

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

Fig. 5. Scalable key lookup using the finger table.

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 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 pseudocode of the *find_successor* operation, extended to use finger tables. If id falls between n and its successor, *find_successor* is finished 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 discover 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 identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier. From this intuition follows a theorem:

Theorem IV.2: With high probability, 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. Consider the i such that node p is in the interval $[n + 2^{i-1}, n + 2^i)$. Since this interval is not empty (it contains p), node n will contact its i^{th} finger, the first 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 the interval $[n + 2^{i-1}, n + 2^i)$, 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 $2 \log N$ forwardings, the distance between the current query node and the key k will be reduced to at most $2^m/N^2$. The probability that any other node is in this interval is at most $1/N$, which is negligible. Thus, the next forwarding step will find the desired node. ∎

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

Although the finger table contains room for m entries, in fact only $O(\log N)$ fingers need be stored. As we just argued in the above proof, no node is likely to be within distance $2^m/N^2$ of any other node. Thus, the i^{th} finger of the node, for any $i \leq m - 2 \log N$, will be equal to the node's immediate successor with high probability and need not be stored separately.

E. 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.

E.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 "stabilization" protocol that each node runs periodically in the background and which updates Chord's finger tables and successor pointers.

Figure 6 shows the pseudocode for joins and stabilization. When node n first starts, it calls n.*join*(n'), where n' is any known Chord node, or $n.create()$ to create a new Chord network. 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.

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

// join a Chord ring containing node n'.
n.join(n')
   predecessor = nil;
   successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells 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';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
   next = next + 1;
   if (next > m)
      next = 1;
   finger[next] = find_successor(n + 2^(next-1));

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
   if (predecessor has failed)
      predecessor = nil;
```

Fig. 6. Pseudocode for stabilization.

Every node runs *stabilize*() periodically to learn about newly joined nodes. Each time node n runs *stabilize*(), 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. In addition, *stabilize*() 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.

Each node periodically calls *fix_fingers* to make sure its finger table entries are correct; this is how new nodes initialize their finger tables, and it is how existing nodes incorporate new nodes into their finger tables. Each node also runs *check_predecessor* periodically, to clear the node's predecessor pointer if n.*predecessor* has failed; this allows it to accept a new predecessor in *notify*.

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. Node n_s, when notified by n, acquires n as its predecessor. When n_p next runs *stabilize*(), it asks n_s for its predecessor (which is now n); n_p then acquires n as its successor. Finally, n_p notifies n, and n acquires n_p as its predecessor. At this point, all predecessor and successor pointers are correct. At each step in the process, n_s is reachable from n_p using successor pointers; this means that lookups concurrent with the join are not disrupted. 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_successor*() will reflect the new node. Newly-joined nodes that are not yet reflected in other nodes' finger tables may cause *find_successor*() to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor (*finger*[1]) pointers through the newly-joined nodes until the correct predecessor is reached. Eventually *fix_fingers*() will adjust finger table entries, eliminating the need for these linear scans.

The following result, proved in [24], shows that the inconsistent state caused by concurrent joins is transient.

Theorem IV.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.

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. These pathological cases cannot be produced by any sequence of ordinary node joins. If produced, these cases can be detected and repaired by periodic sampling of the ring topology [24].

E.2 Impact of Node Joins on Lookups

In this section, we consider the impact of node joins on lookups. We first consider correctness. If joining nodes affect 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.

Now let us consider 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 lookup 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. We call a Chord ring *stable* if all its successor and finger pointers are correct.

Theorem IV.4: If we take a stable network with N nodes with correct finger pointers, 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 $\Omega(\log^2 N)$ rounds of stabilization happen between any N node joins.

E.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 initiated by 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. Assuming each node fails independently with probability p, the probability that all r successors fail simultaneously is only p^r. Increasing r makes the system more robust.

Handling the successor list requires minor changes in the pseudocode 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 n reconciles its list with its successor s by copying s's successor list, removing its last entry, and prepending s to it. 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, *fix_fingers* and *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 succes-

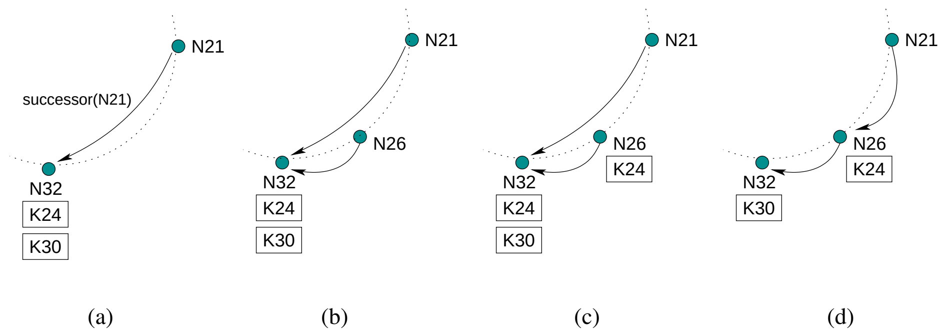


Fig. 7. Example illustrating the join operation. Node 26 joins the system between nodes 21 and 32. The arcs represent the successor relationship. (a) Initial state: node 21 points to node 32; (b) node 26 finds its successor (i.e., node 32) and points to it; (c) node 26 copies all keys less than 26 from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

sor list for the most immediate predecessor of id. In addition, the pseudocode 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 $r = \Omega(\log N)$.

Theorem IV.5: If we use a successor list of length $r = \Omega(\log N)$ 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 any nodes fail, 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 IV.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: Due to space limitations we omit the proof of this result, which can be found in the technical report [24]. ■

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 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. As was discussed above, 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 data to new replicas.

E.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.

F. More Realistic Analysis

Our analysis above gives some insight into the behavior of the Chord system, but is inadequate in practice. The theorems proven above assume that the Chord ring starts in a stable state and then experiences joins or failures. In practice, a Chord ring will never be in a stable state; instead, joins and departures will occur continuously, interleaved with the stabilization algorithm. The ring will not have time to stabilize before new changes happen. The Chord algorithms can be analyzed in this more general setting. Other work [16] shows that if the stabilization protocol is run at a certain rate (dependent on the rate at which nodes join and fail) then the Chord ring remains continuously in an "almost stable" state in which lookups are fast and correct.

V. Simulation Results

In this section, we evaluate the Chord protocol by simulation. The packet-level simulator uses the lookup algorithm in Figure 5, extended with the successor lists described in Section IV-E.3, and the stabilization algorithm in Figure 6.

A. 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 Chord protocol in an iterative style.

During each stabilization step, a node updates its immediate successor and one other entry in its successor list or finger table. Thus, if a node's successor list and finger table contain a total of k unique entries, each entry is refreshed once every k stabilization rounds. Unless otherwise specified, the size of the successor list is one, that is, a node knows only its immediate successor. In addition to the optimizations described on Section IV-E.4, the simulator implements one other optimization. When the predecessor of a node n changes, n notifies its old predecessor p about the new predecessor p'. This allows p to set its successor to p' without waiting for the next stabilization round.

The delay of each packet is exponentially distributed with mean of 50 milliseconds. If a node n cannot contact another node n' within 500 milliseconds, n concludes that n' has left or failed. If n' is an entry in n's successor list or finger table, this entry is removed. Otherwise n informs the node from which it learnt about n' that n' is gone. When a node on the path of a lookup fails, the node that initiated the lookup tries to make progress using the next closest finger preceding the target key.

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.

B. 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 number of keys, we run 20 experiments with different random number generator seeds, counting the number of keys assigned to each node in each experiment. Figure 8(a) plots the mean and the 1st and 99th percentiles of the number of keys per node. The number of keys per node exhibits large variations that increase linearly with the number of keys. For example, in all cases some nodes store no keys. To clarify this, Figure 8(b) plots the probability density function (PDF) of the number of keys per node when there are 5×10^5 keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or $9.1\times$ the mean value. For comparison, the 99th percentile is $4.6\times$ the mean value.

One reason for these variations is that node identifiers do not uniformly cover the entire identifier space. From the perspective of a single node, the amount of the ring it "owns" is determined by the distance to its immediate predecessor. The distance to each of the other $n-1$ nodes is uniformly distributed over the range $[0, m]$, and we are interested in the minimum of these distance. It is a standard fact that the distribution of this minimum is tightly approximated by an exponential distribution with mean $2^m/N$. Thus, for example, the owned region exceeds twice the average value (of $2^m/N$) with probability e^{-2}.

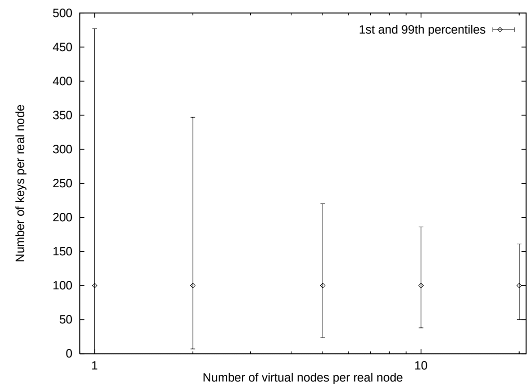


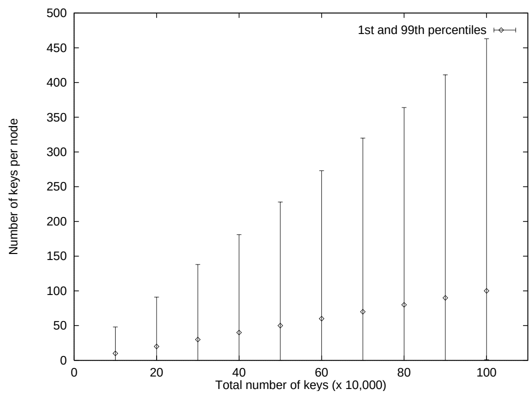
Fig. 9. The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

Chord makes the number of keys per node more uniform by associating keys with virtual nodes, and mapping multiple virtual nodes (with unrelated identifiers) to each real node. This provides 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 $O(\log N)$ virtual nodes [17].

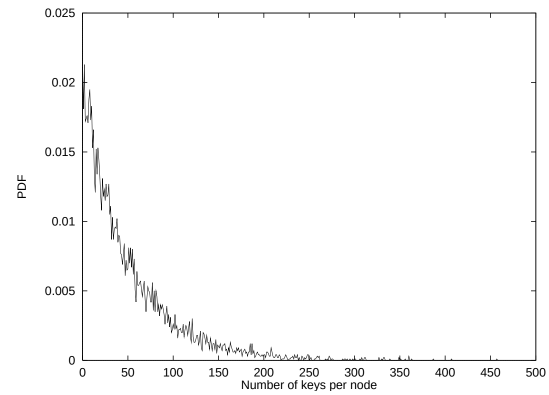
To verify this hypothesis, we perform an experiment in which we allocate r virtual nodes to each real node. In this case keys are associated with virtual nodes instead of real nodes. We consider again a network with 10^4 real nodes and 10^6 keys. Figure 9 shows the 1st and 99th percentiles for $r = 1,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

We make several observations with respect to the complexity incurred by this scheme. First, the asymptotic value of the query path length, which now becomes $O(\log(N \ \log N)) = O(\log N)$, is not affected. Second, the total identifier space covered by the virtual nodes[1] mapped on the same real node is with high probability an $O(1/N)$ fraction of the total, which is the same on average as in the absence of virtual nodes. Since the number of queries handled by a node is roughly proportional to the total identifier space covered by that node, the worst-case number of queries handled by a node does not change. Third, while the routing state maintained by a node is now $O(\log^2 N)$, this value is still reasonable in practice; for $N = 10^6$, $\log^2 N$ is only 400. Finally, while the number of control messages ini-

[1] The identifier space covered by a virtual node represents the interval between the node's identifier and the identifier of its predecessor. The identifier space covered by a real node is the sum of the identifier spaces covered by its virtual nodes.



(a)



(b)

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

tiated by a node increases by a factor of $O(\log N)$, the asymptotic number of control messages received from other nodes is not affected. To see why is this, note that in the absence of virtual nodes, with "reasonable" probability a real node is responsible for $O(\log N/N)$ of the identifier space. Since there are $O(N \log N)$ fingers in the entire system, the number of fingers that point to a real node is $O(\log^2 N)$. In contrast, if each real node maps $\log N$ virtual nodes, with high probability each real node is responsible for $O(1/N)$ of the identifier space. Since there are $O(N \log^2 N)$ fingers in the entire system, with high probability the number of fingers that point to the virtual nodes mapped on the same real node is still $O(\log^2 N)$.

C. Path Length

Chord's performance depends in part on the number of nodes that must be visited to resolve a query. From Theorem IV.2, with high probability, this number 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 each query's path length.

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

Figure 10(a) shows that the path length is about $\frac{1}{2}\log_2 N$. The value of the constant term ($\frac{1}{2}$) can be understood as follows. Consider a node making a query for a randomly chosen key. Represent the distance in identifier space between node and key in binary. 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 node identifiers are randomly distributed, we expect half the of the bits to be ones. As discussed in Theorem IV.2, after the $\log N$ most-significant bits have been fixed, in expectation there is only one node remaining between the current position and the key. Thus the average path length will be about $\frac{1}{2}\log_2 N$.

D. Simultaneous Node Failures

In this experiment, we evaluate the impact of a massive failure on Chord's performance and on its ability to perform correct lookups. We consider a network with $N =$ 1,000 nodes, where each node maintains a successor list of size $r = 20 = 2\log_2 N$ (see Section IV-E.3 for a discussion on the size of the successor list). Once the network becomes stable, each node is made to fail with probability p. After the failures occur, we perform 10,000 random lookups. For each lookup, we record the number of timeouts experienced by the lookup, the number of nodes contacted during the lookup (including attempts to contact failed nodes), and whether the lookup found the key's true current successor. A timeout occurs when a node tries to contact a failed node. The number of timeouts experienced by a lookup is equal to the number of failed nodes encountered by the lookup operation. To focus the evaluation on Chord's performance immediately after failures, before it has a chance to correct its tables, these experiments stop stabilization just before the failures occur and do not remove the fingers pointing to failed nodes from the finger tables. Thus the failed nodes are detected only when they fail to respond during the lookup protocol.

Table II shows the mean, and the 1st and the 99th percentiles of the path length for the first 10,000 lookups after the failure occurs as a function of p, the fraction of failed nodes. As expected, the path length and the number of timeouts increases as the fraction of nodes that fail increases.

To interpret these results better, we next estimate the mean path length of a lookup when each node has a successor list of size r. By an argument similar to the one used in Section V-C, a successor list of size r eliminates the last $\frac{1}{2}\log_2 r$ hops from the lookup path on average. The mean path length of a lookup becomes then $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 r + 1$. The last term (1) accounts for accessing the predecessor of the queried key once this predecessor is found in the successor list of the previous node. For $N = 1,000$ and $r = 20$, the mean path length is 3.82,

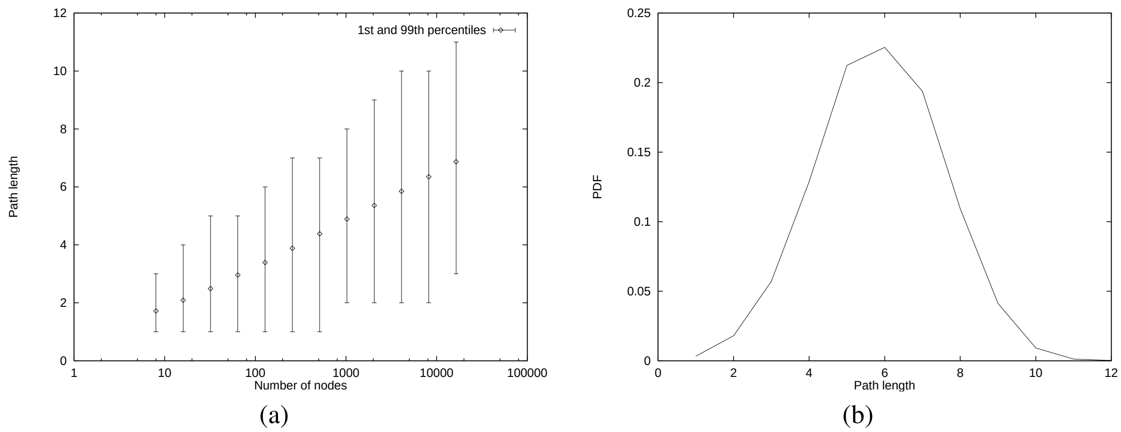


Fig. 10. (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

Fraction of failed nodes	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)
0	3.84 (2, 5)	0.0 (0, 0)
0.1	4.03 (2, 6)	0.60 (0, 2)
0.2	4.22 (2, 6)	1.17 (0, 3)
0.3	4.44 (2, 6)	2.02 (0, 5)
0.4	4.69 (2, 7)	3.23 (0, 8)
0.5	5.09 (3, 8)	5.10 (0, 11)

TABLE II
The path length and the number of timeouts experienced by a lookup as function of the fraction of nodes that fail simultaneously. The 1st and the 99th percentiles are in parenthesis. Initially, the network has 1,000 nodes.

which is very close to the value of 3.84 shown in Table II for $p = 0$.

Let x denote the progress made in the identifier space towards a target key during a particular lookup iteration, when there are no failures in the system. Next, assume that each node fails independently with probability p. As discussed in Section IV-E.3, during each lookup iteration every node selects the largest *alive* finger (from its finger table) that precedes the target key. Thus the progress made during the same lookup iteration in the identifier space is x with probability $(1-p)$, roughly $x/2$ with probability $p*(1-p)$, roughly $x/2^2$ with probability $p^2*(1-p)$, and so on. The expected progress made towards the target key is then $\sum_{i=0}^{\infty} \frac{x}{2^i}(1-p)p^i = x(1-p)/(1-p/2)$. As a result, the mean path length becomes approximately $\frac{1}{2}\log_d N - \frac{1}{2}\log_d r + 1$, where $d = 1.7 = \log_2\left(\frac{1-p/2}{1-p}\right)$. As an example, the mean path length for $p = 0.5$ is 5.76. One reason for which the predicted value is larger than the measured value in Table II is because the series used to evaluate d is finite in practice. This leads us to underestimating the value of d, which in turn leads us to overestimating the mean path length.

Now, let us turn our attention to the number of timeouts. Let m be the mean number of nodes contacted during a lookup operation. The expected number of timeouts experienced during a lookup operation is $m*p$, and the mean path length is $l = m*(1-p)$. Given the mean path length in Table II, the expected number of timeouts is 0.45 for $p = 0.1$, 1.06 for $p = 0.2$, 1.90 for $p = 0.3$, 3.13 for $p = 0.4$, and 5.06 for $p = 0.5$. These values match well the measured number of timeouts shown in Table II.

Finally, we note that in our simulations all lookups were successfully resolved, which supports the robustness claim of Theorem IV.5.

E. Lookups During Stabilization

In this experiment, we evaluate the performance and accuracy of Chord lookups when nodes are continuously joining and leaving. The leave procedure uses the departure optimizations outlined in Section IV-E.4. Key lookups are generated according to a Poisson process at a rate of one per second. Joins and voluntary leaves are modeled by a Poisson process with a mean arrival rate of R. Each node runs the stabilization routine at intervals that are uniformly distributed in the interval $[15, 45]$ seconds; recall that only the successor and one finger table entry are stabilized for each call, so the expected interval between successive stabilizations of a given finger table entry is much longer than the average stabilization period of 30 seconds. The network starts with 1,000 nodes, and each node maintains again a successor list of size $r = 20 = 2\log_2 N$. Note that even though there are no failures, timeouts may still occur during the lookup operation; a node that tries to contact a finger that has just left will time out.

Table III shows the means and the 1st and 90th percentiles of the path length and the number of timeouts experienced by the lookup operation as a function of the rate R at which nodes join and leave. A rate $R = 0.05$ corresponds to one node joining and leaving every 20 seconds on average. For comparison, recall that each node invokes the stabilize protocol once every 30 seconds. Thus, R ranges from a rate of one join and leave per 1.5 stabilization periods to a rate of 12 joins and 12 leaves per one stabilization period.

As discussed in Section V-D, the mean path length in steady state is about $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 r + 1$. Again, since $N = 1{,}000$ and $r = 20$, the mean path length is 3.82. As shown in Table III, the measured path length is very close to this value and does not change dramatically as R increases. This is because the number of timeouts experienced by a lookup is relatively small, and thus it has minimal effect on the path length. On the other hand, the number of timeouts increases with R. To understand this result,

Node join/leave rate (per second/per stab. period)	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)	Lookup failures (per 10,000 lookups)
0.05 / 1.5	3.90 (1, 9)	0.05 (0, 2)	0
0.10 / 3	3.83 (1, 9)	0.11 (0, 2)	0
0.15 / 4.5	3.84 (1, 9)	0.16 (0, 2)	2
0.20 / 6	3.81 (1, 9)	0.23 (0, 3)	5
0.25 / 7.5	3.83 (1, 9)	0.30 (0, 3)	6
0.30 / 9	3.91 (1, 9)	0.34 (0, 4)	8
0.35 / 10.5	3.94 (1, 10)	0.42 (0, 4)	16
0.40 / 12	4.06 (1, 10)	0.46 (0, 5)	15

TABLE III

The path length and the number of timeouts experienced by a lookup as function of node join and leave rates. The 1st and the 99th percentiles are in parentheses. The network has roughly 1,000 nodes.

consider the following informal argument.

Let us consider a particular finger pointer f from node n and evaluate the fraction of lookup traversals *of that finger* that encounter a timeout (by symmetry, this will be the same for all fingers). From the perspective of that finger, history is made up of an interleaving of three types of events: (1) stabilizations of that finger, (2) departures of the node pointed at by the finger, and (3) lookups that traverse the finger. A lookup causes a timeout if the finger points at a departed node. This occurs precisely when the event immediately preceding the lookup was a departure—if the preceding event was a stabilization, then the node currently pointed at is alive; similarly, if the previous event was a lookup, then *that* lookup timed out an caused eviction of that dead finger pointer. So we need merely determine the fraction of lookup events in the history that are immediately preceded by a departure event.

To simplify the analysis we assume that, like joins and leaves, stabilization is run according to a Poisson process. Our history is then an interleaving of three Poisson processes. The fingered node departs as a Poisson process at rate $R' = R/N$. Stabilization of that finger occurs (and detects such a departure) at rate S. In each stabilization round, a node stabilizes either a node in its finger table or a node in its successor list (there are $3 \log N$ such nodes in our case). Since the stabilization operation reduces to a lookup operation (see Figure 6), each stabilization operation will use l fingers on the average, where l is the mean lookup path length.[2] As result, the rate at which a finger is touched by the stabilization operation is $S = (1/30) * l/(3 \log N)$ where $1/30$ is the average rate at which each node invokes stabilization. Finally, lookups using that finger are also a Poisson process. Recall that lookups are generated (globally) as a Poisson process with rate of one lookup per second. Each such lookup uses l fingers on average, while there are $N \log N$ fingers in total. Thus a particular finger is used with probability $l/(N \log N)$, meaning that the finger gets used according to a Poisson process at rate $L = l/(N \log N)$.

We have three interleaved Poisson processes (the lookups, departures, and stabilizations). Such a union of Poisson processes is itself a Poisson process with rate equal to the sum of the three underlying rates. Each time an "event" occurs in this union process, it is assigned to one of the three underlying processes with probability proportional to those processes rates. In other words, the history seen by a node looks like a random sequence in which each event is a departure with probability

$$\begin{aligned} p_t &= \frac{R'}{R' + S + L} = \frac{\frac{R}{N}}{\frac{R}{N} + \frac{l}{90 \log N} + \frac{l}{N \log N}} \\ &= \frac{R}{R + \frac{l*N}{90 \log N} + \frac{l}{\log N}}. \end{aligned}$$

In particular, the event immediately preceding any lookup is a departure with this probability. This is the probability that the lookup encounters the timeout. Finally, the expected number of timeouts experienced by a lookup operation is $l * p_t = R/(R/l + N/(90 \log N) + 1/\log(N))$. As examples, the expected number of timeouts is 0.041 for $R = 0.05$, and 0.31 for $R = 0.4$. These values are reasonable close to the measured values shown in Table III.

The last column in Table III shows the number of lookup failures per 10,000 lookups. The reason for these lookup failures is state inconsistency. In particular, despite the fact that each node maintains a successor list of $2 \log_2 N$ nodes, it is possible that for short periods of time a node may point to an incorrect successor. Suppose at time t, node n knows both its first and its second successor, s_1 and s_2. Assume that just after time t, a new node s joins the network between s_1 and s_2, and that s_1 leaves before n had the chance to discover s. Once n learns that s_1 has left, n will replace it with s_2, the closest successor n knows about. As a result, for any key $id \in (n, s)$, n will return node s_2 instead of s. However, the next time n invokes stabilization for s_2, n will learn its correct successor s.

F. Improving Routing Latency

While Chord ensures that the average path length is only $\frac{1}{2} \log_2 N$, the lookup *latency* can be quite large. This is because the node identifiers are randomly distributed, and therefore nodes close in the identifier space can be far away in the underlying network. In previous work [8] we attempted to reduce lookup latency with a simple extension of the Chord protocol that exploits only the information already in a node's finger table. The idea was to choose the next-hop finger based on both progress in identifier space and latency in the underlying

[2] Actually, since $2 \log N$ of the nodes belong to the successor list, the mean path length of the stabilization operation is smaller than the the mean path length of the lookup operation (assuming the requested keys are randomly distributed). This explains in part the underestimation bias in our computation.

Number of fingers' successors (s)	Stretch (10th, 90th percentiles)			
	Iterative		Recursive	
	3-d space	Transit stub	3-d space	Transit stub
1	7.8 (4.4, 19.8)	7.2 (4.4, 36.0)	4.5 (2.5, 11.5)	4.1 (2.7, 24.0)
2	7.2 (3.8, 18.0)	7.1 (4.2, 33.6)	3.5 (2.0, 8.7)	3.6 (2.3, 17.0)
4	6.1 (3.1, 15.3)	6.4 (3.2, 30.6)	2.7 (1.6, 6.4)	2.8 (1.8, 12.7)
8	4.7 (2.4, 11.8)	4.9 (1.9, 19.0)	2.1 (1.4, 4.7)	2.0 (1.4, 8.9)
16	3.4 (1.9, 8.4)	2.2 (1.7, 7.4)	1.7 (1.2, 3.5)	1.5 (1.3, 4.0)

TABLE IV

The stretch of the lookup latency for a Chord system with 2^{16} nodes when the lookup is performed both in the iterative and recursive style. Two network models are considered: a 3-d Euclidean space, and a transit stub network.

network, trying to maximize the former while minimizing the latter. While this protocol extension is simple to implement and does not require any additional state, its performance is difficult to analyze [8]. In this section, we present an alternate protocol extension, which provides better performance at the cost of slightly increasing the Chord state and message complexity. We emphasize that we are actively exploring techniques to minimize lookup latency, and we expect further improvements in the future.

The main idea of our scheme is to maintain a set of alternate nodes for each finger (that is, nodes with similar identifiers that are roughly equivalent for routing purposes), and then route the queries by selecting the closest node among the alternate nodes according to some network proximity metric. In particular, every node associates with each of its fingers, f, a list of s immediate successors of f. In addition, we modify the *find_successor* function in Figure 5 accordingly: instead of simply returning the largest finger, f, that precedes the queried ID, the function returns the closest node (in terms of networking distance) among f and its s successors. For simplicity, we choose $s = r$, where r is the length of the successor list; one could reduce the storage requirements for the routing table by maintaining, for each finger f, only the closest node n among f's s successors. To update n, a node can simply ask f for its successor list, and then ping each node in the list. The node can update n either periodically, or when it detects that n has failed. Observe that this heuristic can be applied *only* in the recursive (not the iterative) implementation of lookup, as the original querying node will have no distance measurements to the fingers of each node on the path.

To illustrate the efficacy of this heuristic, we consider a Chord system with 2^{16} nodes and two network topologies:

- **3-d space:** The network distance is modeled as the geometric distance in a 3-dimensional space. This model is motivated by recent research [19] showing that the network latency between two nodes in the Internet can be modeled (with good accuracy) as the geometric distance in a d-dimensional Euclidean space, where $d \geq 3$.
- **Transit stub:** A transit-stub topology with 5,000 nodes, where link latencies are 50 milliseconds for intra-transit domain links, 20 milliseconds for transit-stub links and 1 milliseconds for intra-stub domain links. Chord nodes are randomly assigned to stub nodes. This network topology aims to reflect the hierarchical organization of today's Internet.

We use the *lookup stretch* as the main metric to evaluate our heuristic. The lookup stretch is defined as the ratio between the (1) latency of a Chord lookup from the time the lookup is initiated to the time the result is returned to the initiator, and the (2) latency of an optimal lookup using the underlying network. The latter is computed as the round-trip time between the initiator and the server responsible for the queried ID.

Table IV shows the median, the 10th and the 99th percentiles of the lookup stretch over 10,000 lookups for both the iterative and the recursive styles. The results suggest that our heuristic is quite effective. The stretch decreases significantly as s increases from one to 16.

As expected, these results also demonstrate that recursive lookups execute faster than iterative lookups. Without any latency optimization, the recursive lookup style is expected to be approximately twice as fast as the iterative style: an iterative lookup incurs a round-trip latency per hop, while a recursive lookup incurs a one-way latency.

Note that in a 3-d Euclidean space the expected distance from a node to the closest node from a set of $s + 1$ random nodes is proportional to $(s + 1)^{1/3}$. Since the number of Chord hops does not change as s increases, we expect the lookup latency to be also proportional to $(s+1)^{1/3}$. This observation is consistent with the results presented in Table IV. For instance, for $s = 16$, we have $17^{1/3} = 2.57$, which is close to the observed reduction of the median value of the lookup stretch from $s = 1$ to $s = 16$.

VI. Future Work

Work remains to be done in improving Chord's resilience against network partitions and adversarial nodes as well as its efficiency.

Chord can detect and heal 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. 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, this could be an indication for victims that they are not seeing a globally consistent view of the Chord ring.

Even $\frac{1}{2}\log_2 N$ 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 $\log N/(\log(1 + 1/d) \approx O(d \log N)$.

VII. 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 $O(\log N)$ 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 and simulation 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. Our initial experience with Chord has been very promising. We have already built several peer-to-peer applications using Chord, including a cooperative file sharing application [9]. The software is available at `http://pdos.lcs.mit.edu/chord/`.

References

[1] Ajmani, S., Clarke, D., Moh, C.-H., and Richman, S. ConChord: Cooperative SDSI certificate storage and name resolution. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).

[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] Carter, J. L., and Wegman, M. N. Universal classes of hash functions. *Journal of Computer and System Sciences 18*, 2 (1979), 143–154.

[4] 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.

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

[6] 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`.

[7] Cox, R., Muthitacharoen, A., and Morris, R. Serving DNS using Chord. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).

[8] Dabek, F. A cooperative file system. Master's thesis, Massachusetts Institute of Technology, September 2001.

[9] Dabek, F., Kaashoek, F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01* (Banff, Canada, 2001), pp. 202–215.

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

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

[12] 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.

[13] 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.

[14] 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/`.

[15] 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.

[16] Liben-Nowell, D., Balakrishnan, H., and Karger, D. R. Observations on the dynamic evolution of peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).

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

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

[19] Ng, T. S. E., and Zhang, H. Towards global network positioning. In *ACM SIGCOMM Internet Measurements Workshop 2001* (San Francisco, CA, Nov. 2001).

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

[21] 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.

[22] 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.

[23] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001), pp. 329–350.

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

[25] 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.

[26] Zhao, B., Kubiatowicz, J., and Joseph, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.