Analysis of the Evolution of Peer-to-Peer Systems

David Liben-Nowell, Hari Balakrishnan, David Karger
Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square
Cambridge, MA 02139 USA
{dln,hari,karger}@lcs.mit.edu

ABSTRACT

In this paper, we give a theoretical analysis of peer-to-peer (P2P) networks operating in the face of concurrent joins and unexpected departures. We focus on Chord, a recently developed P2P system that implements a distributed hash table abstraction, and study the process by which Chord maintains its distributed state as nodes join and leave the system. We argue that traditional performance measures based on run-time are uninformative for a *continually running* P2P network, and that the *rate* at which nodes in the network need to participate to maintain system state is a more useful metric. We give a general lower bound on this rate for a network to remain connected, and prove that an appropriately modified version of Chord's maintenance rate is within a logarithmic factor of the optimum rate.

1. INTRODUCTION

Peer-to-peer (P2P) systems are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. These systems have recently received significant attention in both academia and industry for a number of reasons. The lack of a central server means that individuals can cooperate to form a P2P network without any investment in additional high-performance hardware to coordinate it. Furthermore, P2P networks suggest a way to aggregate and make use of the tremendous computation and storage resources that remain unused on idle individual computers. Finally, the decentralized, distributed nature of P2P systems makes them robust against certain kinds of faults, making them potentially well-suited for long-term storage or lengthy computations.

P2P systems are, of course, distributed systems, and much traditional distributed systems research is relevant to them. Relatively unusual, however, is the assumption in P2P systems that nodes are continuously joining and leaving the system. This makes challenging several issues which are trivial in a system with a fixed membership. In particular, data items must migrate as nodes come and go; this makes location of a data item at any given time nontrivial. To deal with this problem, a *content addressable network*—as Ratnasamy et al. [10] named this concept—provides a mapping of keys in some keyspace to machines in the network and a lookup protocol to allow any searcher to find the particular machine responsible for any key. For some applications, that node might be responsible for storing a value associated with the key; for others, it might perform computation on that key. The easiest way to implement a content addressable network is to maintain a directory of key assignments; unfortunately, maintaining this directory consistently in a distributed environment is too resource-intensive to scale.

Many P2P routing protocols—like CAN [10], Chord [12], Pastry [3], and Tapestry [14]—induce a connected overlay network across the Internet, with a rich structure that enables efficient key lookups. The typical approach to the design of such overlays is roughly as follows. First, an "ideal" overlay structure is specified, under which key lookups are efficient. Then, a protocol is specified that allows a node to join or leave the network, properly rearranging the ideal overlay to account for their presence or absence. One then can consider the issue of fault tolerance—e.g., showing that the ideal overlay can still route efficiently even after the failure of some fraction of the nodes [3, 4, 12].

The Problem. Unfortunately, the approach described above ignores the fact that a P2P network is a continuously evolving system. The join protocol may work well if joins happen sequentially, but what if many joins happen concurrently? The ideal overlay may tolerate faults, but once those faults occur, the overlay is no longer ideal. What happens as the faults accumulate over time?

To cope with these problems that arise over time, any realistic P2P system must implement some kind of *maintenance protocol* that continuously repairs the overlay as nodes come and go, updating control information and routing tables to ensure that the overlay remains globally connected and supports efficient lookups. In the analysis, we must recognize that the system *almost never* will be in its ideal state. Thus, we must show that lookups, joins, and departures (and, self-referentially, the maintenance protocol itself) behave correctly even in the imperfect overlay.

Evaluating maintenance protocols requires a new performance metric. A P2P system is intended to be running continuously, forever, and system membership is dynamic. Thus, one cannot properly

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

evaluate a maintenance protocol in terms of running time—it runs as long as the network does—or total network bandwidth, which is infinite if the network persists indefinitely. Similarly, the maintenance work done by a single node will tend to grow as the node spends time in the system. Instead, a proper performance measure of a maintenance protocol is the *rate* at which each node must expend resources in the maintenance protocol. This expenditure consumes resources that cannot be used for "real" work, so should be kept small.

We can ask a number of questions in this framework. At what rate must each node in the system do work in order to keep the system in a "good" state? How much work is required to simply provide a connected structure where lookups are correct? How much work is required to provide a richer structure where lookups are correct and also fast?

Our contribution. This paper investigates the per-node *network bandwidth* consumed by maintenance protocols in P2P networks. Network bandwidth (and not storage space or computation time) is presently the most limited resource in P2P systems. If the amount of per-node bandwidth consumed by a maintenance protocol were to grow fairly rapidly (e.g., linearly) as the network size increases, then a system would quickly overwhelm the access bandwidths of its participants and become impractical. Although we focus on bandwidth, both our lower and upper bounds also apply to the rate of CPU usage and storage for maintenance.

Any node joining the network must send at least some number of maintenance messages to let other nodes know of its presence, to provide basic connectivity. Additional messages are usually required to update routing table information on nodes, to support efficient lookups. Similarly, because nodes may fail without any notification, each node must periodically monitor the state of some or all of its neighbors, consuming network bandwidth.[1]

In this paper, we quantify the above observations. First, we give lower bounds on the maintenance protocol bandwidth for basic connectivity in any N-node P2P system as nodes join and leave. We characterize this lower bound in terms of the *half-life* of a distributed system, which essentially measures the time for replacement of half the nodes in the network by new arrivals. We show that per-node maintenance protocol bandwidth, simply to maintain connectivity of the network, is lower-bounded by $\Omega(\log N)$ per half-life. Second, we consider the maintenance protocol used by the Chord P2P routing protocol [12]. We modify this protocol based on our new analysis, and show that the result consumes bandwidth at a polylogarithmic rate, nearly matching our lower bound. Critical to this analysis is a demonstration that Chord's join, lookup, and (properly modified) maintenance protocols work correctly and efficiently even when the system is not in its ideal state.

A note on our model. Our goal in the upcoming sections is to say something formal about the behavior of our system in the real-world setting of the Internet, where nodes can join and leave at arbitrary times, and where messages can be arbitrarily lost or reordered. At the same time, we do not want our algorithm design to overemphasize these worst-case scenarios, since such algorithms are likely to be overcautious and perform poorly in the typical case, when the underlying network is behaving reasonably well. We therefore focus on analyses that model *some* of the problems we might face (e.g., that a large number of nodes may join or fail simultaneously), while making assumptions that preclude more serious failures (e.g., we assume that any two nodes attempting to exchange messages will eventually be able to do so, and that node failures can be detected by their failure to respond "for a long time").

For simplicity, we focus in our presentations on a fail-stop model—that is, we assume perfect failure detection and reliable message delivery. In fact, many of our results apply even in weaker models. None of our algorithms are sensitive to message loss or delivery order; they can provably tolerate these problems to a substantial degree. However, recurring message losses can make a node appear failed when it is actually alive, and our system does not cope with such "false suspicions of failure." In practice, we expect lost messages to lead to timeouts, and consider it reasonable to conclude that repeated timeouts are signs of failure; however, we make no formal claims about this scenario.

To model joins and failures, we postulate an adversary who can specify an arbitrary sequence of node join and failure times and key lookups. We assume the adversary is *oblivious* to the random choices that our protocols will make—that is, that this schedule of joins and failures is specified in advance, independent of behavior of the overlay network. This rules out, e.g., the possibility that the adversary can fail all the nodes that a given node knows about. Against such a powerful, adaptive adversary, the only way to guarantee complete connectivity is to maintain, at tremendous cost, a fully-connected network. Recent work of Saia et al. [11] has begun to tackle the challenge of proving weaker guarantees against an adaptive adversary, e.g., that most of the network remains connected.

2. RELATED WORK

The Chord P2P system was introduced to solve the lookup problem described above. Using storage logarithmic in the size of the network, Chord provides a lookup protocol that can find, in a logarithmic number of routing hops, the node responsible for a given key. Keys in the system are evenly distributed among the participating nodes, providing good load balance. The assignment of keys changes only locally as nodes join and leave the network. Previous publications have described the Chord protocol, the *Cooperative File System* built using it, and experimental results showing that it performs well in practice [1, 2, 12, 13].

Many other recently designed P2P systems provide lookup behavior similar to Chord's. These include CAN [10], Pastry [3], and Tapestry [6, 14] (based on meshes [9]). All offer provably fast lookups and efficient algorithms for node joins. However, the style of evolutionary analysis of P2P networks that we are using has not been well-developed. Many of these P2P systems, such as those of Plaxton et al. [9] and Fiat and Saia [4], focus on models in which nodes join and depart only in a well-behaved fashion, allowing maintenance to happen only at the time of arrival and departure. We believe this kind of well-behaved model is unrealistic. Other protocols such as CAN [10] and Pastry [3] allow for the possibility of unexpected failures, and show that the system is still well-structured after such failures occur. These analyses, however, assume that the system begins in an ideal starting state, and do not show how the system returns to this ideal state after the fail-

[1]Alternatively, a node may choose to detect failures only when it actually needs to contact a neighbor; however, this merely defers the network traffic needed to find a new node when the old one fails. It also raises the risk that all of a node's neighbors fail before it notices any of the failures, permanently disconnecting that node from the network.

ures; thus, accumulation of failures over time eventually disrupts the system.

Recently, Saia et al. [11] have explored the use of a butterfly network in a P2P setting. Their system retains good routing structure even after the *adversarial* removal of a constant fraction of the nodes, and they show how to maintain their network as nodes fail, as long as the number of nodes joining the network is always sufficiently larger than the number of failures. Their assumption is clearly limiting since any system must eventually stop growing.

Perhaps the closest to our evolutionary analysis is the recent work of Pandurangan et al. [8]. They study the problem of maintaining an N-node P2P network as nodes join and depart according to a Poisson process. By using a central server to direct new joins to specific nodes in the network, and to update old nodes' neighbors as nodes depart, they are able to maintain connectivity of the P2P network using only a constant amount of space per node (i.e., each node remembers the identities of only $O(1)$ other nodes). Although this compares favorably to our use of $O(\log N)$ space per node, space is not an expensive resource in practice (at least until we substantially exceed logarithmic space usage). Even more importantly, their scheme does not solve the problem of *routing* within the P2P network: to find the node responsible for a given data item, they propose to flood the network, requiring $\Omega(N)$ messages, while our lookup takes $O(\log N)$ time and $O(\log N)$ messages. Also, their system requires a central server to guarantee connectivity, while ours does not.

3. A HALF-LIFE LOWER BOUND

In this section, we give a general lower bound for the bandwidth of maintenance messages in P2P systems, based on the rate of node joins and departures.

DEFINITION 3.1. *If there are N live nodes at time t, then the* doubling time *from time t is the time that elapses before N additional nodes arrive. The* halving time *from time t is the time required for half of the nodes alive at time t to depart. The* half-life *from time t is the smaller of the doubling and halving times from time t. Finally, the* half-life *of the entire system is the minimum half-life over all times t.*

Intuitively, a half-life of τ means that after time $t + \tau$, at most half the state of the system can be extrapolated from its state at time t. Half-life is a coarse measure of the rate of change of a system; it does not impose any specific conditions on the particular fine-grained pattern of arrivals and departures. Although there are some pathological situations in which the half-life is not a meaningful measure (e.g., the simultaneous failure of almost all nodes in the system), we believe that the concept of half-life is a useful and general characterization of the rate of change of P2P systems in a wide variety of circumstances.

As a specific example, consider a Poisson model of arrivals and departures [8]: nodes arrive according to a Poisson process with rate λ, while a node in the system departs according to an exponential distribution with rate parameter μ (i.e., expected node lifetime is $1/\mu$). If there are N nodes in the system at time t, then the expected doubling time is N/λ and the expected halving time is $(1/\mu)\ln 2$. (The probability p that a node fails in time τ is $1 - e^{-\mu\tau}$; setting $\tau = (1/\mu)\ln 2$ makes $p = 1/2$.) The half-life is then $\min((\ln 2)/\mu, N/\lambda)$.

If λ and μ are fixed and the system is in a steady state, then the arrival rate of λ must be balanced by the departure rate of $N\mu$ (each of N nodes is leaving at rate μ), implying $N = \lambda/\mu$. Then the doubling time is $1/\mu$ and halving time and half-life are both $(1/\mu)\ln 2$. This reflects a general property: in any system where the number of nodes is stable, the doubling time, halving time, and half-life are all equal up to constant factors.

Because nodes in the system are departing frequently, each surviving node must be *notified* of additional nodes in the network to stay connected after its original neighbors fail.

THEOREM 3.2. *There exists a sequence of joins and leaves such that any node that, at any time, has received an average of fewer than k notifications per half-life will be disconnected from the network with probability at least* $(1 - \frac{1}{e-1})^k \approx 0.418^k$.

COROLLARY 3.3. *Any N-node P2P network that remains connected with high probability—i.e., with disconnection probability $O(1/N)$—for any sequence of joins and leaves with half-life τ must notify every node with an average of $\Omega(\log N)$ nodes per τ time.*

PROOF. We consider the above Poisson model, where nodes depart at rate μ (so the half-life is $(1/\mu)\ln 2$). By changing time units, we assume $\mu = 1$. If node n averages fewer than k notifications per half-life then, there must be some time t at which node n has heard about fewer than tk nodes. Consider the minimum such t.

Let $\{n_1, n_2, \dots\}$ be the nodes that node n has heard about by time t, including any nodes of which n was initially aware. Let t_i be the time of notification about n_i (and the last such time if there are several—previous notifications will be irrelevant, since the last notification ensures that n_i is alive at time t_i.) Index such that the t_i are nonincreasing, so that node n_1 was the last node notification.

In our Poisson model, at time t, the probability that node n_i is still alive is $e^{t_i - t}$. It follows that all the nodes n has heard about have failed, disconnecting n from the network, with probability $P = \prod_i (1 - e^{t_i - t})$. By assumption, at any time $t' < t$, at least kt' notifications occurred. It follows that in the time interval $(t - \delta, t)$, fewer than δk notifications occurred. This observation lets us lower bound P. Setting $\delta = 1/k$ tells us that no notifications happened after time $t - 1/k$. In other words, $t_1 \le t - 1/k$. Generalizing, we find $t_i \le t - i/k$. It follows that

$$\begin{aligned} P &= \prod_i (1 - e^{t_i - t}) \;\ge\; \prod_i (1 - e^{-i/k}) \\ &\ge \left(\prod_{j \ge 1} (1 - e^{-j})\right)^k \end{aligned}$$

where the last inequality follows by lower bounding each $e^{-i/k}$ by $e^{-\lceil i/k \rceil}$. Now observe that $\prod(1 - e^{-j}) \ge 1 - \sum e^{-j} = 1 - \frac{1}{e-1}$, and the theorem follows. $\square$

Of course, the node may exceed the k notifications per half-life for quite some time; the theorem does not apply until the average drops sufficiently far. We might worry about an initial condition in which the node is aware of a large number of neighbors. However, the initial number of known neighbors can be at most N and, under the Poisson model, all of these N known neighbors will be gone within $O(\log N)$ half-lives with high probability. Thus the average need be taken only the most recent $O(\log N)$ half-lives.

```
// ask node n to find the successor of id.
$n$.find_successor($id$)
  if ($id \in (n, n.successor]$)
    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 ($finger[i] \in (n, id)$)
      return $finger[i]$;

// join the system using information from node n'.
$n$.join($n'$)
  $predecessor :=$ nil;
  $successor := n'.find\_successor(n)$;
  $build\_fingers(n')$;

// update finger table via searches by node n'.
$n$.build_fingers($n'$)
  $i_0 := \lfloor \log(successor - n) \rfloor + 1$; // first non-trivial finger.
  for each $i \geq i_0$ index into $finger[]$;
    $finger[i] := n'.find\_successor(n + 2^{i-1})$;
```

Figure 1: Pseudocode for the Chord P2P system.

Consider in particular a protocol in which the number of notifications is always bounded by k per half-life. If node n is *not* isolated at the end of those $O(\log N)$ half lives, we can "restart" the above analysis and, after at most $O(\log N)$ additional half-lives, test again whether node n is isolated. Even conditioned on the fact that the node has some neighbors at the beginning of the restart, the above theorem applies. In other words, after each $O(\log N)$ half-lives, the node will become isolated with probability $e^{-\Omega(k)}$. It follows that we expect the node to become isolated within $e^{O(k)}$ half-lives. This result can be strengthened further, under some technical conditions to ensure symmetric behavior of the nodes: given a P2P protocol in which each node sends $o(\log N)$ notifications per half-life in an N-node system, there is a sequence of joins and leaves such that some node becomes isolated from the network with high probability within $O(\log^2 N)$ half-lives. Put more simply, any network involving $o(\log N)$ notifications per half-life will fall apart almost immediately.

4. BACKGROUND ON CHORD

In this section, we outline the Chord P2P system, details of which can be found in an earlier paper [12]. Pseudocode for the protocols is given in Figure 1. The notation $n.f(\cdot)$ means that node n executes procedure $f(\cdot)$, and $n.x$ denotes the value of the variable x stored at node n.

The Chord protocol supports a single operation: given a key, it maps the key onto the node responsible for that key. Chord implements a *distributed hash table*, based on consistent hashing [5, 7]; keys are mapped onto nodes by a hash function that can be resolved by any node in the system, via queries to other nodes. In a steady N-node network, each node needs "routing" information about $O(\log N)$ other nodes, and resolves the hash function by communicating with $O(\log N)$ nodes. We now discuss the mapping, and the mechanism for resolving it, in more detail.

Consistent hashing. Node identifiers (IP addresses) are hashed into m-bit integers where $2^m \gg N$, using some base hash function (we use SHA-1); keys are mapped into the same space. A key is assigned to its *successor* node, the first whose hash follows it modulo 2^m. Pictorially, nodes and keys are mapped onto a circle; the key is assigned to the first node encountered moving clockwise from it; see Figure 2. For ease of exposition, we identify nodes (and keys) and their hashed identifiers. To maintain the mapping when a node n joins, certain keys previously assigned to n's successor are reassigned to n. When node n leaves the network, all of its keys are reassigned to n's successor. No other changes in assignment need to occur. Previous work [5, 7] has shown that consistent hashing does a good job of load balancing keys onto nodes. Intuitively, this follows since the use of an appropriate base hash function means that node and key identifiers can be treated as independent, uniformly distributed random points on the circle. This intuition can be justified formally, and we will make use of it without proof in this paper.[2]

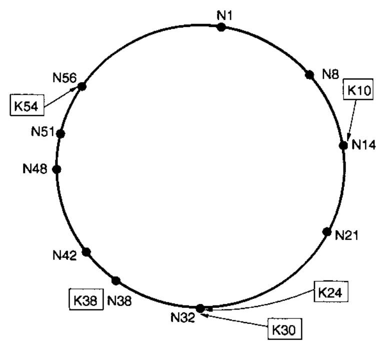

Figure 2: The Chord key-node mapping.

Successor pointers and fingers. Each node stores its *successor node*—the node immediately following it on the circle—so that the successor of a key k can be determined by following successors until we reach a node n with $n < k < n.successor$. Successor pointers are sufficient to guarantee correct lookup of any key's successor. To speed this search, we define the ith *finger* of node n, for $i = 1, \ldots, m$, denoted $n.finger[i]$, the first node to succeed $n + 2^{i-1}$ on the circle. Every node always has some finger pointing halfway to any destination key, so a sequence of $\log N$ "halvings" of the distance take us to the key [12].

Node joins and idealization. When a node n wishes to join the system, it must be integrated into the Chord ring. Node n must set $n.successor$ to point at its immediate successor on the ring, and n's immediate predecessor must update its successor pointer to point at n. Furthermore, node n must set its finger table entries, and certain other nodes should update their fingers to point at n (instead of n's successor).

We allow nodes to join independently without any coordination. These simultaneous joins can destroy the invariants that we want to

[2]We also note that to balance load to within a constant factor, each node must place $\Theta(\log N)$ "virtual copies" of itself on the ring. We ignore this issue in this paper, stating bounds per virtual node. Our bounds are thus precise for a system that uses no virtual nodes, which still yields reasonable load balance, but to within an $O(\log N)$ factor rather than a constant factor. If a system uses multiple virtual nodes, all of our per-node message bounds should be multiplied by the number of virtual copies used by each node.

```
// periodically verify n's successor s, and inform s of n.
// do not run until join() is complete.
n.idealize()
  $x :=$ successor.predecessor;
  if ($x \in (n,$ successor$)$)
    successor $:= x$;
  successor.notify($n$);

// $n'$ thinks it might be our predecessor.
n.notify($n'$)
  if (predecessor $=$ nil or $n' \in$ (predecessor, $n$))
    predecessor $:= n'$;

// periodically refresh finger table entries.
n.fix_fingers()
  build_fingers($n$);
```

Figure 3: Pseudocode for handling joins.

preserve, so each node periodically executes a *idealization* procedure that attempts to reconstruct the desired properties. To perform idealization, each node stores an extra *predecessor* pointer, used to record the closest predecessor that the node has ever heard from. Node n updates its successor to $x = (n.successor).predecessor$ if x falls between n and $n.successor$. Every node runs *idealize*() periodically; this is how older nodes learn about newly joined nodes. Periodically fingers are updated by *fix_fingers*(). See Figure 3.

Departures and fault tolerance. Nodes can also depart the Chord ring, either voluntarily or due to unexpected failures. We might hope that nodes departing voluntarily might "clean up" before departing, but since we need to plan for unexpected failures, which cannot clean up, we make no attempt to define cleanup code for a voluntary departure. Departing nodes simply vanish. In the description above, if a node's successor fails, then the Chord ring is broken and proper lookups cannot take place. To avoid this problem, each node keeps a *successor list* of the first r nodes following it on the ring rather than keeping a single successor pointer.

In Figure 4, we give pseudocode for Chord's operation in the presence of failures. When searching for a node, we may encounter failed nodes along the search path, so *closest_preceding_node*() must check that it is forwarding the search to a live node. Additionally, it must consider nodes in the successor list as candidates for the next hop on the search path. A node n maintains its successor list by repeatedly fetching the successor list of its immediate successor s, removing its last entry, and prepending s to it. If node s fails, then n can replace its successor with the next node on its successor list, and so on. Similarly, node n periodically confirms that its predecessor is alive, and sets $n.predecessor$ to **nil** if not.

5. AN ANALYSIS OF CHORD

As nodes join and leave the system—unexpectedly, and possibly concurrently—the *idealize*() procedure attempts to reconstruct the Chord state described in Section 4. The primary goal of idealization is to achieve that ideal state, but this goal is possible only under certain patterns of joins and leaves. In other cases, the Chord system can only hope to "keep up" with the changes: joins and leaves are happening at too high a rate to progress towards this ideal state, but the system can avoid slipping farther away from ideality. In the remainder of this section, we attempt to quantify the conditions under which this is possible.

A note on our model. For simplicity of presentation, we consider a synchronous model of idealization. With mild complications on

```
// search the local table for the highest predecessor of id.
n.closest_preceding_node(id)
  return the largest node $u$ in finger$[1 \ldots m]$ or successor_list
         so that $u \in (n, id)$ and $u$ is alive;

// periodically reconcile with successor's successor list.
n.fix_successor_list()
  $\langle s_1, \ldots, s_r \rangle :=$ successor.successor_list;
  successor_list $:= \langle$successor$, s_1, s_2, \ldots, s_{r-1}\rangle$;

// periodically update failed successor pointer, if necessary.
n.fix_successor()
  if (successor has failed)
    successor := smallest live node $u$ in
                        finger$[1 \ldots m]$ or successor_list;

// periodically flush predecessor pointer, if necessary.
n.fix_predecessor()
  if (predecessor has failed)
    predecessor := nil;
```

Figure 4: Pseudocode for handling failures.

the definitions that follow, we can handle (without an increase in running time) a network with a reasonable degree of asynchrony, where machines are operating at roughly the same rate, and messages take roughly consistent times to reach their destinations. This is reasonable when differences among machines are small compared to the time between executions of *idealize*(). When the speeds of machines or message deliveries differ by a larger factor f, we can prove analogues of the following results, weakened by that factor f. We refer to a *round* of idealization as the $O(1)$ time required for all nodes to run *idealize*(), disregarding any time required for the transfer of data items.

5.1 What Could Go Wrong

First, we briefly describe some of the problems that might arise in idealizing a network. An immediate concern is the *disconnection* of the network—by changing multiple successor pointers concurrently, we might cause the network to split into two or more separate components and cause any data stored in one component to become inaccessible to nodes in the other. (Recall that we want to use limited space per node, so the simple solution of remembering all nodes ever encountered is infeasible.)

A more subtle difficulty is the creation of a *loopy* cycle. Call a Chord network *weakly ideal* if, for all nodes u in the system, we have $(u.successor).predecessor = u$ and *strongly ideal* if, in addition, for each node u, there is no node v in the same component as u so that $u < v < u.successor$. A *loopy* network is one which is weakly but not strongly ideal. The protocols in Section 4 aim for weak ideality only, a consistency condition that is necessary, but not sufficient, for correct routing in a Chord network. For example, the Chord network shown in Figure 5 is stable under *idealize*(). However, this network is globally inconsistent—in fact, there is no node u so that $u.successor$ is the first node to follow u on the identifier circle. The result of this scenario is that *find_successor*(q) searches from two different nodes in the network, but for the same query q, will return two different nodes, and thus data that is available in the network will appear unavailable to some nodes. In the first part of this section, we show that an initially non-loopy network stays non-loopy through idealization. In Section 5.6, we give a strong idealization algorithm to handle the loopy case if it somehow arises.

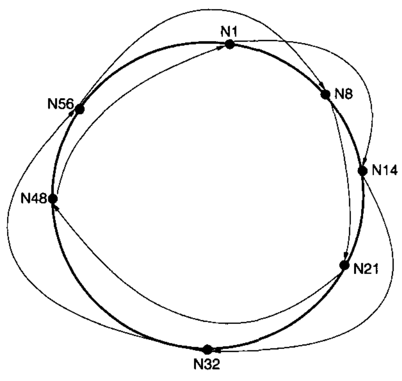

Figure 5: A weakly ideal loopy network. The arrows represent successor pointers, and for every node u, we have $(u.successor).predecessor = u$. However, for every node u, there is a node v_u in the network so that $v_u \in (u, u.successor)$.

5.2 The Ideal Chord State

In our high-level description of the Chord protocol in Section 4, we suggested some of the details of the *ideal state* for Chord; here, we formalize those conditions. Let each successor list have length $c \log N$, for some $c = O(1)$.

Since each Chord node has exactly one successor, 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). In connected networks, this graph is a single *pseudotree*. (When we consider failures, we build this graph using the first live entry in $u.successor\_list$ for each u.)

If this pseudotree does not consist solely of a cycle, then for some nodes u in the cycle, there is a non-empty tree of nodes rooted at u, consisting of nodes that have recently joined the network and are not yet in the cycle. We refer to this rooted tree as u's *appendage*, and denote it $\mathcal{A}_u$. The ideal Chord state has no appendages.

DEFINITION 5.1. *A Chord network is in the* ideal state *if:*

1. **[connectivity]** *There is a path using successor lists and finger tables connecting any two nodes.*
2. **[randomness].** *All the nodes in the system are independently and uniformly distributed around the identifier circle.*
3. **[cycle sufficiency]** *Every node u is on the cycle.*
4. **[non-loopiness]** *For any node u on the cycle, there is no node $v \in (u, u.successor)$.*
5. **[successor list validity]** *Every $u.successor\_list$ contains the first $c \log N$ nodes that follow u.*
6. **[finger validity]** *For every node u and every i, the first node following $u + 2^{i-1}$ is stored as $u.finger[i]$.*

Previous work [12] has established a number of good properties of this ideal state: the procedure *find_successor*(q) returns the true successor of q in time $O(\log N)$, even after all nodes fail independently with constant probability $p < 1$; furthermore, starting from a Chord network in an ideal state and allowing an arbitrary sequence of possibly concurrent joins, the network eventually becomes ideal again. In the remainder of this section, we establish that similar properties hold even in states that are only "close" to ideal.

5.3 A Pure Failure Model

Consider an N-node Chord network in which some nodes have failed recently, and some other nodes may fail soon. In this setting, some of the entries in successor lists may be out of date, containing nodes that have already failed.

Given an N-node ideal network, suppose that $N/2$ nodes in the network fail (obliviously to their identifier, and thus random in identifier space). With high probability, at least one of the nodes in any given $u.successor\_list$ does not fail; thus the network remains connected and non-loopy, and all nodes remain on the cycle. In fact, with high probability, at least one-third of the nodes in any given $u.successor\_list$ do not fail, which means that in the resulting state each successor list consists of at least the first $(c/3) \log N$ live nodes that follow u on the cycle. (This type of argument, which we use frequently in this paper, is based on the Chernoff bound—in expectation, half of the $\Theta(\log N)$ nodes in the successor list fail; thus, with high probability, no more than two-thirds fail.) Any $u.finger[i]$ which did not fail is the first live node following $u + 2^{i-1}$, since all fingers were correct in the ideal state. Consider the resulting $N/2$-node Chord network, and suppose that nodes continue to fail. We would like this network to retain the good properties of the ideal state—namely, this same sort of fault tolerance, and fast and efficient lookups.

DEFINITION 5.2. *A Chord network is in the* cycle with failures *state if:*

1,3,4. As in Definition 5.1.

2. **[randomness]**
   (a) *As in Definition 5.1.*
   (b) *All nodes in the system are alive with probability at least 1/3, even conditioned on the liveness of an arbitrary subset of up to N/4 other nodes.*
5. **[successor list validity]** *For every node u, let L_u denote the live entries in $u.successor\_list$.*
   (a) *Every $|L_u| \geq (c/3) \log N$.*
   (b) *Every L_u contains exactly the first $|L_u|$ live nodes that follow the node u.*
6. **[finger validity]** *For every node u and index i, if $u.finger[i]$ is alive, then it is the first live node following $u + 2^{i-1}$.*

Our analysis will consider starting from a network already in the cycle with failures state, and Constraint 2(b) imposes the condition that any previous failures of nodes in the system were random. (This is in fact somewhat stronger than we need—we only apply this fact to show that, at every stage of a *find_successor*(q) search, with constant probability, the query can be forwarded to the node returned by *closest_preceding_node*(q). Thus we only apply 2(b) to the set of $O(\log^2 N)$ nodes previously encountered along a search path.)

For intuition, consider a network in the cycle with failures state. We claim that if no additional failures occur then the network will become ideal within a small number of rounds of idealization. Each node u has its true live successor as the first live entry in its successor list, so after r rounds of reconciling u's successor list with $u.successor$'s successor list, the first r entries of $u.successor\_list$ will be the first r live successors of u. Similarly, after each finger $u.finger[i]$ is updated by running $u.find\_successor(u+2^{i-1})$, the correctness of *find_successor*() ensures that all fingers will be accurate. In fact, this intuition can be leveraged to show that a net-

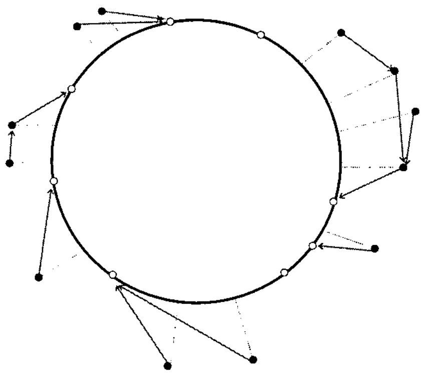

Figure 6: A Chord network in the cycle with appendages state. Unfilled nodes are on the cycle; filled nodes are in appendages.

work in the cycle with failures state remains in the cycle with failures state, as long sufficiently many rounds of idealization occur in a halving time:

LEMMA 5.3. *Consider an N-node Chord network in the cycle with failures state, and suppose that up to $N/2$ oblivious failures occur at any time during the execution of at least $\Omega(\log N)$ rounds of idealization. Then, with high probability,*

1. *Throughout this process, find_successor(q) returns the first living successor of q and runs in time $O(\log N)$.*
2. *The resulting network is in the cycle with failures state.*

The proof of this lemma relies on three key facts:

(1) The network remains connected with high probability—this holds since each successor list has $\Theta(\log N)$ live entries and nodes fail with constant probability;
(2) *find_successor*() is efficient—this holds since, if each finger is up with constant probability, then each forwarding halves the distance to the query with constant probability (even conditioned on the liveness of the nodes encountered so far on the search path), so the total number of hops in the search path is $O(\log N)$ with high probability; and
(3) Successor lists have purged all "old" failures—this holds because, after r rounds of successor list updates, the first r entries must have been alive when the process began.

5.4 A Pure Join Model

We now shift our attention from failures to joins, and consider a Chord network in which nodes can join the system, but no node ever departs. Since node failures are not in our model, we will for simplicity consider a Chord network with no successor lists.

When nodes are joining the network, we must again relax a number of the conditions of Definition 5.1. A node u that has recently joined the network may not be on the cycle because the node $s = u.successor$ has not yet been informed of u's presence, and thus $p = s.predecessor \neq u$. If s and p are nodes that are already on the cycle, then until p learns about u, the node u will not be in the cycle. Furthermore, successor pointers will not be correct with respect to nodes not on the cycle. In addition, finger pointers will not be completely up to date, since nodes may have recently joined between $u + 2^{i-1}$ and the current finger $u.finger[i]$. See Figure 6.

DEFINITION 5.4. *A N-node Chord network is in the* cycle with appendages state *if:*

1-2. *As in Definition 5.1.*
3. **[cycle sufficiency]**
   (a) *Of the nodes on the cycle, a subset of size at least $N/2$ is uniformly and independently distributed around the identifier circle.*
   (b) *For any cycle node u, we have $|\mathcal{A}_u| = O(\log N)$.*
4. **[non-loopiness]**
   (a) *The cycle is non-loopy.*
   (b) *For every node v in the appendage $\mathcal{A}_u$, the path of successors from v to u is increasing.*
5. **[successor validity]** *For every node v:*
   (a) *if v is on the cycle, then $v.successor$ is the first cycle node following v.*
   (b) *if v is in appendage $\mathcal{A}_u$, then u is the first cycle node following v.*
6. **[finger validity]** *There is a set S of $N/2$ nodes on the cycle that are uniformly and independently distributed so that, for every node u and every $1 \leq i \leq m$, no element of S ever falls between $u + 2^{i-1}$ and $u.finger[i]$.*

Constraint 3(a) guarantees that the nodes in the network are "well-distributed" on the identifier circle: most of the nodes are in the cycle, and at least a constant fraction of these nodes are spread randomly across the identifier circle. (There will be bias in the order in which recently joined nodes are incorporated into the cycle—e.g., towards fast joining for nodes that fall nearby nodes already on the cycle, or for nodes that fall between two nodes on the cycle which are close to each other—so it is not the case that the distribution of all nodes on the cycle is uniform and independent.)

Constraint 4(b) ensures that all paths leading to the cycle of the pseudotree are non-loopy, in the sense that following them to the cycle never goes around the identifier space more than once. This is necessary to ensure that *find_successor*() operates correctly when it is invoked by a node in an appendage—without this condition, a search $v.find\_successor(q)$ by a node $v \in \mathcal{A}_u$ can return a result in $\mathcal{A}_u$ even when the correct node is in the cycle.

Constraint 6 ensures that all fingers are sufficiently accurate to allow fast lookups—all fingers are correct with respect to a constant fraction of the nodes in the system.

Given an N-node network in the ideal state, if N additional nodes join (bootstrapping on any node in the network), then the result is a network in the cycle with appendages state. The correctness of search via *find_successor*() implies that a joining node n sets its successor to either its true cycle successor s, or to some other node in $\mathcal{A}_s$. This guarantees properties 1-2 and 4-5. For cycle sufficiency, the existing N nodes form the requisite subset, and, with high probability, only $O(\log N)$ joining nodes have identifiers that fall between any two existing nodes; thus each appendage has size $O(\log N)$. For finger validity, the set S consists of all N existing nodes; since all fingers are built with these already in the cycle, all fingers are correct with respect to the elements of S.

Similarly, consider a network in the cycle with appendages state, and suppose that no further nodes join the system over $\Omega(\log^2 N)$ rounds of idealization. In each round of idealization, a node from any non-empty appendage $\mathcal{A}_u$ enters the cycle, since one cycle node p_c and at least one appendage node p_a both think that u their successor. The idealization procedure will then adjust the successor of the node farther from u (which is p_c) to point the node closer to u (which is p_a), incorporating p_a into the cycle. Thus in $O(\log N)$ rounds, all nodes in appendages are incorporated into the cycle. Within one subsequent full round of finger updates, all fingers are correct with respect to all nodes. The result is an ideal network.

Again, we can combine these two intuitive arguments to show that the cycle with appendages state can be maintained over time:

LEMMA 5.5. *Consider an N-node Chord network in the cycle with appendages state, and suppose that N nodes join the network, each using an arbitrary node to bootstrap on, over at least $\Omega(\log^2 N)$ rounds of idealization. Then, with high probability,*

1. *Throughout this process, find_successor(q) returns the cycle successor s of q or a node u in $\mathcal{A}_s$ so that $q \leq u < s$, and runs in $O(\log N)$ time.*
2. *After this process, the resulting network is in the cycle with appendages state.* □

We establish the efficiency of *find_successor*() by observing that fingers are correct with respect to $N/2$-sized random subset on the cycle, and the remaining nodes are randomly distributed, so with high probability only $O(\log N)$ new nodes fall between the correct node and the result of the search using the old fingers. Thus only $O(\log N)$ additional steps are required to find the true cycle successor of the query. As in the healing of a cycle with appendages network into ideal, some appendage node is incorporated into the cycle in $O(1)$ rounds; since, with high probability, appendages are only $O(\log N)$ in size even after N joins, after $O(\log N)$ rounds all old nodes are incorporated into the cycle. Since *find_successor*() runs in logarithmic time and there are only $O(\log N)$ distinct fingers with high probability, the time required to update fingers is $O(\log^2 N)$, so after $O(\log^2 N)$ rounds of idealization after all the old nodes are incorporated into the cycle, the network is back in the cycle with appendages state.

5.5 A Fully Dynamic Model

Finally, we simultaneously consider joins and failures. The intuition of the previous sections applies here almost directly: most of the conditions that we impose were imposed for either the pure join or failure case. A few conditions must be modified slightly to account for interactions between joins and failures.

DEFINITION 5.6. *A N-node Chord network is in the* cycle with failures and appendages state *if, for some constant D:*

1. **[connectivity].** *The network is connected.*
2. **[randomness].**
   (a) *All the nodes in the system are independently and uniformly distributed around the identifier circle.*
   (b) *All nodes in the system are alive with probability at least $1/3$, even conditioned on the liveness of an arbitrary subset of up to $N/4$ other nodes.*
3. **[cycle sufficiency]**
   (a) *Of the nodes on the cycle, a subset of size at least $N/3$ is uniformly and independently distributed around the identifier circle.*
   (b) *For any consecutive cycle nodes $u_1, \ldots, u_{\log N}$, we have $\sum_{i=1}^{\log N} |\mathcal{A}_{u_i}| = O(\log N)$.*
4. **[non-loopiness]**
   (a) *The cycle is non-loopy.*
   (b) *For every node v in the appendage $\mathcal{A}_u$, the path of successors from v to u is increasing.*
5. **[successor validity]** *For every node v, let L_v denote the live entries in v.successor_list.*
   (a) *Every $|L_v| \geq (c/3)\log N$.*
   (b) *if v is on the cycle, then v.successor is the first live cycle node following v.*
   (c) *if v is in appendage $\mathcal{A}_u$, then u is the first live cycle node following v.*
   (d) *if the successor list of u.successor skips over a live node v, then v is not in u.successor_list.*
   (e) *No successor list contains nodes that failed more than $D\log^2 N$ rounds ago.*
   (f) *No successor list skips any live node that entered the cycle more than $D\log^2 N$ rounds ago.*
6. **[finger validity]**
   (a) *There is a set S of at least $N/3$ nodes on the cycle that are uniformly and independently distributed so that, for every node u and every $1 \leq i \leq m$, no element of S ever falls between $u + 2^{i-1}$ and u.finger$[i]$.*
   (b) *For each i, if u.finger$[i]$ is alive, then it is at least as close to $u + 2^{i-1}$ as the first live node of S following $u + 2^{i-1}$.*

The network in Figure 6 is also in the cycle with failures and appendages state, with appropriate conditions on the state of the successor lists.

Intuitively, condition 5(d) is a consistency condition between the successor lists of adjacent nodes on the cycle; it guarantees that node u only adds new nodes to its successor list by learning them from its successor. Without this condition, the cycle may become loopy as additional nodes fail. Conditions 5(e,f) ensure that the successor lists are reasonably current.

THEOREM 5.7. *Start with a network of N nodes in the cycle with failures and appendages state with successor lists of length $c\log N$, and allow up to N oblivious joins and $N/2$ oblivious failures at arbitrary times over at least $D\log^2 N$ rounds of idealization, for $D = O(1)$. Then, with high probability,*

1. *Throughout this process, find_successor(q) returns the first live cycle successor s of q or a node u in $\mathcal{A}_s$ so that $q \leq u < s$, and runs in $O(\log N)$ time.*
2. *The resulting network is in the cycle with failures and appendages state.*

PROOF. Since the joins and failures are oblivious, they correspond to random identifiers in the network.

In a network with successor lists of length $c\log N$, we will say that a node u is *fully incorporated into the cycle* iff it has been in the cycle for at least $c^2\log N$ 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.

We distinguish between "old" nodes which have been present for longer than $D \log^2 N$ rounds, "middle-aged" nodes which have been present for less time, and the at most N "new" nodes which join during the current $D \log^2 N$ rounds of idealization. By definition of the cycle with failures and appendages 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, only $O(\log N)$ new nodes join between any two old nodes. This implies that no node ends up with too many nodes in its appendage during the time period being analyzed. Cycle nodes can fail, causing their appendages to merge together, but with high probability only $O(\log N)$ consecutive cycle nodes fail, so the size of an appendage is $O(\log N)$ by 3(b), with high probability, including middle-aged and new nodes.

Unfortunately, because of failures, it is not true that a node from each appendage (fully) enters the cycle in each round, since the cycle node that points to it may fail immediately. However, with high probability, within $O(\log N)$ rounds, some node from each appendage will become fully incorporated into the cycle: in $O(\log N)$ attempts for a node $v \in \mathcal{A}_u$ to become incorporated into the cycle, it will begin to join the cycle when the cycle node p that sets $p.successor = v$ does not fail in this entire process. Once a node is fully incorporated into the cycle, with high probability, it never leaves, so within $O(\log^2 N)$ rounds, all $O(\log N)$ middle-aged nodes in each appendage will join the cycle.

The correctness and efficiency of *find_successor*() follow just as in Lemmas 5.3 and 5.5.

After all the middle-aged nodes enter the cycle, we require an additional $O(\log^2 N)$ rounds of idealization to ensure that all of the fingers are correct with respect to the middle-aged nodes, since *fix_fingers*() runs in $O(\log^2 N)$ time since lookups require only $O(\log N)$ time, by the above. □

That is, so long as Chord executes $O(\log^2 N)$ rounds of idealization per half-life, the network remains in this cycle with failures and appendages state, in which search is efficient and correct.

What we have deemed as the "correctness" of the search procedure is somewhat subtle, though the returned node is correct in the following sense: at the instant that each *find_successor*(k) terminates, it yields a node v that is responsible for a key range including k. If v does not hold the key k, one of the following cases holds: (1) k is not yet available because it is being held at a node in an appendage (but, by Condition 5(f), it will join the cycle within a half-life); (2) v is on the ring and responsible for the key k, but is in the process of transferring keys from its successor (but this transfer will complete quickly, and then v will have key k); or (3) v was previously responsible for the key k, but has since transferred k to another node. We can handle (3) by modifying the algorithm to have each node maintain a copy of all transferred data for one half-life after the transfer.

5.6 Loopy States

In Section 5.5, we established that, under our model of joins and departures, Chord's idealization protocol maintains a state in which routing is correct and fast with high probability; the protocols in Figure 1 maintain strong ideality in a strongly ideal network. Thus as long as all nodes operate according to this protocol, it would seem that our network will be strongly ideal, so that our lookups will be correct. But, fearful of bugs in an implementation, or a breakdown in our join/departure model, or the eventual occurrence of low probability events, we now wish to take a more cautious view. (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.)

```
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;
  successor[i].notify(n);

n.idealize()
  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 7: Pseudocode for strong idealization.

In this section, we extend the Chord protocol to idealize the network from an *arbitrary* state, even one not reachable by correct operation of the protocol. This protocol does not reconnect a disconnected network; we rely on some external means to do so. Our approach is in keeping with our focus on the behavior of our system *over time*—over a sufficiently long period of time, extremely unlikely events (such as the simultaneous failure of all nodes in a successor list) can happen; we need to cope with them.

The idealization protocol of Figure 1 guarantees that all nodes have indegree and outdegree one, so a weakly ideal network consists of a topological cycle, but one in which successors might be incorrect. For a node u, call *u's loop* the set of nodes found by following successor pointers starting from u and continuing until we reach a node w so that $w.successor \geq u$. In a loopy network, there is a node u so that u's loop is a strict subset of the set of nodes in the same component as u, and lookups may not be correct.

The fundamental idealization operation by which we unfurl a loopy cycle is based upon *self-search*, wherein a node u searches for itself in the network. If the network is 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 with $w.predecessor < u < w$ found by following successor pointers. We extend our previous idealization protocol by allowing each node u to maintain a second successor pointer, generated by self-search and improved in exactly the same way as in the previous protocol. See Figure 7.

THEOREM 5.8. *Within $O(N^2)$ rounds of strong idealization, an arbitrary connected Chord network becomes strongly ideal.*

PROOF. 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 ideal but not strongly ideal, then at least one node will find a improved second successor when it performs its self-search. Having ruled out the "wrong" weak idealization, 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 $\mathit{successor}[0] \neq \mathit{successor}[1]$) 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 in weak idealization, this is not a stable situation: the closer predecessor p will 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 node's true successor in the network.

Observe that a loopy Chord network will never permit any nodes to join until its loops merge—in a loopy network, for all u, we have $u.\mathit{on\_cycle}$ = **false**, since u's self-search never returns u. Thus, if the network somehow finds its way into a loopy state, it will heal itself within $O(N^2)$ rounds: each of the N successor pointers can improve at most N times. □

While the runtime of our strong idealization protocol is large, recall that it needs to be invoked *only* when the system gets into a pathological state. Such pathologies ought to be extremely rare, which means that the lengthy recovery is a small fraction of the overall lifetime of the system. Nonetheless, it would clearly be preferable to develop a strong idealization protocol that, like weak idealization, simply executes at a low rate in the background, rather than bringing everything else to a halt for lengthy periods.

Strong idealization in the presence of failures. As before, maintaining a successor list of length $\Theta(\log N)$ will ensure that our graph, with high probability, 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, at most N failures can occur before the network is strongly ideal (or has disappeared), since, as discussed above, no nodes can join a loopy network. 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 $u.\mathit{successor\_list}$. So we may slip backwards using the sense of "progress" from Theorem 5.8. However, there are at most N failures before the network empties. We can only have $O(N^2)$ improvements after any of the N failures before we are strongly ideal, so we have the following:

THEOREM 5.9. *Start from an arbitrary connected state with successor lists of length $\Theta(\log N)$. Allow $O(N)$ failures over $\Omega(\log N)$ steps. Then, with high probability, in $O(N^3)$ rounds, the network is strongly ideal.*

6. CONCLUSION

We have described the operation of Chord in a general model of evolution involving joins and departures. We have shown that a limited amount of housekeeping work per node allows the system to resolve queries efficiently. There remains the possibility of reducing this housekeeping work by logarithmic factors.

Our current scheme postulates that the half-life of the system is known; an interesting question is whether the correct maintenance rate can be learned from observation of the behavior of neighbors. Another direction for further work is the analysis of Chord with limited forms of message loss: the eventual idealization from any non-loopy state into an ideal network continues to hold as long as any two machines that attempt to communicate eventually succeed, and we have some preliminary results suggesting that something much like Theorem 5.7 can be extended to such a model. A significant challenge is identifying a good mildly pessimistic model for loss of messages.

A further area to address is recovery from pathological situations. Our protocol exhibits slow recovery from certain pathological "disorderings" of the Chord ring. Although it is of course impossible to recover from total disconnection, an ideal protocol would recover quickly from any state in which the system remained connected.

7. REFERENCES

[1] 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 *Proc. IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (2001).

[2] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. SOSP* (2001).

[3] 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-VIII)* (2001), pp. 65–70.

[4] FIAT, A., AND SAIA, J. Censorship resistant peer-to-peer content addressable networks. In *Proc. SODA* (2002).

[5] 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 *Proc. STOC* (1997).

[6] 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 *Proc. ASPLOS* (2000).

[7] 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/`.

[8] PANDURANGAN, G., RAGHAVAN, P., AND UPFAL, E. Building low-diameter P2P networks. In *Proc. FOCS* (2001).

[9] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. SPAA* (1997).

[10] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. SIGCOMM* (2001).

[11] SAIA, J., FIAT, A., GRIBBLE, S., KARLIN, A. R., AND SAROIU, S. Dynamically fault-tolerant content addressable networks. In *Proc. IPTPS* (2002).

[12] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM* (2001).

[13] 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/`.

[14] 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.