# Observations on the Dynamic Evolution of Peer-to-Peer Networks

David Liben-Nowell, Hari Balakrishnan, and David Karger*

Laboratory for Computer Science
Massachusetts Institute of Technology
{dln,hari,karger}@lcs.mit.edu

http://pdos.lcs.mit.edu/chord

**Abstract.** A fundamental theoretical challenge in peer-to-peer systems is proving statements about the evolution of the system while nodes are continuously joining and leaving. Because the system will operate for an infinite time, performance measures based on runtime are uninformative; instead, we must study the *rate* at which nodes consume resources in order to maintain the system state.

This "maintenance bandwidth" depends on the rate at which nodes tend to enter and leave the system. In this paper, we formalize this dependence. Having done so, we analyze the Chord peer-to-peer protocol. We show that Chord's maintenance bandwidth to handle concurrent node arrivals and departures is near optimal, exceeding the lower bound by only a logarithmic factor. We also outline and analyze an algorithm that converges to a correct routing state from an arbitrary initial condition.

## 1 Introduction

Peer-to-peer (P2P) routing protocols like CAN [?], Chord [?], Pastry [?], and Tapestry [?] 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 goes roughly as follows. First, an "ideal" overlay structure is specified, under which key lookups are efficient. Then, a protocol is specified that allows nodes to join or leave the network, properly rearranging the ideal overlay to account for their presence or absence. Finally, fault tolerance may be discussed: one can show that the ideal overlay can still route efficiently even after the failure of some fraction of the nodes.

Such an approach 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 happen concurrently? The ideal overlay may tolerate faults,

* 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, by NSF contract CCR-9624239, and by a Packard Foundation fellowship.

but once those faults occur, the overlay is no longer ideal. So what happens as the faults continue to accumulate over time?

To cope with these problems, any realistic P2P system must implement some kind of *maintenance protocol* that continuously repairs the overlay as nodes come and go, ensuring that the overlay remains globally connected and supports efficient lookups. In analyzing this maintenance protocol, we must recognize that the system is unlikely ever to be in its ideal state. Thus, we must show that lookups and joins (and the maintenance protocol itself) occur correctly even in the imperfect overlay.

Because a P2P system is intended to be running continuously and system membership is dynamic, the time taken to maintain the system's state is not a proper measure of resource usage; rather, what matters is how much resource bandwidth is consumed by nodes in maintaining control information in the form of routing tables and other such data structures.

This paper investigates the per-node network bandwidth consumed by maintenance protocols in P2P networks. We are motivated by the observation that this property—which addresses how much work each node must do in the interests of providing connectivity and a good topological structure—may be an important factor in determining the long-term viability of large-scale, dynamic P2P systems. For instance, if the per-node bandwidth consumed by these maintenance protocols 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.

Any node joining the network must send at least some number of housekeeping 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, so that efficient lookups can then occur. 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]

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 simply to 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?

To answer these questions, we make two kinds of observations about P2P maintenance protocols. First, we give lower bounds on the maintenance protocol bandwidth for connectivity in any P2P network as nodes join and leave. We characterize this lower bound using the notion of *half-life*, 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 is lower-bounded by $\Omega(\log N)$ per half-life for any P2P system that wishes to remain connected

[1] Alternatively, a node may detect failures only when it actually needs to contact a neighbor; however, this merely defers the network traffic for finding a new neighbor until the old one fails. It also raises the risk that all of a node's neighbors fail without the failures being noticed, permanently disconnecting that node from the network.

with high probability.[2] Second, we analyze the maintenance protocol used by Chord [?], a P2P routing protocol. We show that Chord consumes bandwidth only logarithmically larger than our lower bound. It is noteworthy our system provides fast lookup at a resource cost not much greater than the minimum necessary merely to maintain connectivity. Critical to this analysis is a demonstration that Chord's join, lookup, and maintenance protocols work correctly even when the system is not in its idealized stable state.

This style of evolutionary analyses of P2P networks has not been well-developed. Many P2P systems 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 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 failures; thus, accumulation of failures over time eventually disrupts the system. (See, e.g., [?,?,?,?,?].)

Recently, Saia et al. [?] 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. In their system, keeping a well-structured network as nodes join is more difficult, and their results to do not apply to a steady state in which the number of nodes in the system remains constant.

Perhaps the closest to our evolutionary analysis is the recent work of Pandurangan et al. [?], who study a centralized, flooding-based P2P protocol. Using a Poisson arrival/departure model, they show that their protocol results in an overlay network whose diameter remains logarithmic, with high probability. However, 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 flooding the network, requiring $\Omega(N)$ messages. Also, their system requires a central server to guarantee connectivity.

We believe that our evolutionary analysis, with its recognition that the ideal state will rarely occur, is crucial for proper understanding of P2P protocols in practice.

This note summarizes work which is reported in full in [?].

[2] Throughout this paper, *with high probability* (abbreviated *whp*) means with probability at least $1-1/N$. In a system with $N$ nodes, events that happen with probability exceeding $1/N$ become "expected" (i.e., the expected number of nodes at which the event occurs exceeds one). Thus, it is standard in theoretical analysis to attempt to ensure that bad events occur with probability at most $1/N$. As is usual, the proof shows how parameters can be varied to achieve any desired bound on the failure probability, e.g., $1/N^2$ as opposed to $1/N$.

## 2 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 1.** *Consider a P2P system with $N$ nodes alive at time $t$. The* doubling time at time $t$ *is time that it takes for $N$ additional nodes to arrive. The* halving time at time $t$ *is the time that elapses before half of the nodes alive at time $t$ depart. The* half-life at time $t$ *is the smaller of the doubling and halving times at time $t$. Finally, the* half life *of the entire system is the minimum half-life over all times $t$.*

Intuitively, a half-life of $\tau$ means that after time $t + \tau$, only 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, and does not impose any specific conditions on the particular fine-grained pattern of arrivals and departures; a half-life of $\tau$ can result from a steady stream of node joins and failures or from the simultaneous joins or failures of a massive number of nodes. 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/departures [**?**]: nodes arrive according to a Poisson process with rate $\lambda$, while each node in the system departs independently according to an exponential distribution with rate parameter $\mu$ (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 $N/\lambda$ and the expected halving time is $(1/\mu)\ln 2$. (The probability $p$ that a node fails in time $\tau$ 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 $\lambda$ and $\mu$ are fixed and the system is in a steady state, then the arrival rate of $\lambda$ must be balanced by the departure rate of $N\mu$ (each of $N$ nodes is leaving at rate $\mu$), 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 to within constant factors.

Using this Poisson model, we derive a lower bound on the rate at which bandwidth must be consumed to maintain connectivity of the P2P network.

**Theorem 2.** *Consider any P2P system with any initial configuration. Suppose that there is some node $n$ that, on average, receives notification about fewer than $k$ new nodes per $\tau$ time.*

*Then there is a sequence of joins and leaves with half-life $\tau$ and a time $t$ so that node $n$ is disconnected from the network by time $t$ with probability $(1 - \frac{1}{e-1})^k$.*

**Corollary 3.** *Consider any $N$-node P2P network that remains connected with high probability for every sequence of joins and leaves with half-life $\tau$.*

*Then every node must be notified with an average of $\Omega(\log N)$ new nodes per $\tau$ time.*

The corollary follows from the theorem by setting $(1 - 1/(e-1))^k = 1/N$ in the theorem. The intuition behind the theorem is as follows. In a half-life, the probability that any particular node in the network fails is $1/2$. Thus, if any node has fewer than $\log N$ neighbors, then the probability that they all fail during this half life is larger than $1/N$. So each node must maintain a set of $\log N$ neighbors. In each half-life, then, each node loses about $(\log N)/2$ neighbors; it must replace its failed neighbors to remain connected in the next half-life.[3]

## 3 A Dynamic Model for Chord

This section outlines and analyzes two maintenance protocols in Chord. The first is *weak stabilization* from [**?**], which maintains a small amount of correct routing information in the face of concurrent arrivals and departures. The second is *strong stabilization*, which ensures a correct routing overlay from an arbitrary initial condition.

### 3.1 Background on Chord

Chord nodes[4] and keys are hashed into a random location on the unit circle; a key is assigned to the first node encountered moving clockwise from it. Each node knows its *successor node*—the node immediately following it on the circle—which allows correct lookup of any key $k$ by walking around the circle until reaching $k$'s successor. We speed this search using *fingers*: $n.\mathit{finger}[i]$ is the first node following $n + 2^i$ on the identifier circle. Intuitively, any node always has a finger pointing halfway to any destination, so that a sequence of $\log N$ "halvings" of the distance take us to the key. Each node $u$ also maintains its *predecessor*, the node closest to $u$ that has $u$ as its successor.

Each node $n$ periodically executes a *weak stabilization* procedure to maintain the desired routing invariants: it contacts its successor $s$, and if $s.\mathit{predecessor} = p$ falls between nodes $n$ and $s$, then node $n$ sets $n.\mathit{successor} := p$. To maintain finger pointers, each node $n$ periodically searches for improved fingers by running $\mathit{find\_successor}(n + 2^{i-1})$ for each finger $i$.

A node departing the Chord ring can cause disconnection of the ring because another node may no longer be able to contact its successor. To prevent this disconnection, each node keeps a *successor list* of the first $\Theta(\log N)$ nodes following

[3] Note that this does not require that each node $u$ learn about $\Omega(\log N)$ nodes in every half-life, since $u$ may receive a message containing information about many new nodes; instead, it requires that $u$ receive information about new nodes at an *average rate* of $\Omega(\log N)$ per half-life.

[4] For load balancing, each "real" Chord node maintains $\log N$ *virtual nodes* with different identifiers; For simplicity, we analyze work per *virtual* node. A system that does not need perfect load balancing can run one virtual node per real node; one that does require load balancing will need to do $\log N$ times as much work.

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

// join the system using information from node n′.
n.join(n′)
  predecessor := nil;
  s := n′.find_successor(n);
  build_fingers(s);
  successor := s;

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

// update finger table via searches by node n′.
n.build_fingers(n′)
  // get first non-trivial finger entry.
  i_0 := ⌊log(successor − n)⌋ + 1;
  for each i ≥ i_0 index into finger[];
    finger[i] := n′.find_successor(n + 2^(i−1));

// 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;

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

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

// update successor list using successor's successor list.
n.fix_successor_list()
   ⟨s_1, ..., s_k⟩ := successor.successor_list;
   successor_list := ⟨successor, s_1, s_2, ..., s_{k−1}⟩;
```

**Fig. 1.** Pseudocode for the Chord P2P system.

it on the ring. A node $n$ maintains its successor list by repeatedly fetching the successor list of $s = n.successor$, removing its last entry, and prepending $s$ to it. If node $s$ fails, then $n$ sets $n.successor$ to the next node on its successor list. Node $n$ also periodically confirms that its predecessor has not failed; if so, it sets $n.predecessor = \mathbf{nil}$.

See Figure 1 for pseudocode.

*A note on our model.* For simplicity, we limit ourselves to a synchronous model of stabilization. We can thus refer to a *round* of stabilization. With mild complications, we can handle (without an increase in running time) a network with a reasonable degree of asynchrony, where machines operate at roughly the same rate, and messages take roughly consistent times to reach their destinations.

In the remainder of this work, we consider independent, random joins and failures. Because Chord identifiers are generated randomly, any correlations among failures or joins in the physical world disappear in the logical Chord world. Thus adversarial real node failures translate to random Chord node failures. This contrasts with the stronger notion of adversarial failures of Saia et al. [**?**]—our model handles an adversary oblivious to the structure of the Chord overlay, while their network is robust against an omniscient adversary.

### 3.2 The Ring-Like State in Chord

The state of a correct Chord ring can be characterized as follows. Each node has exactly one successor, so the graph defined by successor pointers is a *pseudoforest*, a graph in which all components are directed trees pointing towards a root cycle (instead of a root node). We will limit our consideration to connected networks, where the graph is a *pseudotree*. The network is (weakly) stable when all nodes are in the cycle. For each cycle node $u$, there is a tree rooted at $u$ which we call $u$'s *appendage*, denoted $\mathcal{A}_u$. We insist that a node $u$ joining the system invoke $u.join(n)$ for an existing node $n$ that is already on the cycle.

**Definition 4.** *A Chord network with successor lists of length $\Theta(\log N)$ is* ring-like *if, for some $c$,*

1. *Each cycle node's successor is the cycle node with the next-highest identifier. The nodes in each appendage $\mathcal{A}_u$ fall between $u$ and $u$'s cycle predecessor. Every node's path of successor pointers to the cycle has increasing identifiers.*
2. *Every node $u$ that joined the network at least $c \log^2 N$ rounds ago is "good": $u$ is on the cycle and $u$ never lies between $v + 2^i$ and $v.finger[i]$, for any $v$ and $i$.*
3. *At least a third of the nodes are good.*
4. *Any $\log N$ consecutive appendages $\mathcal{A}_u$ contain only $O(\log N)$ nodes in total.*
5. *Nodes that failed at least $c \log^2 N$ rounds ago are not contained in any successor lists, and no more than a quarter of the nodes in any successor list have failed at all. Successor lists are consistent—no $u.successor\_list$ skips over a live node that is contained in $(u.predecessor).successor\_list$—and include all nodes that joined the cycle at least $c \log^2 N$ rounds ago.*

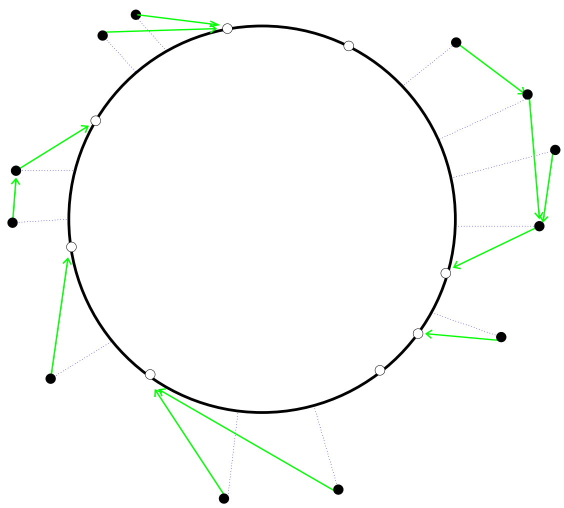

**Fig. 2.** An example of the ring-like state—unfilled nodes are on the cycle, filled nodes are in appendages.

An example is given in Figure 2.

The ring-like state is the "normal" operating condition of a Chord network. Our main result is that a Chord network in the ring-like state remains in the ring-like state, as long as nodes send $\Omega(\log^2 N)$ messages before $N$ new nodes join or $N/2$ nodes fail.

**Theorem 5.** *Start with a network of $N$ nodes in the ring-like state with successor lists of length $\Theta(\log N)$, and allow $N$ random joins and $N/2$ random failures at arbitrary times over at least $c \log^2 N$ rounds. Then, with high probability, we end up in the ring-like state.*

Intuitively, the theorem follows because appendages are not too big, and not too many nodes join them. Thus over $c \log^2 N$ rounds, the appendage nodes have time to join the cycle.

**Theorem 6.** *In the ring-like state, lookups require $O(\log N)$ time.*

This theorem follows from Properties 2 and 3 of Definition 4. For every node $u$ and $i$, the pointer $u.\mathit{finger}[i]$ is accurate with respect to good nodes. Thus our analysis showing logarithmic time search when all fingers are correct can be easily adapted to show that, in logarithmically many steps, a *find_successor*$(k)$ search ends up at the last good node $n$ preceeding key $k$. Since at least a third of the nodes in the network are good, there are, with high probability, only $O(\log N)$ non-good nodes between $n$ and the successor of $k$. Even passing over these one-by-one using successor pointers requires only logarithmically many additional steps.

The correctness of lookups is somewhat subtle in this dynamic setting since, e.g., searches by nodes on the cycle will only return other nodes on the cycle (even if the "correct" answer is on an appendage). However, lookups arrive at a "correct" node, in the following sense: each $\mathit{find\_successor}(k)$ is correct at the instant that it terminates, i.e., 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 Property 2, 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.

### 3.3 Strong Stabilization

The previous section proved, given our model, that Chord's stabilization protocol maintains a state in which routing is done correctly and quickly. But, fearful of bugs in an implementation, or a breakdown in our model,[5] we now wish to take a more cautious view. In this section, we extend the Chord protocol to one that will stabilize 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.

This 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.

A Chord network is *weakly stable* if, for all nodes $u$ in the network, we have $(u.\mathit{successor}).\mathit{predecessor} = u$ and *strongly stable* if, in addition, for each node $u$, there is no node $v$ so that $u < v < u.\mathit{successor}$. A *loopy* network is one which is weakly but not strongly stable; see Figure 3. Previous Chord protocols guaranteed weak stability only; however, such networks can be globally inconsistent—e.g., no node $u$ in Figure 3 has the correct $\mathit{successor}(u)$. The result of this scenario is that $u.\mathit{find\_successor}(q) \neq v.\mathit{find\_successor}(q)$ for some nodes $u$ and $v$ and some query $q$, and thus data available in the network will appear unavailable to some nodes.

The previous Chord stabilization protocol guarantees that all nodes have indegree and outdegree one, so a weakly stable 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 $\mathit{successor}(w) \geq u$. In a loopy network, there is a node $u$ so that $u$'s loop is a strict subset of $u$'s component; here, lookups may not be correct.

[5] 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.

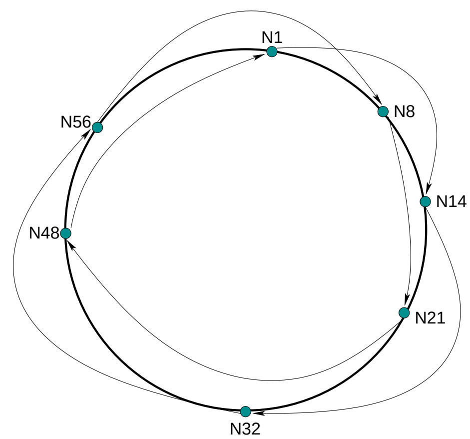


**Fig. 3.** An example of a network that is weakly stable but not strongly stable.

The fundamental stabilization operation by which we unfurl a loopy cycle is based upon *self-search*, wherein a node $u$ searches for itself in the network. 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 $w$ found by following successor pointers so that $\mathit{predecessor}(w) < u < w$. We extend our previous stabilization protocol by allowing each node $u$ to maintain a second successor pointer. This second successor is generated by self-search, and improved in exactly the same way as in the previous protocol. See Figure 4.

**Theorem 7.** *A connected Chord network strongly stabilizes within $O(N^2)$ rounds if no nodes join it, and in $O(N^3)$ rounds if there are no joins and at most $O(N)$ failures occur over $\Omega(\log N)$ rounds.*

**Corollary 8.** *A connected loopy Chord network strongly stabilizes within $O(N^2)$ rounds with no failures, and $O(N^3)$ rounds if there are at most $O(N)$ failures occur over $\Omega(\log N)$ rounds.*

The requirement on the failure rate exists solely to allow us to maintain a successor list with sufficiently many live nodes, and thus maintain connectivity. The intuition for the theorem is that cycles are the only configurations which are not improved by weak stabilization, and self-search turns any loopy cycle into a "non-cycle" by adding a second successor pointer. Therefore, the only configuration not improved by these two operations taken together is a non-loopy (i.e., strongly stable) cycle.

The corollary follows because a loopy Chord network will never permit any new nodes to join until its loops merge—in a loopy network, for all $u$, we have $u.\mathit{on\_cycle} = \mathbf{false}$, since $u$'s self-search never returns $u$ in a loopy network.

```
n.join(n')
  on_cycle := false;
  predecessor := nil;
  s := n'.find_successor(n);
  while (¬ s.on_cycle) do
    s := s.find_successor(n');
  successor[0] := s;
  successor[1] := s;
```

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

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

**Fig. 4.** Pseudocode for strong stabilization.

Thus, no node attempting to join can ever find a node $s$ on the cycle to choose as its successor.

While the runtime of our strong stabilization protocol is large, recall that strong stabilization 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. For example, if pathological states occur only once every $N^4$ rounds, then the system will only be spending a $1/N$ fraction of its time on strong stabilization. Nonetheless, it would clearly be preferable to develop a strong stabilization protocol that, like weak stabilization, simply executes at a low rate in the background, rather than bringing everything else to a halt for lengthy periods.

## 4 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 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.