Self-Stabilizing, Cost-Effective, and Fast-Convergent Structured Overlay Maintenance

Yu Chen
Microsoft Research Asia
ychen@microsoft.com

Wei Chen
Microsoft Research Asia
weic@microsoft.com

Abstract

In this paper we study the self stabilization of structured overlay maintenance in decentralized peer-to-peer (P2P) systems. Our study addresses a number of limitations of existing overlay maintenance protocols, such as the reliance on a continuously available bootstrap system, the assumption of a known system stabilization time, and the need to maintain large local membership lists. In particular, we present a precise specification for self-stabilizing overlay maintenance protocols, with additional requirement on messaging and local state costs. All properties of the specification are desired by applications, while together they prohibit protocols with the limitations existed in previous proposals. We then provide a complete protocol with proof showing that it satisfies the specification. Finally, we show how to improve our self-stabilizing protocol to significantly reduce topology convergence time.

Keywords: structured overlay, self stabilization, peer-to-peer, fault tolerance

MSR-TR-2006-56

1 Introduction

Since their introduction in [18, 20, 23, 24], structured overlays have been used as an important substrate for many peer-to-peer applications such as P2P storage [6, 9, 13, 21] and P2P multicast [4]. In a structured peer-to-peer overlay, each node maintains a partial list of other nodes in the system, and these partial lists together form an overlay topology that satisfies certain structural properties (e.g., a ring). Various system conditions, such as node joins and leaves, message delays and network partitions, affect overlay topology, so overlay topology should adjust itself appropriately to maintain structural properties. Topology maintenance is crucial to the correctness and the performance of applications built on top of the overlay.

Most structured overlays are based on a logical key space, and they can be conceptually divided into two components: leafset tables and finger tables.[1] The leafset table of a node keeps its logical neighbors in the key space, while the finger table keeps relatively faraway nodes in the key space to enable fast routing along the overlay topology. The leafset tables are the key for maintaining a correct overlay topology since finger tables can be constructed efficiently from the correct leafset tables. Therefore, in this paper we focus on leafset maintenance. In particular, we focus on one-dimensional circular key space and the ring-like leafset topology in this space, similar to many studies such as [20] and [23].

Leafset maintenance is a continuously running protocol that needs to deal with various system conditions. One important criterion for leafset maintenance is self stabilization, a criterion frequently used in studying robust and fault-tolerant distributed protocols [7]. Informally, a leafset maintenance protocol is self-stabilizing if it is able to stabilize the leafset tables to the correct configuration after the underlying system stabilizes (but without knowing about system stabilization), no matter how adverse the system conditions were before system stabilization. Besides self stabilization, the protocol should also be cost-effective because it is always running, and should stabilize the leafset fast to reduce transient periods with incorrect topologies.

Existing studies on overlay maintenance have various limitations. Some investigate system level improvements without formal proofs on protocol guarantees [19, 3, 10]; some provide formal proofs to their protocols but do not address fault tolerance and self stabilization [14]; and some propose one-shot protocols for fast overlay construction under known system stabilization conditions without considering adverse effects before system stabilization [1]. Among the studies of self-stabilizing protocols, some rely on a continuously available bootstrap system to actively participate in the self stabilization process [8, 22]; some incur a significant amount of cost by maintaining a large membership list [12, 16]; and some only provides a special case for self stabilization [2].

In this paper, we present a self-stabilizing protocol that removes the limitations in existing protocols. In particular, we first provide a precise specification for self-stabilizing leafset maintenance protocols with cost effectiveness requirements. All properties of the specification are desired by applications, while together they prohibit protocols with the above limitations. We then provide a complete protocol with proof showing that it satisfies the specification. Finally, we show how to improve our self-stabilizing protocol to significantly reduce the topology convergence time. We now explain our contributions in more detail.

Our study is based on a symmetric system model in which any node may join and leave the system or crash, and there is no special group of nodes that is always available to act as a bootstrap system. For studying self stabilization, we assume the system eventually stabilizes. But the protocol does not know the system stabilization time, so it cannot easily nullify the impact of system conditions before system stabilization.

Based on the system model, we provide a set of properties as a rigorous specification of self-stabilizing

[1]The term leafset is originally used in Pastry [20] while the term finger is originally used in Chord [23].

leafset maintenance protocol. One important property is Connectivity Preservation: If the underlying system stabilizes and the topology is still connected, the maintenance protocol should not break the connectivity of the topology while it evolves the topology towards the correct configuration. In addition to self-stabilizing properties, we explicitly put requirements on cost effectiveness: The messaging and local state costs on a node should only be related to the size of its leafset table, which is independent of the size of the system.

To deal with topology partitions caused by network partitions, we define a simple $\texttt{add}(contacts)$ interface, through which an application can add new contact nodes into a leafset to heal topology partitions. Our specification makes it clear that after the underlying system stabilizes, the $\texttt{add}(contacts)$ interface only needs to be invoked once at one node to bridge the partitioned topology. Afterwards, the protocol should continue to preserve connectivity and stabilize the topology by itself without further help. Hence, the reliance on an outside mechanism such as a bootstrap system is kept at the minimum.

Our specification prohibits protocols that either rely on a continuously available bootstrap system, or maintain large local membership lists that is related to the size of the system, or assume that the system stabilization condition is known. Therefore, it only permits protocols that remove these limitations existed in previous protocols.

Next, we present a decentralized self-stabilizing protocol. To be cost-effective, the protocol needs to remove extra entries in leafsets, but such removals may break topology connectivity, especially when there are concurrent removals. We employ several mechanisms to deal with various concurrency issues, some of which handle concurrent removals to nullify potential adverse impacts by system conditions before system stabilization, while others avoid subtle livelock scenarios that would prevent the progress of the stabilization process. We have a complete proof to show that the protocol satisfies our specification, and thus is both self-stabilizing and cost-effective.

Finally, we provide new mechanisms to improve the convergence speed of the topology. These mechanisms utilize finger tables, and deal with topologies that are difficult to converge and are not addressed by previous studies. Through simulations and analysis, we show that our fast convergence mechanisms can shorten the convergence time from $O(N)$ to $O(\log N)$, where N is the number of online nodes in the system.

The rest of the paper is organized as follows. Section 2 discusses the related work. The system model and the protocol specification are described in Sections 3 and 4. Section 5 provides the details of our self-stabilizing protocol and the proof of its correctness. Section 6 describes our improvements to achieve fast convergence. We conclude the paper in Section 7.

2 Related Work

Many existing structured P2P overlay proposals mention that each node should have a leafset table. However, those such as Pastry [20], CAN [18], and SkipNet [11] only provide brief descriptions on what a correct leafset table looks like and how to fix it when the leafset table becomes incorrect due to system churns. These proposals assume that there is a correct leafset table on each node to begin with, then give methods to repair the leafset tables in response to various system events. Bamboo DHT [19] and the latest Pastry improvements [3, 10] adopt practical mechanisms to improve overlay maintenance and routing correctness in a dynamic environment. These mechanisms are system level improvements, while there are no proofs or formal studies on protocol guarantees, such as connectivity preservation and self stabilization.

In [2], Balakrishnan et. al. point out the topology maintenance issues of the original Chord [23] and propose an "idealize" process to adjust the immediate successor of each node to improve topology maintenance. This approach is essentially a self-stabilizing method, but it restricts itself to immediate successor data structure. Therefore, it is only a special case of our protocol, is less robust, and is difficult to accommodate partition healing, which requires to maintain multiple links together to bridge partitioned components.

Some recent overlay maintenance protocols, such as T-Man [12] and TChord [16], address self stabilization. However, they focus on creating an overlay with desired topology without considering global membership changes due to system churns. They require to keep a large membership list on each node, so the cost increases significantly when the system is large or the membership changes over time.

Authors of [8] and [22] also propose self-stabilizing overlay maintenance protocols. But their protocols and proofs depend on the existence of a *continuously available* bootstrap system. In [8], the bootstrap system needs to handle all join and repair requests, and needs to issue periodic broadcast messages for self stabilization purpose; while in [22] each node must periodically initiate lookups to the bootstrap system. These protocols impose significant load and availability requirement on the bootstrap system. In contrast, our protocol only needs an outside mechanism such as a bootstrap system when the topology is partitioned, and it only needs the bootstrap system once after system stabilization. Therefore, the load and availability requirements on the bootstrap system is minimized.

Authors of Ranch [14] provide an overlay maintenance protocol with formal proof of correctness. However, they do not consider fault tolerance: all nodes leaves are "active leave", in which case all nodes invoke a special leave protocol before getting offline. They also do not show the self stabilization feature of their protocol. We believe silent failures must be considered in a wide area peer-to-peer environment, and we explicitly address this in our system model, our specification, and our protocol.

The above topology maintenance studies do not address the fast convergence issue for the special topologies that are difficult to converge, which are considered in this paper. In [1], Angluin et. al. proposed a method for fast construction of an overlay network by a tree merging process. Their protocol is not a self-stabilizing overlay maintenance protocol, because they assume that overlay construction is executed when the underlying system is known to have stabilized and they do not consider adverse impacts of system conditions before system stabilization.

In [5], the authors provide formal specifications for weakly and strongly consistent key-based routing protocols in peer-to-peer systems, and they focus on strong consistency. The self-stabilizing maintenance protocol presented in this paper can be used to support weakly consistent key-based routing.

3 System Model

We consider a distributed peer-to-peer system consisting of nodes (peers) from the set $\Sigma = \{x_1, x_2, \dots, \}$. Each node has a unique numerical ID drawn from a one-dimensional circular key space $\mathcal{K}$. We use x to represent both a node $x \in \Sigma$ and its ID in $\mathcal{K}$. For convenience, we set $\mathcal{K} = [0, 1)$, all real numbers between 0 and 1. We define the following distances in key space $\mathcal{K}$: For all $x, y \in \mathcal{K}$, (a) the *clockwise distance* $d^+(x, y)$ is $y - x$ when $y \geq x$ and $1 + y - x$ when $y < x$; (b) the *counter-clockwise distance* $d^-(x, y) = d^+(y, x)$, and (c) the *circular distance*,$d(x, y) = \min(d^+(x, y), d^-(x, y))$.

Throughout the paper, we use continuous global time to describe system and protocol behavior, but individual nodes do not have access to global time.

Nodes may join and leave the system or crash at any time. We treat node leave and crash as the same type of event, that is, a node disappears from the system without notifying other nodes in the system. We define a *membership pattern* Π as a function from time t to a finite subset of Σ, such that $\Pi(t)$ refers to all of the *online* nodes at time t. Nodes not in $\Pi(t)$ are considered *offline*.

Each node has access to a local clock. The local clocks are not synchronized, but for simplicity we assume that the drifts among local clocks are negligible (our protocol is still correct when we adjust timers to accommodate bounded drifts among local clocks). A node can set a timer to be expired at a later time, and it can cancel a pending timer or reset it with different values.

Every node in the system executes protocols by taking steps triggered by events, which include input events invoked by applications, message receipt events, and timer expiration events. In each step, a node may change its local state, set/cancel/reset some timers, and send out a finite number of messages. We assume that the time to execute a step is negligible, but a node may fail during the execution of a step. For any finite time interval, only a finite number of steps are executed by nodes in the system.

Nodes communicate with one another by sending and receiving messages through channels. The channels cannot create or duplicate messages, but they may delay or drop messages. We say that a message sent from node x to node y is Δ-*timely* if the time elapsed from x sending m to y receiving m is at most Δ.

For the purpose of studying self stabilization, we assume that the system eventually stabilizes. In particular, there is an unknown time t_0 and a known constant Δ such that (a) the set of online nodes $\Pi(t)$ for all $t \geq t_0$ does not change, and (b) all messages sent at or after time t_0 between online nodes are Δ-timely.[2] We denote $sset(\Pi)$ to be the finite set of stable nodes after time t_0, i.e., $sset(\Pi) = \Pi(t_0)$. Note that in our model we do not assume that there is a known group of nodes that are always online or always in the stable set $sset(\Pi)$.

4 Self-Stabilization in Leafset Maintenance

We now define what we mean by self stabilization for a leafset maintenance protocol. Our specification always refers to an arbitrary execution of the protocol with an arbitrary membership pattern Π.

First, we define the function $\mathsf{leafset}(x, set)$ as follows. We have a fixed constant $L \geq 1$, which informally means that the leafset of a node should have L closest nodes on each side of it in the circular space. Given a finite subset $set \subseteq \Sigma$ and a node x, If $|set \setminus \{x\}| < 2L$, then $\mathsf{leafset}(x, set) = set \setminus \{x\}$. Otherwise, sort $set \setminus \{x\}$ as (a) $\{x_{+1}, x_{+2}, \ldots\}$ such that $d^+(x, x_{+1}) < d^+(x, x_{+2}) < \ldots$, and (b) $\{x_{-1}, x_{-2}, \ldots\}$ such that $d^-(x, x_{-1}) < d^-(x, x_{-2}) < \ldots$, then, we have $\mathsf{leafset}(x, set) = \{x_{+1}, x_{+2}, \ldots, x_{+L}\} \cup \{x_{-1}, x_{-2}, \ldots, x_{-L}\}$.

In the leafset maintenance protocol, each node x maintains a variable $neighbors$, the value of which is a finite subset of Σ. Informally, $x.neighbors$ should eventually stabilize on the correct leafset of x, meaning $x.neighbors = \mathsf{leafset}(x, sset(\Pi))$, in which case the final topology resembles a ring structure.

Each node also has an interface function $\mathsf{add}(contacts)$, where $contacts$ are a finite subset of Σ. This function is used to bridge partitioned components. In particular, it can be used in the following situations: (a) adding initial contacts when the system is initially bootstrapped; (b) introducing contact nodes when a new node joins the system; and (c) introducing nodes in other partitioned components after the overlay is partitioned (perhaps due to transient network partitions).

To formalize our requirements, we first need to address the connectivity of the leafset topology. Let $x.neighbors_t$ be the value of $neighbors$ on x at time t after node x takes a step at time t (if there is a step). The *leafset topology* at time t is a directed graph $G_t = \langle \Pi(t), E(t) \rangle$, where $E(t) = \{\langle x, y \rangle | x, y \in \Pi(t) \ \wedge \ y \in x.neighbors_t\}$. We say that G_t is *strongly connected* if there is a directed path between any pair of nodes in $\Pi(t)$, and G_t is *weakly connected* (or simply *connected*) if there is an undirected path (when treating edges in $G(t)$ as undirected) between any pair of nodes in $\Pi(t)$. G_t is *disconnected* if it is not weakly connected.

Leafset stabilization requires the protocol to keep topology connectivity eventually, but many adverse conditions before system stabilization time t_0 may have lingering effects to connectivity even after t_0. Thus we need to define a time after which it is reasonable to require the protocol to nullify the adverse impacts of system conditions before t_0. We first define time $t'_0 \geq t_0$ such that all messages sent before t_0 are either

[2]If the constant Δ is unknown, our protocol can gradually increase the timeout value such that it still eventually stabilizes the system topology.

lost or received before time t'_0, and all timers started before t_0 have expired or been canceled/reset before t'_0. Such a time t'_0 must exist because there are only a finite number of messages sent and a finite number of timers started before t_0. After t'_0, system conditions before t_0 will not directly affect the protocol behavior. We add a Δ beyond t'_0 so that indirect effects carried by messages sent before t'_0 are already shown before time $t'_0 + \Delta$, and we require that after $t'_0 + \Delta$ the protocol should have the ability to nullify the indirect effects of system conditions before t_0. This is specified as the following property.

- *Connectivity Preservation*: If for some time $t \geq t'_0 + \Delta$, G_t is weakly connected, then for all time $t' > t$, $G_{t'}$ is weakly connected.

Connectivity Preservation is a key property of our protocol, but it is not explicitly addressed or enforced by previous protocols in a purely peer-to-peer environment. With the connectivity issue addressed, we now define the requirement for self stabilization. We say that a leafset maintenance protocol is *self-stabilizing* if it satisfies the following properties.

- *Leafset Stabilization*: If for some time $t \geq t'_0 + \Delta$, G_t is weakly connected, then there exist a time $t' > t$, such that for all $t'' \geq t'$ and all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors_{t''}) = \mathsf{leafset}(x, sset(\Pi))$.
- *Partition Healing*: Suppose that there is a time $t \geq t'_0 + \Delta$ such that G_t is partitioned into k disconnected components, and we have k nodes in G_t, $x_1, x_2, \ldots, x_k$, such that they are in k different components. If there is an invocation of $\mathsf{add}(\{x_2, x_3, \ldots, x_k\})$ on x_1 at time $t' > t$, then there is a time $t'' > t'$ such that $G_{t''}$ is weakly connected.
- *Leafset Cleanup*: If there is a time t after which no $\mathsf{add}()$ is invoked at any node in the system, then there is a time t' such that for all time $t'' \geq t'$ and all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors_{t''}) = x.neighbors_{t''}$.

The Leafset Stabilization property requires that after system stabilization the leafset table on every node eventually contains the correct leafset entries, as long as the topology is connected at some time after $t'_0 + \Delta$. Note that the Leafset Stabilization property should holds no matter if there are invocations of $\mathsf{add}()$ after the topology is weakly connected. Leafset Stabilization includes the aspect of Connectivity Preservation, so we do not include Connectivity Preservation in our specification. The Partition Healing property requires that, if the topology is partitioned, the application only needs to invoke the $\mathsf{add}()$ interface once to heal the partition. Then Leafset Stabilization guarantees to further stabilize the topology without any more help. The Leafset Cleanup property requires that eventually the leafset maintenance protocol should only maintain the actual leafset entries.

Besides self stabilization, the leafset maintenance protocol should also be cost-effective in terms of both messaging costs and local state costs in the stable period. The messaging cost is defined in terms of a *detection-repair* cycle, as defined below. In the steady state when the system and the leafset topology stabilize, a *detection-repair cycle* is the length of the worst-case period from the time when a new crash occurs to the time when the system detects the failure and repairs the leafset tables back to the steady state. Measuring messaging costs based on one detection-repair cycle is appropriate for assessing and comparing messaging costs among the leafset maintenance protocols, because a protocol may choose to artificially slow down or speed up the pace of sending messages, but it correspondingly increases or decreases the detection time and repair time. The requirement on local state and messaging cost is reflected by the following property.

- *Cost Effectiveness*: If there is a time t after which no $\mathsf{add}()$ is invoked at any node in the system, then in the steady state when the system and the leafset topology stabilize, the size of the local state on each node is $O(Poly(L))$, and the total size of all messages sent by each node in one detection-repair cycle is $O(Poly(L))$, where $Poly(L)$ is a polynomial of L.

On node x:

```
Data structure:
    $neighbors$: set of nodes intended for leafset entries, initially $\emptyset$.
add($contacts$)
    **foreach** $y \in contacts$
        send PING-CONTACT to $y$
Upon receipt of PING-CONTACT from $y$:
    send PONG-CONTACT to $y$
Upon receipt of PONG-CONTACT from $y$:
    $neighbors \leftarrow neighbors \cup \{y\}$
Repeat periodically with interval $I_p$:
    **foreach** $y \in neighbors$, send PING-ALIVE to $y$
Upon receipt of PING-ALIVE from a node $y$:
    send PONG-ALIVE to $y$
Repeat periodically with interval $I_c$:
    **foreach** $y \in neighbors$
        **if** not received PONG-$\{$CONTACT, ALIVE, INVITE, REPLACE$\}$ from $y$ in the past $T_c$ time units
            **then** $neighbors \leftarrow neighbors \setminus \{y\}$
```

Figure 1: Self-stabilizing leafset maintenance protocol, Part I: Add new contacts and check liveness.

The Cost Effectiveness property requires that eventually the communication and local state costs on each node is only related to the size of the leafset table, not to the total number of online nodes in the system. The consideration of both self stabilization and cost effectiveness is important to applications, but it makes the protocol design more challenging. We show in the next section how to satisfy both requirements with our leafset maintenance protocol.

5 Self-Stabilizing Leafset Maintenance Protocol

5.1 Protocol description

Our leafset maintenance protocol consists of five sub-protocols: (a) the add() protocol to add new contacts supplied by the application (Fig. 1, lines 3–9); (b) the liveness-checking protocol to check the liveness of nodes in the leafset (Fig. 1, lines 10–17); (c) the invite protocol to invite closer nodes into leafset (Fig. 2); (d) the replacement protocol to replace faraway nodes that should not be in the leafset with closer nodes (Fig. 3); [3] and (e) the deloopy protocol to detect and resolve a special incorrect topology called loopy topology (Fig. 5). The replacement protocol (Fig. 3) is our key contribution, so we focus our attention on this sub-protocol while briefly explaining other sub-protocols. Even though each sub-protocol has its own functionality, they have to work together to provide the desired self-stabilizing and cost-effective features specified in the previous section.

All these sub-protocols use a periodic ping-pong messaging structure. For ease of understanding, each type of ping-pong message is sent independently. In actual implementations, one can unify all periodic ping-pong messages together for efficiency.

On each node, the protocol maintains a *neighbors* set as required by the specification. The protocol

[3] Technically, the faraway nodes for a node x are those in $x.neighbors \setminus \mathsf{leafset}(x, x.neighbors)$. Whenever necessary, we use $x.var$ to denote the variable var on x.

```
On node $x$:
18 Data structure:
19   $cand$: candidate nodes for $neighbors$, initially $\emptyset$.
20 Repeat periodically:
21   foreach $y \in neighbors$, send PING-ASK-INV to $y$
22 Upon receipt of PING-ASK-INV from a node $y$:
23   $view \leftarrow \mathsf{leafset}(y, neighbors)$
24   send (PONG-ASK-INV, $view$) to $y$
25   $cand \leftarrow cand \cup \{y\}$
26 Upon receipt of (PONG-ASK-INV, $view$) from $y$
27   $cand \leftarrow cand \cup view$
28 Repeat periodically                                   /* invite closer nodes */
29   foreach $y \in cand \setminus neighbors$
30     if $y \in \mathsf{leafset}(x, cand \cup neighbors)$ then
31       send PING-INVITE to $y$
32   $cand \leftarrow \emptyset$
33 Upon receipt of PING-INVITE from $y$:
34   send PONG-INVITE to $y$
35 Upon receipt of PONG-INVITE from $y$:
36   if $y \in \mathsf{leafset}(x, neighbors \cup \{y\}) \setminus neighbors$
37     then $neighbors \leftarrow neighbors \cup \{y\}$
```

Figure 2: Self-stabilizing leafset maintenance protocol, Part II: Invite closer nodes in the key space.

keeps an invariant that a node y is added into $x.neighbors$ only after x receives a pong message directly from y. The $\mathsf{add}(contacts)$ protocol (Fig. 1, lines 3–9) uses a ping-pong message loop to add nodes in $contacts$ to $x.neighbors$. The liveness-checking protocol (Fig. 1, lines 10–17) uses periodic ping-pong messages to detect node departures and remove a node from $neighbors$ if not receiving any pong messages from the node for a time period T_c.

The invite protocol (Fig. 2) uses a variable $cand$ to store candidate nodes to be invited into the $neighbors$ set. The candidate nodes are discovered by exchanging local leafset views through the PING-ASK-INV and PONG-ASK-INV messages. Once node x discovers some new candidates, it uses the periodic PING-INVITE and PONG-INVITE message loop to invite these candidates into $x.neighbors$. The invite is successful when the candidate y sends back the PONG-INVITE message to x and x verifies that y is indeed qualified to be in x's leafset (lines 36–37). The invite protocols is in principle similar to other leafset maintenance protocols (e.g. [23, 19, 12, 22]).

The replacement protocol (Fig. 3) is responsible of removing faraway nodes from the $neighbors$ sets to keep $neighbors$ sets small. This protocol is our key contribution to provide cost-effective and self-stabilizing leafset maintenance and the key differentiator with other protocols. When removing the faraway nodes, we need to ensure both safety (Connectivity Preservation) and liveness (Leafset Cleanup and Leafset Stabilization), in the presence of concurrent replacements and other system events.

To ensure safety, we use a closer node to replace a faraway node instead of removing it directly. The basic replacement flow consists of two ping-pong loops. Suppose a node x intends to remove a node z since z is not in $\mathsf{leafset}(x, x.neighbors)$. Node x uses the PING-ASK-REPL and PONG-ASK-REPL loop (lines 42–49) with node z to obtain a replacement node y, which is recorded by x in $x.z.repl$. Then x uses

```
On node $x$:
38 Data structure:
39   $repl$: for each $z \in neighbors$, $z.repl$ is a node to replace $z$, initially $\bot$.
40   $round$: current round number of the replacement round, initially 0
41   $commit$: for each $z \in neighbors$, $z.commit$ is the minimum replacement round number that $x$ can commit
         to replace $z$, initially 0
42 Repeat periodically:
43   **foreach** $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$
44     send PING-ASK-REPL to $z$
45 Upon receipt of PING-ASK-REPL from $z$:
46   $y \leftarrow v$ such that $v \in \mathsf{leafset}(x, neighbors)$ **and** $d(z,v) < d(z,x)$ **and**
       $d(z,v) = \min_{u \in \mathsf{leafset}(x,neighbors)} d(z,u)$
47   send $($PING-ASK-REPL$, y)$ to $z$
48 Upon receipt of $($PONG-ASK-REPL$, y)$ from $z$
49   **if** $z \in neighbors$ **then** $z.repl \leftarrow y$
50 Repeat periodically:
51   $round \leftarrow round + 1$
52   **foreach** $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$ **and** $z.repl \neq \bot$
53     send $($PING-REPLACE$, z, round)$ to $z.repl$
54 Upon receipt of $($PING-REPLACE$, z, rnd)$ from $y$:
55   if $z \in neighbors$ **then**
56     $z.commit \leftarrow round + 1$
57     send $($PONG-REPLACE$, z, rnd)$ to $y$
58 Upon receipt of $($PONG-REPLACE$, z, rnd)$ from $y$:
59   **if** $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$ **and** $y = z.repl$ **then**
60     $neighbors \leftarrow neighbors \cup \{y\}$
61     **if** $z.commit \leq rnd$ **then**
62       $neighbors \leftarrow neighbors \setminus \{z\}$
63       $y.commit \leftarrow round + 1$
```

Figure 3: Self-stabilizing leafset maintenance protocol, Part III: Replace faraway nodes.

the PING-REPLACE and PONG-REPLACE message loop to verify with y about the replacement (lines 50–63). If y finds z in $y.neighbors$ at the time it receives the PING-REPLACE message from x, it acknowledges x with a PONG-REPLACE message. Only after receiving the PONG-REPLACE message from y, x may replace z with y in $x.neighbors$. This method tries to ensure that after the removal of edge $\langle x, z\rangle$ from the overlay, there is still a path from x to z via y.

The above method, however, cannot nullify the indirect effects of system conditions before time t_0 when there are concurrent replacements, and thus the topology connectivity could still be jeopardized. For example, in Fig. 4, x replaces z with y after time $t'_0 + \Delta$ when it receives the PONG-REPLACE message sent by y after time t'_0. In the meantime, there is a concurrent task in which y wants to replace z with u. After sending the PONG-REPLACE message to x, y receives the PONG-REPLACE message from u so y successfully replaces z with u. By definition of t'_0, u must send the PONG-REPLACE message to y after t_0, but could be before t'_0. However, because several recent PONG-ALIVE messages sent by z to u before time t_0 are lost, the liveness checking timer on u triggered right after sending the PONG-REPLACE message causes u to incorrectly remove z from $u.neighbors$. Thus even though the replacement of z on x happens

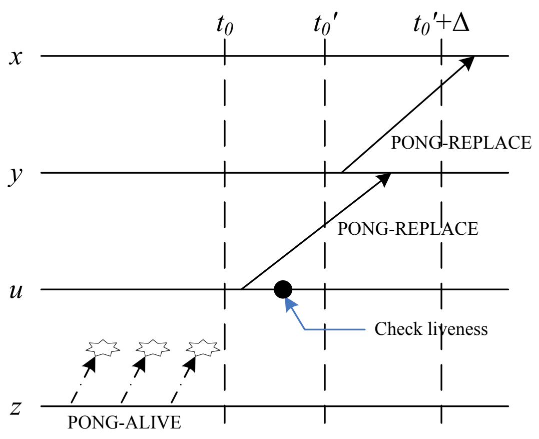


Figure 4: Concurrent replacement tasks introduce indirect effects of system conditions before t_0 and break topology connectivity.

after $t_0' + \Delta$, the path from x to z is unavailable after the replacement, due to an indirect effect of system conditions before t_0. A similar danger exists when x tries to replace z and y concurrently.

We introduce variables $round$ and $commit$ to eliminate these dangerous concurrent replacements. Variable $round$ is a counter incremented every time a node sends out PING-REPLACE messages (line 51). The value of $round$ is piggybacked with the PING-REPLACE and PONG-REPLACE messages to indicate the replacement round of the current replacement task. For each $z \in x.neighbors$, variable $z.commit$ records the lowest replacement round that x can commit to replace z (enforced in line 61). When y verifies the replacement of z for x, y sets $z.commit$ on it to be $round + 1$ (line 56), which disables any concurrent replacement tasks of z on y. Similarly, when x replaces z with y, it also sets $y.commit$ to $round + 1$ (line 63) to disable any concurrent replacement tasks of y. As shown by our proof, the use of $round$ and $commit$ variables is the core mechanism to satisfy the Connectivity Preservation property.

Next, we restrict the selection of replacement node y in order to guarantee the Leafset Cleanup property. A node y can be a replacement of z for x only when y is closer to x than z and is in z's leafset (line 46). The distance constraint avoids circular replacement, while the leafset constraint guarantees that y can successfully verifies the replacement. The later is true because our invite protocol guarantees that eventually the leafsets are mutual, so z will be in y's leafset. These two replacement selection constraints guarantee the progress of the replacement tasks, and thus the Leafset Cleanup property.

The mechanisms introduced so far are not enough to guarantee the Leafset Stabilization property, however. During the proof of an earlier version of the protocol, we uncover the following subtle livelock scenario in which the add() invocations interfere with leafset stabilization. Whenever node x wants to replace z with y, the replacement is rejected because x just replaced another node u with z and therefore only committed to replace z in a higher round. The rejections can keep happening if an application keeps invoking add($\{u\}$) on x at inopportune times such that the edge from x to u is kept being added back to the topology. The inability for x to replace z with y is not an issue by itself. However, it is possible that there is a node v that should be in x's leafset, and the only way x learns about v is through z by the replacement protocol (the invite protocol will not help if all nodes in $z.neighbors$ are outside x's leafset range). In this case, x cannot replace z with y and thus will not learn about v, so the leafset stabilization will not occur.

To fix this problem, we break the replacement of z with y on node x into two phases. First, x can add node y into $x.neighbors$ (line 60), without checking the constraint of $z.commit \leq rnd$. Next, x can remove

```
On node $x$:
64 Data structure:
65     $succ$: a derived variable, $succ = x$ if $neighbors = \emptyset$
           else $succ = y \in neighbors$ such that $d^+(x, y) = \min\{d^+(x, z) : z \in neighbors\}$
66 Repeat periodically:
67     if $neighbors \neq \emptyset$ and $d^+(x, 0) < d^+(x, succ)$
68         send $(\text{PING-DELOOPY}, x)$ to $succ$
69 Upon receipt of $(\text{PING-DELOOPY}, u)$ from $y$:
70     if $x = u$ then return
71     if $neighbors = \emptyset$ or $d^+(x, 0) < d^+(x, succ)$ then
72         $cand \leftarrow cand \cup \{u\}$
73         send PONG-DELOOPY to $u$
74     else
75         send $(\text{PING-DELOOPY}, u)$ to $succ$
76 Upon receipt of PONG-DELOOPY from $y$:
77     $cand \leftarrow cand \cup \{y\}$
```

Figure 5: Self-stabilizing leafset maintenance protocol, Part IV: Loopy detection.

z only when the condition $z.commit \leq rnd$ holds (lines 61–62). With this change, x can still find closer nodes through z even if x cannot replace z.

We also find another similar livelock scenario if the replacement node is selected from z's $neighbors$ set rather than its leafset (leafset$(z, z.neighbors)$) in line 46. The discovery of these subtle and even counter-intuitive livelock scenarios shows that a rigorous and complete proof helps us in discovering subtle concurrency issues that are otherwise difficult to discern.

With the sub-protocols explained so far, the topology may still not be correct, because it can be in a special state called *loopy state* as defined in [2]. A node's *successor* is the closest node in its $neighbors$ set according to the clockwise distance. A topology is in the loopy state if following the successor links one may traverse the entire key space more than once before coming back to the starting point. We use a deloopy protocol (Fig. 5) similar to the one in [2] to detect the loopy state and resolve it. The protocol essentially initiates a PING-DELOOPY message along the successor links to see if the message makes a complete traversal of the logical space before coming back to the initiator. If so, a loopy state is found, and the protocol puts the two end nodes of this traversal into each other's $cand$ sets, so that the invite protocol is triggered to resolve the loopy state.

5.2 Proof of correctness for the self-stabilizing protocol

Our proof always refers to a particular execution of the algorithm with a membership pattern Π.

For two different points x and y in the key space $\mathcal{K}$, we denote the interval (x, y) as the interval from x to y in the clockwise direction, excluding point x and y. That is, $(x, y) = \{z \in \mathcal{K} : 0 < d^+(x, z) < d^+(x, y)\}$.

Let I_p and I_c be the intervals for a node to periodically send PING-ALIVE messages and check liveness of nodes, and let T_c be the liveness timeout value, as shown in the protocol (Fig. 1).

Lemma 1 *If $I_c, T_c \geq I_p + 2\Delta$, then at or after time t'_0, liveness checking protocol (lines 14–17) will not remove any online nodes in $sset(\Pi)$ from the $neighbors$ set of any node.*

Proof. Suppose, for a contradiction, that there exist nodes $x, y \in sset(\Pi)$ and a time $t \geq t'_0$ such that

x removes y from $x.neighbors$ at time t in the liveness checking protocol (lines 14–17). Without loss of generality, we assume t is the first such time. From the algorithm (line 16), we know that x does not receive any PONG-{CONTACT, ALIVE, INVITE, REPLACE} messages from y in the time period $[t - T_c, t]$. Since $T_c \geq I_p + 2\Delta$, the above is true for the period $[t - (I_p + 2\Delta), t]$. Note that the periodic timer for checking liveness is scheduled at time $t - I_c$ and expires at time $t \geq t'_0$. By the definition of t'_0, $t - I_c \geq t_0$. Since $I_c > I_p + 2\Delta$, we have $t - (I_p + 2\Delta) \geq t_0$.

We claim that y is in $x.neighbors$ in the period $[t - (I_p + 2\Delta), t]$. To see that this is true, first, x removes y from $x.neighbors$ at time t when executing line 17, so y was in $x.neighbors$ right before time t. If y was not in $x.neighbors$ in the entire period in $[t - (I_p + 2\Delta), t]$, then y must be added back to $x.neighbors$ at some time in this period. However, according to the algorithm, a node y can be added to $x.neighbors$ only upon the receipt of a message PONG-CONTACT (line 9), or PONG-INVITE (line 37), or a message PONG-REPLACE (line 60). Since we know that x does not receive any of these messages from y in the time period $[t - (I_p + 2\Delta), t]$, y must be in $x.neighbors$ through the entire period $[t - (I_p + 2\Delta), t]$.

Given the period $[t - (I_p + 2\Delta), t - 2\Delta]$ with length I_p, x must send one PING-ALIVE message to y since $y \in x.neighbors$ in this time period. Since $t - (I_p + 2\Delta) \geq t_0$, this PING-ALIVE message is received by y within Δ time units. Upon the receipt of this PING-ALIVE message, y sends to x a PONG-ALIVE message, which should be received by x within Δ time units. Therefore, x should have received a PONG-ALIVE message from y in the time period $[t - (I_p + 2\Delta), t]$, which is a contradiction. □

In the following descriptions, we assume that $I_c, T_c \geq I_p + 2\Delta$.

Since the liveness checking protocol (Figure 1) will not remove any online nodes from the $neighbors$ set after time t'_0, the removal of such nodes must be caused by the replacement protocol (Figure 3). With the replacement protocol, if a node u wants to remove a node w from its $neighbors$ set, it must find another node v as the replacement, in order to preserve the connectivity from u to w. Lines 42–49 in Figure 3 shows how u could find v. Then u sends a PING-REPLACE to v to ask for v's approval for the replacement. If v agrees with the replacement, v will send a PONG-REPLACE back to u, and u can replace w with v upon the receipt of the approval.

We call the step that successfully executes lines 54–57 *v's verification of the replacement of w for node u*, and the step that successfully executes lines 58–63 *u's replacement of w with v*. These two steps are the key in the process for connectivity preservation. We define a *replacement sequence* as $R = (u, v, w, t_{vf}, t_{rp})$, in which u replaces w with v, with t_{vf} and t_{rp} as the time points at which v's verification and u's replacement of w occur, respectively.

Lemma 2 *Suppose there are three replacement sequences $R_1 = (u, v, w, t^1_{vf}, t^1_{rp})$, $R_2 = (u, r, v, t^2_{vf}, t^2_{rp})$, and $R_3 = (v, s, w, t^3_{vf}, t^3_{rp})$ in the execution. Then we have*

1). $t^2_{rp} > t^1_{rp}$ implies $t^2_{vf} > t^1_{rp}$.

2). $t^3_{rp} > t^1_{vf}$ implies $t^3_{vf} > t^1_{vf}$.

Proof. Proof of 1). At time t^1_{rp}, v is merged into $u.neighbors$ (line 60) and $v.commit$ on u is set to $u.round + 1$ (line 63). At time t^2_{rp}, u replaces v with r (lines 60 and 62). According to the algorithm, this implies that $v.commit \leq rnd$ at time t^2_{rp}, where rnd is the round number variable embedded in the PONG-REPLACE message from r (line 61). Thus the corresponding PING-REPLACE message is sent at some time t with $u.round \geq v.commit$. This implies that $u.round$ is increased comparing to its value at time t^1_{rp}. Since $u.round$ only monotonically increases, we have $t \geq t^1_{rp}$. Because t is the time that u sends the PING-REPLACE to r, and t^2_{vf} is the time that r receives that message, it must be true that $t^2_{vf} > t$. Therefore, $t^2_{vf} > t \geq t^1_{rp}$.

Proof of 2). At time t^1_{vf}, v verifies the replacement of w so $w.commit$ on v is set to $v.round + 1$ (line 56). At a later time t^3_{rp}, v replaces w with s, which implies that the round number rnd embedded in the PONG-REPLACE message from s to v satisfies $w.commit \leq rnd$ (line 61). This means that the corresponding PING-REPLACE message with rnd is sent at a time t at or after time t^1_{vf} such that rnd is greater than the value of $v.round$ at time time t^1_{vf}. Since at time t^3_{vf} this PING-REPLACE message is received at s, we have $t^3_{vf} > t \geq t^1_{vf}$. □

Lemma 3 *For a replacement sequence $R = (u, v, w, t_{vf}, t_{rp})$ with $w \in sset(\Pi)$, if $t_{vf} \geq t'_0$, then at any time $t \geq t_{vf}$, there is a path from v to w in the directed graph G_t.*

Proof. To prove the lemma, we prove the following statement by induction: For any replacement sequence $R = (u, v, w, t_{vf}, t_{rp})$ with $w \in sset(\Pi)$ and $t_{vf} \geq t'_0$, for any finite step sequence σ started at the step of v's verification of the replacement at time t_{vf}, there is a path $p = (v_0 = v, v_1, \ldots, v_m = w)$ from v to w in the directed graph G, where G is derived from the $neighbors$ sets of all online nodes after the step sequence σ, and p and σ have the following property:

(*) For any edge $\langle v_i, v_{i+1}\rangle$ on the path, either v_i verifies the replacement of v_{i+1} for some node x (lines 54–57) in the step sequence σ, or v_i replaces some node x with v_{i+1} (lines 58–63) in the step sequence σ.

By our system model, there are only a finite number of steps that can occur in any finite time interval, so the above statement will cover all time after t_{vf}. We prove the above statement by an induction on the number of steps k in the step sequence σ.

In the base case where $k = 1$, σ has one step, which is v's verification of the replacement of w at time t_{vf}. According to the algorithm, right after this step we have $w \in v.neighbors$. Since $t_{vf} \geq t'_0$, we have $v \in sset(\Pi)$. Since we also have $w \in sset(\Pi)$, we know that edge $\langle v, w\rangle$ is in G where G is the directed graph after v's verification step. So the path we need is $p = (v, w)$. The (*) property holds for p and σ, since v verifies the replacement of w for node u in σ (the only step in σ).

We now suppose that the statement is true for less than k steps and we need to show it is also true for k steps where $k > 1$. Let σ' be the step sequence with k steps and σ be the prefix of σ with $k-1$ steps. Let s_k be the k-th step in σ'. Let G be the graph after the sequence σ and G' be the graph after sequence σ'. By the induction hypothesis, there is a path $p = (v_0 = v, v_1, \ldots, v_m = w)$ from v to w in G. By definition, all nodes on the path are in $sset(\Pi)$. If step s_k does not affect path p, then we are done. If step s_k does affect path p, it could be one of the following two types: (a) s_k is the removal of v_{i+1} from $v_i.neighbors$ when v_i executes the liveness checking protocol (lines 14–17) for some $i < m$; or (b) s_k is the replacement of v_{i+1} with some node z in $v_i.neighbors$ (lines 58–63), for some $i < m$. By Lemma 1, case (a) cannot occur because all v_i's are in $sset(\Pi)$, and all steps in σ' occur at or after t'_0. Therefore we only need to consider case (b).

Let this replacement sequence be $R_1 = (v_i, z, v_{i+1}, t_1, t'_1)$. By the algorithm, step s_k occurs when v_i receives a $(\text{PONG-REPLACE}, v_{i+1}, r)$ message from z, where r is a replacement round number. According to the algorithm, we know that before step s_k, $v_i.v_{i+1}.commit \leq r$ (line 61), and after step s_k, $v_i.z.commit = v_i.round + 1$ (line 63). Since step s_k is the replacement of v_{i+1} with z in $v_i.neighbors$, there is a corresponding step before s_k at which z verifies the replacement of v_{i+1} for v_i. Let this step be s.

We claim that step s must be in σ' but it is not the first step in σ'. To show the claim, we use the induction hypothesis. By the (*) property of the induction hypothesis, in σ either v_i verifies the replacement of v_{i+1} for some node x or v_i replaces some node x with v_{i+1}. In the first case, v_i verifies the replacement of v_{i+1}

for some node x in σ. Let this step be s'. Because s_k is after s', we can apply Lemma 2 2) and conclude that step s is after s', so s is in σ' but not the first one in σ'. In the second case, v_i replaces some node x with v_{i+1} in σ. Let this step be s'. Because s_k is after s', we can apply Lemma 2 1) and conclude that step s is after s', so s is in σ' but not the first one in σ'.

Now let σ_1 be the suffix of σ' started with the step s. The length of σ_1 is less than k. By the definition of the first step s in σ_1, z receives the $(\text{PING-REPLACE}, v_{i+1}, r)$ message from v_i at step s. This implies that during the entire execution of σ_1, we have $v_i.round \geq r$.

For the replacement sequence $R_1 = (v_i, z, v_{i+1}, t_1, t'_1)$, we can apply induction hypothesis on σ_1 and know that there is a path p_1 from z to v_{i+1} in G', and the (*) property holds for p_1 and σ_1.

We now claim that $\langle v_i, v_{i+1}\rangle$ is not on path p_1. To show this claim, suppose, for a contradiction, that $\langle v_i, v_{i+1}\rangle$ is on the path p_1. By the (*) property, there are two cases. In the first case, v_i verifies the replacement of v_{i+1} for some node x in σ_1. Suppose this step is s'', which must be after s and before s_k. By the algorithm, after s'' we have $v_i.v_{i+1}.commit = v_i.round + 1$. Since $v_i.round \geq r$ during σ_1, we have $v_i.v_{i+1}.commit \geq r + 1$, which contradicts to our earlier conclusion that $v_i.v_{i+1}.commit \leq r$ before s_k. In the second case, v_i replaces some node x with v_{i+1} in σ_1. By the algorithm, after this step $v_i.v_{i+1}.commit = v_i.round + 1$. Thus $v_i.v_{i+1}.commit \geq r + 1$ during σ_1, again contradicting with our conclusion that $v_i.v_{i+1}.commit \leq r$ before s_k.

With the claim that $\langle v_i, v_{i+1}\rangle$ is not on path p_1, we can see that $\langle v_i, v_{i+1}\rangle$ is removed from G', but instead we have $\langle v_i, z\rangle$, and a path p_1 from z to v_{i+1} in G'. Thus, there is still a path from v_i to v_{i+1}, and we can use this path to replace $\langle v_i, v_{i+1}\rangle$ in path p, such that in G' we still have a path p' from v to w.

Finally, we need to show that path p' and step sequence σ' satisfy the (*) property. We only need to show edge $\langle v_i, z\rangle$ for the property, since all other edges are either from path p or path p_1, and by the induction hypothesis, they satisfy the (*) property. For edge $\langle v_i, z\rangle$, we know that v_i replaces v_{i+1} with z in step s_k, the last step of σ'. So the (*) property holds for edge $\langle v_i, z\rangle$. We now finish the induction step. □

Corollary 4 *If a node u replaces node $w \in sset(\Pi)$ with node v at time $t \geq t'_0 + \Delta$, then there is still a path from u to w in G_t after the replacement.*

Proof. If u replaces w with v at time $t \geq t'_0 + \Delta$, then v verifies this replacement at or after time t'_0. The corollary then follows directly from Lemma 3. □

Lemma 5 (Connectivity Preservation) *If for some time $t \geq t'_0 + \Delta$, G_t is weakly connected, then for all time $t' > t$, $G_{t'}$ is weakly connected.*

Proof. After time $t'_0 + \Delta$, some edge in $G_{t'_0+\Delta}$ may be removed in G_t only by some replacement steps in the replacement protocol (part III) or node removal steps in the liveness checking protocol (part I). Lemma 1 shows that no edges in $G_{t'}$ for any $t' \geq t'_0 + \Delta$ can be removed by the liveness checking protocol. Corollary 4 shows that after time $t'_0 + \Delta$, any replacement that removes an edge $\langle u, w\rangle$ with $w \in sset(\Pi)$ still keeps a path from u to w. Therefore, all replacements keep connectivity, and thus $G_{t'}$ for all time $t' > t \geq t'_0 + \Delta$ is still weakly connected if G_t is weakly connected. □.

Lemma 6 (Partition Healing) *Suppose that there is a time $t \geq t'_0 + \Delta$ such that G_t is partitioned into k disconnected components, and we have k nodes in G_t, $x_1, x_2, \ldots, x_k$, such that they are in k different components. If there is an invocation of* $\mathsf{add}(\{x_2, x_3, \ldots, x_k\})$ *on x_1 at time $t' > t$, then there is a time $t'' > t'$ such that $G_{t''}$ is weakly connected.*

Proof. Let the k disconnected components of G_t be $P_1, P_2, \ldots, P_k$. By Corollary 4 and a similar argument as in the proof of Lemma 5, we know that P_i will keep to be connected after time t for all $i = 1, 2, \ldots, k$. During the invocation of $\mathsf{add}(\{x_2, x_3, \ldots, x_k\})$ on x_1 at time $t' > t$, x_1 sends a PING-CONTACT message to nodes $x_2, x_3, \ldots, x_k$. By the definition of G_t, these nodes are online, so they receives the PING-CONTACT message from x_1 and reply with PONG-CONTACT messages to x_1. When x_1 receives these messages, it add x_i into $x_1.neighbors$ (line 9), and thus component P_1 and P_i become connected. Therefore, when x_1 receives all the PONG-CONTACT messages at time $t'' > t'$, graph $G_{t''}$ is weakly connected. □

Lemma 7 *Let* $t_1 = t'_0 + T_c + I_c$. *We have* $\forall t > t_1, \forall x \in sset(\Pi), x.neighbors_t \subseteq sset(\Pi)$.

Proof. After t'_0, all the messages sent from nodes that are not in $sset(\Pi)$ have been delivered. For any node x, y is added into $x.neighbors$ only if x receives any of the PONG-CONTACT, PONG-ALIVE, PONG-INVITE, or PONG-REPLACE messages from y directly. So our algorithm will not add any offline nodes into the $neighbors$ set of any online nodes after t'_0.

Let $y \in x.neighbors_{t'_0}$ for arbitrary nodes $x \in sset(\Pi)$ and $y \notin sset(\Pi)$. At any time after $t'_0 + T_c$, it must be the case that x has not received any messages from y for more than T_c time. During the time interval $[t'_0 + T_c, t'_0 + T_c + I_c]$, the check freshness timer must have expired at least once, which triggers the logic in the Part I of our protocol to remove y. So after $t'_0 + T_c + I_c$, $y \notin x.neighbors$. □

From now on, Let t_1 be as defined in Lemma 7.

In the following lemmas, for any node x, we denote $x.neighbors$ both $\{x_{+1}, x_{+2}, \ldots\}$ and as $\{x_{-1}, x_{-2}, \ldots\}$, such that $d^+(x, x_{+i}) < d^+(x, x_{+(i+1)})$ and $d^-(x, x_{-i}) < d^-(x, x_{-(i+1)})$, for all $i = 1, 2, \ldots$.

Lemma 8 $\exists t_2 \geq t_1, \forall t, t' > t_2, \forall x \in sset(\Pi), \mathsf{leafset}(x, x.neighbors_t) = \mathsf{leafset}(x, x.neighbors_{t'})$

Proof. To prove the lemma, we define the following derived variables for each node $x \in sset(\Pi)$.

$$x.R^+ = \begin{cases} L - |s.neighbors| & \text{if } |x.neighbors| < L \\ d^+(x, x_{+L}) & \text{otherwise} \end{cases}$$

$$x.R^- = \begin{cases} L - |s.neighbors| & \text{if } |x.neighbors| < L \\ d^-(x, x_{-L}) & \text{otherwise} \end{cases}$$

Let $x.sum = x.R^+ + x.R^-$.

According to Lemmata 1 and 7, the replacement protocol is the only one that will remove nodes from $x.neighbors$ after t_1 for any $x \in sset(\Pi)$. Because the replacement protocol only removes nodes that are not in $\mathsf{leafset}(x, x.neighbors)$ (line 59), the value of $x.sum$ will not change due to the replacement.

Therefore, after t_1, the change of $x.sum$ is only caused by the additions of nodes into $x.neighbors$. It is easy to verify that when nodes are added into $x.neighbors$, variables $x.R^+$ and $x.R^-$ either remain the same or decrease, and thus $x.sum$ either remains the same or decreases. Moreover, $x.sum$ remains the same if and only if $\mathsf{leafset}(x, x.neighbors)$ remains the same.

Since $sset(\Pi)$ is a finite set, and by Lemma 7 after t_1 $x.neighbors \subseteq sset(\Pi)$, there are only a finite number of possible values of $x.sum$ after t_1. So there exist a time $t_{2,x} > t_1$ after which $x.sum$ does not change. So after $t_{2,x}$, $\mathsf{leafset}(x, x.neighbors)$ remains the same. Let $t_2 = \max\{t_{2,x} : x \in sset(\Pi)\}$, and the lemma holds. □

From now on, let t_2 be as defined in Lemma 8.

Corollary 9 *After time t_2, the invite protocol (Part II) will not add any node into the neighbors set of any online node in line 37.*

Proof. It is clear that every execution of line 37 that adds a node y into $x.neighbors$ changes the leafset of node x, so following Lemma 8, no online nodes executes line 37 to add another node into $x.neighbors$ after time t_2. $\square$

Lemma 10 $\forall t > t_2, y \in \mathsf{leafset}(x, x.neighbors_t)$ *if and only if* $x \in \mathsf{leafset}(y, y.neighbors_t)$.

Proof. Suppose $\exists x, y \in sset(\Pi), \exists t > t_2$, $y \in \mathsf{leafset}(x, x.neighbors_t)$ but $x \notin \mathsf{leafset}(y, y.neighbors_t)$. According to Lemma 8, $\mathsf{leafset}(x, x.neighbors_t)$ and $\mathsf{leafset}(y, y.neighbors_t)$ will not change any more after t_2. So we can use $\mathsf{leafset}(x, x.neighbors)$ and $\mathsf{leafset}(y, y.neighbors)$ to refer to x and y's leafset after time t_2. Since $y \in \mathsf{leafset}(x, x.neighbors)$, x will send PING-ASK-INV to y sometime after t (line 21). When y receives this message at some time $t' > t$, y adds x into $y.cand$ (line 25).

Consider y's leafset $\mathsf{leafset}(y, y.neighbors)$, and it is represented as $\{y_{-1}, y_{-2}, \ldots\}$. If $|\mathsf{leafset}(y, y.neighbors)| < L$ or $d^-(y, x) < d^-(y, y_{-L})$, then $x \notin y.neighbors$, because otherwise x is in $\mathsf{leafset}(y, y.neighbors)$. In this case, at the next time after t' when y invites closer nodes (lines 28–32), we have $x \in y.cand \setminus y.neighbors$ and $x \in \mathsf{leafset}(y, y.cand \cup y.neighbors)$, so y sends a PING-INVITE message to x (line 31). Node x will respond to y with a PONG-INVITE message. When y receives this PONG-INVITE, we have $x \in \mathsf{leafset}(y, y.neighbors \cup \{x\})$, so y adds x into $y.neighbors$, and thus $\mathsf{leafset}(y, y.neighbors)$ now include x, contradicting to Lemma 8 stating that no leafset changes after time t_2. Therefore, we know that $|\mathsf{leafset}(y, y.neighbors)| \geq L$ and $d^-(y, x) > d^-(y, y_{-L})$. By a symmetric argument on $\{y_{+1}, y_{+2}, \ldots\}$, we also know that $d^+(y, x) > d^+(y, y_{+L})$.

Hence, we now have $2L$ different nodes $\{y_{-1}, y_{-2}, \ldots, y_{-L}\}$ and $\{y_{+1}, y_{+2}, \ldots, y_{+L}\}$ such that for all $i \in \{1, 2, \ldots, L\}$, $d^-(y, y_{-i}) < d^-(y, x)$ and $d^+(y, y_{+i}) < d^+(y, x)$. Since $y \in \mathsf{leafset}(x, x.neighbors)$, there exists some $j \in \{1, 2, \ldots, L\}$ such that $y = x_{-j}$ or $y = x_{+j}$. Without loss of generality, suppose $y = x_{+j}$. Let the interval $I = (x, y)$. We have that there are less than L nodes in $x.neighbors$ in interval I since $y = x_{+j}$, but there are at least L nodes $\{y_{-1}, y_{-2}, \ldots, y_{-L}\}$ in $y.neighbors$ in interval I.

Therefore, when y receives the PING-ASK-INV message from x at time t', the $view$ that y calculates for x in line 23, which is $\mathsf{leafset}(x, y.neighbors)$, includes at least L nodes in interval I. Therefore there exists at least one node $z \in view \setminus x.neighbors$. Node y sends this $view$ to x (line 24). When x receives the view, it add it into $x.cand$ (line 27). Next time when x invites closer nodes (lines 28–32), we have $z \in x.cand \setminus x.neighbors$ and $z \in \mathsf{leafset}(x, x.cand \cup x.neighbors)$, because $y \in \mathsf{leafset}(x, x.neighbors)$ and $d^+(x, z) < d^+(x, y)$. So x sends a PING-INVITE message to z (line 31). Since $z \in y.neighbors$, by Lemma 7, $z \in sset(\Pi)$. Therefore, z receives the PING-INVITE message from x and sends a PONG-INVITE message back to x. When x receives this PONG-INVITE message from z, it adds z into $x.neighbors$ (line 37), unless z is already in $x.neighbors$. In either case, the leafset of x changes, contradicting to Lemma 8. $\square$

We define two helper functions $\mathsf{succ}(x, set)$ and $\mathsf{pred}(x, set)$ as the following. When $set \setminus \{x\}$ is empty, $\mathsf{succ}(x, set) = \mathsf{pred}(x, set) = x$. When $set \setminus \{x\}$ is not empty, $\mathsf{succ}(x, set)$ is the node $y \in set \setminus \{x\}$ such that $d^+(x, y) = \min\{d^+(x, v) : v \in set \setminus \{x\}\}$, and $\mathsf{pred}(x, set)$ is the node $z \in set \setminus \{x\}$ such that $d^-(x, z) = \min\{d^-(x, v) : v \in set \setminus \{x\}\}$. We also define two derived variables $x.succ$ (the successor of x) and $x.pred$ (the predecessor of x) for node x, $x.succ = \mathsf{succ}(x, x.neighbors)$ and $x.pred = \mathsf{pred}(x, x.neighbors)$.

Given a topology graph G_t at time t, we say that the graph is *loopy* if it satisfies the following conditions: (a) there exists a node v_0 such that $d^+(v_0, 0) < d^+(v_0, v_0.succ_t)$; and (b) on the sequence $v_0, v_1, v_2, \ldots$ with $v_{i+1} = v_i.succ_t$, there exists a node $v_j \neq v_0$ such that $d^+(v_j, 0) < d^+(v_j, v_j.succ_t)$.

Lemma 11 *For all time $t > t_2$, G_t is not loopy. Moreover, there exists time $t' > t_2$ such that no node sends* PONG-DELOOPY *message (line 73) after time t'.*

Proof. Suppose, for a contradiction, that at time $t > t_2$ graph G_t is loopy. By Lemma 8 we know that after time t_2 the leafset of every node remains unchanged, and therefore, the successor of every node remains unchanged. We can use $x.succ$ to represent $x.succ_t$ for all time $t > t_2$. If G_t is loopy, there exists a node v_0 such that $d^+(v_0, 0) < d^+(v_0, v_0.succ)$, and on the sequence $v_0, v_1, v_2, \ldots$ with $v_{i+1} = v_i.succ_t$, there exists a node $v_j \neq v_0$ such that $d^+(v_j, 0) < d^+(v_j, v_j.succ)$. Without loss of generality, let v_j be the first node in the sequence that has the property. According to the deloopy protocol (Part IV), node v_0 will send a $(\text{PING-DELOOPY}, v_0)$ message to $v_0.succ$ (line 68). By Lemma 7, every node v_i on the sequence is online, so every node v_i relay the $(\text{PING-DELOOPY}, v_0)$ message to $v_i.succ$ (line 75), until the message reaches v_j. On v_j, the loopy detection condition on line 71 is true, so v_j adds v_0 into $v_j.cand$ (line 72) and send a PONG-DELOOPY message back to v_0 (line 73). When v_0 receives this message, it adds v_j into $v_0.cand$ (line 77).

By the condition $d^+(v_0, 0) < d^+(v_0, v_0.succ)$ and $d^+(v_j, 0) < d^+(v_j, v_j.succ)$, it is straightforward to see that either v_0 is in the interval $(v_j, v_j.succ)$, or v_j is in the interval $(v_0, v_0.succ)$. If v_0 is in the interval $(v_j, v_j.succ)$, then we know that $v_0 \notin v_j.neighbors$ and $v_0 \in \mathsf{leafset}(v_j, v_j.neighbors \cup \{v_0\})$. Since v_0 is added into $v_j.cand$, v_0 will send a PING-INVITE to v_0 (line 31), which will lead to the addition of v_0 into v_j's leafset, contradicting to Lemma 8 stating that the leafset of any node will not change after t_2. If v_j is in the interval $(v_0, v_0.succ)$, similarly we can show that v_j will be added into v_0's leafset, again a contradiction. Therefore, G_t is not loopy for all time $t > t_2$.

Since G_t is not loopy for all time $t > t_2$, any $(\text{PING-DELOOPY}, x)$ message sent by x in line 68 after time t_2 will not trigger loopy detection, i.e., it will not trigger some node y to send PONG-DELOOPY to x. Since all messages sent before time t_2 eventually disappear from the system, there is a time t' after which no node sends PONG-DELOOPY message. □

Lemma 12 *Suppose that there is a time $t > t_2$ and two nodes $x, z \in sset(\Pi)$ such that $z \in x.neighbors_t \setminus \mathsf{leafset}(x, x.neighbors_t)$.*
1). There exist a time $t' > t$ and a node $y \in \mathsf{leafset}(z, z.neighbors_t)$ such that $d(x,y) < d(x,z)$ and $y \in x.neighbors_{t'}$.
2). Suppose further that node x and z are such that $d(x,z) = \max\{d(u,w) : w \in u.neighbors_t \setminus \mathsf{leafset}(u, u.neighbors_t), u, w \in sset(\Pi)\}$, and there is no invocation of $\mathsf{add}()$ at or after time $t - 2\Delta$. Then there is a time $t' > t$ and a node $y \in \mathsf{leafset}(z, z.neighbors_t)$ such that $d(x,y) < (x,z)$ and x replaces z with y at time t'.

Proof. By Lemma 8, the leafset of every node does not change after time t_2, so we just use $\mathsf{leafset}(v, v.neighbors)$ to represent the leafset of v after time t_2.

We prove (1) first. Suppose, for a contradiction, that at time $t > t_2$ we have $x, z \in sset(\Pi)$ such that $z \in x.neighbors_t \setminus \mathsf{leafset}(x, x.neighbors_t)$, but for all time $t' > t$ and all $y \in \mathsf{leafset}(z, z.neighbors_t)$ such that $d(x,y) < d(x,z)$, we have $y \notin x.neighbors_{t'}$. Note that, this means that z is not replaced by x after time t in line 62. Because if x replaces z with a node y, then y is a replacement provided by z, which means $y \in \mathsf{leafset}(z, z.neighbors)$ and $d(x,y) < d(x,z)$. According to line 60, y is added to $x.neighbors$ before z is replaced. Since no such y is found after time t, we know that z is never replaced.

By Lemmata 1 and 7, after time t_2 only the replacement protocol can remove a node from a *neighbors* set. Thus we know that z is always in $x.neighbors$ after time t. Since the leafset of x does not change after time t_2, z is always in $x.neighbors \setminus \mathsf{leafset}(x, x.neighbors)$ after time t.

By Lemma 10, we know that $x \notin \mathsf{leafset}(z, z.neighbors)$, because otherwise $z \in \mathsf{leafset}(x, x.neighbors)$. Then $\mathsf{leafset}(z, z.neighbors)$ must have $2L$ nodes. Otherwise, $z.neighbors_{t'}$ is the same as $\mathsf{leafset}(z, z.neighbors)$ with less than $2L$ nodes for all time $t' \geq t$. In this case, x cannot be in $z.neighbors_{t'}$. Since z is always in $x.neighbors$, x will send a PING-ASK-INV message z, and z will add x to $z.cand$ after receiving the message (line 25). Then later z will send x a PING-INVITE and eventually x will be added into $z.neighbors$ and thus changes $\mathsf{leafset}(z, z.neighbors)$, contradicting to Lemma 8.

Because z is always in $x.neighbors \setminus \mathsf{leafset}(x, x.neighbors)$ after time t, x periodically send PING-ASK-REPL messages to z, and z will try to find a replacement in $\mathsf{leafset}(z, z.neighbors)$ (line 46). Among the $2L$ nodes in $\mathsf{leafset}(z, z.neighbors)$, there must be some node v that satisfies $d(x, v) < d(x, z)$. In fact, it is easy to verify that if $d(x, z) = d^+(x, z)$, then there are L nodes in the interval (x, z) satisfying the above condition, and if $d(x, z) = d^-(x, z)$, there are L nodes in the interval (z, x) satisfying the above condition. This means that z will be able to find a proper replacement y and sends $(\text{PONG-ASK-REPL}, y)$ back to x. Because y is selected deterministically from $\mathsf{leafset}(z, z.neighbors)$, z will always provide the same y to x after t_2. When x receives this message, it sets $x.z.repl$ to y.

Then x will send $(\text{PING-REPLACE}, z, rnd)$ to y (line 53). By Lemma 10, $y \in \mathsf{leafset}(z, z.neighbors)$ implies that $z \in \mathsf{leafset}(y, y.neighbors)$. So when y receives the $(\text{PING-REPLACE}, z, rnd)$ from x, y sends $(\text{PONG-REPLACE}, z, rnd)$ back to x (line 57). When x receives this message from x at a time $t' > t$, we know that $z \in x.neighbors_{t'} \setminus \mathsf{leafset}(x, x.neighbors)$ and $y = x.z.repl$. Therefore, the condition in line 59 holds and x adds y into $x.neighbors$ (line 60). Since we have $d(x, y) < d(x, z)$ and $y \in \mathsf{leafset}(z, z.neighbors)$, Part (1) of the lemma holds.

We now prove Part (2). We continue the proof in Part (1) with the added assumption that $d(x, z) = \max\{d(u, w) : w \in u.neighbors_t \setminus \mathsf{leafset}(u, u.neighbors_t)\}$, and there is no invocation of $\mathsf{add}()$ after time t. Under these conditions and x's repeatedly attempts to replace z, we show that at some time the condition in line 61 will become true and thus the replacement will succeed eventually.

If the condition in line 61 is not true, then after x sends out the $(\text{PING-REPLACE}, z, rnd)$ to y, either x verifies the replacement of z for another node u (line 56), or x replaces another node u with z (line 63).

For the first case, we consider $y.z.repl$ for all y such that $z \in y.\mathsf{leafset} \setminus \mathsf{leafset}(y, y.neighbors)$. Because every y periodically refreshes $y.z.repl$ by sending PING-ASK-REPL to z and z always returns nodes in its own leafset, there exist a time τ after which $x \neq y.z.repl$ for all y. So after τ, nobody will send $(\text{PING-REPLACE}, z, rnd)$ to x, and x will never verify the replacement of z after $\tau + \Delta$. Therefore, $x.z.commit$ will not increase due to the verifications of z for some other nodes after $\tau + \Delta$.

For the second case, it implies $d(x, z) < d(x, u)$ and $u \in x.neighbors_{t'} \setminus \mathsf{leafset}(x, x.neighbors)$ for some time $t' > t$. According to the choice of x and z, we conclude that u cannot be in $x.neighbors_t$. Therefore u is added after time t but before time t'. From Corollary 9, no nodes are added into $x.neighbors$ after time t_2 by the invite protocol. By our condition, no $\mathsf{add}()$ is invoked at or after time $t - 2\Delta$ so no node is added into $x.neighbors$ at or after time t due to the receipt of PONG-CONTACT message. So the only place that x can add u into $x.neighbors$ after time t is in the replacement protocol (line 60). If so, it means there is yet another node v such that $d(x, u) < d(x, v)$ and $v \in x.neighbors_{t''} \setminus \mathsf{leafset}(x, x.neighbors)$ and $t < t'' < t'$. However, we know that v cannot be in $x.neighbors_t$ either, so v is added into $x.neighbors$ after time t. We cannot repeat this argument forever since there are only a finite number of nodes, so we reach a contradiction. Therefore, after t $x.z.commit$ will not increase due to x's own replacement of some other nodes.

Since $x.z.commit$ stops increase after $\max(\tau + \Delta, t)$, we can always find a time $t'' \geq \max(\tau + \Delta, t)$ at which $x.round \geq x.z.commit$ and x sends PING-REPLACE to a node $y \in \mathsf{leafset}(z, z.neighbors_{t''})$ and $d(x, y) < d(x, z)$. After x gets the corresponding PONG-REPLACE from y, the condition in line 61 must be true and x replaces z with y. □

Lemma 13 (Leafset Cleanup) *If there is a time t after which no* add() *is invoked at any node in the system, then there is a time t' such that for all time $t'' \geq t'$ and all $x \in sset(\Pi)$,* $\mathsf{leafset}(x, x.neighbors_{t''}) = x.neighbors_{t''}$*.*

Proof. Let t'_1 be $\max(t_2, t + 2\Delta)$. Given any time τ, we define a metric $m(\tau) = \max\{d(x,z) : z \in x.neighbors_\tau \setminus \mathsf{leafset}(x, x.neighbors_\tau), x, z \in sset(\Pi)\}$ if there exists x and z such that $z \in x.neighbors_\tau \setminus \mathsf{leafset}(x, x.neighbors_\tau)$, otherwise $m(\tau) = 0$. We show that after time t'_1, $m(\tau)$ is non-increasing as τ increases and eventually it becomes 0.

First, we notice that $m(\tau)$ only changes when some $neighbors$ set changes. After time t_2, we know that no node is added into any sink set due to the invocation of add(), no node is added by the invite protocol (Corollary 9), and no node is removed by the liveness check protocol (Lemmata 1 and 7). Therefore, a node may only be added or removed by the replacement protocol. If a node y is added into $x.neighbors$ by the replacement protocol in line 60 at a time τ, there must be a node z such that $z \in x.neighbors_\tau \setminus \mathsf{leafset}(x, x.neighbors_\tau)$ (line 59) and $d(x,y) < d(x,z)$ (line 46). Hence this step of adding y will not affect metric $m(\tau)$. The removal of a node cannot increase $m(\tau)$, therefore after time t'_1, $m(\tau)$ is non-increasing.

If at a time $\tau > t'_1$ we have $m(\tau) > 0$, let x and z be the nodes such that $z \in x.neighbors_\tau \setminus \mathsf{leafset}(x, x.neighbors_\tau)$ and $d(x,z) = m(\tau)$. By Lemma 12 2), there is a time $\tau' > \tau$ such that x replaces z with a node y with $d(x,y) < d(x,z)$. If there are multiple such pairs of x and z with $d(x,z) = m(\tau)$, they will all be replaced by Lemma 12 2). Therefore, there is a time $\tau' > \tau$ such that $m(\tau') < m(\tau)$. Since there are finite nodes in $sset(\Pi)$, metric $m(\tau)$ can only take finitely many values, therefore eventually $m(\tau)$ will become 0 at some time t' and stays as 0 afterwards. When this occurs, we know that for every node $x \in sset(\Pi)$, $x.neighbors_{t''} = \mathsf{leafset}(x, x.neighbors_{t''})$ for all time $t'' > t'$. □

We sort a node x's $neighbors$ set as the following:
$x.neighbors^+ = \{x_{+1}, x_{+2}, \ldots\}$ s.t. $d^+(x, x_{+i}) < d^+(x, x_{+(i+1)}), i = 1, 2, \ldots$
$x.neighbors^- = \{x_{-1}, x_{-2}, \ldots\}$ s.t. $d^-(x, x_{-i}) < d^-(x, x_{-(i+1)}), i = 1, 2, \ldots$

Lemma 14 *1). $\forall t, t' > t_2$, $x.succ_t = x.succ_{t'}$ and $x.pred_t = x.pred_{t'}$.*
2). $\forall t > t_2$, $x.succ_t = y$ is equivalent to $y.pred_t = x$.
3). Suppose $x.neighbors_t \neq \emptyset$. $\forall t > t_2$, $\forall x \in sset(\Pi)$, consider the nodes $\{x_{+1}, x_{+2}, \ldots, x_{+M}\}$ in $x.neighbors^+$ and $\{x_{-1}, x_{-2}, \ldots, x_{-M}\}$ in $x.neighbors^-$ with $M = \min(L, |x.neighbors|)$. We have $x_{+i}.succ_t = x_{+(i+1)} \wedge x_{-i}.pred_t = x_{-(i+1)}, 1 \leq i \leq M-1$.

Proof.

Proof of 1). It is immediate from Lemma 8 and the fact that $x.succ_t$ and $x.pred_t$ is only determined by $\mathsf{leafset}(x, x.neighbors_t)$.

Proof of 2). The statement is trivially true when $x.neighbors_t = \emptyset$, so we consider $x.neighbors_t \neq \emptyset$. From $x.succ_t = y$, we know that $y \in \mathsf{leafset}(x, x.neighbors_t)$. By Lemma 10 it must be true that $x \in \mathsf{leafset}(y, y.neighbors_t)$. Suppose $y.pred_t = z \neq x$, it must be true that $d^-(y,z) < d^-(y,x)$. According to the definition of distance function d^- and d^+, we know that $d^+(x,z) < d^+(x,y)$.

Since $\mathsf{leafset}(x, x.neighbors)$ never change, x will send PING-ASK-INV message to y later, according to our protocol part II. Upon the receipt of x's PING-ASK-INV message, y calculates the *view* as $\mathsf{leafset}(x, y.neighbors)$. This implies that y will put a node $u \in y.neighbors$ with $d^+(x,u) \leq d^+(x,z)$ in the *view* variable in the acknowledged PONG-ASK-INV to x, thus allowing u to enter $x.cand$. Since

$d^+(x,u) < d^+(x,y)$, next time when x invites closer nodes (lines 28–32), x must send a PING-INVITE to some node v such that $d^+(x,v) \leq d^+(x,u) < d^+(x,y)$ and v is online. Node v will reply with a PONG-INVITE to x, and when x receives this message, x adds v into its *neighbors* set since $v \in \mathsf{leafset}(x, x.neighbors \cup \{v\})$, unless v is already in $x.neighbors$ by that time. In either case, it contradicts to Lemma 8 saying that the leafset never changes. Therefore, we have $y.pred_t = x$.

We can also prove that $y.pred_t = x$ implies $x.succ_t = y$ in a similar way.

Proof of 3). First we prove $x_{+i}.succ_t = x_{+(i+1)}, 1 \leq i \leq M-1$. Suppose this is not true. There must exists a j $(1 \leq j \leq M-1)$, such that $x_{+j}.succ_t = z \neq x_{+(j+1)}$.

Suppose $d^+(x_{+j}, z) > d^+(x_{+j}, x_{+(j+1)})$. Because $x_{+j} \in \mathsf{leafset}(x, x.neighbors_t)$ implies $x \in \mathsf{leafset}(x_{+j}, x_{+j}.neighbors_t)$ and $\mathsf{leafset}(x_{+j}, x_{+j}.neighbors)$ never changes, x_{+j} will send PING-ASK-INV message to x some time after t, according to our protocol part II. Upon the receipt of the PING-ASK-INV message, x will put $x_{+(j+1)}$ (or some other nodes in the interval (x_{+j}, z)) in the *view* variable in the PONG-ASK-INV message as the reply. Using a similar reasoning in the proof of 2), we know that the leafset of x_{+j} changes, a contradiction.

Suppose $d^+(x_{+j}, z) < d^+(x_{+j}, x_{+(j+1)})$. Because $x_{+j} \in \mathsf{leafset}(x, x.neighbors_t)$ and $\mathsf{leafset}(x, x.neighbors)$ never changes, x will send PING-ASK-INV message to x_{+j} some time after t, according to our protocol part II. Upon the receipt of the PING-ASK-INV message, x_{+j} will put z in the *view* variable in the PONG-ASK-INV message as the reply, leading to the change of $\mathsf{leafset}(x, x.neighbors)$, again a contradiction.

So it must be true that $x_{+i}.succ_t = x_{+(i+1)}, 1 \leq i \leq M-1$. $x_{-i}.pred_t = x_{-(i+1)}, 1 \leq i \leq M-1$ could be proved in the similar way. □

After time t_2, since the leafset does not change, $x.succ$ and $x.pred$ do not change either, so we just use them to represent their values at any time after time t_2. After time t_2, given any node $x \in sset(\Pi)$, let *closed sequence* $\rho(x) = (x = x_0, x_1, \ldots, x_k)$ such that $x_i.succ = x_{i+1}$ for all $i = 0, 1, \ldots, k-1$, $x_i = x_j$ iff. $i = j$, and $x_k.succ = x_j$ for some $j = 0, 1, \ldots, k$. We say that $\rho(x)$ is a *closed loop* if $x_k.succ = x_0 = x$. For convenience, we also use $\rho(x)$ to represent the set of nodes in the sequence.

Corollary 15 *For any node $x \in sset(\Pi)$, $\rho(x)$ exists, is unique, and is a closed loop. Moreover, for all $y \in \rho(x)$, $\mathsf{leafset}(y, y.neighbors_t) \subseteq \rho(x)$ for all time $t > t_2$.*

Proof. Closed sequence $\rho(x)$ exists because $sset(\Pi)$ is a finite set. It is unique because the $x.succ$ variable has a unique value. Suppose $x_k.succ = x_j \neq x_0$ with $0 < j \leq k$. If $x_k.succ = x_k$, $x_k.neighbors \setminus \{x_k\}$ must be empty according to the definition of helper function succ. However, by Lemma 14 2) we have $x_k.pred = x_{k-1} \neq x_k$, which means that $x_k.neighbors \setminus \{x_k\}$ is not empty. This is contradictory. If $x_k.succ = x_j (j < k)$, then according to Lemma 14 2), $x_k = x_j.pred$. Since by definition we also have $x_{j-1} = x_j.pred$, so $x_{j-1} = x_k$, still a contradiction. Therefore, $x_k.succ = x_0$ and $\rho(x)$ is a closed loop. By Lemma 14 3), it is straightforward to see that for all $y \in \rho(x)$, $\mathsf{leafset}(y, y.neighbors_t) \subseteq \rho(x)$ for all time $t > t_2$. □

Lemma 16 *If for some time $t \geq t'_0 + \Delta$, G_t is weakly connected, then after time t_2, for any node $x \in sset(\Pi)$, $\rho(x) = sset(\Pi)$.*

Proof. Suppose, for a contradiction that there exists $x, z \in sset(\Pi)$ such that $z \in sset(\Pi) \setminus \rho(x)$. By Lemma 5, the topology is always connected after time t. Thus, for a time $t' > \max(t, t_2)$, $G_{t'}$ is connected. So there is a path from x to z when treating edges in $G_{t'}$ as undirected. Along the path, we can find node x_0 and z_0 such that $x_0 \in \rho(x)$ and $z_0 \notin \rho(x)$, and either $\langle x_0, z_0 \rangle$ is in $G_{t'}$ or $\langle z_0, x_0 \rangle$ is in $G_{t'}$.

Consider the first case where $\langle x_0, z_0\rangle$ is in $G_{t'}$. By Corollary 15, $z_0 \in x_0.\mathit{neighbors}_{t'} \setminus \mathsf{leafset}(x_0, x_0.\mathit{neighbors}_{t'})$. By Lemma 12 1), there is a time $\tau_0 > t'$ and a node $y_0 \in \mathsf{leafset}(z_0, z_0.\mathit{neighbors}_{t'})$ such that $d(x_0, y_0) < d(x_0, z_0)$ and $y_0 \in x_0.\mathit{neighbors}_{\tau_0}$. By Corollary 15, we know that $y \notin \rho(x)$. Otherwise since $y \in \rho(x)$ leads to $\mathsf{leafset}(y_0, y_0.\mathit{neighbors}_{\tau_o}) \subseteq \rho(x)$, and $z_0 \in \mathsf{leafset}(y_0, y_0.\mathit{neighbors}_{\tau_0})$ by Lemma 10, $z_0 \in \rho(x)$, a contradiction. Since $\mathsf{leafset}(x_0, x_0.\mathit{neighbors}_{\tau_0}) \subseteq \rho(x)$, we also have $y_0 \notin \mathsf{leafset}(x_0, x_0.\mathit{neighbors}_{\tau_0})$. Thus we find a node y_0 such that $y_0 \in x_0.\mathit{neighbors}_{\tau_0} \setminus \mathsf{leafset}(x_0, x_0.\mathit{neighbors}_{\tau_0})$ and $d(x_0, y_0) < d(x_0, z_0)$. We can continue applying Lemma 12 1) to find nodes $y_1, y_2, \ldots$ and time points $\tau_1, \tau_2, \ldots$ such that $y_i \in x_0.\mathit{neighbors}_{\tau_i} \setminus \mathsf{leafset}(x_0, x_0.\mathit{neighbors}_{\tau_i})$, $d(x_0, y_{i+1}) < d(x_0, y_i)$, and $\tau_i < \tau_{i+1}$. However, since there are only a finite number of nodes in $\mathit{sset}(\Pi)$, this process cannot continue indefinitely, a contradiction.

Consider the second case where $\langle z_0, x_0\rangle$ is in $G_{t'}$. In this case, we consider the closed loop $\rho(z_0)$. We have $x_0 \notin \rho(z_0)$, otherwise it means $\rho(x) = \rho(x_0) = \rho(z_0)$ and thus $z_0 \in \rho(x)$. Then we can apply the same argument as in case 1 with the roles of x_0 and z_0 reversed and reaches a contradiction. Therefore the lemma holds. □

Lemma 17 *If for some time $t \geq t'_0 + \Delta$, G_t is weakly connected, then for all time $t' > t_2$ and all $x \in \mathit{sset}(\Pi)$, $x.\mathit{succ}_{t'} = \mathsf{succ}(x, \mathit{sset}(\Pi))$.*

Proof. We consider the time after t_2 when the leafset on every node does not change. In this case, we omit the subscript in $x.\mathit{succ}$.

We only consider the cases that $\mathit{sset}(\Pi) \neq \emptyset$. If $\mathit{sset}(\Pi) = \{x\}$, according to Lemma 7, $x.\mathit{neighbors}$ becomes empty after t_1. So $x.\mathit{succ} = x$ and $\mathsf{succ}(x, \mathit{sset}(\Pi)) = x$, the lemma holds. Now consider the case that $|\mathit{sset}(\Pi)| > 1$. Suppose, for a contradiction, that there is some node $x \in \mathit{sset}(\Pi)$ such that $x.\mathit{succ} \neq \mathsf{succ}(x, \mathit{sset}(\Pi)) = y$.

Consider the closed loop $\rho(x)$. By Lemma 16, $\rho(x) = \mathit{sset}(\pi)$, so $|\rho(x)| > 1$, and $y \in \rho(x)$. We claim that for any leafset topology G_t containing the above closed loop $\rho(x)$, G_t must be loopy, which contradicts to Lemma 11, and thus the lemma holds.

To prove the claim, we use the following properties of the circular space $\mathcal{K}$. For $u, v \in \mathcal{K}$ and $u \neq v$, we use the notion $[u, v)$ to denote the interval $(u, v) \cup \{u\}$. It is obvious that $w \in [u, v)$ is equivalent to $d^+(u, w) < d^+(u, v)$. Moreover, it is also straightforward to verify that $0 \in [u, v)$ is equivalent to $u = 0$ or $v < u$. With this, we know that given a sequence $u_0, u_1, \ldots, u_k$ where all nodes in the sequence are different, if $0 \in [u_0, u_k)$, then $0 \in [u_i, u_{i+1})$ for some $i \in \{0, 1, \ldots, k-1\}$. This is because, if some $u_i = 0$, then $0 \in [u_i, u_{i+1})$; if no u_i is 0, then $u_k < u_0$, which means there must exist $u_{i+1} < u_i$, which implies $0 \in [u_i, u_{i+1})$.

Let $z = x.\mathit{succ}$. Since $y = \mathsf{succ}(x, \mathit{sset}(\Pi))$ and $x.\mathit{neighbors} \subseteq \mathit{sset}(\Pi)$, we have $d^+(x, y) < d^+(x, z)$, i.e. $y \in [x, z)$. We now consider the following two possible cases. In the first case, we have $0 \in [x, y)$. It is easy to verify that in this case $0 \in [x, z)$ and $0 \in [z, y)$. Thus in the sequence from z to y in $\rho(x)$, there is a node u such that $0 \in [u, u.\mathit{succ})$. Since all nodes in the sequence from z to y is different from x, we find two nodes x and u in $\rho(x)$ such that $0 \in [x, x.\mathit{succ})$ and $0 \in [u, u.\mathit{succ})$. By definition this means that any leafset topology containing $\rho(x)$ is loopy.

In the second case, $0 \in [y, x)$. Thus in the sequence from y to x in $\rho(x)$ (but excluding x) we have a node u such that $0 \in [u, u.\mathit{succ})$. Now consider the interval $[x, z)$. If $0 \in [x, z)$, we already find two different nodes x and u such that $0 \in [x, x.\mathit{succ})$ and $0 \in [u, u.\mathit{succ})$, which means the leafset topology containing $\rho(x)$ is loopy. If $0 \notin [x, z)$, then we have $0 \in [z, x)$. Together with $0 \in [y, x)$ and $y \in [x, z)$, it is easy to verify that $0 \in [z, y)$. Thus in the sequence from z to y in $\rho(x)$ (but excluding y) we have a node v such that $0 \in [v, v.\mathit{succ})$. The sequence from z to y excluding y has no overlap with the sequence from y to

x, so $u \neq v$. Therefore we again find two nodes u and v such that $0 \in [u, u.succ)$ and $0 \in [v, v.succ)$, which implies that the leafset topology containing $\rho(x)$ is loopy. In all cases, we reach a contradiction. Therefore, the lemma holds. $\square$

Lemma 18 (Leafset Stabilization) *If for some time $t \geq t'_0 + \Delta$, G_t is weakly connected, then there exist a time $t' > t$, such that for all $t'' \geq t'$ and all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors_{t''}) = \mathsf{leafset}(x, sset(\Pi))$.*

Proof. After time t_2, there is no leafset change, and thus we omit the subscript in $x.succ$ and $\mathsf{leafset}(x, x.neighbors)$. It is sufficient to show that for all time after t_2 and for all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors) = \mathsf{leafset}(x, sset(\Pi)$.

By Lemma 17, for all $x \in sset(\Pi)$, $x.succ = \mathsf{succ}(x, sset(\Pi))$. According to the definition of succ and pred, we know that $\mathsf{succ}(x, sset(\Pi)) = y$ is equivalent to $\mathsf{pred}(y, sset(\Pi)) = x$. According to Lemma 14 1) and 2), we also have $x.pred = \mathsf{pred}(x, sset(\Pi))$.

Now consider $\{x_{+1}, x_{+2}, \ldots, x_{+M}\}$ in $x.neighbors^{+}$, and $\{x_{-1}, x_{-2}, \ldots, x_{-M}\}$ in $x.neighbors^{-}$ with $M = \min(L, |x.neighbors|)$, for an arbitrary node $x \in sset(\Pi)$. By Lemma 14 3) we have $x_{+1} = x.succ = \mathsf{succ}(x, sset(\Pi))$ and $x_{+(i+1)} = x_{+i}.succ = \mathsf{succ}(x_{+i}, sset(\Pi))$, $x_{-1} = x.pred = \mathsf{pred}(x, sset(\Pi))$ and $x_{-(i+1)} = x_{-i}.pred = \mathsf{pred}(x_{-i}, sset(\Pi))$, for all $i = 1, 2, \ldots, M-1$. This directly implies that $\mathsf{succ}(x_{+i}, sset(\Pi)) \in \mathsf{leafset}(x, sset(\Pi))$ and $\mathsf{pred}(x_{-i}, sset(\Pi)) \in \mathsf{leafset}(x, sset(\Pi))$, for all $i = 1, 2, \ldots, M-1$. Thus we have $\mathsf{leafset}(x, x.neighbors) \subseteq \mathsf{leafset}(x, sset(\Pi))$.

If $|\mathsf{leafset}(x, x.neighbors)| < |\mathsf{leafset}(x, sset(\Pi))|$, then $\mathsf{leafset}(x, x.neighbors) = x.neighbors$, and since $\rho(x) = sset(\Pi)$, there must be some node $y \in \rho(x)$ such that $y \notin x.neighbors$. Let y be the first such one following the sequence $\rho(x)$, i.e., $y = z.succ$ while $z \in x.neighbors$. In this case the invite protocol (part II) will cause z to introduce new nodes to x and thus x's leafset will add new nodes in.

Therefore we have $|\mathsf{leafset}(x, x.neighbors)| = |\mathsf{leafset}(x, sset(\Pi))|$. In this case, if $\mathsf{leafset}(x, sset(\Pi)) \not\subseteq \mathsf{leafset}(x, x.neighbors)$, there exists a node $y \in \mathsf{leafset}(x, sset(\Pi)) \setminus \mathsf{leafset}(x, x.neighbors)$. So either there exists a x_{+j} such that $y \in (x_{+j}, x_{+(j+1)})$ for some $j = 1, 2, \ldots, M-1$, or there exists a x_{-j} such that $y \in (x_{-j}, x_{-(j+1)})$ for some $j = 1, 2, \ldots, M-1$. In the first case, we have $x_{+j}.succ = x_{+(j+1)} \neq \mathsf{succ}(x_{+j}, sset(\Pi))$, contradicting to Lemma 17. In the second case, we have $x_{-(j+1)}.succ = x_{-j} \neq \mathsf{succ}(x_{-(j+1)}, sset(\Pi))$, again contradicting to Lemma 17. Therefore $\mathsf{leafset}(x, sset(\Pi)) \subseteq \mathsf{leafset}(x, x.neighbors)$, and thus $\mathsf{leafset}(x, sset(\Pi)) = \mathsf{leafset}(x, x.neighbors)$. $\square$

Theorem 1 *When $I_c, T_c \geq I_p + 2\Delta$, the leafset maintenance protocol provided in Fig. 1, 2, 3, and 5 is self-stabilizing. In particular it satisfies the Leafset Stabilization, Partition Healing, and Leafset Cleanup properties.*
Proof. The theorem follows from Lemmata 6, 13, and 18. $\square$

Theorem 2 *When $I_c, T_c \geq I_p + 2\Delta$, the leafset maintenance protocol provided in Fig. 1, 2, 3, and 5 satisfies the Cost Effectiveness property. In particular, in the steady state the size of local state on each node is $O(L)$ and the total size of messages sent by each node in one detection-repair cycle is $O(L^2)$.*
Proof (Sketch). In the steady state when both the system and the leafset topology are stabilized, each node only maintains at most $2L$ nodes in its $neighbors$ set, at most $4L$ nodes in its $cand$ set, and for each node y in the $neighbors$ set $y.repl$ is a single node. Thus the size of the local state is $O(L)$.

Considering the messaging cost, each node (a) periodically sends at most $2L$ PING-ALIVE messages with at most $2L$ PONG-ALIVE responses; (b) periodically sends at most $2L$ PING-ASK-INV messages with at most $2L$ PONG-ASK-INV messages, each of which contains at most $2L$ nodes. The deloopy protocol sends at most N messages in one deloopy period among all nodes where N is the number of online nodes in the steady state. So each node only sends out one message on average in one deloopy period. Nodes will not send PING-CONTACT, PONG-CONTACT, PING-INVITE, PONG-INVITE, PING-ASK-REPL, PONG-ASK-REPL, PING-REPLACE, PONG-REPLACE, and PONG-DELOOPY messages in the steady state. Therefore, in a constant period of time the total size of all messages one node sends is at most $O(L^2)$.

We now consider the length of the detection-repair cycle. In the worst case, a node sends our a PONG-ALIVE message to all nodes in its $neighbors$ set and then crashes immediately at time t, the PONG-ALIVE message takes Δ to arrive at all nodes, and this message makes others not detecting the failure for another T_c time units. Right before time $t + \Delta + T_c$, all nodes checked the liveness once, so it takes another I_c to do the next check. Thus by time $t+\Delta+T_c+I_c$, all processes detect the failure and removed the crashed node from their $neighbors$. Then these nodes need to add new nodes back via the invite protocol. Let I_a be the interval of sending periodic PING-ASK-INV messages. Then it may take another I_a to send PING-ASK-INV out, which takes another 2Δ to collect views back. Let I_i be the interval of sending periodic PING-INVITE messages. It then takes another I_i to send out PING-INVITE, and takes another 2Δ to receive PONG-INVITE, by which the leafset should be repaired. Therefore, in the worst case, the detection-repair cycle takes $\Delta + T_c + I_c$ time for detection, and takes $I_a + I_i + 4\Delta$ for repair. Therefore, the detection-repair cycle is constant. Since each node only sends out $O(L^2)$ messages in a constant period, the total messaging cost on each node in one detection-repair cycle is $O(L^2)$. □

6 Improvement for Fast Convergence

In this section, we describe several optimizations to our self-stabilizing protocol to significantly speed up the stabilization process.

Leafset topology stabilization consists of two periods. The first period starts at the system stabilization time t_0 and ends when all $neighbors$ sets on all nodes contain the correct leafset members, and it corresponds to the Leafset Stabilization property. The second period starts at the end of the first period and ends when all $neighbors$ sets only contain leafset members (provided there are no more add() invocations), and it corresponds to the Leafset Cleanup property. We call the length of the first period *convergence time* and the length of the second period *cleanup time*. Reducing the convergence time is important because it significantly reduces the transition period where overlay routings may be incorrect, and it makes the topology more robust under churn. Thus, we focus our discussion on reducing convergence time. Reducing cleanup time is listed as one of our future work items.

To reduce convergence time, we make use of finger tables, since they maintain faraway links so that a node may learn about other nodes in its leafset faster through finger tables. Our first optimization is on the invite protocol. In addition to provide $neighbors$ set to other nodes as leafset candidates (line 23), a node can also provide its finger table entries as candidates. We can adapt our proof easily and show that as long as the finger tables do not contain offline entries eventually, our protocol is still correct. This optimization is similar to ones used by other protocols.

There are several special classes of topologies in which the above optimization is not helpful. We find that these special topologies are more difficult to converge than random topologies, but they are not addressed by previous studies. We now propose a few additional optimizations that handle these cases.

6.1 Merging of the multi-ring topology

One special class of topologies is *multi-ring* topologies, in which several ring structures are connected by a few cross-ring links (Fig. 6). Multi-ring topologies can be generated due to network partitions. After a network partition, the topology may be broken into several disconnected components, all of which stabilize

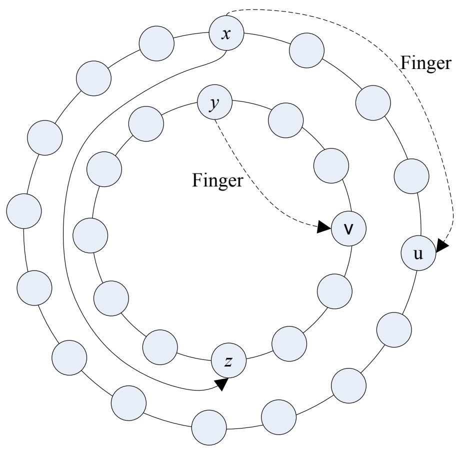


Figure 6: Multi-ring topology.

into a ring structure. When the network recovers from the partition, the application or a bootstrap system may add a few cross-ring links to connect the topology.

In Fig. 6, there are two separate rings and the only cross-ring link is from x to z (z is in x's $neighbors$ set). The invite protocol will not be helpful for this topology when all candidates that z can provide to x are outside x's current leafset range. In this case, through the replacement protocol node x will eventually learn a node in z's ring that is within x's leafset range, for example, node y in Fig. 6. We call two close nodes x and y learning about each other the creation of the first *healing point* between the two separate rings. Once the first healing point is created, the two rings will merge by the invite protocol along the two directions on the rings. Let N be the number of online nodes in the multi-ring topology. With our current protocol, it may take $O(N)$ time to create the first healing point and take another $O(N)$ time to merge the two rings. To reduce the convergence time, we need to shorten both the period to create the first healing point and the period to merge rings.

To speed up the creation of the first healing point, we allow node x to issue a special routing request starting from z using x's own ID as the routing key. With the help of fingers, the routing takes $O(\log N)$ steps to reach the routing destination y. So within $O(\log N)$ time, x can learn about y, creating the first healing point.

To speed up the ring merging process after the first healing point, we want to spawn more healing points to merge the rings in parallel instead of merging in linear fashion along the two directions. To do so, we let leafset neighbors exchange their finger tables. For example, as shown in Figure 6, suppose u is a finger of x and v is a finger of y, and the first healing point is already created between nodes x and y. Through finger table exchange between x and y, x learns about v. When x probes its finger u, x tells u about v since v is close to u from x's point of view. On receiving the probe message, u puts v in its $cand$ set. If v is indeed in u's leafset range, v will be pulled into $u.neighbors$ by the invite protocol. When this happens, a new healing point between u and v is created. This process is carried out for all finger table entries in order to spawn as many healing points as possible.

Although there is no guarantee that every round of leafset exchanges and finger probes at a healing point generates new healing points, we anticipate that well distributed fingers (which is satisfied by most finger protocols) will lead to exponentially fast creations of new healing points. Therefore, we conjecture that the above fast convergence process will lead to $O(\log N)$ convergence time for multi-ring topologies and

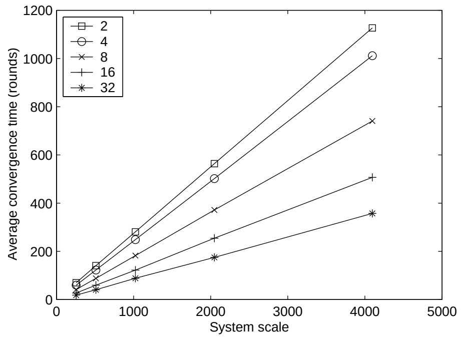
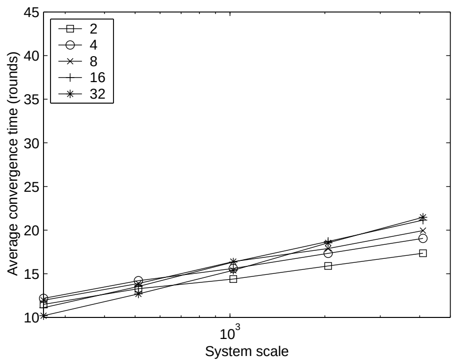
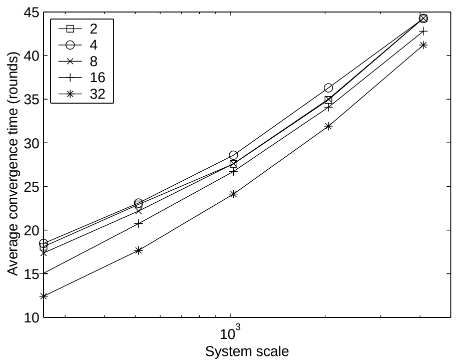


Figure 7: Convergence time vs. system scale without fast convergence algorithm on multi-ring topologies.



(a) With Chord finger



(b) With Symphony finger

Figure 8: Convergence time vs. system scale using fast convergence algorithm.

even other topologies. We verified the conjecture by simulations, while we plan to conduct a mathematical analysis to prove it.

The simulations are conducted by running our protocol with an initial multi-ring topology in an environment without node churns. Each multi-ring topology may be composed of 2, 4, 8, 16, and 32 rings. For each type of the multi-ring topology, we generate 100 instances for each of the following system scale (number of online nodes): 256, 512, 1024, 2048 and 4096, and take the average convergence time among these instances. We set the intervals of all periodic ping-sending timers in our protocol to be equal, so that we can use a single round number to measure the convergence time.

In Fig. 7, we show the convergence time before applying the fast convergence algorithm. The result indicates that the convergence time of multi-ring topologies increases linearly with the system scale. In Fig. 8, we show the convergence time on multi-ring topologies when the protocol uses the fast convergence algorithm with two types of fingers: the Chord fingers [2] and the Symphony fingers [15]. With both types of fingers, the convergence time is $O(\log N)$. Therefore, our simulation shows that our fast convergence mechanisms reduces convergence time to $O(\log N)$.

We also conducted simulations with random initial topologies, and the results show that even without the fast convergence algorithm random topologies converge in $O(\log N)$ time. This indicates that the convergence of multi-ring topologies are indeed more difficult than random topologies.

6.2 Fast loopy detection

Another special case we want to deal with is the loopy topology. Our current deloopy protocol (Part IV) may take $O(N)$ time to find a *deloopy point* — two different nodes that are both immediately preceding point 0 in the key space. To speed up loopy detection, we would like to use finger tables to forward the deloopy message faster. However, if a finger crosses point 0 in the key space, it may miss the deloopy point and forward the deloopy message back to the initiator. Therefore, we propose to use a special finger structure that we call *perfect skip list*, since it resembles a special centralized skip list data structure ([17]).

The finger table is a simple recursive structure. The i-th level finger on node x is calculated as follows: $x.\mathit{fingers}[0] = x.\mathit{succ}$, and $x.\mathit{fingers}[i+1] = (x.\mathit{fingers}[i]).\mathit{fingers}[i]$. We also need to know whether the i-th level finger crosses the point 0 in the key space. This is done by recursively calculating a boolean variable $\mathit{crossed}[i]$: $x.\mathit{crossed}[0] = (x.\mathit{succ}$ crossed $0)$, and $x.\mathit{crossed}[i+1] = (x.\mathit{crossed}[i]$ **or** $(x.\mathit{fingers}[i]).\mathit{crossed}[i])$. If the topology is in the loopy state, the correct $\mathit{fingers}$ and $\mathit{crossed}$ variables can be computed in parallel in $O(\log N)$ time.

The PING-DELOOPY messages are passed by a node along its highest level finger that does not cross point 0. We proved that if the topology is loopy, the above mechanism will find the deloopy point in $O(\log N)$ time. Once the deloopy point is found, we can use the parallel merging process described in Section 6.1 to quickly converge the topology into a correct ring structure.

7 Conclusion and Future Work

In this paper, we propose a formal specification of self-stabilizing structured overlays, and introduce a complete protocol that matches the specification. The protocol is able to preserve overlay connectivity in a purely peer-to-peer manner while maintaining a small leafset, and it is able to converge any connected topology to the correct configuration. We then consider the convergence speed of our protocol and provide algorithms to achieve $O(\log N)$ convergence time. Our protocol removes the limitations existed in previous protocols. Our future work includes theoretical analysis on our fast convergence algorithm, improvements to reduce cleanup time, and generalizations of our results to other structured overlay topologies.

References

[1] D. Angluin, J. Aspnes, and J. Chen. Fast construction of overlay networks. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, Las Vegas, Nevada, USA, July 2005.

[2] H. Balakrishnan, D. Karger, and D. Liben-Nowell. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, USA, July 2002.

[3] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the International Conference on Dependable Systems and Networks 2004*, Palazzo dei Congressi, Florence, Italy, June 2004.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), Oct. 2002.

[5] W. Chen and X. Liu. Enforcing routing consistency in structured peer-to-peer overlays: Should we and could we? In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, Santa Babara, California, USA, Feb. 2006.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[7] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[8] S. Dolev and R. I. Kat. Hypertree for self-stabilizing peer-to-peer systems. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications*, 2004.

[9] P. Druschel and A. Rowstron. PAST: Persistent and anonymous storage in a peer-to-peer networking enviroment. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Operation Systems*, Elmau/Oberbayern, Germany, May 2001.

[10] A. Haeberlen, J. Hoye, A. Mislove, and P. Druschel. Consistent key mapping in structured overlays. Technical Report TR05-456, Rice Computer Science Department, Aug. 2005.

[11] N. J. A. Harvey, M. B. Jones, S. Saroin, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, USA, Mar. 2003.

[12] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In *Proceedings of the 3rd International Workshop on Engineering Self-Organising Applications*, Utrecht, The Netherlands, July 2005.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, USA, Nov. 2000.

[14] X. Li, J. Misra, and C. G. Plaxton. Active and concurrent topology maintenance. In *Proceedings of the 18th International Symposium on Distributed Computing*, Trippenhuis, Amsterdam, the Netherlands, Oct. 2004.

[15] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, USA, Mar. 2003.

[16] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, Konstanz, Germany, Aug. 2005.

[17] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the SIGCOMM'01 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Deigo, California, Aug. 2001.

[19] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, Boston, Massachusetts, USA, June 2004.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, Heidelberg, Germany, Nov. 2001.

[21] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[22] A. Shaker and D. S. Reeves. Self-stabilizing structured ring topology p2p systems. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, Konstanz, Germany, Aug. 2005.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM'01 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Deigo, California, USA, Aug. 2001.

[24] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.