Decentralized, Connectivity-Preserving, and Cost-Effective Structured Overlay Maintenance

Yu Chen
Microsoft Research Asia
ychen@microsoft.com

Wei Chen
Microsoft Research Asia
weic@microsoft.com

Abstract

In this paper we present a rigorous treatment to structured overlay maintenance in decentralized peer-to-peer (P2P) systems subject to various system and network failures. we present a precise specification that requires the overlay maintenance protocols to be decentralized, preserve connectivity of the overlay, always converge to the desired structure whenever possible, and only maintain a small local state independent of the size of the system. We then provide a complete protocol with proof showing that it satisfies the specification. The protocol solves a number of subtle issues caused by decentralization and concurrency in the system. Our specification and the protocol overcomes a number of limitations of existing overlay maintenance protocols, such as the reliance on a centralized and continuously available bootstrap system, the assumption of a known system stabilization time, and the need to maintain large local membership lists.

Keywords: structured overlay maintenance, peer-to-peer, fault tolerance

MSR-TR-2007-84

1 Introduction

Since their introduction, structured overlays have been used as an important substrate for many peer-to-peer applications. 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 events, such as node joins, leaves and crashes, message delays and network partitions, affect overlay topology. Thus, an overlay topology should adjust itself appropriately to maintain its 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 a 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 vital for maintaining a correct overlay topology since finger tables can be constructed efficiently from the correct leafset tables. Therefore, our study focuses 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 [17, 19].

Leafset maintenance is a continuously running protocol that needs to deal with various system events. An important criterion for leafset maintenance is convergence. That is, the leafset topology can always converge back to the desired structure after the underlying system stabilizes (but without knowing about system stabilization), no matter how adverse the system events were before system stabilization.

In this paper, we provide a rigorous treatment to leafset convergence. Our contributions are mainly twofold. First, we provide a precise specification for leafset maintenance protocols with cost effectiveness requirements. All properties of the specification are desired by applications, while together they prohibit protocols with various limitations appeared in previous work (see Section 2 for a detailed comparison with previous work). Second, We provide a complete protocol that is proven to satisfy our specification.

There are several distinct features in our specification. First, our specification explicitly emphasizes connectivity preservation: the connectivity of the leafset topology may only be broken by adverse system events such as node crashes and network failures, but it should not be broken by the maintenance protocol itself. Some previous protocols such as Chord [11] and Pastry [17] allow runs in which the topology is broken due to protocol logic itself. Specifying the Connectivity Preservation property is not simple. We need to define a system stabilization time after which no adverse system events occur and require that the maintenance protocol no longer disconnect any nodes in the system afterwards. We dedicate a section to show that defining such a system stabilization time is subtle in that any time earlier will not guarantee connectivity perservation.

Second, we explicitly put requirements on cost effectiveness: the size of the local state maintained by the protocol in the steady state only depends on the size of its leafset table, but should not depend on the system's size. To be cost-effective, a protocol inevitably needs to remove some extra entries in the leafset (as in many existing protocols), but such removals may jeopardize the connectivity of the topology. Therefore, handling the apparent conflict between connectivity preservation and cost effectiveness is the key in our protocol design. Some existing protocols ([9, 13]) rely on the maintenance of a large membership list to preserve connectivity, and thus is not cost-effective.

Third, we explicitly address the how to heal topology partition by introducing an interface function $\mathtt{add}(contacts)$. Although the overlay could be more resistant to topology partition by maintaining more entries in the routing tables [11], network partitions are still inevitable, especially when failures on major network links happen. Therefore, we believe partition healing is an indispensable part of the protocol. The interface $\mathtt{add}(contacts)$ and its specification cleanly separates partition detection from partition healing: A separate mechanism may be

[1] The term leafset is originally used in Pastry [17] while the term finger is originally used in Chord [19].

used to detect topology partition, and then to call the add$(contacts)$ interface (only once) to bridge the partitioned components, while afterwards the maintenance protocol will automatically converge the topology. Our specification keeps the dependency on an external mechanism such as a bootstrap system at the minimum, while some previous protocols heavily rely on continuously available bootstrap systems to keep connectivity [6, 18].

In this paper, we provide a complete protocol and prove that it satisfies our specification. As indicated already, the core of the protocol is to handle the conflict between connectivity preservation and cost effectiveness: The protocol should remove extra entries in the leafset while preserving the topology's connectivity. The protocol addresses a couple of subtle issues: one is how to nullify the effects of adverse system events without knowing when the system stabilizes, and the second is to avoid livelocks that may be caused by inopportune invocations of the add$(contacts)$ interface. The correctness proof is technically involved and long, because our protocol needs to deal with system asynchrony and various system failures and events.

The correctness of our protocol is based on the availability of a dynamic failure detector that eventually can correctly detect failures of neighbors of a node, and we show that such a failure detector can be implemented in partially synchronous systems. One may argue that in peer-to-peer environments, such failure detectors or the partial synchrony assumption cannot be satisfied. We justify our model with the following reasons. First, studying the convergence behavior of a dynamic protocol under system failures is important to understand the correctness and the efficiency of the protocol, and to compare different protocols under the same condition. Such studies naturally assume a model in which system failures eventually stop, for which the paradigm of self stabilization is a direct example.[2] Second, the theoretical assumption that the system stabilizes after a certain time point means in practice a long enough stable period for the topology to converge. Based on our simulation study (Section 8) we show that with some optimizations the convergence speed of our protocol is fast ($O(\log N)$) where N is the number of nodes in the system), so system stabilization assumption may not seem so unreasonable in certain settings. Third, failure detection accuracy can be greatly improved if we consider voluntary leaves, in which a leaving node notifies its neighbors before leaving the system. Therefore, the failure detection requirement in the model may be more easily achieved for a sufficiently long time than considering only node crashes.

To our knowledge, our protocol is the first one that satisfies all the properties required by the specification with a complete correctness proof. We believe that our work could compensate many system-level studies on structured overlay maintenance and provide a more formal approach to study the correctness of overlay maintenance protocols.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines our system model with the failure detector specification. Section 4 introduces the complete specification of covergent leafset maintenance protocols. Section 5 justifies our definition of system stabilization time in our specification. Section 6 presents the complete leafset maintenance protocol. Section 7 provides a sample implementation of the failure detector in one partially synchronous model. Section 8 discusses optimizations for fast converging the topology and show simulation results. We conclude the paper and discuss future work in Section 9. Complete proofs of all technical results are included in the appendix.

2 Related Work

Many existing structured P2P overlay proposals mention that each node should have a leafset table. However, those such as Pastry [17], CAN [15], and SkipNet [8] only provide brief descriptions on what a correct leafset table looks like and how to fix it when the leafset table becomes incorrect because of 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 [16] and the latest Pastry improvements [2, 7] 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 convergence.

In [11], Liben-Nowell et. al. point out the topology maintenance issues of the original Chord proto-

[2]In Section 4 we will elaborate the relationship between our specification and self stabilization.

col [19] and propose an "idealize" process to adjust the immediate successor of each node to improve topology maintenance. This approach is essentially a method to guarantee convergence, but the maintenance restricts itself to immediate successor data structure. Although each node stores a successor list to handle successor failures, the node does not actively maintain the list. Instead, it uses the successor list of its immediate successor to overwrite its own one, and thus it could be disconnected from other nodes in its own list. Therefore, it is only a special case of our protocol, is less robust, and is difficult to accommodate partition healing, which requires maintaining multiple links together to bridge partitioned components.

Some recent work uses the approach of self stabilization [4, 5] to study overlay maintenance. The T-Man [9] and TChord [13] protocols are self-stabilizing, but they do not consider global membership changes from system churns. They require to keep an essentially full membership list on each node, so the maintenance cost increases significantly when the system is large or the membership changes over time. Authors of [6] and [18] also propose self-stabilizing overlay maintenance protocols. But their protocols and proofs depend on the existence of a *continuously available* bootstrap system. In [6], the bootstrap system needs to handle all join and repair requests, and needs to issue periodic broadcast messages for self stabilization purpose, while in [18] each node must periodically initiate look-ups to the bootstrap system. These protocols impose significant load and availability requirements on the bootstrap system. In contrast, our protocol only needs an external 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 are minimized.

Authors of Ranch [10] provide an overlay maintenance protocol with a 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. We believe silent failures must be considered in a wide area peer-to-peer environment, and the treatment of silent failures makes the model, the specification and the protocol design significantly depart from those studied in [10].

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 convergent 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 the adverse impacts of system conditions before system stabilization.

3 System Model

We consider a distributed peer-to-peer system consisting of nodes (peers) from the set $\Sigma = \{x_1, x_2, \ldots\}$. 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 have local clocks, which are used to generate increasing timestamps and periodic events on the nodes. Local clocks are not synchronized with one another. They provide an interface function getClockValue(), which is only required to return monotonically increasing time values on a node even if the node has failures between the calls to the function.

Nodes may join and leave the system or crash at any time. We treat a 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, and we refer to such an event as a *failure* in the system. We define a *membership pattern* Π as a function from time t to a finite and nonempty 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*. For the purpose of studying overlay convergence, we assume that the set of online nodes $\Pi(t)$ eventually stabilizes. That is, there is an unknown time t such that for all $t' \geq t$, $\Pi(t')$ remains the same, which we denote as $sset(\Pi)$. Let GST_N (N stands for nodes) be the *global stabilization time* of the nodes, which is the earliest time after which $\Pi(t)$ does not change any more. Henceforth, all specification properties refer to an arbitrary membership pattern Π.

Nodes communicate with one another by sending and receiving messages through asynchronous communication channels. We assume that there is a bidirectional channel between any pair of nodes. The channels cannot create or duplicate messages, but they might delay or drop messages. The channels are eventually reliable in the following sense: There exists an earliest time $GST_M \geq GST_N$ such that for any message m sent by $x \in sset(\Pi)$ to $y \in sset(\Pi)$ after time GST_M, m is eventually received by y.

To deal with failures in asynchronous environments, we assume the availability of a failure detector, which is a powerful abstraction that encapsulates all timing assumptions on message delays, processing speed, and local clock drifts [3]. Unlike the original model in [3], our failure detector is for dynamic environments, and we do not assume that the failure detector knows a priori a set of nodes to monitor. Instead, the failure detector provides an input interface $\mathsf{register}(S)$ for a node to register a set of nodes $S \subset \Sigma$ to be monitored by the failure detector. A node may invoke $\mathsf{register}(S)$ many times with a different set S to change the set to be monitored. The failure detector also provides an output interface $\mathsf{detected}(x)$ to notify a node that it detects the failure of a node $x \in \Sigma$.

Informally, the failure detector should eventually detect all failures among all registered nodes, and should eventually not make any wrong detections on nodes that are still online. More rigorously, the failure detector satisfies the following properties:

- *Strong Completeness*: For all $x \in sset(\Pi)$ and all $y \notin sset(\Pi)$, if x invokes $\mathsf{register}(S)$ with $y \in S$ at some time t, then there is a time $t' > t$ at which either the failure detector outputs $\mathsf{detected}(y)$ on x or x invokes $\mathsf{register}(S')$ with $y \notin S'$.

- *Eventual Strong Accuracy*: For all $x, y \in sset(\Pi)$, there is a time t such that for all $t' \geq t$, the failure detector will not output $\mathsf{detected}(y)$ on x at time t'.

Our failure detector differs from the eventually perfect failure detector $\diamond\mathcal{P}$ in the static environment [3] in that our failure detector relies on application inputs to learn the set of processes to monitor. We denote our failure detector as $\diamond\mathcal{P}_D$ (D stands for dynamic). In our protocol $\diamond\mathcal{P}_D$ is only used for each node to monitor its neighbors, so it is easier to achieve than $\diamond\mathcal{P}$ that requires monitoring all nodes in the system.

Every node in the system executes protocols by taking *steps* triggered by events, which include input events invoked by applications, message receipt events, periodic events generated by the local clock, and failure detection events $\mathsf{detected}()$. In each step, a node may change its local state, register with the failure detector, and send out a finite number of messages. For simplicity we assume that the time to execute a step is negligible, but a node might fail during the execution of a step. We also assume that there are only a finite number of steps taken during any finite time interval, and at each time point, there is at most one step taken by one node.[3]

A *run* of a leafset maintenance protocol is an infinite sequence of steps together with the increasing time points indicating when the steps occur, such that it conforms with the above assumptions on membership pattern, message delivery, and failure detection.

4 The Specification for Leafset Maintenance

We now specify the desired properties 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 converge onto the correct leafset, meaning x.*neighbors* $= \mathsf{leafset}(x, sset(\Pi))$, in which case the final topology resembles a ring structure.

[3]Our results also work if each step is not instantaneous or there are multiple concurrent steps at the same time, but it would make our description and proof more cumbersome to handle these situations.

Each node also has an interface function $\mathsf{add}(\mathit{contacts})$, where $\mathit{contacts}$ is 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. For any directed graph G, we say that 1) it is *strongly connected* if there is a directed path between any pair of nodes in G, 2) it is *weakly connected* (or simply *connected*) if there is an undirected path (when treating edges in G as undirected) between any pair of nodes in G, and 3) it is *disconnected* if it is not weakly connected. The *leafset topology* at time t is a directed graph $G(t) = \langle \Pi(t), E(t) \rangle$, where $E(t) = \{\langle x, y\rangle | x, y \in \Pi(t) \ \wedge \ y \in x.\mathit{neighbors}_t\}$. For any node $x \in \Pi(t)$, we denote $P_x(t)$ as the set of nodes in the *connected* subgraph of $G(t)$ that contains x; that is, $P_x(t)$ is the set of nodes that have undirected paths to x.

A key property we require on leafset maintenance is that the protocol should not break the connectivity of the topology. However, the topology might also be broken by underlying system behaviors out of protocol control, such as node failures and message delays. To factor out system-induced topology break-ups, we only require that the topology is not broken once the underlying system is stabilized. To do so, we first need to define the stabilization time of the system.

Let GST_D (D stands for detector) be the global stabilization time of the failure detector $\diamond\mathcal{P}_D$, which is the earliest time $t \geq \mathit{GST}_N$ such that $\diamond\mathcal{P}_D$ will not output $\mathsf{detected}(y)$ on any $x \in \mathit{sset}(\Pi)$ for any $y \in \mathit{sset}(\Pi)$ after time t. That is, GST_D is the earliest time after which the failure detector does not make wrong detections on online nodes any more. After GST_D, both the nodes and the failure detector stabilize, but nodes might still receive old messages sent before GST_D that may adversely affect the convergence of the topology. Thus, we define GST_S (S stands for system) to be the global stabilization time of the system, which is the earliest time $t \geq \max(\mathit{GST}_D, \mathit{GST}_M)$ such that all messages sent before GST_D or GST_M have been delivered by time t or are lost. Since there are only a finite number of messages that could have been sent before GST_D or GST_M, we know GST_S must be a finite value. Note that these stabilization times are defined for each run of the leafset maintenance protocol.

Our connectivity preservation property is defined based on GST_S as follows:

- *Connectivity Preservation*: For any $t \geq \mathit{GST}_S$, for any directed path from x to y in $G(t)$, for any time $t' > t$, there is a directed path from x to y in $G(t')$.

Connectivity Preservation is a key property to guarantee leafset convergence, but it is not explicitly addressed or enforced by previous protocols in a purely peer-to-peer environment. In Section 5, we show that condition $t \geq \mathit{GST}_S$ is necessary in the sense that no algorithm can guarantee connectivity preservation starting from a time earlier than GST_S.

By the Connectivity Preservation property, we know that the connected component $P_x(t)$ can only grow after time GST_S. Since $\Pi(t)$ does not change after GST_S and is finite, we know that $P_x(t)$ eventually stabilizes. The next property requires that the leafset of x eventually contains the correct leafset in the connected component of x:

- *Eventual Inclusion*: There is a time t such that for all $t' \geq t$ and for all $x \in \mathit{sset}(\Pi)$, $\mathsf{leafset}(x, x.\mathit{neighbors}_{t'}) = \mathsf{leafset}(x, P_x(t'))$.

If the topology becomes connected at some time after GST_S, then Eventual Inclusion together with Connectivity Preservation means that eventually $\mathsf{leafset}(x, x.\mathit{neighbors}_{t'}) = \mathsf{leafset}(x, \mathit{sset}(\Pi))$ for all $x \in \mathit{sset}(\Pi)$. The properties also imply that the weakly connected component $P_x(t)$ will become strongly connected eventually. Note that the Eventual Inclusion property should hold no matter if there are invocations of $\mathsf{add}()$ after GST_S.

If the topology is partitioned, the application should be able to use the $\mathsf{add}()$ interface to heal the partition. This is specified by the following property:

- *Partition Healing*: For any $x, y \in \mathit{sset}(\Pi)$, if there is an invocation of $\mathsf{add}(S)$ on x at time $t > \mathit{GST}_S$ with $y \in S$, then there is a time $t' > t$ such that x and y are connected in $G(t')$ (i.e., $P_x(t') = P_y(t')$).

The Partition Healing property ensures that only one invocation of $\mathsf{add}()$ on one node is necessary to bridge the partition, as long as we use an S that contains a node from every component in $\mathsf{add}(S)$. Afterwards, Eventual Inclusion and Connectivity

Preservation properties guarantee the autonomous convergence of the topology without any further help.

The following property requires that eventually the leafset maintenance protocol should only maintain the actual leafset entries, provided that the application eventually stops invoking $\mathsf{add}()$.

- *Eventual 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 $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors_{t''}) = x.neighbors_{t''}$.

We call a leafset maintenance protocol *convergent* if it satisfies Connectivity Preservation, Eventual Inclusion, Partition Healing, and Eventual Cleanup. If an external mechanism guarantees to call $\mathsf{add}()$ as described in Partition Healing, then the convergent protocol ensures that the topology is eventually connected and the leafset of every node is correct, i.e., $x.neighbors = \mathsf{leafset}(x, sset(\Pi))$.

One informative way to understand the specification is to see how it avoids a trivial implementation that always splits every node into a singleton, i.e., sets $x.neighbors$ to $\emptyset$ on every node x. This implementation would correctly satisfy the specification if there were no Partition Healing property. With Partition Healing, however, after GST_S the protocol is forced to reconnect nodes after $\mathsf{add}()$ invocations, and by Connectivity Preservation, the protocol has to keep these connections, and then by Eventual Inclusion and Eventual Cleanup, the protocol has to converge to a correct leafset structure. Thus trivially splitting nodes is prohibited by the specification.

Besides convergence, the leafset maintenance protocol should also be cost-effective in terms of the cost to maintain the *neighbors* set on the nodes. We look at the maintenance cost when the protocol reaches its *steady state*: that is, assuming that there is no more $\mathsf{add}()$ invoked at any node, the *neighbors* set of each online node has already included the correct leafset entries in its stabilized connected component and nothing more. The cost effectiveness is characterized 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 of the protocol, on each node the size of the local state and the number of nodes registered to the failure detector are both $O(L)$.

When counting the size, we assume that each node ID and each clock value take a constant number of bits to represent. The property specifies that in the steady state the local state and the number of nodes monitored by the failure detector on each node is linear to the size of the leafset and is not related to the system's size. The requirement of $O(L)$ nodes registered to the failure detector prevents a protocol from monitoring a large set of nodes in the steady state. The property also implies that in the steady state each node can only send messages to $O(L)$ nodes and the size of each message is at most $O(L)$.

Our specification of convergent overlay maintenance protocols is similar to self stabilization [4, 5] in that we require the leafset topology to eventually converge to the desired structure (each connected component is a ring structure) no matter what the topology was before the underlying system stabilizes. Our specification differs from self stabilization in the following aspects: First, we consider an open system where applications may invoke $\mathsf{add}()$ to add new contact nodes at any time, while self stabilization considers a closed system without any application interference. Second, unlike in the self stabilization model, we do not assume that all system states can be arbitrarily corrupted before system stabilization (e.g., local clock values cannot go backwards).

5 Necessity of GST_S

The Connectivity Preservation property requires the preservation of connectivity after time GST_S. One might wonder if we can find a protocol that preserves connectivity starting from an earlier time point. In particular, we know that after time $t = \max(GST_D, GST_M)$, both failure detection and message delivery are reliable. So can we guarantee connectivity preservation starting from time t? Why do we have to wait for all messages before t to be delivered to reach the time GST_S and then only guarantee connectivity preservation after GST_S?

The following theorem provides the justification for GST_S by showing that any convergent leafset maintenance protocol has a run in which the topology is connected right before GST_S but becomes disconnected after GST_S. This means no protocol can guarantee connectivity preservation starting from a time earlier than GST_S.

Theorem 1 *For any convergent leafset maintenance protocol A and any small real value $\epsilon > 0$, there exists a run in which G_t is weakly connected for some*

t such that $GST_S - \epsilon < t < GST_S$, but at a later time $t' \geq GST_S$, $G_{t'}$ is not weakly connected.

We now give an outline of the proof that covers the intuitive ideas, and move the complete proof to the appendix due to space constraints. For any protocol A, by our specification it has to guarantee convergence of each connected component to a single ring structure eventually. Then by the Partition Healing property, when two disconnected components are linked by some add() invocations, these components have to be merged into a single ring structure. This inevitably leads to some node u removing another node v from its *neighbors* set during the convergence process. Suppose this removal step on u is triggered by some message m received by u. If m is sent before GST_D, then it is possible that after m is sent, all processes other than u and v are crashed. Then v receives some wrong failure detection events claiming that u have crashed, which causes v to remove u from v.*neighbors*. However, u is not aware of these system events and when it receives m it still removes v from u.*neighbors*. This removal breaks u and v into two disconnected components. Therefore, any message sent before GST_D may potentially break the connectivity of the topology in any protocol. Hence, we can only ask a protocol to guarantee connectivity after these messages are delivered, which means after GST_S.

6 Leafset Maintenance Protocol

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 failure-handling protocol to remove the failed nodes from the leafset upon the notification of failure detector (Fig. 1, lines 11–12); (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); [4] 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.

```
On node $x$:
1  Data structure:
2    neighbors: set of nodes intended for
         leafset entries, initially $\emptyset$.
3  add($contacts$)
4    foreach $y \in contacts$
5      send PING-CONTACT to $y$
6  Upon receipt of PING-CONTACT from $y$:
7    send PONG-CONTACT to $y$
8  Upon receipt of PONG-CONTACT from $y$:
9    $neighbors \leftarrow neighbors \cup \{y\}$
10   register($neighbors$)
11 Upon detected($y$):
12   $neighbors \leftarrow neighbors \setminus \{y\}$
```

Figure 1: Leafset maintenance protocol, Part I: Add new contacts and handle failures.

All of these sub-protocols (except the failure-handling one) 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 keeps an invariant that a node y is added into x.*neighbors* only after x receives a pong message directly from y. This invariant verifies the liveness of any nodes to be added into the *neighbors* set and prevents different unwanted behaviors in different sub-protocols.

In the add() protocol, if the nodes were added directly into the *neighbors* set without any verification, the property *Eventual Inclusion* would not be satisfied because the application might keep inserting failed nodes via add(). To solve this problem, the add($contacts$) protocol (Fig. 1, lines 3–9) uses a ping-pong message loop to check the liveness of the nodes being added. In this way, the add() invoked after GST_N will not add any failed nodes into the *neighbors* set of any online nodes, since the failed nodes cannot respond to the PING-CONTACT messages.

The invite protocol (Fig. 2) uses a variable *cand* to store candidate nodes to be invited into the

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

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

Figure 2: Leafset maintenance protocol, Part II: Invite closer nodes in the key space.

neighbors set. The candidate nodes are discovered by exchanging local leafset views through the PING-ASK-INV and PONG-ASK-INV messages. Once a 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 invitation 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 31–32). The invite protocol is in principle similar to other leafset maintenance protocols (e.g. [19, 16, 9, 18]), except that we use PING-INVITE and PONG-INVITE messages to prevent a phenomenon called *ghost entry*. A *ghost entry* is an entry of a failed node that keeps bouncing among the *neighbors* sets of two or more online nodes, as explained below.

In the above example, suppose y is a failed node with ID adjacent to x and z. We also suppose y is still in z.*neighbors*. When x sends PING-ASK-INV message to z, z returns y. Without the message loop of PING-INVITE and PONG-INVITE, x would add y into x.*neighbors* directly. After z told x about y, its failure detector reports y's failure and y is removed from z.*neighbors*. Later z contacts x to find some nodes to be invited, and x returns y. So y is added back to z.*neighbors*. Then y could be removed from x.*neighbors* by a failure detector notification on x, and added back again by the PONG-ASK-INV message from z.

This process can repeat forever, making y bouncing back and forth between x.*neighbors* and z.*neighbors*. The *ghost entry* phenomenon violates the property of *Eventual Inclusion*. It could be eliminated by the PING-INVITE and PONG-INVITE message loop. With this message loop, a failed node will not be added into the *neighbors* set by the invitation protocol since it cannot send any PONG-INVITE messages. Therefore, it will not be returned to other nodes as an invitation candidate, either.

The replacement protocol (Fig. 3) is responsible for removing faraway nodes from the *neighbors* sets to keep *neighbors* sets small. This protocol is our key contribution to provide *Cost Effectiveness*, and the key differentiator from other protocols. When removing the faraway nodes, we need to ensure both safety (*Connectivity Preservation*) and liveness (*Eventual Inclusion* and *Eventual Cleanup*), 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 leafset(x, x.*neighbors*). Node x uses the PING-ASK-REPL and PONG-ASK-REPL loop (lines 39–46) with node z to obtain a replacement node y, which is recorded by x in x.*repl*[z]. (If there does not exist a node v satisfy the condition at line 43, y is set to $\perp$ and returned to x.) Then x uses the PING-REPLACE and PONG-REPLACE message loop to verify with y about the replacement (lines 47–61). 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 first ping-pong loop tries to find an alternative path to replace $\langle x, z\rangle$. The second ping-pong loop tries to ensure y's liveness and the validity of the path.

The above basic flow alone, however, cannot nul-

```
On node $x$:
35 Data structure:
36     repl[ ]: for each $z \in neighbors$, $repl[z]$ is a node
           to replace $z$, initially $\perp$
37     commit[ ]: for each $z \in neighbors$, $commit[z]$ is the
           time when $x$ commits to $z$ in a
           replacement task, initially 0
       /* repl[ ] and commit[ ] only maintains entries for nodes
           in neighbors */
38     ts: timestamp of the replacement task, initially 0
39 Repeat periodically:
40     foreach $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$
41         send PING-ASK-REPL to $z$
42 Upon receipt of PING-ASK-REPL from $z$:
43     $y \leftarrow v$ such that $v \in \mathsf{leafset}(x, neighbors)$
           and $d(z, v) < d(z, x)$ and
           $d(z, v) = \min_{u \in \mathsf{leafset}(x, neighbors)} d(z, u)$
44     send $(\text{PONG-ASK-REPL}, y)$ to $z$
45 Upon receipt of $(\text{PONG-ASK-REPL}, y)$ from $z$
46     if $z \in neighbors$ then $repl[z] \leftarrow y$
47 Repeat periodically:
48     $ts \leftarrow \mathsf{getClockValue}()$
49     foreach $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$
               and $repl[z] \neq \perp$
50         send $(\text{PING-REPLACE}, z, ts)$ to $repl[z]$
51 Upon receipt of $(\text{PING-REPLACE}, z, ts)$ from $y$:
52     if $z \in neighbors$ then
53         $commit[z] \leftarrow \mathsf{getClockValue}()$
54         send $(\text{PONG-REPLACE}, z, ts)$ to $y$
55 Upon receipt of $(\text{PONG-REPLACE}, z, ts)$ from $y$:
56     if $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$
               and $y = repl[z]$ then
57         $neighbors \leftarrow neighbors \cup \{y\}$
58         if $commit[z] < ts$ then
59             $neighbors \leftarrow neighbors \setminus \{z\}$
60             $commit[y] \leftarrow \mathsf{getClockValue}()$
61         register(neighbors)
```

Figure 3: Leafset maintenance protocol, Part III: Replace faraway nodes.

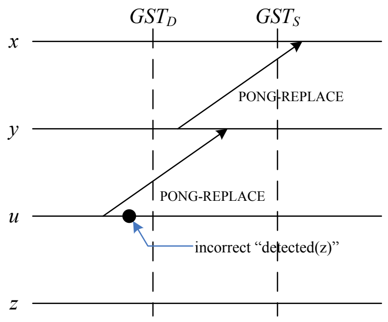


Figure 4: Concurrent replacement tasks introduce indirect effects of system conditions before GST_D and break topology connectivity.

lify the indirect effects of adverse system events before time GST_D 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 GST_S when it receives the PONG-REPLACE message sent by y after time GST_D. 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 and successfully replaces z with u. However, the time that u sends the PONG-REPLACE message to y could be before GST_D. So an erroneous "detected(z)" on u immediately after the sending of the message could remove z from $u.neighbors$. As the result, x is relying on the alternative path $x \rightarrow y \rightarrow u \rightarrow z$ to remove z from $x.neighbors$, but the path is broken since u removed z from $u.neighbors$. However, x is not aware of these concurrent events, and it still removes z after GST_S, which breaks the connectivity. This shows the indirect effect of adverse system events before GST_D. A similar danger exists when x tries to replace z and y concurrently.

We introduce variables *ts* and *commit*[ ] to eliminate these dangerous concurrent replacements. Variable *ts* is a timestamp identifying the current replacement task when a node sends out PING-REPLACE messages (line 48), and its value is piggybacked with the PING-REPLACE and PONG-REPLACE messages. For each $z \in x.neighbors$, variable $x.commit[z]$ records the time when x commits to z in a replacement task, either when x verifies the replacement of z for another node y (line 53), or when x uses z to replace another node y (line 60). The key condition is that x can only successfully replace z in a replacement task whose timestamp *ts* is higher than $commit[z]$ on x (line 58). The use of *ts* and *commit*[ ] variables avoids any dangerous concurrent replacement tasks in the system. In the example of Fig. 4, after y sends the PONG-REPLACE message to confirm the replacement of z for x, $y.commit[z]$ is updated to a new timestamp that is larger than the timestamp of y's own concurrent replacement task to z. So when y receives the PONG-REPLACE from u, it will not remove z from $y.neighbors$. As shown by our proof, it is the core mechanism to satisfy the *Connectivity Preservation* property.

Next, we restrict the selection of replacement node y to guarantee the *Eventual 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 43). The distance constraint avoids circular replacement, while the leafset constraint guarantees that y can successfully verify the replacement. The latter 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 *Eventual Cleanup* property.

The mechanisms introduced so far are not enough to guarantee the *Eventual Inclusion* property, however. During the proof of an earlier version of the protocol, we uncovered the following subtle livelock scenario in which the add() invocations interfere with leafset convergence. Whenever node x wants to replace z with y, the replacement is rejected because x just committed to z in a replacement task that replaces another node u with z. 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 continually 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 convergence 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 57), without checking the constraint of $z.commit < ts$. Next, x can remove z only when the condition $z.commit < ts$ holds (lines 58–59). 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 43. 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 still might be incorrect, because it can be in a special state called the *loopy state* as defined in [11]. 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 [11] 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.

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

Figure 5: Leafset maintenance protocol, Part IV: Loopy detection.

Our protocol is cost-effective because in the steady state each node only maintain sets *neighbors* and *cand*, mappings *repl*[ ] and *commit*[ ], which contain $O(L)$ number of nodes, and only nodes in the *neighbors* set are eventually registered with the failure detector.

Putting all sub-protocols together, we have a full protocol that satisfies all properties in our specification, as summarized by the following theorem.

Theorem 2 *The leafset maintenance protocol provided in Fig. 1, 2, 3, and 5 is both convergent and cost-effective, which means it satisfies the Connectivity Preservation, Partition Healing, Eventual Cleanup, Eventual Inclusion, and Cost Effectiveness properties.*

```
On node $x$:
Data structure:
  mset: set of nodes being monitored, initially $\emptyset$
  ts[ ]: for each $y \in$ mset, ts[$y$] is a timestamp for $y$.
register($S$):
  foreach $y \in$ mset $\setminus S$
    remove $y$'s entry from ts[ ]
  foreach $y \in S \setminus$ mset
    ts[$y$] $\leftarrow$ getClockValue()
  mset $\leftarrow S$
Repeat periodically with interval $I_p$:
  foreach $y \in$ mset, send PING-ALIVE to $y$
Upon receipt of PING-ALIVE from a node $y$:
  send PONG-ALIVE to $y$
Upon receipt of PONG-ALIVE from $y$
  if $y \in$ mset then
    ts[$y$] $\leftarrow$ getClockValue()
Repeat periodically with interval $I_c$:
  foreach $y \in$ mset
    if ts[$y$] $+ T_c <$ getClockValue() then
      output detected($y$)
      mset $\leftarrow$ mset $\setminus \{y\}$
      remove $y$'s entry from ts[ ]
```

Figure 6: Implementation of $\Diamond\mathcal{P}_D$.

7 Implementation of $\Diamond\mathcal{P}_D$

Failure detector $\Diamond\mathcal{P}_D$ could be implemented in a number of different partially synchronous systems. As an example, in this section we provide one implementation in a system model that guarantees message delays being bounded by a known constant Δ eventually. More precisely, we assume that local clocks are drift-free (our protocols can be easily adjusted to accommodate bounded clock 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. 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. We assume that there is an unknown time GST_M and a known constant bound Δ such that all messages sent at or after GST_M are Δ-timely.[5]

As shown in Figure 6, our implementation works in a heartbeat manner. The failure detector on a node x, periodically (with interval I_p) sends PING-ALIVE messages to nodes being monitored (in *mset*). When a node y receives a PING-ALIVE message, it immediately responds with a PONG-ALIVE message. Meanwhile, x also checks all the nodes in its *mset* periodically with interval I_c. If a node y enters *mset* for more than T_c time and also fails to send any PONG-ALIVE messages to x for at least T_c time, a failure notification about y will be reported.

It is obvious that our implementation will report failure to failed nodes eventually, since the failed nodes cannot send PONG-ALIVE messages any more. For the property of *Eventual Strong Accuracy*, we show that when the liveness checking interval I_c and the timeout threshhold T_c are set to be at least $I_p + 2\Delta$, the online nodes can return PONG-ALIVE messages on time, and thus no false failure notifications will be reported eventually. The proof of correctness is included in the appendix.

8 Improvement for Fast Convergence

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

Leafset topology convergence consists of two periods. The first period starts at the system stabilization time GST_S and ends when all *neighbors* sets on all nodes contain the correct leafset members, and it corresponds to the Eventual Inclusion 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 Eventual 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.

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 18), 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

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

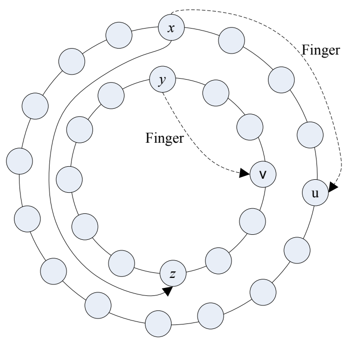


Figure 7: Multi-ring topology.

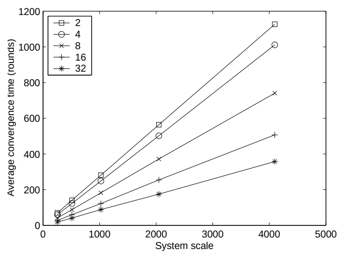


Figure 8: Convergence time vs. system scale without fast convergence algorithm on multi-ring topologies.

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.

8.1 Merging of the multi-ring topology

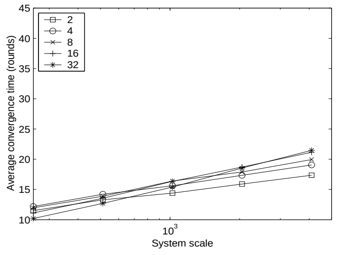
One special class of topologies is *multi-ring* topologies, in which several ring structures are connected by a few cross-ring links (Fig. 7). Multi-ring topologies can be generated due to network partitions. After a network partition, the topology may be broken into several disconnected components, all of which stabilize 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. 7, 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. 7. 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.

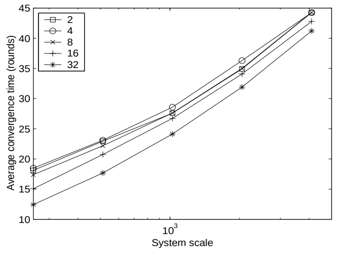
To speed up the creation of the first healing point, we allow node x to issue a special routing request starting from z using x's own ID as the routing key. With the help of fingers, the routing takes $O(\log N)$ steps to reach the routing destination y. So within $O(\log N)$ time, x can learn about 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 7, 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 even other topologies. We verified the conjecture by simulations, while we plan to conduct a mathematical analysis to prove it.



(a) With Chord finger



(b) With Symphony finger

Figure 9: Convergence time vs. system scale using fast convergence algorithm.

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. 8, 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. 9, 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 [11] and the Symphony fingers [12]. 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.

8.2 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 ([14]).

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 *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 *fingers* and *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 8.1 to quickly converge the topology into a correct ring structure.

9 Conclusions and Future Work

In this paper, we propose a formal specification of peer-to-peer structured overlay maintenance, 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 some heuristics to achieve $O(\log N)$ convergence time where N is the total number of nodes in the system. Our future work includes theoretical analysis of the fast convergence protocols. Another direction is to generalize 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] 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.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

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

[5] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. Liben-Nowell, H. Balakrishnan, and D. Karger. 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.

[12] 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.

[13] 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.

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

[15] 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 Diego, California, Aug. 2001.

[16] 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.

[17] 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.

[18] 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.

[19] 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 Diego, California, USA, Aug. 2001.

Appendix

A Proof of Theorem 1

Theorem 1 *For any convergent leafset maintenance protocol A and any small real value $\epsilon > 0$, there exists a run in which G_t is weakly connected for some t such that $GST_S - \epsilon < t < GST_S$, but at a later time $t' \geq GST_S$, $G_{t'}$ is not weakly connected.*
Proof. We construct a series runs of protocol A to reach the conclusion.

Run R_1: In this run, $GST_N = GST_D = GST_M = GST_S = 0$, i.e., there is no membership change, no message loss and no failure detection mistakes in the run. Let P_1 to be the finite set of online nodes with at least $2L + 1$ entries. At some time after $GST_N = 0$ we invoke $\mathsf{add}(P_1)$ on one of the nodes in P_1. By the Partition Healing property, if the leafset topology is not connected, it will become connected. By the Connectivity Perservation, Eventual Inclusion, and Eventual Cleanup properties, there exists a time t_1 such that any time $t > t_1$ and any node $x \in P_1$, $x.\mathit{neighbors}_t = \mathsf{leafset}(x, P_1)$.

Run R_2: R_2 is the same as R_1, except that the protocol now executes on another set of nodes P_2 with $P_2 \cap P_1 = \emptyset$, and an $\mathsf{add}(P_2)$ invocation on some node in P_2. So there exists a time t_2, for any time $t > t_2$ and any node $y \in P_2$, $y.\mathit{neighbors}_t = \mathsf{leafset}(y, P_2)$.

Run R_3: In this run, we still have $GST_N = GST_D = GST_M = GST_S = 0$. The run contains all the nodes in P_1 and P_2. Let $t_3 = \max(t_1, t_2)$. We merge the steps of R_1 and R_2 by time t_3 together as the steps of R_3 by time t_3. This means we delay all messages between P_1 and P_2, if there are any, and let the two components run alone. Thus we have that at time t_3, for any node $x \in P_1, y \in P_2$, $x.\mathit{neighbors}_{t_3} = \mathsf{leafset}(x, P_1)$, $y.\mathit{neighbors}_{t_3} = \mathsf{leafset}(y, P_2)$. In other words, the system topology is two disjoint "rings" at t_3.

At some time $t_4 > t_3$, we invoke $\mathsf{add}(\{y\})$ on x for two arbitrary nodes $x \in P_1$ and $y \in P_2$. Then we deliver the delayed messages between P_1 and P_2 after t_4. According to the Partition Healing property, the system topology eventually becomes weakly connected. By the other properties in the specification, we know that there exists a time $t_5 > t_4$ such that for any time $t > t_5$ and any node $z \in P_1 \cup P_2$, $z.\mathit{neighbors}_t = \mathsf{leafset}(z, P_1 \cup P_2)$. That is, the topology merges into a single ring after t_5.

This leads to a fact that there exists a node $u \in P = P_1 \cup P_2$ and for any two time points t, t' such that $t_3 < t < t_4$ and $t_5 < t'$, $u.\mathit{neighbors}_t \neq u.\mathit{neighbors}_{t'}$. Since both P_1 and P_2 contain at least $2L + 1$ nodes, it must be true that $|u.\mathit{neighbors}_t| = |u.\mathit{neighbors}_{t'}| = 2L$. Therefore, there must be some entry being removed from $u.\mathit{neighbors}$ after t_4. Let s be the first step after t_4 at which u removes v ($v \in u.\mathit{neighbors}_t$) from its *neighbors* set. It's obvious that s is not the first step in R_3, since the invocation of add is before it. Let s' be the step immediately preceeding s in run R_3, and t_c be the time when s' occurs. We know that G_{t_c} contains edge $\langle u, v \rangle$, and G_{t_c} is connected by our specification. Let σ' be the step sequence from the first step to step s'.

Run R_4: R_4 is exactly the same as R_3 until step s'. After step s', we make all the nodes in P offline except v. According to our specification, there exists a step s'' of v such that after step s'' $v.\mathit{neighbors}$ does not contain any nodes in $P \setminus \{v\}$. Let σ'' be the step sequence from the step after s' to step s''.

Run R_5: R_5 is exactly the same as R_3 until step s' at time t_c. After step s', we immediately make all the nodes in P offline except u and v. So GST_N is a value greater than t_c but smaller than the time when the first step in σ'' is taken. Then we simulate the execution in R_4 by letting v taking the steps in σ'' as in run R_4. After step s'', the last step in σ'', we schedule u to take step s at time t_f. After step s'', we also drop all messages that have not been delivered except the message that triggers step s (in the case s is triggered by a message). Thus the step sequence in R_5 until time t_f is $\sigma' \cdot \sigma'' \cdot s$. Figure 10 shows an example of the step sequence. Since u is online but does not take any step in sequence σ'', we know that for any time t with $GST_N \leq t < t_f$, G_t contains two nodes u and v and an edge $\langle u, v \rangle$, so G_t is connected. For any t with $t_c \leq t < GST_N$, G_t is connected because G_{t_c} is connected and there is no step after t_c and before GST_N. However, after step s'' $v.\mathit{neighbors}$ only contains v, and in step s at time t_f, u removes v from $u.\mathit{neighbors}$, so G_{t_f} is not connected.

Now we argue why we can schedule run R_5 as described above. First, for v to take the same steps in σ'' as in run R_4, v needs to have the same message receipt and failure detection events. It is easy to ensure the same message receipt events because it runs the same as in run R_4 until the last step s''

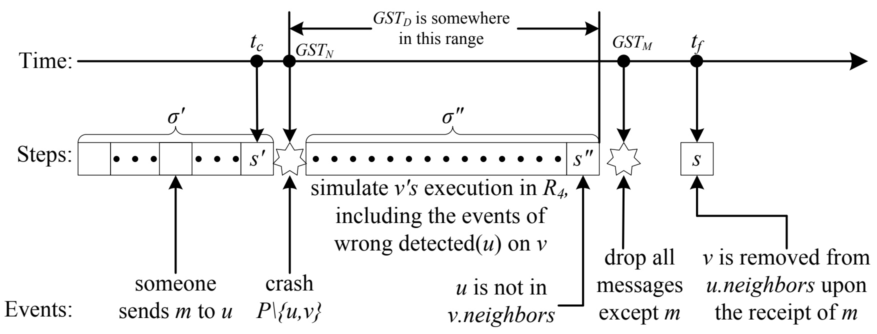

Time:
t_c
GST_N
GST_D is somewhere in this range
GST_M
t_f
Steps:
σ'
σ''
s'
s''
s
simulate v's execution in R_4, including the events of wrong detected(u) on v
Events:
someone sends m to u
crash $P\setminus\{u,v\}$
u is not in $v.neighbors$
drop all messages except m
v is removed from $u.neighbors$ upon the receipt of m

Figure 10: Run R_5 until step s that is triggerred by the receipt of a message m.

and node u does not take any step in σ''. For failure detection events, in σ'' it may have one or more $\mathsf{detected}(u)$ events on v since u is offline during the steps in σ'' in run R_4, but in R_5 u is online during this period. However, we can still have the same $\mathsf{detected}(u)$ events since the failure detector may make mistakes. Second, node u should be able to take step s as in run R_3. If step s is triggered by a message receipt event in R_3, the message is generated at some step in σ', so the same message exists in R_5, and we can delay its delivery to time t_f in the asynchornous model. If step s is triggered by a failure detection event $\mathsf{detected}(x)$ in R_3, then $x \notin P$ since $GST_D = 0$ in run R_3. In this case $\mathsf{detected}(x)$ can still be triggered at step s in run R_5. If step s is triggered by a periodic event of u's local clock in R_3, we can still trigger s with the same periodic event in R_5 since u's local clock has no timing guarantee and can be slowed down. Finally, step s cannot be triggered by an $\mathsf{add}()$ event in R_3 since in R_3 all $\mathsf{add}()$ events occur before step s. Therefore, run R_5 is a legitimate run in our model.

We now verify the time GST_S and conclude the proof. For GST_D and GST_M, by definition they are at least GST_N, and they are before t_f because there could be wrong $\mathsf{detected}(u)$ events during step sequence σ'', and there could be messages sent during σ'' that are lost, but after σ'', there is no more wrong detection and no more message loss. There are two cases to consider for GST_S. In the first case, step s is not triggered by a message receipt event. In this case, we know that GST_S is before time t_f, because after step s'' and before t_f, we know that all messages sent before GST_D or GST_M either have been received or dropped. Since we know that for all t with $t_c \leq t < t_f$, G_t is connected but G_{t_f} is not connected, and GST_S is such that $t_c < GST_N \leq GST_S < t_f$, so we can certainly find a t such that $GST_S - \epsilon < t < GST_S$ and G_t is connected, and $t' = t_f > GST_S$ with $G_{t'}$ not connected.

In the second case, step s is triggered by the receipt of a message m. In this case, $GST_S = t_f$, because by time t_f, every message sent before GST_D or GST_M is either received before t_f, or dropped, or is message m received at time t_f. Again since the topology was connected before t_f but disconnected after step s at t_f, we can certainly find t with $GST_S - \epsilon < t < GST_S$ and G_t is connected, and find a time $t' = GST_S$ such that $G_{t'}$ is not connected.

Therefore, R_5 is a run satisfying the theorem. □

B 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, $(x, y) = \{z \in \mathcal{K} : 0 < d^+(x, z) < d^+(x, y)\}$.

Lemma 1 *After time GST_D, for all node $x \in sset(\Pi)$, x will not remove any online nodes in $sset(\Pi)$ from its neighbors set by failure detection.*

Proof. According to the property of *Eventual Strong Accuracy* of $\diamond\mathcal{P}_D$, line 12 will never be invoked on x for a online node y. □

Since the liveness checking protocol (Figure 1) will not remove any online nodes from the *neighbors* set after time GST_D, 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 39–46 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 51–54 *v's verification of the replacement of w for node u*, and the step that successfully executes lines 55–61 *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 *global* 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 57) and *commit*$[v]$ on u is set to the local time of u (line 60). So t^1_{rp} is the corresponding global time of u's local time *commit*$[v]$ (i.e. $t^1_{rp} = \mathsf{globalTime}(u, \mathit{commit}[v])$). At time t^2_{rp}, u replaces v with r (lines 57 and 59). According to the algorithm, this implies that *commit*$[v] <$ *ts* at time t^2_{rp}, where *ts* is the timestamp variable embedded in the PONG-REPLACE message from r (line 58). Now let $t = \mathsf{globalTime}(u, \mathit{ts})$. We have $t > t^1_{rp}$. 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, $t^2_{vf} > t > t^1_{rp}$.

Proof of 2). At time t^1_{vf}, v verifies the replacement of w so *commit*$[w]$ on v is set to the local time of v (line 53). In other words, $t^1_{vf} = \mathsf{globalTime}(v, \mathit{commit}[w]$. At a later time t^3_{rp}, v replaces w with s, which implies that the timestamp *ts* embedded in the PONG-REPLACE message from s to v satisfies *commit*$[w] <$ *ts* (line 58). Let $t = \mathsf{globalTime}(v, \mathit{ts})$, we have $t > t^1_{vf}$. Since the corresponding PING-REPLACE message is sent at t and t^3_{vf} is the time at which this message is received by s, we have $t^3_{vf} > t > t^1_{vf}$. □

Lemma 3 *For a replacement sequence $R = (u, v, w, t_{vf}, t_{rp})$ with $w \in sset(\Pi)$, if $t_{vf} \geq GST_D$, 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 $t_{vf} \geq GST_D$, 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_σ, where G_σ 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 51–54) in the step sequence σ, or v_i replaces some node x with v_{i+1} (lines 55–61) 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 $t_{vf} \geq GST_D \geq GST_N$, 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_σ. 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'. By the induction hypothesis, there is a path $p = (v_0 = v, v_1, \ldots, v_m = w)$ from v to w in G_σ. 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* as a failure handling on v_i (line 12) for some $i < m$; or (b) s_k is the replacement of v_{i+1} with some node z in v_i.*neighbors* (lines 55–61), 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 GST_D. 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 (PING-REPLACE, v_{i+1}, *ts*) message from z, where *ts* is v_i's local time at which the corresponding PING-REPLACE message was sent. According to the algorithm, we know that before step s_k, $v_i.v_{i+1}.commit < ts$ (line 58), and after step s_k, $t'_1 = \mathsf{globalTime}(v_i, v_i.z.commit)$ (v_i set $v_i.z.commit$ to its local time at global time t'_1 at line 60). 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 (PING-REPLACE, v_{i+1}, *ts*) message from v_i at step s. This implies that during the entire execution of σ_1, v_i's local time is greater than *ts*.

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 $G_{\sigma'}$, 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 $v_i.v_{i+1}.commit = ts' > ts$, which contradicts to our earlier conclusion that $v_i.v_{i+1}.commit < ts$ before s_k. In the second case, v_i replaces some node x with v_{i+1} in σ_1. By the algorithm, we still have $v_i.v_{i+1}.commit = ts' > ts$, again contradicting with our conclusion that $v_i.v_{i+1}.commit < ts$ 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 $G_{\sigma'}$, but instead we have $\langle v_i, z\rangle$, and a path p_1 from z to v_{i+1} in $G_{\sigma'}$. 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 $G_{\sigma'}$ 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 $t \geq GST_S$, 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 $t \geq GST_S$, then v verifies this replacement at or after time GST_D. The corollary then follows directly from Lemma 3. □

Lemma 5 (Connectivity Preservation) *If for some time $t \geq GST_S$ there is a directed path from x to y in $G(t)$, then for all time $t' > t$, there is also a directed path from x to y in $G(t')$.*

Proof. At time t, suppose the direct path from x to y is $p = (v_0 = x, v_1, \ldots, v_m = y$. By the definition of G_t, we know $v_i \in sset(\Pi)$ for all i. Lemma 1 shows that no edges in $G(t')$ for any $t' \geq GST_S$ can be removed by the failure handling protocol. Corollary 4 shows that after time GST_S, any replacement that removes an edge $\langle v_i, v_{i+1}\rangle$ still keeps a path from v_i to v_{i+1}, for all $i < m$.

Therefore, we can construct a directed path p' from x to y in $G(t')$ in the following way: (1) if $\langle v_i, v_{i+1}\rangle$ is still in $G(t')$, we keep the edge; (2) if $\langle v_i, v_{i+1}\rangle$ is not in $G(t')$, there must be a path p_i in $G(t')$ from v_i to v_{i+1}, and we use p_i to replace $\langle v_i, v_{i+1}\rangle$. □.

Corollary 6 *If for some time $t \geq GST_S$, P is a weakly connected component of $G(t)$, then for all time $t' > t$, nodes in P are still weakly connected in $G(t')$.*

Proof. According to Lemma 5, every directed path in P is preserved for all time $t' > t$. Since every undirected path in P is the concatenation of several directed path in P, the undirected paths in P are also preserved. Therefore, nodes in P are still weakly connected in $G(t')$. $\square$.

Lemma 7 (Partition Healing) *For any $x, y \in sset(\Pi)$, if there is an invocation of* $\mathsf{add}(S)$ *on x at time $t > GST_S$ with $y \in S$, then there is a time $t' > t$ such that x and y are connected in $G(t')$ (i.e., $P_x(t') = P_y(t')$).*

Proof. At time t, let the component containing x to be $P_x(t)$ and the component containing y to be $P_y(t)$. If $P_x(t) = P_y(t)$, x and y are always connected at any time after t, according to Corollary 6. Suppose $P_x(t) \neq P_y(t)$. During the invocation of $\mathsf{add}(S)$ on x at time t, x sends a PING-CONTACT message to y. Since $y \in sset(\Pi)$ and $t > GST_S$, y eventually receives the PING-CONTACT message and reply with PONG-CONTACT messages to x. When x receives the PONG-CONTACT message from y at t', it adds y into $x.neighbors$ (line 9), and thus x and y become connected in $G(t')$. $\square$.

Lemma 8 *There exists $t_1 > GST_S$, such that $\forall t > t_1, \forall x \in sset(\Pi), x.neighbors_t \subseteq sset(\Pi)$.*

Proof. After GST_S, 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-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 GST_S.

Let $y \in x.neighbors_{GST_S}$ for arbitrary nodes $x \in sset(\Pi)$ and $y \notin sset(\Pi)$. In our protocol, there are only two ways for y to be removed from $x.neighbors$: (1) y is removed by the replacement protocol. (2) y is removed by the failure handling protocol.

If y is removed at some time $t_{x,y} > GST_S$ by the replacement protocol, it is not possible for y to be added back later. If the replacement protocol does not remove y, y will be always monitored by the failure detector on x, because x always ask the failure detector to monitor the whole *neighbors* set. According to the "Strong Completeness" property of $\diamond\mathcal{P}_D$, there exists a time $t_{x,y} > GST_S$ at which the failure detector will output $\mathsf{detected}(y)$ on x, thus removing y from $x.neighbors$ (line 12). Therefore, there always exists a time $t_{x,y}$, at which y is removed from $x.neighbors$ by either the replacement protocol or the failure handling protocol.

Let $t_1 = \max\{t_{x,y}\}$ for all $x \in sset(\Pi)$, $y \notin sset(\Pi)$, and $y \in x.neighbors_{GST_S}$. We have, for all $t > t_1$, for all $x \in sset(\Pi)$, $x.neighbors_t \subseteq sset(\Pi)$. $\square$

From now on, Let t_1 be as defined in Lemma 8.

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 9 *There exists $t_2 \geq t_1$, such that for all $t, t' > t_2$ and for all $x \in sset(\Pi)$,* $\mathsf{leafset}(x, x.neighbors_t) = \mathsf{leafset}(x, x.neighbors_{t'})$

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

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

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

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

According to Lemmata 1 and 8, 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 56), 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 8 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. $\square$

From now on, let t_2 be as defined in Lemma 9.

Corollary 10 *After time t_2, the invite protocol (Part II) will not add any node into the neighbors set of any online node in line 32.*

Proof. It is clear that every execution of line 32 that adds a node y into $x.\textit{neighbors}$ changes the leafset of node x, so following Lemma 9, no online nodes executes line 32 to add another node into $x.\textit{neighbors}$ after time t_2. □

Lemma 11 $\forall t > t_2, y \in \mathsf{leafset}(x, x.\textit{neighbors}_t)$ *if and only if* $x \in \mathsf{leafset}(y, y.\textit{neighbors}_t)$.

Proof. Suppose $\exists x, y \in \textit{sset}(\Pi), \exists t > t_2$, $y \in \mathsf{leafset}(x, x.\textit{neighbors}_t)$ but $x \notin \mathsf{leafset}(y, y.\textit{neighbors}_t)$. According to Lemma 9, $\mathsf{leafset}(x, x.\textit{neighbors}_t)$ and $\mathsf{leafset}(y, y.\textit{neighbors}_t)$ will not change any more after t_2. So we can use $\mathsf{leafset}(x, x.\textit{neighbors})$ and $\mathsf{leafset}(y, y.\textit{neighbors})$ to refer to x and y's leafset after time t_2. Since $y \in \mathsf{leafset}(x, x.\textit{neighbors})$, x will send PING-ASK-INV to y sometime after t (line 16). When y receives this message at some time $t' > t$, y adds x into $y.\textit{cand}$ (line 20).

Consider y's leafset $\mathsf{leafset}(y, y.\textit{neighbors})$, and it is represented as $\{y_{-1}, y_{-2}, \ldots\}$. If $|\mathsf{leafset}(y, y.\textit{neighbors})| < L$ or $d^-(y, x) < d^-(y, y_{-L})$, then $x \notin y.\textit{neighbors}$, because otherwise x is in $\mathsf{leafset}(y, y.\textit{neighbors})$. In this case, at the next time after t' when y invites closer nodes (lines 23–27), we have $x \in y.\textit{cand} \setminus y.\textit{neighbors}$ and $x \in \mathsf{leafset}(y, y.\textit{cand} \cup y.\textit{neighbors})$, so y sends a PING-INVITE message to x (line 26). Node x will respond to y with a PONG-INVITE message. When y receives this PONG-INVITE, we have $x \in \mathsf{leafset}(y, y.\textit{neighbors} \cup \{x\})$, so y adds x into $y.\textit{neighbors}$, and thus $\mathsf{leafset}(y, y.\textit{neighbors})$ now include x, contradicting to Lemma 9 stating that no leafset changes after time t_2. Therefore, we know that $|\mathsf{leafset}(y, y.\textit{neighbors})| \geq L$ 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 $y \in \mathsf{leafset}(x, x.\textit{neighbors})$, 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 $x.\textit{neighbors}$ in interval I since $y = x_{+j}$, but there are at least L nodes $\{y_{-1}, y_{-2}, \ldots, y_{-L}\}$ in $y.\textit{neighbors}$ in interval I.

Therefore, when y receives the PING-ASK-INV message from x at time t', the $\textit{view}$ that y calculates for x in line 18, which is $\mathsf{leafset}(x, y.\textit{neighbors})$, includes at least L nodes in interval I. Therefore there exists at least one node $z \in \textit{view} \setminus x.\textit{neighbors}$. Node y sends this $\textit{view}$ to x (line 19). When x receives the view, it add it into $x.\textit{cand}$ (line 22). Next time when x invites closer nodes (lines 23–27), we have $z \in x.\textit{cand} \setminus x.\textit{neighbors}$ and $z \in \mathsf{leafset}(x, x.\textit{cand} \cup x.\textit{neighbors})$, because $y \in \mathsf{leafset}(x, x.\textit{neighbors})$ and $d^+(x, z) < d^+(x, y)$. So x sends a PING-INVITE message to z (line 26). Since $z \in y.\textit{neighbors}$, by Lemma 8, $z \in \textit{sset}(\Pi)$. 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 $x.\textit{neighbors}$ (line 32), unless z is already in $x.\textit{neighbors}$. In either case, the leafset of x changes, contradicting to Lemma 9. □

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

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 $d^+(v_0, 0) < d^+(v_0, v_0.\textit{succ}_t)$; and (b) on the sequence $v_0, v_1, v_2, \ldots$ with $v_{i+1} = v_i.\textit{succ}_t$, there exists a node $v_j \neq v_0$ such that $d^+(v_j, 0) < d^+(v_j, v_j.\textit{succ}_t)$.

Lemma 12 *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 71) after time t'.*

Proof. Suppose, for a contradiction, that at time $t > t_2$ graph $G(t)$ is loopy. By Lemma 9 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 $x.\textit{succ}$

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 66). By Lemma 8, 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 73), until the message reaches v_j. On v_j, the loopy detection condition on line 69 is true, so v_j adds v_0 into $v_j.cand$ (line 70) and send a PONG-DELOOPY message back to v_0 (line 71). When v_0 receives this message, it adds v_j into $v_0.cand$ (line 75).

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 26), which will lead to the addition of v_0 into v_j's leafset, contradicting to Lemma 9 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 66 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 13 *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 no invocation of add() *or delivery of* PONG-CONTACT *message happens at or after time t. 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 9, 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 59. 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 57, 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 8, 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 11, 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 20). 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 9.

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 43). 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 $\mathsf{leafset}(z, z.\textit{neighbors})$, z will always provide the same y to x after t_2. When x receives this message, it sets $x.\textit{repl}[z]$ to y.

Then x will send $(\text{PING-REPLACE}, z, \textit{rnd})$ to y (line 50). By Lemma 11, $y \in \mathsf{leafset}(z, z.\textit{neighbors})$ implies that $z \in \mathsf{leafset}(y, y.\textit{neighbors})$. So when y receives the $(\text{PING-REPLACE}, z, \textit{rnd})$ from x, y sends $(\text{PONG-REPLACE}, z, \textit{rnd})$ back to x (line 54). When x receives this message from x at a time $t' > t$, we know that $z \in x.\textit{neighbors}_{t'} \setminus \mathsf{leafset}(x, x.\textit{neighbors})$ and $y = x.\textit{repl}[z]$. Therefore, the condition in line 56 holds and x adds y into $x.\textit{neighbors}$ (line 57). Since we have $d(x, y) < d(x, z)$ and $y \in \mathsf{leafset}(z, z.\textit{neighbors})$, Part (1) of the lemma holds.

We now prove Part (2). We continue the proof in Part (1) with the added assumption that $d(x, z) = \max\{d(u, w) : w \in u.\textit{neighbors}_t \setminus \mathsf{leafset}(u, u.\textit{neighbors}_t)\}$, 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 58 will become true and thus the replacement will succeed eventually.

If the condition in line 58 is not true, then after x sends out the $(\text{PING-REPLACE}, z, \textit{ts})$ to y, either x verifies the replacement of z for another node u (line 53), or x replaces another node u with z (line 60).

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

For the second case, it implies $d(x, z) < d(x, u)$ and $u \in x.\textit{neighbors}_{t'} \setminus \mathsf{leafset}(x, x.\textit{neighbors})$ for some time $t' > t$. According to the choice of x and z, we conclude that u cannot be in $x.\textit{neighbors}_t$. Therefore u is added after time t but before time t'. From Corollary 10, no nodes are added into $x.\textit{neighbors}$ after time t_2 by the invite protocol. By our condition, no invocation of $\mathsf{add}()$ or delivery of PONG-CONTACT message happens at or after time t so no node is added into $x.\textit{neighbors}$ at or after time t due to the receipt of PONG-CONTACT message. So the only place that x can add u into $x.\textit{neighbors}$ after time t is in the replacement protocol (line 57). If so, it means there is yet another node v such that $d(x, u) < d(x, v)$ and $v \in x.\textit{neighbors}_{t''} \setminus \mathsf{leafset}(x, x.\textit{neighbors})$ and $t < t'' < t'$. However, we know that v cannot be in $x.\textit{neighbors}_t$ either, so v is added into $x.\textit{neighbors}$ 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 $x.\textit{commit}[z]$ will not increase due to x's own replacement of some other nodes.

Since $x.\textit{commit}[z]$ stops increase after $\max(\tau', t)$, we can always find a time $t'' \geq \max(\tau', t)$ at which $x.\mathsf{getClockValue}() \geq x.\textit{commit}[z]$ and x sends PING-REPLACE to a node $y \in \mathsf{leafset}(z, z.\textit{neighbors}_{t''})$ and $d(x, y) < d(x, z)$. After x gets the corresponding PONG-REPLACE from y, the condition in line 58 must be true and x replaces z with y. □

It is reasonable to assume that applications can only invoke finite number of $\mathsf{add}()$ in a fixed-length time interval. So if applications stop invoking $\mathsf{add}()$, the delivery of PONG-CONTACT message will also stop some time later. After applications stop interfering with our protocol, the size of each node's *neighbors* set eventually become less than or equal to $2L$.

Lemma 14 (Eventual Cleanup) *If there is a time t after which no $\mathsf{add}()$ is invoked at any node in the system and no delivery of* PONG-CONTACT *message happens, then there is a time t' such that for all time $t'' \geq t'$ and all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.\textit{neighbors}_{t''}) = x.\textit{neighbors}_{t''}$.*

Proof. Let t'_1 be $\max(t_2, t)$. Given any time τ, we define a metric $m(\tau) = \max\{d(x, z) : z \in x.\textit{neighbors}_\tau \setminus \mathsf{leafset}(x, x.\textit{neighbors}_\tau), x, z \in sset(\Pi)\}$ if there exists x and z such that $z \in x.\textit{neighbors}_\tau \setminus \mathsf{leafset}(x, x.\textit{neighbors}_\tau)$, 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'_1, we know that no node is added into any *neighbors* set due to the invocation of $\mathsf{add}()$, no node is added by the invite protocol (Corollary 10), and no node is removed by the liveness check protocol (Lemmata 1 and 8). Therefore, a node may only be added or removed by the replacement protocol. If a node y is added into $x.\mathit{neighbors}$ by the replacement protocol in line 57 at a time τ, there must be a node z such that $z \in x.\mathit{neighbors}_\tau \setminus \mathsf{leafset}(x, x.\mathit{neighbors}_\tau)$ (line 56) and $d(x, y) < d(x, z)$ (line 43). 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 $z \in x.\mathit{neighbors}_\tau \setminus \mathsf{leafset}(x, x.\mathit{neighbors}_\tau)$ and $d(x, z) = m(\tau)$. By Lemma 13 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 13 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 $x \in sset(\Pi)$, $x.\mathit{neighbors}_{t''} = \mathsf{leafset}(x, x.\mathit{neighbors}_{t''})$ for all time $t'' > t'$. □

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

Lemma 15 *1). $\forall t, t' > t_2$, $x.\mathit{succ}_t = x.\mathit{succ}_{t'}$ and $x.\mathit{pred}_t = x.\mathit{pred}_{t'}$.*
2). $\forall t > t_2$, $x.\mathit{succ}_t = y$ is equivalent to $y.\mathit{pred}_t = x$.
3). Suppose $x.\mathit{neighbors}_t \neq \emptyset$. $\forall t > t_2$, $\forall x \in sset(\Pi)$, consider the nodes $\{x_{+1}, x_{+2}, \ldots, x_{+M}\}$ in $x.\mathit{neighbors}^+$ and $\{x_{-1}, x_{-2}, \ldots, x_{-M}\}$ in $x.\mathit{neighbors}^-$ with $M = \min(L, |x.\mathit{neighbors}|)$. We have $x_{+i}.\mathit{succ}_t = x_{+(i+1)} \wedge x_{-i}.\mathit{pred}_t = x_{-(i+1)}, 1 \le i \le M - 1$.

Proof.

Proof of 1). It is immediate from Lemma 9 and the fact that $x.\mathit{succ}_t$ and $x.\mathit{pred}_t$ is only determined by $\mathsf{leafset}(x, x.\mathit{neighbors}_t)$.

Proof of 2). The statement is trivially true when $x.\mathit{neighbors}_t = \emptyset$, so we consider $x.\mathit{neighbors}_t \neq \emptyset$. From $x.\mathit{succ}_t = y$, we know that $y \in \mathsf{leafset}(x, x.\mathit{neighbors}_t)$. By Lemma 11 it must be true that $x \in \mathsf{leafset}(y, y.\mathit{neighbors}_t)$. Suppose $y.\mathit{pred}_t = z \neq x$, 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 $\mathsf{leafset}(x, x.\mathit{neighbors})$ 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 $\mathsf{leafset}(x, y.\mathit{neighbors})$. This implies that y will put a node $u \in y.\mathit{neighbors}$ with $d^+(x, u) \le d^+(x, z)$ in the *view* variable in the acknowledged PONG-ASK-INV to x, thus allowing u to enter $x.\mathit{cand}$. Since $d^+(x, u) < d^+(x, y)$, next time when x invites closer nodes (lines 23–27), x must send a PING-INVITE to some node v such that $d^+(x, v) \le d^+(x, u) < d^+(x, y)$ 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 $v \in \mathsf{leafset}(x, x.\mathit{neighbors} \cup \{v\})$, unless v is already in $x.\mathit{neighbors}$ by that time. In either case, it contradicts to Lemma 9 saying that the leafset never changes. Therefore, we have $y.\mathit{pred}_t = x$.

We can also prove that $y.\mathit{pred}_t = x$ implies $x.\mathit{succ}_t = y$ in a similar way.

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

Suppose $d^+(x_{+j}, z) > d^+(x_{+j}, x_{+(j+1)})$. Because $x_{+j} \in \mathsf{leafset}(x, x.\mathit{neighbors}_t)$ implies $x \in \mathsf{leafset}(x_{+j}, x_{+j}.\mathit{neighbors}_t)$ and $\mathsf{leafset}(x_{+j}, x_{+j}.\mathit{neighbors})$ 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 $x_{+j} \in \mathsf{leafset}(x, x.\mathit{neighbors}_t)$ and $\mathsf{leafset}(x, x.\mathit{neighbors})$ 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

$\mathsf{leafset}(x, x.\textit{neighbors})$, again a contradiction.

So it must be true that $x_{+i}.\textit{succ}_t = x_{+(i+1)}, 1 \le i \le M-1$. $x_{-i}.\textit{pred}_t = x_{-(i+1)}, 1 \le i \le M-1$ could be proved in the similar way. □

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

Corollary 16 *For any node $x \in \textit{sset}(\Pi)$, $\rho(x)$ exists, is unique, and is a closed loop. Moreover, for all $y \in \rho(x)$, $\mathsf{leafset}(y, y.\textit{neighbors}_t) \subseteq \rho(x)$ for all time $t > t_2$.*

Proof. Closed sequence $\rho(x)$ exists because $\textit{sset}(\Pi)$ is a finite set. It is unique because the $x.\textit{succ}$ variable has a unique value. Suppose $x_k.\textit{succ} = x_j \neq x_0$ with $0 < j \le k$. If $x_k.\textit{succ} = x_k$, $x_k.\textit{neighbors} \setminus \{x_k\}$ must be empty according to the definition of helper function succ. However, by Lemma 15 2) we have $x_k.\textit{pred} = x_{k-1} \neq x_k$, which means that $x_k.\textit{neighbors} \setminus \{x_k\}$ is not empty. This is contradictory. If $x_k.\textit{succ} = x_j (j < k)$, then according to Lemma 15 2), $x_k = x_j.\textit{pred}$. Since by definition we also have $x_{j-1} = x_j.\textit{pred}$, so $x_{j-1} = x_k$, still a contradiction. Therefore, $x_k.\textit{succ} = x_0$ and $\rho(x)$ is a closed loop. By Lemma 15 3), it is straightforward to see that for all $y \in \rho(x)$, $\mathsf{leafset}(y, y.\textit{neighbors}_t) \subseteq \rho(x)$ for all time $t > t_2$. □

Lemma 17 *For all $t > t_2$, for any node $x \in \textit{sset}(\Pi)$, $\rho(x) = P_x(t)$. $P_x(t)$ is the weakly connected component containing x in $G(t)$.*

Proof. Suppose, for a contradiction that there exists $x, z \in P_x(t)$ such that $z \in P_x(t) \setminus \rho(x)$. By definition of $P_x(t)$, there is a path from x to z when treating edges in $P_x(t)$ 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 $P_x(t)$ or $\langle z_0, x_0\rangle$ is in $P_x(t)$.

Consider the first case where $\langle x_0, z_0\rangle$ is in $P_x(t)$. By Corollary 16, $z_0 \in x_0.\textit{neighbors}_t \setminus \mathsf{leafset}(x_0, x_0.\textit{neighbors}_t)$. By Lemma 13 1), there is a time $\tau_0 > t$ and a node $y_0 \in \mathsf{leafset}(z_0, z_0.\textit{neighbors}_t)$ such that $d(x_0, y_0) < d(x_0, z_0)$ and $y_0 \in x_0.\textit{neighbors}_{\tau_0}$. By Corollary 16, we know that $y \notin \rho(x)$. Otherwise since $y \in \rho(x)$ leads to $\mathsf{leafset}(y_0, y_0.\textit{neighbors}_{\tau_o}) \subseteq \rho(x)$, and $z_0 \in \mathsf{leafset}(y_0, y_0.\textit{neighbors}_{\tau_0})$ by Lemma 11, $z_0 \in \rho(x)$, a contradiction. Since $\mathsf{leafset}(x_0, x_0.\textit{neighbors}_{\tau_0}) \subseteq \rho(x)$, we also have $y_0 \notin \mathsf{leafset}(x_0, x_0.\textit{neighbors}_{\tau_0})$. Thus we find a node y_0 such that $y_0 \in x_0.\textit{neighbors}_{\tau_0} \setminus \mathsf{leafset}(x_0, x_0.\textit{neighbors}_{\tau_0})$ and $d(x_0, y_0) < d(x_0, z_0)$. We can continue applying Lemma 13 1) to find nodes $y_1, y_2, \ldots$ and time points $\tau_1, \tau_2, \ldots$ such that $y_i \in x_0.\textit{neighbors}_{\tau_i} \setminus \mathsf{leafset}(x_0, x_0.\textit{neighbors}_{\tau_i})$, $d(x_0, y_{i+1}) < d(x_0, y_i)$, and $\tau_i < \tau_{i+1}$. However, since there are only a finite number of nodes in $\textit{sset}(\Pi)$, this process cannot continue indefinitely, a contradiction.

Consider the second case where $\langle z_0, x_0\rangle$ is in $P_x(t)$. 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 18 *For all time $t > t_2$ and all $x \in \textit{sset}(\Pi)$, $x.\textit{succ}_t = \mathsf{succ}(x, P_x(t))$.*

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 $x.\textit{succ}$.

We only consider the cases that $\textit{sset}(\Pi) \neq \emptyset$. If $P_x(t)$ only contains a single node x, according to Lemma 8, $x.\textit{neighbors}$ becomes empty after t_1. So $x.\textit{succ} = x$ and $\mathsf{succ}(x, P_x(t)) = x$, the lemma holds. Now consider the case that $P_x(t)$ contains at least 2 nodes. Suppose, for a contradiction, that there is some node $x \in P_x(t)$ such that $x.\textit{succ} \neq \mathsf{succ}(x, P_x(t)) = y$.

Consider the closed loop $\rho(x)$. By Lemma 17, $\rho(x) = P_x(t)$, 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 12, 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 $y = \mathsf{succ}(x, P_x(t))$ and $x.\mathit{neighbors} \subseteq P_x(t)$, 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 $0 \in [u, u.\mathit{succ})$ and $0 \in [v, v.\mathit{succ})$, which implies that the leafset topology containing $\rho(x)$ is loopy. In all cases, we reach a contradiction. Therefore, the lemma holds. □

Lemma 19 (Eventual Inclusion) *For all time $t > t_2$, and all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.\mathit{neighbors}_t) = \mathsf{leafset}(x, P_x(t))$.*

Proof. It is sufficient to show that for all time after t_2 and for all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.\mathit{neighbors}_t) = \mathsf{leafset}(x, P_x(t))$.

By Lemma 18, for all $x \in sset(\Pi)$, $x.\mathit{succ}_t = \mathsf{succ}(x, P_x(t))$. According to the definition of succ and pred, we know that $\mathsf{succ}(x, P_x(t)) = y$ is equivalent to $\mathsf{pred}(y, P_x(t)) = x$. According to Lemma 15 1) and 2), we also have $x.\mathit{pred}_t = \mathsf{pred}(x, P_x(t))$.

Now consider $\{x_{+1}, x_{+2}, \ldots, x_{+M}\}$ in $x.\mathit{neighbors}_t^+$, and $\{x_{-1}, x_{-2}, \ldots, x_{-M}\}$ in $x.\mathit{neighbors}_t^-$ with $M = \min(L, |x.\mathit{neighbors}|)$, for an arbitrary node $x \in P_x(t)$. By Lemma 15 3) we have $x_{+1} = x.\mathit{succ} = \mathsf{succ}(x, P_x(t))$ and $x_{+(i+1)} = x_{+i}.\mathit{succ} = \mathsf{succ}(x_{+i}, P_x(t))$, $x_{-1} = x.\mathit{pred} = \mathsf{pred}(x, P_x(t))$ and $x_{-(i+1)} = x_{-i}.\mathit{pred} = \mathsf{pred}(x_{-i}, P_x(t))$, for all $i = 1, 2, \ldots, M-1$. This directly implies that $\mathsf{succ}(x_{+i}, P_x(t)) \in \mathsf{leafset}(x, P_x(t))$ and $\mathsf{pred}(x_{-i}, P_x(t)) \in \mathsf{leafset}(x, P_x(t))$, for all $i = 1, 2, \ldots, M-1$. Thus we have $\mathsf{leafset}(x, x.\mathit{neighbors}_t) \subseteq \mathsf{leafset}(x, P_x(t))$.

If $|\mathsf{leafset}(x, x.\mathit{neighbors})| < |\mathsf{leafset}(x, P_x(t))|$, then $\mathsf{leafset}(x, x.\mathit{neighbors}_t) = x.\mathit{neighbors}_t$, and since $\rho(x) = P_x(t)$, there must be some node $y \in \rho(x)$ such that $y \notin x.\mathit{neighbors}_t$. Let y be the first such one following the sequence $\rho(x)$, i.e., $y = z.\mathit{succ}$ while $z \in x.\mathit{neighbors}$. 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 $|\mathsf{leafset}(x, x.\mathit{neighbors}_t)| = |\mathsf{leafset}(x, P_x(t))|$. In this case, if $\mathsf{leafset}(x, P_x(t)) \nsubseteq \mathsf{leafset}(x, x.\mathit{neighbors}_t)$, there exists a node $y \in \mathsf{leafset}(x, P_x(t)) \setminus \mathsf{leafset}(x, x.\mathit{neighbors}_t)$. 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 $x_{+j}.\mathit{succ} = x_{+(j+1)} \neq \mathsf{succ}(x_{+j}, P_x(t))$, contradicting to Lemma 18. In the second case, we have $x_{-(j+1)}.\mathit{succ} = x_{-j} \neq \mathsf{succ}(x_{-(j+1)}, P_x(t))$, again contradicting to Lemma 18. Therefore $\mathsf{leafset}(x, P_x(t)) \subseteq \mathsf{leafset}(x, x.\mathit{neighbors}_t)$, and thus $\mathsf{leafset}(x, P_x(t)) = \mathsf{leafset}(x, x.\mathit{neighbors}_t)$. □

Corollary 20 *If for some time $t \geq GST_S$, $G(t)$ is weakly connected, then for all time $t' > t_2$, for any node $x \in sset(\Pi)$, $\mathsf{leafset}(x, sset(\Pi)) = \mathsf{leafset}(x, x.\mathit{neighbors}_{t'})$.*

Proof. Because $G(t)$ is weakly connected, we have $P_x(t'') = G(t'')$ for all $t'' \geq t$, according to Lemma 5. According to Lemma 17, we know that for all $t'' > \max(t, t_2)$ and for all $x \in sset(\Pi)$, $\rho(x) = sset(\Pi)$. Because $\rho(x)$ only depends on the leafset of all the nodes and Lemma 9 ensures that leafsets of all nodes stop change after t_2. So it must be true that $t_2 \geq t$. Therefore, according to Lemma 19, $\mathsf{leafset}(x, sset(\Pi)) = \mathsf{leafset}(x, x.neighbors_{t'})$. □

Lemma 21 (Cost Effectiveness) *In the steady state the size of local state on each node is $O(L)$ and the number of nodes registered to the failure detector is no more than $2L$.*

Proof (Sketch). After t_2, 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 *repl*$[y]$ is a single node. Thus the size of the local state is $O(L)$.

On every change of the *neighbors* set, except for the one caused by the detected notification, our protocol re-registers the set to be monitored by the failure detector. If the failure detector removes the node detected as failed from its monitor set automatically, the monitor set is eventually the same as the *neighbors* set. Since the *neighbors* set contains no more than $2L$ nodes, the number of nodes registered to the failure detector is eventually no more than $2L$. □

Theorem 2 *The leafset maintenance protocol provided in Fig. 1, 2, 3, and 5 is both convergent and cost-effective, which means it satisfies the Connectivity Preservation, Partition Healing, Eventual Cleanup, Eventual Inclusion, and Cost Effectiveness properties.*

Proof. The theorem follows from Lemmata 5, 7, 14, 19, and 21. □

C Proof of Correctness for $\Diamond\mathcal{P}_D$ implementation

We prove that the protocol in Figure 6 correctly implement a failure detector in $\Diamond\mathcal{P}_D$.

Lemma 22 (Strong Completeness) *For all $x \in sset(\Pi)$ and all $y \notin sset(\Pi)$, if x invokes* $\mathsf{register}(S)$ *with $y \in S$ at some time t, then there exists a time $t' > t$ at which either our $\Diamond\mathcal{P}_D$ protocol (Figure 6) outputs* $\mathsf{detected}(y)$ *or x invokes* $\mathsf{register}(S')$ *with $y \notin S'$.*

Proof. We only need to prove that $\mathsf{detected}(y)$ will be outputted, if x never invokes any $\mathsf{register}(S')$ after t, or every $\mathsf{register}(S')$ has y in S'.

After GST_N, y cannot send messages any more since $y \notin sset(\Pi)$. Let $t(y)$ be the time after which no messages from y will be delivered. So after $t(y)$, x will not be able to receive any message from y. Let $t'' = \max(t, t(y))$. During the interval $(t'' + T_c, t'' + T_c + I_c]$, there exists a time t' at which x checks the liveness in $x.mset$. If $y \in x.mset$ at t', the checking at line 19 will be passed, because $t' - t'' > T_c$. Therefore, a $\mathsf{detected}(y)$ is outputted on x. If $y \notin x.mset$ at t', since every registration after t contains y, an earlier detection must have been issued. In either case, the lemma holds. □

Lemma 23 (Eventual Strong Accuracy) *If $I_c, T_c \geq I_p + 2\Delta$, then after time $GST_M + I_c$, our $\Diamond\mathcal{P}_D$ protocol (Figure 6) will not report failures of any online nodes in $sset(\Pi)$.*

Proof. Suppose, for a contradiction, that there exist nodes $x, y \in sset(\Pi)$ and a time $t > GST_M + I_c$ such that our $\Diamond\mathcal{P}_D$ protocol invokes $\mathsf{detected}(y)$ on x. Without loss of generality, we assume t is the first such time. From the algorithm (line 19), we know that x does not receive any PONG-ALIVE 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 $[t-(I_p+2\Delta), t]$. Since $I_c > I_p + 2\Delta$ and $t > GST_M + I_c$, we have $t - (I_p + 2\Delta) > GST_M$.

We claim that y is in $x.mset$ in the period $[t-(I_p+2\Delta), t]$. To see that this is true, first, $\mathsf{detected}(y)$ is invoked on x at time t when executing line 20, so y was in $x.mset$ right before time t. If y was not in $x.mset$ in the entire period in $[t - (I_p + 2\Delta), t]$, then y must be added back to $x.mset$ at some time in this period. However, according to the algorithm, a node y can be added to $x.mset$ only through the $\mathsf{register}()$ interface. And line 19 ensures that y enters $x.mset$ for at least T_c time. Since $T_c > I_p + 2\Delta$, y must be in $x.mset$ 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 $y \in x.mset$ in this time period. Since $t - (I_p + 2\Delta) > GST_\Delta$, 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. □

Theorem 3 *The protocol in Figure 6 implements a failure detector in $\Diamond\mathcal{P}_D$.*

Proof. By Lemmata 22 and 23. □