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

Yu Chen and Wei Chen

Microsoft Research Asia
{ychen,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 overlay connectivity, 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

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

[1] The term leafset is originally used in Pastry [19] while the term finger is originally used in Chord [21].

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 [19, 21].

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. 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 [14] and Pastry [19] 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 preservation.

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 ([11, 15]) rely on the maintenance of a large membership list to preserve connectivity, and thus is not cost-effective.

Third, we explicitly address how to heal topology partition by introducing an interface function $\mathsf{add}(\mathit{contacts})$. Although the overlay could be more resistant to topology partition by maintaining more entries in the routing tables [14], 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 $\mathsf{add}(\mathit{contacts})$ and its specification cleanly separates partition detection from partition healing: A separate mechanism may be used to detect topology partition, and then to call the $\mathsf{add}(\mathit{contacts})$ 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 [7, 20].

Moreover, 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. One may argue that in peer-to-peer environments, such failure detectors are unrealistic. 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 [4], 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 convergent leafset maintenance protocols. Section 5 presents the complete leafset maintenance protocol. We conclude the paper and discuss future work in Section 6. Our full technical report [4] contains further results including complete proofs, a sample implementation of

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

the failure detector in one partially synchronous model, and optimizations for fast convergence of the protocol.

2 Related Work

Many existing structured P2P overlay proposals mention that each node should have a leafset table. However, those such as Pastry [19], CAN [17], and SkipNet [10] 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 [18], the latest Pastry improvements [2, 9], and $\mathcal{DKS}$ [8] 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 [14], Liben-Nowell et al. point out the topology maintenance issues of the original Chord protocol [21] 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 [5, 6] to study overlay maintenance. The T-Man [11] and TChord [15] 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 [7] and [20] also propose self-stabilizing overlay maintenance protocols. But their protocols and proofs depend on the existence of a *continuously available* bootstrap system. In [7], the bootstrap system needs to handle all join and repair requests, and needs to issue periodic broadcast messages for self stabilization purpose, while in [20] 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.

The "Linearization" method in [16] describes a self-stabilizing algorithm to transform any connected graph into a sorted list. Although a bootstrap sys-

tem is not required, the algorithm does not consider node churns and asynchronous/concurrent effects in a distributed message passing environment.

Authors of Ranch [13] 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 environment, and dealing with them makes the model, the specification and the protocol design significantly depart from those studied in [13].

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 bidi-

rectional 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 still online. More rigorously, it 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

[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.

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.

Each node also has an interface function $\mathsf{add}(contacts)$, where *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 $E(t) = \{\langle x, y \rangle | x, y \in \Pi(t) \ \wedge \ y \in x.neighbors_t\}$. 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 $t \geq GST_N$ such that $\diamond\mathcal{P}_D$ will not output $\mathsf{detected}(y)$ on any $x \in sset(\Pi)$ for any $y \in sset(\Pi)$ 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 $t \geq \max(GST_D, GST_M)$ 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 $t \geq GST_S$, 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. The following theorem shows the necessity of GST_S, meaning that no algorithm can guarantee connectivity preservation starting from a time earlier than GST_S. The proof of the theorem can be found in [4].

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.*

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

If the topology becomes connected at some time after GST_S, then Eventual Inclusion together with Connectivity Preservation means that eventually $\mathsf{leafset}(x, x.neighbors_{t'}) = \mathsf{leafset}(x, sset(\Pi))$ for all $x \in sset(\Pi)$. 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, an application (or even a user) should be able to use the $\mathsf{add}()$ interface to heal the partition. This is specified by the following property:

- *Partition Healing*: For any $x, y \in sset(\Pi)$, 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., $P_x(t') = P_y(t')$).

The Partition Healing property ensures that only one invocation of 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 add().

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

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 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 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 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 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 [5, 6] 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 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 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 3–8); (b) the failure-handling protocol to remove the failed nodes from the leafset upon the notification of failure detector (Fig. 1, lines 9–10); (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.

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.

5.1 Add new contacts and handle failures

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 3–8) uses a ping-pong message loop to check the liveness of the nodes being added. In this way, the

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

On node x:

```
1  Data structure:
2    neighbors: set of nodes intended for leafset entries, initially ∅.
3  add(contacts)
4    foreach y ∈ contacts, send PING-CONTACT to y
5  Upon receipt of PING-CONTACT from y:
6    send PONG-CONTACT to y
7  Upon receipt of PONG-CONTACT from y:
8    neighbors ← neighbors ∪ {y}; register(neighbors)
9  Upon detected(y):
10   neighbors ← neighbors \ {y}
```

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

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.

5.2 Invite closer nodes

The invite protocol (Fig. 2) uses a variable *cand* to store candidate nodes to be invited into the *neighbors* set. The candidate nodes are discovered by exchanging local leafset views through the PING-ASK-INV and PONG-ASK-INV messages. Once 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 27).The invite protocol is in principle similar to other leafset maintenance protocols (e.g. [21, 18, 11, 20]), 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

On node x:

```
11  Data structure:
12    cand: candidate nodes for neighbors, initially ∅.
13  Repeat periodically:
14    foreach y ∈ neighbors, send PING-ASK-INV to y
15  Upon receipt of PING-ASK-INV from a node y:
16    view ← leafset(y, neighbors); send (PONG-ASK-INV, view) to y
17    cand ← cand ∪ {y}
18  Upon receipt of (PONG-ASK-INV, view) from y
19    cand ← cand ∪ view
20  Repeat periodically                                  /* invite closer nodes */
21    foreach y ∈ cand \ neighbors
22      if y ∈ leafset(x, cand ∪ neighbors) then send PING-INVITE to y
23    cand ← ∅
24  Upon receipt of PING-INVITE from y:
25    send PONG-INVITE to y
26  Upon receipt of PONG-INVITE from y:
27    if y ∈ leafset(x, neighbors ∪ {y}) \ neighbors then
28        neighbors ← neighbors ∪ {y}; register(neighbors)
```

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

PONG-INVITE messages. Therefore, it will not be returned to other nodes as an invitation candidate, either.

5.3 Replace faraway nodes

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 $\mathsf{leafset}(x, x.neighbors)$. Node x uses the PING-ASK-REPL and PONG-ASK-REPL loop (lines 33–39) 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 36, 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 40–52). 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,

On node x:

29 Data structure:
30 $repl[\,]$: for each $z \in neighbors$, $repl[z]$ is a node to replace z, initially $\perp$
31 $commit[\,]$: 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 $neighbors$ */
32 : timestamp of the replacement task, initially 0
33 Repeat periodically:
34 **foreach** $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$, send PING-ASK-REPL to z
35 Upon receipt of PING-ASK-REPL from z:
36 $y \leftarrow v$ such that $v \in \mathsf{leafset}(x, neighbors)$ **and** $d(z, v) < d(z, x)$ **and** $d(z, v) = \min_{u \in \mathsf{leafset}(x, neighbors)} d(z, u)$
37 send (PONG-ASK-REPL, y) to z
38 Upon receipt of (PONG-ASK-REPL, y) from z
39 **if** $z \in neighbors$ **then** $repl[z] \leftarrow y$
40 Repeat periodically:
41 $\leftarrow \mathsf{getClockValue}()$
42 **foreach** $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$ **and** $repl[z] \neq \perp$
43 send (PING-REPLACE, z,) to $repl[z]$
44 Upon receipt of (PING-REPLACE, z,) from y:
45 if $z \in neighbors$ **then**
46 $commit[z] \leftarrow \mathsf{getClockValue}()$; send (PONG-REPLACE, z,) to y
47 Upon receipt of (PONG-REPLACE, z,) from y:
48 **if** $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$ **and** $y = repl[z]$ **then**
49 $neighbors \leftarrow neighbors \cup \{y\}$
50 **if** $commit[z] <$ **then**
51 $neighbors \leftarrow neighbors \setminus \{z\}$; $commit[y] \leftarrow \mathsf{getClockValue}()$
52 $\mathsf{register}(neighbors)$

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

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 nullify 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 "$\mathsf{detected}(z)$" 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

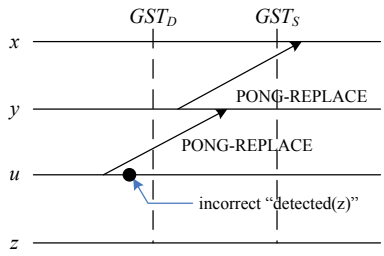

Fig. 4. Concurrent replacement tasks introduce indirect effects of adverse system events before GST_D and break topology connectivity.

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 and *commit*[ ] to eliminate these dangerous concurrent replacements. Variable is a timestamp identifying the current replacement task when a node sends out PING-REPLACE messages (line 41), 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 46), or when x uses z to replace another node y (line 51). The key condition is that x can only successfully replace z in a replacement task whose timestamp is higher than $commit[z]$ on x (line 50). The use of 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 36). 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.

On node x:

53 Data structure:
54 $\succ$: a derived variable, $\succ = x$ if $neighbors = \emptyset$ else $\succ = y \in neighbors$ such that $d^+(x, y) = \min\{d^+(x, z) : z \in neighbors\}$
55 Repeat periodically:
56 **if** $neighbors \neq \emptyset$ **and** $d^+(x, 0) < d^+(x, \succ)$ **then**
57 send (PING-DELOOPY, x) to $\succ$
58 Upon receipt of (PING-DELOOPY, u) from y:
59 **if** $x = u$ **then return**
60 **if** $neighbors = \emptyset$ **or** $d^+(x, 0) < d^+(x, \succ)$ **then**
61 $cand \leftarrow cand \cup \{u\}$; send PONG-DELOOPY to u
62 **else**
63 send (PING-DELOOPY, u) to $\succ$
64 Upon receipt of PONG-DELOOPY from y:
65 $cand \leftarrow cand \cup \{y\}$

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

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 49), without checking the constraint of $z.commit <$. Next, x can remove z only when the condition $z.commit <$ holds (lines 50–51). 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 ($\mathsf{leafset}(z, z.neighbors)$) in line 36. 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.

5.4 Detect loopy structure

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 [14]. 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 [14] 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.

Our protocol is cost-effective because in the steady state each node only maintains 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.*

6 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.

The primary focus of this paper is the formal treatment of ring-based overlay maintenance. For a more practical implementation, a number of issues need to be addressed, which can be regarded as the future directions of our work. First, potential optimizations is possible to save the maintenance bandwidth of our protocol. Second, we may be able to weaken our model assumptions such as the availability of the dynamic failure detector $\diamond\mathcal{P}_D$ and the existence of the global stabilization time GST_S to match closer to the dynamic peer-to-peer environments, by following the similar approach in [12] for example. Another direction is to study the convergence speed of our protocol. On this front, we have conducted simulation studies with some heuristics to achieve an $O(\log N)$-level convergence time where N is the total number of nodes in the system [4]. We are looking into theoretical analysis of the fast convergence protocols. Finally, generalizing our results to other structured overlay topologies is also useful.

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*, 2005.
2. M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2004.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

4. Y. Chen and W. Chen. Decentralized, connectivity-preserving, and cost-effective structured overlay maintenance. Technical Report MSR-TR-2007-84, Microsoft Research, 2007.
5. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
6. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
7. 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.
8. A. Ghodsi, L. O. Alima, and S. Haridi. Low-bandwidth topology maintenance for robustness in structured overlay networks. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Track 9*, 2005.
9. A. Haeberlen, J. Hoye, A. Mislove, and P. Druschel. Consistent key mapping in structured overlays. Technical Report TR05-456, Rice Computer Science Department, 2005.
10. 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*, 2003.
11. M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In *Proceedings of the 3rd International Workshop on Engineering Self-Organising Applications*, 2005.
12. I. Keidar and A. Shraer. How to choose a timing model? In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.
13. X. Li, J. Misra, and C. G. Plaxton. Active and concurrent topology maintenance. In *Proceedings of the 18th International Symposium on Distributed Computing*, 2004.
14. 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*, 2002.
15. A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, 2005.
16. M. Onus, A. Richa, and C. Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, 2007.
17. 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*, 2001.
18. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, 2004.
19. 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*, 2001.
20. 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*, 2005.
21. 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*, 2001.