Avatar: A Time- and Space-Efficient Self-stabilizing Overlay Network

Andrew Berns(✉)

Department of Computer Science, University of Wisconsin-La Crosse, La Crosse, WI, USA
aberns@uwlax.edu

Abstract. Overlay networks present an interesting challenge for fault-tolerant computing. Many overlay networks operate in dynamic environments (e.g. the Internet), where faults are frequent and widespread, and the number of processes in a system may be quite large. Recently, self-stabilizing overlay networks have been presented as a method for managing this complexity. *Self-stabilizing overlay networks* promise that, starting from any weakly-connected configuration, a correct overlay network will eventually be built. To date, this guarantee has come at a cost: nodes may either have high degree during the algorithm's execution, or the algorithm may take a long time to reach a legal configuration. In this paper, we present the first self-stabilizing overlay network algorithm that does not incur this penalty. Specifically, we (i) present a new locally-checkable overlay network based upon a binary search tree, and (ii) provide a randomized algorithm for self-stabilization that terminates in an expected polylogarithmic number of rounds *and* increases a node's degree by only a polylogarithmic factor in expectation.

1 Introduction

Today's distributed systems are quite different from those only a decade ago. Pervasive network connectivity and an increase in the number of computational devices has ushered in an era of large-scale distributed systems operating in highly-dynamic environments. One type of distributed system that has gained popularity recently is the overlay network. An *overlay network* is a network where communication occurs over *logical* links, where each logical link consists of zero or more *physical* links. The use of logical links allows the design of efficient logical topologies (e.g. topologies with low diameter and/or low degree) irrespective of the physical topology, enabling efficient data structures to be constructed from large systems with arbitrary physical networks.

The dynamic nature of many overlay networks makes fault tolerance extremely important. *Self-stabilization*, first presented by Dijkstra in 1974 [6], is an elegant fault-tolerant paradigm promising that, after *any* memory-corrupting transient fault, the system will eventually recover to a correct configuration. *Self-stabilizing overlay networks* are logical networks that guarantee a correct topology will be restored after any such transient fault.

A. Pelc and A.A. Schwarzmann (Eds.): SSS 2015, LNCS 9212, pp. 233–247, 2015.
DOI: 10.1007/978-3-319-21741-3_16

1.1 Related Work

Many overlay networks include a mechanism to tolerate a subset of possible faults. For instance, Chord [15] defines a procedure for nodes to join the network efficiently. The Forgiving Graph [9] presents a self-healing overlay network which maintains connectivity while limiting degree increases and stretch despite periodic adversarial node insertions and deletions.

Self-stabilizing overlay networks, however, are a relatively new area of research. In 2007, Onus et al. presented the first silent self-stabilizing overlay network, building a linear topology in linear (in the number of nodes) rounds [14]. The Skip+ graph, presented in 2009 by Jacob et al. [10], was the first self-stabilizing overlay network with polylogarithmic convergence time. Berns et al. presented a generic framework capable of building any locally-checkable overlay network, and proved that their result was near-optimal in running time [4].

Current self-stabilizing overlay networks have suffered from one of two limitations. First, some self-stabilizing overlay networks require a long time to reach a correct configuration. For instance, ReChord [11], a self-stabilizing variant of Chord, requires $\mathcal{O}(n \log n)$ rounds to reach a correct configuration. Other self-stabilizing overlay networks that converge quickly have required a large amount of space. For example, Skip+ [10] has a polylogarithmic convergence time, but may increase a node's degree to $\mathcal{O}(n)$ during convergence. The Transitive Closure Framework [4] requires $\Theta(n)$ space. To date, no work has achieved efficient convergence in both time and space.

1.2 Contributions

In Section 3, we present Avatar, a generic locally checkable overlay network, and describe a specific "instance" of the network called $\text{Avatar}_{\text{Cbt}}$ which is based upon a binary search tree. Section 4 presents a randomized self-stabilizing algorithm for creating the $\text{Avatar}_{\text{Cbt}}$ network, as well as an analysis sketch of the algorithm's performance in both convergence time and space (using a new metric we call the *degree expansion*).

2 Model of Computation

We model the distributed system as an undirected graph $G = (V, E)$, with nodes V representing the processes of the system, and edges E representing the communication links. Each node u is assigned an identifier from the function $id : V \to \mathbb{Z}^+$. We assume each node stores $id(u)$ as immutable data. Where clear from context, we refer to a node u by its identifier $id(u)$.

Each node $u \in V$ has a *local state* $S(u)$ consisting of a set of variables and their values. We assume all nodes have access to a shared immutable random sequence Ψ. A node u can modify the values of its variables using *actions* defined in the *program* of u. All nodes execute the same program. We use a *synchronous* model of computation, where in one *round* each node executes its program and

communicates with its neighbors. We use the *message passing* model of communication, where a node u can communicate with a node v in its *neighborhood* $N(u) = \{v \in V : (u, v) \in E\}$ by sending node v (called a *neighbor*) a message. A node can send unique messages to every neighbor in every round. We assume reliable and bounded capacity communication channels where a message is received by a node u at the beginning of round i if and only if it was sent to u by some $v \in N(u)$ in round $(i - 1)$.

In the overlay network model, a node's neighborhood is part of its state, allowing a node to change its neighborhood with program actions. In a round i, a node u can delete any subset of edges incident upon it, and add edges to any nodes currently at distance 2 from u. Specifically, let G_i be the configuration in round i. A node u can (i) delete any edge $(u, v) \in E(G_i)$, resulting in $(u, v) \notin E(G_{i+1})$, and (ii) create the edge (u, w) if $(u, v), (v, w) \in E(G_i)$, resulting in $(u, w) \in E(G_{i+1})$. We restrict edge additions to only those nodes at distance 2 to reflect the fact that only nodes at distance 2 share a common neighbor through which they can be "connected". We assume that $v \in N(u) \Leftrightarrow (u, v) \in E$ – that is, every neighbor of u is known, and u has no "false neighbors" in $N(u)$ (this can be achieved with the use of a "heartbeat" message).

Our problem is to take a set of nodes V and create a *legal configuration*, where a legal configuration is defined by some predicate taken over the state of all nodes in V. Since edges are state in an overlay network, the legal configuration predicate often includes the requirement that the topology matches a particular *desired topology* $ON(V) = (V, E)$. The *self-stabilizing overlay network problem* is to design an algorithm $\mathcal{A}$ such that, when executed on nodes V with arbitrary initial state in an arbitrary weakly-connected initial topology, the system eventually reaches a legal configuration. Furthermore, once the network is in a legal configuration, it remains in a legal configuration until an external fault perturbs the system.

Performance of an overlay network algorithm can be measured in terms of both time and space. To analyze the worst-case performance, it is assumed that an *adversary* creates the initial configuration using full knowledge of the nodes and the algorithm they will execute. The adversary does not, however, know the value of the shared random sequence Ψ. The maximum number of rounds required for a legal configuration to be reached, taken over all possible configurations, is the *convergence time* of $\mathcal{A}$. One measure of space complexity is the number of incident edges on a node, as each incident edge requires memory and maintenance (heartbeat messages). We introduce the *degree expansion* as a space complexity measurement. The degree expansion, informally, is the amount a node's degree may grow "unnecessarily" during convergence. For a graph G with node set V, let Δ_G be the maximum degree of nodes in G. For a self-stabilizing algorithm $\mathcal{A}$ executing on G, let $\Delta_{\mathcal{A},G}$ be the maximum degree of any node from V during execution of $\mathcal{A}$ beginning from configuration G. We define degree expansion as follows.

Definition 1. *The* degree expansion of $\mathcal{A}$ on G, *denoted* $DegExp_{\mathcal{A},G}$, *is equal to* $(\Delta_{\mathcal{A},G}/\max(\Delta_G, \Delta_{ON(\lambda)}))$. *Let the* degree expansion of $\mathcal{A}$ *be* $DegExp_{\mathcal{A}} = \max_{G\in\mathcal{G}}(DegExp_{\mathcal{A},G})$

The degree expansion is meant to capture the degree growth of the algorithm while excluding clever initial configurations from the adversary resulting in a high degree increase.

A self-stabilizing overlay network algorithm is *silent* if and only if the algorithm brings the system to a configuration where the messages exchanged between nodes remains fixed until a fault perturbs the system [7]. Traditionally, these messages consist of a node's state. In order for a silent self-stabilizing overlay network algorithm to exist using only this information, the network must be locally checkable. An overlay network is *locally checkable* if and only if each illegal configuration has at least one node (called a *detector*) which detects that the configuration is not legal using only its state and the state of its neighbors, and all legal configurations have no detectors.

3 The Avatar Network

3.1 Avatar Specification

One of the challenges with creating silent self-stabilizing overlay network algorithms is designing a locally checkable topology as many previous overlay networks are *not* locally checkable. For instance, Skip+ [10] was created to have a locally-checkable variant of the Skip graph [1]. Similarly, the self-stabilizing ReChord network [11] is a locally-checkable Chord [15] derivative built using real and virtual nodes. Simplifying this network design task is the motivation for Avatar. Avatar easily allows many different topologies to be "simulated" while ensuring local checkability.

Avatar is based around the idea of network embeddings. A *network embedding* Φ maps the node set of a *guest network* $G_g = (V_g, E_g)$ onto the node set of a *host network* $G_h = (V_h, E_h)$ [13]. The *dilation* of Φ is defined as the maximum distance between any two nodes $\Phi(u), \Phi(v) \in V_h$ such that $(u, v) \in E_g$. The Avatar network is an overlay network realizing a dilation-1 embedding for a guest network using logical overlay links. To do this *and* ensure local checkability, the (host) overlay edges of Avatar consist of the successor and predecessor edges from a linearized graph (ensuring host nodes can check which guest nodes map to them) as well as the overlay edges necessary for host nodes of two neighboring guest nodes to be at most distance 1 apart.

Formally, for any $N \in \mathbb{N}$, let $[N]$ be the set of nodes $\{0, 1, \ldots, N-1\}$. Let $\mathcal{F}$ be a family of graphs such that, for each $N \in \mathbb{N}$, there is exactly one graph $F_N \in \mathcal{F}$ with node set $[N]$. We use $\mathcal{F}(N)$ to denote F_N. We call $\mathcal{F}$ a *full graph family*, capturing the notion that the family contains exactly one topology for each "full" set of nodes $[N]$ (relative to the identifiers). For any $N \in \mathbb{N}$ and $V \subseteq [N]$, $\text{Avatar}_{\mathcal{F}}(N, V)$ is a network with node set V that realizes a dilation-1 embedding of $F_N \in \mathcal{F}$. The specific embedding is given below. We also show

that, when N is known, Avatar is locally checkable (N can be viewed as an upper bound on the number of nodes in the system).

Definition 2. *Let $V \subseteq [N]$ be a node set $\{u_0, u_1, \ldots, u_{n-1}\}$, where $u_i < u_{i+1}$ for $0 \leq i < n-1$. Let the* range *of a node u_i be range$(u_i) = [u_i, u_{i+1})$ for $0 < i < n-1$. Let range$(u_0) = [0, u_1)$ and range$(u_{n-1}) = [u_{n-1}, N)$. Avatar$_{\mathcal{F}}(N, V)$ is a graph with node set V and edge set consisting of two edge types:*

Type 1: $\{(u_i, u_{i+1}) | i = 0, \ldots, n-1\}$
Type 2: $\{(u_i, u_j) | u_i \neq u_j \wedge \exists (a, b) \in E(F_N), a \in range(u_i) \wedge b \in range(u_j)\}$

Theorem 1. *Let $\mathcal{F}$ be an arbitrary full graph family, and let Avatar$_{\mathcal{F}}(N, V)$ be an overlay network for an arbitrary N and V, and let each $u \in V$ know N. Avatar$_{\mathcal{F}}(N, V)$ is locally checkable.*

Proof sketch: Note each node can calculate its range using only its neighborhood. Using the state of its neighbors, u can also calculate the range of each neighbor. This information is sufficient for each node u to verify every neighbor $v \in N(u)$ is either from a type 1 or type 2 edge. As all nodes know N and there is exactly one $F_N \in \mathcal{F}$, all nodes can verify their type 1 and type 2 edges correctly map to the given network. Interestingly, Avatar$_{\mathcal{F}}$ is locally checkable using only $\mathcal{O}(\log n)$ bits from each neighbor: each node need share only (i) its identifier, and (ii) the identifier of its predecessor and successor.

3.2 The Full Graph Family Cbt

Our goal is to create a self-stabilizing Avatar network which stabilizes quickly and maintains low degree during convergence. To this end, we simulate a simple data structure with constant degree and logarithmic diameter: a binary search tree. As we show, an embedding in Avatar of a binary search tree maintains a low degree and diameter.

Formally, consider the graph family based upon *complete binary search trees*. Below we define the full graph family Cbt by defining Cbt(N) recursively.

Definition 3. *For $a \leq b$, let* Cbt$[a, b]$ *be a binary tree rooted at $r = \lfloor (b+a)/2 \rfloor$. Node r's left cluster is* Cbt$[a, r-1]$*, and r's right cluster is* Cbt$[r+1, b]$*. If $a > b$, then* Cbt$[a, b] = \perp$*. We define* Cbt(N) = Cbt$[0, N-1]$*. Let the* level *of a node d in* Cbt$[0, N-1]$ *be the distance from d to root $\lfloor N - 1/2 \rfloor$.*

Diameter and Maximum Degree of Avatar$_{\text{Cbt}}$. All dilation-1 embeddings preserve the diameter of the guest network, meaning Avatar$_{\text{Cbt}}$ has $\mathcal{O}(\log N)$ diameter. Note a node v in our embedding may have a large $\Phi^{-1}(v)$ – that is, many nodes from the guest network may map to a single host node. Surprisingly, the host nodes for Avatar$_{\text{Cbt}}$ have a small degree *regardless of* Φ^{-1}, as we show below.

Theorem 2. *For any node set $V \subseteq [N]$, the maximum degree of any node $u \in V$ in* $\textsc{Avatar}_{\textsc{Cbt}}(N, V)$ *is at most* $2 \cdot \log N + 2$.

Proof sketch: Consider $\Phi^{-1}(u)$, the subset of nodes from $[N]$ mapped to node u. Let $[N]_j$ be the set of all nodes at level j of $\textsc{Cbt}(N)$. There are at most 2 nodes in $\Phi^{-1}(u) \cap [N]_j$ with a neighbor not in $\Phi^{-1}(u)$ – that is, there are at most 2 edges from the range of a node u to any other node outside this range for a particular level j of the tree. As there are only $\log N + 1$ levels, the total degree of any node in $\textsc{Avatar}_{\textsc{Cbt}}$ is at most $2 \cdot \log N + 2$.

4 A Self-Stabilizing Algorithm

4.1 Algorithm Overview

At a high level, our self-stabilizing algorithm works on the same principle as Gallager, Humblet, and Spira's algorithm for constructing a minimum-weight spanning tree [8]. The network is organized into disjoint clusters, each with a leader. The cluster leaders coordinate cluster merges until a single cluster remains, at which point the network is in a legal configuration.

Self-stabilizing overlay networks introduce a complication to this pattern. Converging from an arbitrary weakly-connected configuration while limiting a node's degree increase requires coordinated merges, which requires either time (additional rounds) or bandwidth (additional edges). In the overlay network model, we can increase both: we can add edges to the network and add steps to our algorithm. Our algorithm balances these additions using four components, discussed below, to achieve expected polylogarithmic convergence time and degree growth.

1. **Clustering:** As any weakly-connected initial configuration is possible, we must ensure all nodes join a cluster and have a way to efficiently communicate within their cluster. We define a cluster for $\textsc{Avatar}_{\textsc{Cbt}}$ and present mechanisms for cluster creation and intra-cluster communication.
2. **Matching:** Progress comes from clusters merging, moving towards a single-cluster configuration. However, we will show merging clusters results in an $\mathcal{O}(\log^2 N)$ degree increase for each involved cluster. To control degree growth, we limit a cluster to merge with at most one other cluster at a time. We create a matching to determine which clusters should merge. Using the overlay network model's ability to add edges, we introduce a mechanism to create "sufficiently-many" matchings on *any* topology.
3. **Merging:** Once two clusters are matched they merge together into a single cluster. Merging quickly requires sufficient "bandwidth" (in the form of edges) between two clusters. To limit degree increases, these edges must be created carefully. We present an algorithm for merging two clusters quickly while still limiting the number of additional edges that are created.
4. **Termination Detection:** Finally, to ensure our algorithm is silent, we define a simple mechanism for detecting when the legal configuration has been reached, allowing our algorithm to terminate.

We discuss these components below, providing sketches of the algorithms and analysis. Complete algorithms and analysis can be found in the full version of this work [3].

4.2 Clustering

Defining a Cluster. In the overlay network model, we can create clusters by defining the nodes of the cluster as well as the *topology* of the cluster.

Definition 4. *Let G be a graph with node set V. A* Cbt cluster *is a set of nodes $V' \subseteq V$ in graph G such that $G[V']$, the subgraph of G induced by V', is* $\text{Avatar}_{\text{Cbt}}(N, V')$.

Notice our cluster can be thought of on two levels: on one level, it consists of an N-node guest Cbt network, while on the other level, it consists of host nodes V'. We call the root of the guest Cbt network the *root* of the cluster. Figure 1 contains the (host) network G with two clusters: T and T'. The two (guest) Cbt networks corresponding to these clusters are given in Figure 2.

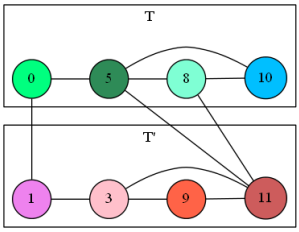

Fig. 1. Host nodes of clusters T (top) and T' (bottom)

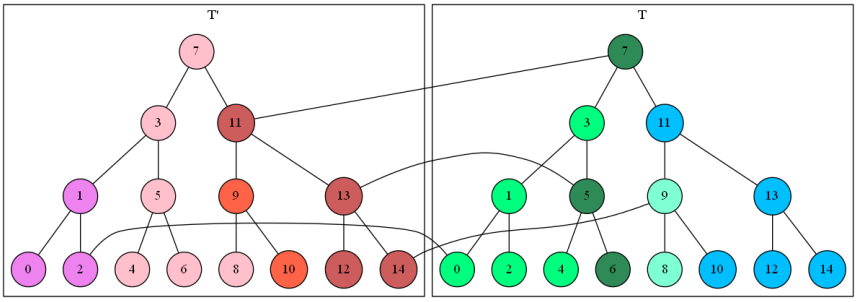

Fig. 2. Guest Nodes for T (right) and T' (left)

To ensure nodes quickly perform the necessary actions to join a cluster, we make our CBT clusters (from here on, simply clusters) locally checkable. To do this, we add three variables to each node: a cluster identifier $cluster_u$ containing the identifier of the host of the root node in the cluster, and a cluster predecessor $clusterPred_u$ and successor $clusterSucc_u$, set to the closest identifiers in the subgraph induced by nodes with the same cluster identifier. Let the *cluster range of* u be the range of u defined by the *cluster* predecessor and successor. We say a set of nodes V' is a valid cluster if and only if (i) the subgraph induced by V' matches $\text{AVATAR}_{\text{CBT}}(N, V')$, and (ii) the legal range in $\text{AVATAR}_{\text{CBT}}(N, V')$ matches the *cluster range* of u in the configuration G.

Like $\text{AVATAR}_{\text{CBT}}(N, V')$, our cluster is locally checkable. The proof follows closely the proof that $\text{AVATAR}_{\text{CBT}}$ itself is locally checkable: nodes in a cluster V' can calculate their cluster range and the cluster range of their cluster neighbors, allowing them to check that their intra-cluster edges are from $\text{AVATAR}_{\text{CBT}}(N, V')$. Furthermore, if the cluster identifier is invalid, at least one node will detect this. During convergence, there are some cases where a node is not a member of a cluster due to program actions (i.e. merge). Later we show this is also locally checkable.

In the self-stabilizing setting, there is no guarantee that each node begins execution belonging to a valid cluster. Therefore, we define a "reset" operation which a node executes when a subset of faulty configurations are detected. We say that a node u has detected a *reset fault* if u detects (i) it is not a member of a cluster, (ii) it is not in a state reachable from a "legal" merge (as we shall see, merges occur in a way allowing nodes to differentiate fault-induced invalid clusters from merging clusters), and (iii) it did not reset in the previous round. When u detects a reset fault, it "resets" to a cluster of size 1.

Intra-Cluster Communication. Our algorithms require a systematic and reliable means of intra-cluster communication. For this, we use a non-snap-stabilizing variant of the *propagation of information with feedback and cleaning* (*PFC*) algorithm [5], which we "simulate" on the guest CBT network for a cluster T (denoted $\text{CBT}_T(N)$). The root node initiates a *PFC wave*, which (i) propagates information down the tree level-by-level until reaching the leaves, (ii) sends a feedback wave from the leaves to the root, passing along any requested feedback, and (iii) prepares all nodes for another *PFC* wave. To allow the host network to simulate the *PFC* algorithm, it is sufficient to append the "level" of the sender in the guest network to each message in the host network. For instance, a guest root initiating a *PFC* wave with message m corresponds to the host of the root sending the message $(m, 0)$ to the (at most two) hosts of the root's children.

Analysis of Cluster Creation and Communication

Lemma 1. *Every node u will be a member of a cluster in $\mathcal{O}(\log N)$ rounds.*

Proof sketch: The lemma holds easily for nodes that are members of a cluster initially. Consider a node u that is not a member of a cluster. If a reset fault is detected by u, then u becomes a size-1 cluster in one round. If no reset fault is detected, either (i) u believes it is participating in a merge, or (ii) u believes it is a member of a cluster. For case (i), we will show later that the merge process is locally checkable (that is, if a configuration is reached that is not a valid merge, at least one node detects this), and that every node that detects an invalid cluster from a merge will either complete the merge in $\mathcal{O}(\log N)$ rounds, or reset in $\mathcal{O}(\log N)$ rounds, satisfying our claim. For case (ii), as clusters are locally checkable, there must be a shortest path of nodes $u, v_0, v_1, \ldots, v_k$, with $k = \mathcal{O}(\log N)$, such that all nodes in the path have a cluster identifer matching the identifier of u, and v_k detects a reset fault. When v_k executes a reset, it will cause node v_{k-1} to detect a reset fault in the next round, causing it to reset and v_{k-2} to detect a reset fault, and so on. In this way, the reset will "spread" to u in $\mathcal{O}(\log N)$ rounds, resulting in u executing a reset, satisfying our claim.

Lemma 2. *After $\mathcal{O}(\log N)$ rounds, if a set of nodes $T \subseteq V$ forms a cluster, no node in T will execute a reset action until an external fault perturbs the system.*

Proof sketch: Note that once u is part of a cluster T, no action u executes will cause it to leave cluster T unless it is merging with another cluster T'. If T and T' are merging, they will successfully create a new valid cluster T'', with $V(T) \subseteq V(T'')$. Our lemma's initial "delay" of $\mathcal{O}(\log N)$ rounds handles the case where the initial configuration contains a node which is part of a cluster with a corrupted PFC mechanism. This can only happen in an initial configuration, and it is corrected (either through the PFC mechanism or through resets) in $\mathcal{O}(\log N)$ rounds, confirming our claim.

From this point forward, our analysis shall assume the system is in a "reset-free" configuration consisting of valid clusters and merging clusters.

4.3 Matching

We can add edges during a merge to increase "bandwidth" and thus decrease the time required for the merge. However, we must be careful to limit the resulting degree increase. Therefore, a cluster T can only merge one other cluster at a time. For this, we calculate a matching between clusters. We say that a cluster T has been assigned a *merge partner* T' if and only if the roots of T and T' have been connected by the matching process described below. We say that a cluster T is *matched* if it has been assigned a merge partner, and unmatched otherwise.

We say that the *cluster graph* G_c of G is the graph induced by the clusters in configuration G, where a node v_T in G_c corresponds to a cluster T in G, and an edge $(v_T, v_{T'})$ corresponds to an edge between at least one node $u \in T$ and node $u' \in T'$. Our goal is to find a large matching on the cluster graph. To find this matching, we use a randomized symmetry-breaking technique. "Traditional" matching algorithms, however, are insufficient, as there are topologies where even a maximum matching consists of only a small number of nodes

(e.g. a star topology has a maximum matching of a single pair). In these cases, only a small number of merges would occur at a time, resulting in slow convergence. Note, however, that one can identify large matchings on the square of the cluster graph, G_c^2 (the graph resulting from connecting all nodes of distance at-most 2 in G). Since we are in the overlay network model, a matching on G_c^2 can become a (distance-one) matching by adding a single edge between matched clusters. Our matching algorithm creates a matching on G_c^2. We provide a sketch of the matching algorithm in Algorithm 1, and a discussion below.

Our matching algorithm uses two different *roles* selected by the cluster root: *leaders* and *followers*. Leaders connect followers together to form a matching on G_c^2. A cluster root chooses the cluster's role uniformly at random, with the exception of one special case (as discussed below, clusters which are merging become leaders if they were "followed" during their merge). When the root has selected the role of follower (leader), we say that the entire cluster is a follower (leader).

Consider first a follower cluster T. Each node $u \in T$ will check $N(u)$ for a node v such that v is in another cluster T' and v is a potential leader. A *potential leader* is a node which either (i) has the role of leader and is "open" (see below), or (ii) is merging, and thus "available" for followers. A node $u \in T$ will (i) mark one potential leader as "followed" (if one such neighbor exists), (ii) receive at most two edges to potential leaders from its children, and (iii) forward at most one edge incident on a potential leader to u's parent. Eventually, at most two such edges reach the root of the cluster. At this point, the root waits for the selected leader to assign it a merge partner. We define two types of followers: *long followers* and *short followers*. Short followers will only search for a leader for a "short" amount of time ($4 \log N$ rounds), while long followers will search for a (slightly) "longer" time ($24 \log N$). Long and short followers are used to make the scenario where a cluster and all of its neighbors are "stuck" searching for a leader sufficiently rare.

If the root of T has selected the leader role, the root begins by communicating the leader role to all nodes in the cluster. At this point, nodes are considered *open leaders*, and neighboring follower nodes can "follow" these leaders. After this PFC wave completes, the root sends another PFC wave asking nodes in T to (i) become *closed leaders* (no node can select them as a potential leader), and (ii) connect any current followers as merge partners. Nodes in T will connect all followers incident upon them as merge partners, thus creating a matching on G_c^2. If a node $u \in T$ has an odd number of followers, it simply matches as many pairs as possible and forwards the one "extra" follower to u's parent. This guarantees all followers of T will find a merge partner, as the root of T either receives no followers, matches two received followers, or sets the single received follower as the merge partner for T itself. Once this PFC wave completes, the root either (i) begins the merge process with a follower T' (if a merge partner was found), or (ii) randomly selects a new role.

Algorithm Sketch 1. The Matching Algorithm for Cluster T

1. If no role, root r_T selects a role uniformly at random: leader or follower.
2. If r_T is a *leader*:
3. r_T uses PFC to set all nodes as *open leaders*
4. Upon completion of the wave, r_T uses PFC set all nodes as *closed leader*
5. Upon completion of the wave, nodes connect all incident followers, and forward to their parent the (at-most-one) unmatched follower
6. r_T matches any received followers
7. r_T either repeats the matching algorithm (if unmatched), or begins merging (if matched)
8. Else
9. r_T selects uniformly at random the role of *long* or *short* follower
10. Nodes in T search for a leader. Short followers search for 2 PFC waves, while long followers search for 12 PFC waves
11. If a leader was found r_T waits to be matched with a merge partner
12. Else T repeats the matching algorithm
13. Endif

Analysis of Matching

Lemma 3. *Consider a cluster T in a configuration G_i. With probability at least $1/4$, all nodes in T will be a potential leader for at least one round in the next $\mathcal{O}(\log N)$ rounds.*

Proof sketch: There are four cases to consider based on the state of T in G_i: T is an open leader, T is a closed leader, T is a follower, and T is merging. If T is an open leader, our claim holds. If T is a closed leader, in $\mathcal{O}(\log N)$ rounds T will either begin a merge (becoming a potential leader) or select the new role of leader with probability $1/2$. If T is a follower, T is a short follower with probability $1/2$, and after $4(\log N + 1) + 4$ rounds T will select a new role of leader with probability $1/2$, or begin a merge (becoming a potential leader). If T is currently merging, then every node will be a potential leader for at least one round during the merging process, which will complete in $\mathcal{O}(\log n)$ rounds.

Lemma 4. *Consider a cluster T in configuration G_i. With probability at least $1/16$, T is assigned a merge partner at least once over $\mathcal{O}(\log N)$ rounds.*

Proof sketch: This proof combines the previous lemma with the fact that a cluster has probability $1/4$ of being a long follower, which will ensure the cluster searches sufficiently long to detect at least one potential leader in a neighboring cluster.

4.4 Merging

After being matched, two clusters can merge. Our merging algorithm adds edges in a systematic fashion to ensure there is enough "bandwidth" for two clusters

to merge quickly, yet still limits degree increases. Our merging algorithm can be discussed from two points of view: one which considers two N-node clusters in the guest network merging into a single N-node cluster, and another which considers two clusters in the host network systematically updating their cluster successors and predecessors. Below, we present a discussion from both viewpoints for clarity. Note these are simply different ways of thinking about the same algorithm.

From the point of view of the guest network, merging can be thought of as (i) connecting guest nodes with identical identifiers from the two clusters, beginning with the roots, (ii) determining which of these guest nodes will remain in the new network (using the ranges of their hosts), and (iii) transferring the links from the "deleted" node to the "winning" node (the node remaining in the single merged N-node cluster). The remaining node can then connect its children with the received children from the former root, and these nodes then repeat the "merge" process. This proceeds level-by-level until only a single N-node cluster remains.

From the point of view of the host network, the merge involves (i) connecting the two hosts of two guest nodes with the same identifiers (beginning with the hosts of the root), and then (ii) updating the cluster ranges of these hosts, transferring any links from the "lost range" of one host node to another. Note that initially the hosts of the roots are connected and the cluster ranges of these two hosts overlap. Given the presence of the other host, each host will update their cluster successor or predecessor (if needed), and the host whose cluster range was reduced will transfer any outgoing intra-cluster links to its new successor/predecessor, effectively allowing one host to "take over" the cluster range of another. This change in the cluster range corresponds to the "deleting" of a guest node discussed above. The hosts from the next level in the tree are then connected and the process repeats recursively until all cluster successors, predecessors, and (by implication) Type 2 edges are updated and a new cluster is formed. The merge algorithm is sketched in Algorithm 2.

Algorithm Sketch 2. The Merging Algorithm for Cluster T

// Cluster T has been assigned merge partner T'
1. Root r_T notifies all nodes of merge partner T' and its view of the random sequence Ψ_r
2. Edges between T and T' are removed if $\Psi_r = \Psi$
3. Beginning with the roots r_T and $r_{T'}$:
4. Node r_T updates its range based upon the identifier of $r_{T'}$ (if needed)
5. Node r_T sends any edges not in its new range to $r_{T'}$, and receives edges from $r_{T'}$
6. Children of r_T and r'_T are connected, and process repeats concurrently
7. Once process reaches leaves, pass feedback wave to new root
8. New root $r_{T''}$ sends PFC wave to update nodes in T'' of new cluster identifier.

Note every merge begins with a pre-processing stage which removes all links between merge partners T and T' other than the edge between the roots of

T and T'. To prevent the network from being partitioned, no edge is deleted unless both incident nodes receive from their respective cluster roots a message matching the shared random sequence Ψ. As this sequence is unknown to the adversary, network partitions are prevented with high probability.

Analysis of Merge

Lemma 5. *Consider two clusters T and T' such that T and T' are merge partners. In $\mathcal{O}(\log N)$ rounds, T and T' have formed a single cluster T'' consisting of all nodes in $T \cup T'$.*

Proof sketch: The proof follows from the fact that the merge process requires $\mathcal{O}(\log N)$ rounds of pre-processing, and then resolves at least one level of the guest network in a constant number of rounds. Since there are $\log N + 1$ levels, our lemma holds.

Lemma 6. *The degree of a node $u \in T$ will increase by $\mathcal{O}(\log^2 N)$ during a merge, and will return to within $\mathcal{O}(\log N)$ of its initial degree when the merge is complete.*

Proof sketch: This proof follows from Theorem 2. For any set of nodes T forming $\text{AVATAR}_{\text{CBT}}$ (including a cluster), a node $u \in T$ has at most $\mathcal{O}(\log N)$ edges amongst nodes in T. As merging involves transferring the $\mathcal{O}(\log N)$ edges from a contiguous portion of $range(u)$ to some node v at most once per level in the guest network, no node will "take over" more than $\mathcal{O}(\log^2 N)$ edges during a merge. Once the merge is completed, any node in the new cluster T'' has at most $\mathcal{O}(\log N)$ edges in T'', again by Theorem 2.

4.5 Termination Detection

Note the root of a cluster repeatedly executes the matching algorithm. For silent stabilization, the root must know when a legal configuration has been reached so it can cease the matching algorithm. For this, we add a "faulty bit" to the feedback wave sent after a merge has completed. If a node (i) detects the configuration is faulty, or (ii) received a faulty bit of 1 from at least one child, the node sets its faulty bit to 1 and appends this to the feedback message sent to the node's parent. If the root receives a feedback wave without the faulty bit set (i.e. a value of 0), it stops executing the algorithm. If a node u completes this wave with its faulty bit set to 0 and it either (i) detects a faulty configuration, or (ii) detects a neighbor with a faulty bit not equal to 0, u will detect a reset fault. This ensures our algorithm is silent and stabilizing.

Lemma 7. *When our algorithm builds a legal $\text{AVATAR}_{\text{CBT}}$ network, the faulty bit will be set to 0, and remain 0 until a transient fault again perturbs the system.*

Proof sketch: Since $\text{AVATAR}_{\text{CBT}}$ is locally checkable, a faulty configuration has at least one detector which will set its faulty bit to 1. By similar argument to Lemma 1, in $\mathcal{O}(\log N)$ rounds, all nodes will have their faulty bit set to 1 and begin executing our algorithm. Once the last merge occurs, no node will detect a fault, and all faulty bits will remain 0 until another fault occurs.

4.6 Combined Analysis

Theorem 3. *The algorithm in Section 4 is a self-stabilizing algorithm for the* $\textsc{Avatar}_{\textsc{Cbt}}$ *network with expected convergence time of* $\mathcal{O}(\log^2 N)$.

Proof sketch: All nodes are members of a cluster in $\mathcal{O}(\log N)$ rounds, at which point the number of clusters will only decrease. Each time a merge occurs, the number of clusters is reduced by 1, and the probability that a cluster merges over a span of $\mathcal{O}(\log N)$ rounds is constant (1/16). In expectation, then, every cluster has merged in $\mathcal{O}(\log N)$ rounds, halving the number of clusters. After $\mathcal{O}(\log^2 N)$ rounds, we are left with a single cluster, which is the legal configuration.

Theorem 4. *The degree expansion of the self-stabilizing* $\textsc{Avatar}_{\textsc{Cbt}}$ *algorithm from Section 4 is* $\mathcal{O}(\log^2 N)$ *in expectation.*

Proof sketch: A node's degree will increase under only a small number of circumstances. A node will only have $\mathcal{O}(\log N)$ edges to nodes in its cluster (except during merges). During a merge, the degree can increase to at most $\mathcal{O}(\log^2 N)$ (Lemma 6). Each time a cluster selects the leader role, a node in the cluster may have its degree increase by 1. As the algorithm will terminate in $\mathcal{O}(\log^2 N)$ rounds in expectation, there are an expected $\mathcal{O}(\log N)$ such increases. Finally, consider an invalid initial configuration which causes a node u to receive many edges while not in a cluster or merging with another cluster. The only way for a node u to receive additional edges in a round and not execute a reset action is for only a single node to add edges to u. Since no node will send more than $\mathcal{O}(\log N)$ edges in a single round, and u will be in a cluster in $\mathcal{O}(\log N)$ rounds, the degree expansion of our algorithm is $\mathcal{O}(\log^2 N)$.

5 Discussion and Future Work

As it is based on a binary tree, $\textsc{Avatar}_{\textsc{Cbt}}$ has poor load balancing properties. However, it can be useful as an intermediate step in creating other topologies using a mechanism we call *network scaffolding*. In this approach, $\textsc{Avatar}_{\textsc{Cbt}}$ is used as an intermediate topology from which another network is built (much like a scaffold is used for construction). Our technique has already been successful in building a self-stabilizing Chord network with polylogarithmic convergence time and degree expansion [2].

To build on this work, we would like to remove the requirement that all nodes know N, perhaps using a self-stabilizing protocol. We also are examining how much state nodes must continuously exchange to guarantee local checkability with *mutable* state, unlike immutable proof labels [12]. Reducing this exchanged state can reduce the maintenance for correct configurations. We are also investigating bounds for the degree expansion to determine how efficient this algorithm is in this self-stabilizing overlay network setting.

Acknowledgments: I would like to thank my advisors, Dr. Sriram V. Pemmaraju and Dr. Sukumar Ghosh, for their guidance and discussions on this paper.

References

1. Aspnes, J., Shah, G.: Skip graphs. In: SODA 2003: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 384–393. Society for Industrial and Applied Mathematics, Philadelphia (2003)
2. Berns, A.: Self-Stabilizing Overlay Networks. Ph.D. thesis, University of Iowa (December 2012)
3. Berns, A.: Avatar: A Time- and Space-Efficient Self-Stabilizing Overlay Network (2015). 1506.0168
4. Berns, A., Ghosh, S., Pemmaraju, S.V.: Building self-stabilizing overlay networks with the transitive closure framework. Theor. Comput. Sci. **512**, 2–14 (2013). http://dx.doi.org/10.1016/j.tcs.2013.02.021
5. Bui, A., Datta, A.K., Petit, F., Villain, V.: State-optimal snap-stabilizing pif in tree networks. In: Workshop on Self-stabilizing Systems, ICDCS 1999, pp. 78–85. IEEE Computer Society, Washington, DC (1999). http://dl.acm.org/citation.cfm?id=647271.721996
6. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
7. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
8. Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. **5**(1), 66–77 (1983). http://doi.acm.org/10.1145/357195.357200
9. Hayes, T., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. Distributed Computing **25**(4), 261–278 (2012). http://dx.doi.org/10.1007/s00446-012-0160-1
10. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: PODC 2009: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 131–140. ACM, New York (2009)
11. Kniesburges, S., Koutsopoulos, A., Scheideler, C.: Re-chord: a self-stabilizing chord overlay network. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011, pp. 235–244. ACM, New York (2011). http://doi.acm.org/10.1145/1989493.1989527
12. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. In: Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, pp. 9–18. ACM, New York (2005). http://doi.acm.org/10.1145/1073814.1073817
13. Leighton, F.T.: Introduction to parallel algorithms and architectures: array, trees, hypercubes. Morgan Kaufmann Publishers Inc., San Francisco (1992)
14. Onus, M., Richa, A.W., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: ALENEX. SIAM (2007)
15. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. **31**(4), 149–160 (2001)