Fall 2012

# Self-stabilizing overlay networks

Andrew David Berns
*University of Iowa*

# SELF-STABILIZING OVERLAY NETWORKS

by

Andrew David Berns

<u>An Abstract</u>

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2012

Thesis Supervisors: Professor Sukumar Ghosh
Associate Professor Sriram V. Pemmaraju

## ABSTRACT

Today's distributed systems exist on a scale that was unimaginable only a few decades ago. Distributed systems now can consist of thousands or even millions of computers spread across the entire world. These large systems are often organized into *overlay networks* – networks composed of virtual links, with each virtual link realized by one or more physical links. *Self-stabilizing overlay networks* promise that, starting from any weakly-connected configuration, the correct network topology is always built. This area of research is young, and prior examples of self-stabilizing overlay networks have either been for simple topologies, or involved complex algorithms that were difficult to verify and extend. We address these limitations in this thesis.

First, we present the *Transitive Closure Framework*, a generic framework to transform any locally-checkable overlay network into a self-stabilizing network. This simple framework has a running time which is at most a logarithmic number of rounds more than optimal, and in fact is optimal for a particular class of overlay networks. We also prove the only known non-trivial lower bound on the convergence time of any self-stabilizing overlay network. To allow fast and efficient repairs for local faults, we extend the Transitive Closure Framework to the Local Repair Framework. We demonstrate this framework by implementing an efficient algorithm for node joins in the Skip+ graph.

Next, we present the Avatar network, which is a generic locally checkable overlay network capable of simulating many other overlay networks. We design a self-stabilizing algorithm for a binary search tree embedded onto the Avatar network, and prove this algorithm requires only a polylogarithmic number of rounds to converge *and* limits degree increases to within a polylogarithmic factor of optimal. This algorithm is the first to achieve such efficiency, and its modular design makes it easy to extend. Finally, we introduce a technique called *network scaffolding*, which builds other overlay network topologies using the Avatar network.

Abstract Approved: ________________________________
Thesis Supervisor

________________________________
Title and Department

________________________________
Date

________________________________
Thesis Supervisor

________________________________
Title and Department

________________________________
Date

# SELF-STABILIZING OVERLAY NETWORKS

by

Andrew David Berns

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2012

Thesis Supervisors: Professor Sukumar Ghosh
Associate Professor Sriram V. Pemmaraju

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Andrew David Berns

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2012 graduation.

Thesis Committee:

Sukumar Ghosh, Thesis Supervisor

Sriram Pemmaraju, Thesis Supervisor

Ted Herman

Kishore Kothapalli

Jon Kuhl

Kasturi Varadarajan

To my family.

## ACKNOWLEDGEMENTS

I would like to thank my advisors, Dr. Sukumar Ghosh and Dr. Sriram V. Pemmaraju. Their guidance for this dissertation, my graduate school career, and my future has been invaluable. I would also like to recognize my committee members, Dr. Ted Herman, Dr. Kishore Kothapalli, Dr. Jon Kuhl, and Dr. Kasturi Varadarajan. Their generous offering of time and feedback has certainly improved this dissertation. Special thanks are also due to all members of the Department of Computer Science at The University of Iowa, including faculty, students, and program administrators. I could not have made it through my doctoral program without their help, and I will carry many memories of them with me.

Finally, I would like to thank my close friends and family for supporting me unconditionally. I have not been alone in enjoying the successes and bearing the stresses of the past years, and I am in debt to those that have journeyed with me. Thank you for going with me to reach my goals.

## ABSTRACT

Today's distributed systems exist on a scale that was unimaginable only a few decades ago. Distributed systems now can consist of thousands or even millions of computers spread across the entire world. These large systems are often organized into *overlay networks* – networks composed of virtual links, with each virtual link realized by one or more physical links. *Self-stabilizing overlay networks* promise that, starting from any weakly-connected configuration, the correct network topology is always built. This area of research is young, and prior examples of self-stabilizing overlay networks have either been for simple topologies, or involved complex algorithms that were difficult to verify and extend. We address these limitations in this thesis.

First, we present the *Transitive Closure Framework*, a generic framework to transform any locally-checkable overlay network into a self-stabilizing network. This simple framework has a running time which is at most a logarithmic number of rounds more than optimal, and in fact is optimal for a particular class of overlay networks. We also prove the only known non-trivial lower bound on the convergence time of any self-stabilizing overlay network. To allow fast and efficient repairs for local faults, we extend the Transitive Closure Framework to the Local Repair Framework. We demonstrate this framework by implementing an efficient algorithm for node joins in the Skip+ graph.

Next, we present the Avatar network, which is a generic locally checkable overlay network capable of simulating many other overlay networks. We design a self-stabilizing algorithm for a binary search tree embedded onto the Avatar network, and prove this algorithm requires only a polylogarithmic number of rounds to converge *and* limits degree increases to within a polylogarithmic factor of optimal. This algorithm is the first to achieve such efficiency, and its modular design makes it easy to extend. Finally, we introduce a technique called *network scaffolding*, which builds other overlay network topologies using the Avatar network.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF ALGORITHMS

# CHAPTER 1
# INTRODUCTION

Now more than ever, network connectivity is ubiquitous for many. Not only are residential broadband connections common, mobile network connections are also seemingly omnipresent through Wi-Fi hot-spots and cellular data networks. For many, connecting to another computer anywhere in the world is simply a matter of pressing a few buttons, anywhere, anytime. The scale of today's network connectivity is impressive.

This new connectivity has brought about a new paradigm for distributed computing. Now communication need not be modeled as occurring over a few static links, but rather can be modeled as using any number of dynamic virtual links, each consisting of one or more physical links. Networks that use these virtual links are called *overlay networks*. The Internet – itself an overlay network – hosts many other overlay networks, such as Gnutella [33], BitTorrent [8], and Skype [35].

Many overlay networks, including those hosted by the Internet, operate in dynamic environments. These environments present an interesting challenge for fault-tolerant computing. Faults are frequent occurrences, node membership in the system may change dramatically in a short amount of time, and the system size may be quite large. Early research into structured overlay networks (networks with exactly one correct configuration) was ill-equipped to operate in these fragile environments. Recently, however, a classic fault-tolerance paradigm has been combined with overlay networks to elegantly handle the complex nature of faults inherent in these networks. These new networks are called self-stabilizing overlay networks.

*Self-stabilizing overlay networks* are overlay networks that can form a correct topology regardless of their current state. While there has been several self-stabilizing overlay networks proposed, the field is still new, and generic techniques for building self-stabilizing networks (and proving them correct) are still absent, as are results on efficient stabilization. This thesis hopes to address this missing piece of the self-stabilizing overlay network puzzle.

## 1.1 Related Work

Self-stabilizing overlay networks bring together two areas of distributed computing, one established and one more recent. In this section, we review this prior work.

### 1.1.1 Self-Stabilization

First introduced by Dijkstra in 1974 [13], self-stabilization has been a well-studied and productive area of research for several decades. *Self-stabilization* is the ability of a system to recover from any transient fault, usually modeled as the system starting in an arbitrary initial state. Specifically, a self-stabilizing system satisfies two properties: closure and convergence [1]. Informally, *closure* is the property that a system in a legal configuration remains in a legal configuration when executing program actions, while *convergence* is the property that a system in an illegal configuration will eventually reach a legal configuration by executing program actions.

More formally, let $\mathcal{P}$ be a predicate defined over the state of a system $\mathcal{S}$. We say that $\mathcal{S}$ is in a *legal* or *valid* state if $\mathcal{P}$ evaluates to *true*, and $\mathcal{S}$ is in an *illegal* or *invalid* state otherwise. Let every process in $\mathcal{S}$ execute some program $\mathcal{A}$. We say that $\mathcal{A}$ satisfies the *closure* property if, from any state $S$ where $\mathcal{P}$ is *true*, executing actions of $\mathcal{A}$ always results in a configuration $S'$ where $\mathcal{P}$ remains *true*. We say that $\mathcal{A}$ satisfies the *convergence* property if, from any state $S$ where $\mathcal{P}$ is *false*, executing actions of $\mathcal{A}$ will always eventually result in a configuration $S'$ where $\mathcal{P}$ is *true*. A self-stabilizing algorithm satisfies both closure and convergence.

#### 1.1.1.1 Randomization

Many self-stabilizing algorithms rely upon randomization. These randomized algorithms rely on the output of a random number generator. Randomization is a powerful technique that may provide solutions to problems that are unsolvable with deterministic algorithms. The use of randomized algorithms does, however, complicate analysis of worst-case convergence time and state space bounds, as now performance must be measured with respect to either expected performance or with high probability (or both).

To see the power of randomization, consider the problem of orienting the edges of a ring. This problem begins with each node being a member of a ring topology. A node has two incident edges, and these edges must be "directed" – that is, each edge must have a source node and destination node designated such that a cycle exists in the network going only from sources to destinations. Said another way, a legal configuration has exactly one incoming and one outgoing edge incident upon every node. Israeli and Jalfon [22] consider ring orientation with anonymous nodes (that is, nodes without identifiers), and show a solution is impossible with a deterministic algorithm. They then prove orienting a ring *is* possible using randomization, and provide a two-part algorithm to do so. In the first part, every pair of neighbors agrees on a direction for their shared incident edge using randomization. In part two, nodes create "tokens" based upon their incident edges, and these tokens are circulated around the system, being removed when certain conditions are met. Once all remaining tokens are being forwarded in the same direction, and every node has received at least one of these tokens, the system is in a correct configuration. This example demonstrates how randomization allows self-stabilizing solutions to problems that would be impossible with a deterministic algorithm.

As another example, consider several randomized versions of Dijkstra's original work in self-stabilization [13], which circulated a token around a ring. Dijkstra's algorithm guaranteed that, regardless of how many "tokens" were originally present, eventually only a single token would exist, and this token would be circulated amongst all nodes infinitely often. Dijkstra's (deterministic) algorithm required a minimum of $n$ states per node. In 1992, Herman [21] designed a randomized algorithm that required only 3 states per node. In 2009, we designed a randomized algorithm which circulated $k$ tokens around a ring using only $k + 1$ states, which we proved was optimal [4]. Both of these results would be impossible with a deterministic algorithm. Randomization can increase the convergence time, however – for instance, Nakata [28] proved the convergence time for Herman's 3-state token ring algorithm was $O(n^2)$.

#### 1.1.1.2 Transformers

One way to create self-stabilizing systems is to use a *transformer* – an algorithm that takes any distributed algorithm as input, and produces a self-stabilizing version of the algorithm as output. Transformers usually sacrifice either convergence time or state space requirements to guarantee stabilization, but they significantly simplify the task of the algorithm designer.

Katz and Perry presented one of the first transformers in 1993 [24]. This transformer works in two steps. First, a global snapshot is taken of the system by a designated process. Next, this designated process decides if the system is in a legal configuration, and issues a global *reset* command if it is not in a legal configuration. The reset command brings every node in the system back to some predetermined legal state. This global reset is simple and self-stabilizing. However, it requires snapshots to be continuously collected, even in a legal state.

Awerbuch et al. presented another transformer in 1994 [3]. Their transformer relies on processes having the ability to check *locally* if there was a fault, without gathering the state of the entire system. With this local checking, a faulty configuration leads to the issuance of an *internal reset* – that is, a reset that is initiated by any node in the system. After issuing an internal reset, nodes must allow sufficient time for the system to correct itself before another reset is issued.

#### 1.1.1.3 Fault Containment

Certain refinements of self-stabilization are concerned with performance measures besides worst-case convergence time or state space requirements. For example, *fault containment* promises that a system with a single fault requires only a constant time for its output to converge. Considering that the time to failure in modern computing hardware is sufficiently long enough that most faults are isolated incidents, fault containment is useful from both an applied and theoretical standpoint.

The first fault containment algorithm was presented by Ghosh et al. in 1996 [19]. This paper presented several measures to evaluate fault containment. The *containment time* is the time it takes the system's primary variables (the output variables of the system) to reach a legal state after a single fault. Similarly, the *contamination number* is the number of processes that must change their state during the recovery from a single fault. The *fault gap* is the minimum distance two faults must be apart so that each fault is repaired in constant time. Finally, the *containment space* is the extra state space required to implement a fault-containing algorithm. Ghosh et al. also presented a transformer for taking a non-reactive self-stabilizing system and converting it to a fault-containing system. This was done using two protocols. The first protocol takes a system with a single fault and, in constant time, sets the primary variables to satisfy a legal configuration. The second protocol converges the remaining system state.

#### 1.1.1.4 Superstabilization

Some self-stabilizing algorithms do not restrict the states of the system during convergence – any arbitrary behavior is acceptable until a legal configuration is reached. *Superstabilization*, first introduced by Dolev and Herman in 1997 [14], guarantees some predicate holds during convergence.

Superstabilization focuses on dynamic systems, where topology changes caused by link failures or node crashes should be expected. Such changes ideally would not cause the entire system to “start over” with convergence. A superstabilizing system is a self-stabilizing system that also guarantees a weaker predicate holds during a set of topology changes originating in a legal configuration (that is, the system was in a legal configuration, and then the topology changed). This weaker predicate is called the *passage predicate*, while the time it takes the system to return to a legal state after the topology change is called the *superstabilization time*. Finally, the number of processes that must change state as a result of the topology change is called the *adjustment measure*. Superstabilization quantifies not only how the system behaves after a topology change, but also how much time and space is expended during this convergence.[1]. Dolev and Herman also offer several examples of superstabilization. One of these examples addresses the problem of graph coloring. A legal graph coloring is an assignment of colors to nodes such that no node in the graph has a neighbor assigned the same color. Dolev and Herman present a superstabilizing algorithm with

[1]Notice that fault containment provides quick stabilization after any transient fault, while superstabilization allows topology changes, as well as guarantees the passage predicate

a passage predicate that allows nodes with a special color ($\perp$) to be neighbors. The algorithm considers topology changes where either a node crashes or resumes execution, or a link recovers (at most $k$ links from some node $p$ are allowed to recover when $p$ resumes execution). After the occurrence of such a topology change, some process $t$ (selected using a set of given rules) sets its color to $\perp$. With these modifications, a superstabilizing algorithm is possible with superstabilization time of $O(k)$, and an adjustment measure of $(k + 1)$. From a legal configuration, then, this system can undergo a change in topology and still satisfy the passage predicate, and within $O(k)$ time steps, will again be a legally-colored graph.

#### 1.1.1.5 Snap-Stabilization

*Snap stabilization* is another refinement of self-stabilization. Snap-stabilization guarantees the system *always* satisfies the problem specification predicate. These predicates are based upon external requests – the system receives a request, processes this request, and "returns" a legal value. In this way, snap-stabilization is an "as-needed" stabilization – if the system does not receive a request, it need not stabilize.

The first snap-stabilizing algorithm was presented by Bui et al. in 1999 [9], and is built on top of the propagation of information with feedback (PIF) scheme. In PIF, the root processor of a tree initiates a broadcast phase by broadcasting a message to all its children. These children in turn broadcast this message to all of their children. This continues until the broadcast reaches a leaf node, at which point the leaf node initiates the feedback phase by sending feedback to its parent. Once a parent receives feedback from all of its children, it sends feedback to its parent, until finally, the root node receives feedback from all of its children. At this point, the root has aggregated feedback from all nodes in the tree. To make PIF snap-stabilizing, the broadcast and feedback stages are augmented with a third stage called the *cleaning* stage. This cleaning stage is executed after a node has sent feedback and before it initiates another broadcast. This additional stage makes PIF snap-stabilizing – that is, every request from the root to propagate some information and request feedback completes successfully.

Delaët et al. consider snap-stabilization with a message-passing communication model [12]. They show that, if channels are allowed unbounded capacity, then snap-stabilization is impossible for problems with a safety-distributed specification. A *safety-distributed specification* is one whose safety property can only be violated by considering the behavior of multiple processes. A classic example is mutual exclusion – the safety property can only be violated if more than one process is in the critical section at the same time. If channel capacity is allowed to be unbounded, then there may be an arbitrary number of messages in the channel upon initialization, which can "trick" a node into believing neighbors are in a correct configuration, which can enable a node to execute an incorrect action. If channels are bounded, snap-stabilizing solutions are possible by the use of a data link protocol similar to the one presented by Dolev and Tzachar [15]. This data link protocol requires a message to be received several times before it is "accepted" by the algorithm (specifically, a message is not considered valid until it is received $2 \cdot k + 1$ times, where $k$ is the channel capacity).

### 1.1.2 Overlay Networks

Overlay networks are networks composed of *logical* links, where each logical link consists of one or more *physical* links. The canonical example of an overlay network is the Internet – computers usually have a single physical connection (e.g. to a router), but make many logical connections (e.g. to various web servers). Using logical links allows for the creation and deletion of edges in the network, meaning specific topologies can be created which ensure desirable properties, such as low diameter, low node degree, and efficient routing.

Overlay networks can be partitioned into two sets: structured and unstructured. In an *unstructured* network, there are no specific requirements concerning the desired topology (except that the network cannot be partitioned). Routing is often quite inefficient with such networks since there are no rules for determining the best path (or even any path) between two nodes. In some cases, routing is reduced to executing a random walk. While routing efficiently may be difficult, maintaining the "correct" topology is trivial, since any topology suffices. A popular unstructured overlay network is the Gnutella file-sharing network [33]. While unstructured networks have no strict topological constraints, they may focus on *self-optimization* – modifying their topologies not to reach a specific configuration, but rather to ensure some operations are completed efficiently (for instance, queries for objects).

The other type of overlay networks are structured network. *Structured* networks have constraints on the network topology. Specifically, structured networks have a single correct configuration. Examples of structured overlay networks include

Chord [36] and Skip+ [23]. Structured networks usually allow for efficient routing and low node degree. Many structured networks are organized as *distributed hash tables*, or DHTs, where nodes and objects generate identifiers through the use of a hash function, and every object is stored at a particular node based upon their identifiers. The canonical example of a structured P2P network is the Chord network, a ring-based network with additional links (discussed in Chapter 2).

Overlay networks may be expected to operate in hostile environments where node and link failures are common. It is not surprising, therefore, that researchers have considered several forms of fault tolerance in overlay networks. Many overlay networks include protocols to manage *churn* – the addition and deletion of nodes from a valid system. Some of these protocols, nodes are assumed to leave "gracefully" – that is, leave after executing a leave protocol. Other protocols, such as the one from Kuhn, Schmid, and Wattenhofer [26], ensure that the network continues to operate (and no data is lost) even after nodes crash. Skip graphs [2] consider not only adversarial failures (i.e. a powerful adversary forcing nodes to crash), but also random failures, showing that 85% of queries complete successfully even when nodes have a 50% chance of failure. Notice that these methods of fault tolerance, however, are for a subset of possible failures. Specifically, most overlay network protocols consider fault tolerance only in cases where the network begins in a correct state, and node addition and deletion is bounded for any particular time period.

### 1.1.3 Self-Stabilizing Overlay Networks

Self-stabilizing overlay networks provide an elegant solution to fault tolerance in overlay networks by guaranteeing that, from *any* weakly-connected topology, the correct configuration is reached. Work on self-stabilizing overlay networks began in 2005 with two works: the Iterative Successor Pointer Rewiring Protocol [11] and the Ring Network [34]. Both algorithms create a ring topology from any weakly-connected initial state. Each algorithm requires nodes to constantly "search" the network to discover faults, which means, even in correct configurations, nodes must continue to exchange messages.

*Silent self-stabilizing algorithms* avoid this continuous search. A silent self-stabilizing algorithm has no enabled actions in the correct configuration. For silent self-stabilizing algorithms to exist, the overlay network being built must be *locally checkable*. Informally, this means that all incorrect configurations contain at least one node who can detect (based upon a predicate $P$, defined over a node's local state, evaluating to *false*) that the network is incorrect, while all nodes in the correct configuration evaluate $P$ to *true*. One of the first silent self-stabilizing overlay network algorithms was published in 2007 by Onus et al. [30]. This work builds the Linear topology, which is a "line" graph of nodes sorted by identifier. The authors present two algorithms for Linear which stabilize in linear time. They also present a modified Linear topology which uses "shortcut neighbors" to achieve logarithmic diameter. The authors run several simulations which suggest the convergence time of their algorithm is logarithmic. The Linear topology was examined again by Gall et al. [17], with the authors demonstrating the effects on convergence time if every node could execute only a constant amount of work per round (where *work* is defined as adding or deleting an incident edge). Gall et al. found that, with a worst-case scheduling of actions while limiting work to a constant, convergence requires time polynomial in $n$.

In 2009, Jacob et al. presented a silent self-stabilizing algorithm for creating a Skip+ graph (a locally-checkable variant of the Skip graph). Their algorithm runs in a polylogarithmic number of rounds. Besides being self-stabilizing, the authors show their protocol can resolve node joins and leaves in a logarithmic number of rounds using only a polylogarithmic amount of work.

A self-stabilizing algorithm for the Chord network was created by Kniesburges et al. in 2011 [25]. In this work, the authors present a locally checkable version of the Chord network called Re-Chord (for "reactive Chord"). The Re-Chord network is defined over a set of both real and virtual nodes. Every real node simulates zero or more virtual nodes, with each node determining which nodes to simulate based upon the distribution of other nodes in the network. The algorithm works by taking all nodes (real and virtual) and forming a Linear graph, and then executing additional actions to create the edges necessary for the Chord network. The convergence time is $\mathcal{O}(n \log n)$ due to the linearization step (and the presence of $\log n$ virtual nodes).

Also in 2011, Richa et al. presented a self-stabilizing version of the de Bruijn graph called the "Linearized De Bruijn Network" (LDB) [32]. This work again relies upon virtual nodes, although in this case only two virtual nodes are created per real node. The LDB network is *not* locally checkable, however, and the authors rely upon a "linear probing" scheme to constantly search the network for faults. This linear probing scheme requires $\mathcal{O}(n)$ time to complete, again resulting in linear convergence time for LDB.

## 1.2 Contributions

In this thesis, we provide several new results in the area of self-stabilizing overlay networks. Specifically, we show the following three results:

1. In Chapter 3, we present the *Transitive Closure Framework*. This framework transforms any locally-checkable overlay network into a self-stabilizing network, and has a running time which is within at most a logarithmic number of rounds from optimal. We extend this framework to allow for the efficient repairs of a subset of faults, which we call the *Local Repair Framework*.

2. In Chapter 4, we present the Avatar network, a generic locally checkable overlay network which simulates other overlay networks. Using a topology based upon an embedded binary search tree, we present a self-stabilizing algorithm which converges in only a polylogarithmic number of rounds *and* limits degree increases to within a polylogarithmic factor of optimal. This algorithm is the first to achieve such efficiency.

3. In Chapter 5, we introduce a technique called *network scaffolding*, which builds other overlay network topologies using the Avatar network. We show how one can easily create a self-stabilizing Chord overlay network with convergence time which is polylogarithmic while limiting degree increases.

# CHAPTER 2
# MODEL AND PROBLEM SPECIFICATIONS

In this chapter, we formally define the model of computation and the self-stabilizing overlay network problem. We also discuss appropriate complexity measures for the overlay network model. Finally, we review several common overlay networks used throughout the remainder of this thesis.

## 2.1 Model of Computation

We model the distributed system as an undirected graph $G = (V, E)$, with $V$ being the set of nodes representing the processors of the distributed system, and $E$ being the set of edges representing communication links. Every node $u \in V$ is mapped to a unique identifier, and we use the function $\texttt{id} : V \to \mathbb{Z}^+$ to access this identifier. For some systems, every node also is assigned a random string of bits, which we access through the function $\texttt{rs} : V \to \{0, 1\}^*$. We shall see in Section 2.4 how these random bits are used. We shall use $id_u$ and $rs_u$ to denote the identifier and random sequence associated with node $u$, which node $u$ stores as immutable data. Where clear from context, we also refer to nodes simply by their identifier (e.g. node 10 is a node $u$ such that $id_u = 10$). Let $\lambda$ be the collection of nodes and associated identifiers and random sequences ($\lambda = (V, \texttt{id}, \texttt{rs})$).

Every node in the network has a set of local variables called its *state*. The set of all node states at some time $i$ is called a *configuration* and is denoted $S_i$. The initial configuration is denoted $S_0$. A node may change its state by executing a *program*. We use a synchronous model of computation – that is, in one *round*, all nodes execute their programs, potentially changing their state. A *computation* is a series of configurations $S_0, S_1, \ldots, S_c$ such that, for all configurations $0 \leq i < c$, $S_{i+1}$ is reachable from $S_i$ by every node executing its program. We say a self-stabilizing algorithm is *silent* if all computations from any initial state are finite – that is, all computations result in a configuration $S_c$, for some finite $c$, such that $S_c$ is a valid state, and no node changes the configuration by executing its program. Informally, one can think of a silent algorithm as an algorithm that reaches the correct configuration and then "ceases" execution (i.e. does not change the configuration).

A node $u$ can communicate with any node in its *neighborhood* $N(u)$ by sending the nodes (called *neighbors*) a message. We use a synchronous message passing model of communication where messages sent to node $u$ in round $i$ are received by node $u$ at the start of round $(i + 1)$. Messages may be of unbounded size, although the preference is to limit them to polylogarithmic in the number of nodes in the network. We assume reliable communication channels where all sent messages are received and all received messages are sent.

In the message passing communication model, "silent" self-stabilizing algorithms may continue to exchange messages even in the correct configuration. Without this, nodes cannot detect when the configuration is correct and when it is incorrect. Therefore, when we speak of silent self-stabilization, we shall mean the computation terminates *except for* the exchange of messages used to determine the state of a node's neighbors. A silent self-stabilizing algorithm in the message passing model should reach a configuration $S_c$ such that the only action executed by every node in every round is to exchange information concerning a node's local state.

For our overlay network model, we shall assume a node's neighborhood $N(u)$ is part of a node's state. We assume that a node's view of its neighborhood matches its actual neighborhood in the network – that is, no node can have a neighbor which it is not aware of, and no node believes it has a neighbor which does not exist (in the synchronous model, we can assume all nodes exchange a "heartbeat" message to ensure this). Since a node's neighborhood is part of its state, a node's neighborhood can change due to program actions. This means that the neighbors of a node $u$ in configuration $S_i$ may be different than the neighbors of $u$ in configuration $S_j$, changing $u$'s knowledge of the network. This ability to change the network topology is quite different from traditional models of distributed computation such as $\mathcal{CONGEST}$ and $\mathcal{LOCAL}$ [31].

Changes in the network topology can happen in one of two ways: in a round $i$, a node $u$ can delete any edge incident upon it, as well as add an edge between any two neighbors. 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 $(w, v)$ if $(u, w), (u, v) \in E(G_i)$, resulting in $(w, v) \in E(G_{i+1})$. Restricting edge creation to between nodes at distance 2 is reasonable since, in practice, the address space of nodes is so large as to make blind "probes" impractical (e.g. a computer cannot reasonably expect to find a neighbor by randomly sampling IP addresses in the network). To maintain the decentralized nature of overlay networks, we also do not assume the existence of an oracle that can return a link to a node of distance greater than 2 (for instance, there is no centralized server containing identities of nodes in the network).

## 2.2 Definition of Self-Stabilizing Overlay Networks

We define a *family of overlay networks* as a mapping $ON : \Lambda \to \mathcal{G}$, where $\Lambda$ is the set of all $\lambda = (V, \texttt{id}, \texttt{rs})$, and $\mathcal{G}$ is the set of all undirected graphs. For any set of nodes $V$ and associated identifiers and random sequences, then, there is exactly one "correct" network $ON(\lambda)$. We say that an overlay network algorithm is *self-stabilizing* if it takes a weakly-connected network $G = (V, E)$ as input and transforms it through program actions into $ON(\lambda)$.

One can consider the process of stabilization as a walk through the state space of a system. Program actions move the system configuration along a path towards a stable state. Early work on stabilization focused on finding *any* path to a legal state. Later work focused on finding *short* paths from all configurations to a legal configuration. In this thesis, we will consider both short paths, and short paths that only go through intermediate states with low degree. To date, the existence of such paths, and how to design a program to realize them, has not been considered.

We define $N_{ON(\lambda)}(u)$ to be the neighborhood of node $u$ in $ON(\lambda)$. We say that $u$ is *faulty* in configuration $S_i$ if $N(u)$ in configuration $S_i$ is not equal to $N_{ON(\lambda)}(u)$. Similarly, we say the network is *faulty* in configuration $S_i$ if there exists at least one faulty node in $S_i$. It is possible that a node is faulty in the network but cannot detect it locally due to limited network knowledge. Specifically, a node $u$ only knows (i) its local state (including $N(u)$), and (ii) the messages it has received from its neighbors. This knowledge may be insufficient to detect that $N(u) \neq N_{ON(\lambda)}(u)$.

### 2.2.1 Local Checkability

A family of overlay networks $ON$ is *locally checkable* if and only if all faulty configurations contain at least one node $u$ that detects the configuration is faulty using its local information. Such a node is called a *detector*. As stated earlier, this "local" information consists of (i) a node $u$'s local state, including the neighbors of $u$, and (ii) the messages received from $u$ from nodes in $N(u)$. For a silent self-stabilizing algorithm to exist for an overlay network family $ON$, $ON$ must be locally checkable.

An interesting question is, given a locally checkable overlay network $ON$, what messages need to be exchanged between nodes so that nodes can detect a fault? One approach is to simply have every node $u$ share all of its state information with its neighbors. In this approach, every node knows its 2-neighborhood. Specifically, let the set of nodes within distance 2 of node $u$ be $N^2(u)$, and let the associated identifiers and random sequences be $\mathtt{id}_u^2$ and $\mathtt{rs}_u^2$, respectively. Furthermore, let $E(u)$ be the set of edges between nodes in $u$'s 2-neighborhood – that is, $E(u) = \{(u,v)|v \in N(u)\} \cup \{(v,w)|v \in N(u) \wedge w \in N(v)\}$. Note that the size of messages that are exchanged even in a correct (silent) state can be quite large – at least a logarithmic factor of the size of $N(u)$.

To save bandwidth, we examine in Chapter 4 of this thesis how we can maintain the local checkability of a network while exchanging only a (relatively) small amount of information between nodes. Specifically, every node maintains a string of bits (called the *proof label*), which it exchanges with all of its neighbors in every round. In this case, a silent self-stabilizing algorithm eventually reaches a state where only small messages are being exchanged between neighbors. These messages can accompany the heartbeat messages, for instance. We describe proof labels in more detail in Chapter 4.

The local checkability of an overlay network has played a major role in current self-stabilizing overlay network research. Because many overlay networks are *not* locally checkable, to achieve silent stabilization, researchers have had to first create a locally checkable version of an overlay network, and *then* create the algorithm. For its prominent role in self-stabilizing overlay networks, local checkability has received little formal attention – no current work offers explicit proofs of local checkability, nor a formal definition with respect to overlay networks.

## 2.3 Measures of Complexity

Many of the traditional complexity measures for distributed computing apply to overlay networks. Perhaps the most common measure is the time required for an algorithm to bring the system into a legal configuration. Our model uses synchronous networks where computation proceeds in rounds. In every round, every node in the network receives messages sent to it in the prior round, executes all enabled actions, and sends messages as specified by program actions. The *round complexity*, also called the *stabilization time* or *convergence time*, for a network $G$ and algorithm

$\mathcal{A}$ is the number of synchronous rounds required for the algorithm $\mathcal{A}$ to reach the correct configuration $ON(\lambda)$ when starting from $G = (V, E)$ and $\lambda = (V, \texttt{id}, \texttt{rs})$. The convergence time for algorithm $\mathcal{A}$ is the maximum convergence time for network $G$ taken over all $G \in \mathcal{G}$. We assume that overlay networks may consist of a large number of nodes. Therefore, a "good" algorithm should have convergence time that is sub-linear in $n$, with logarithmic in $n$ being the best achieved to date.

One measure of complexity that has been proposed specifically for overlay networks is work. *Work*, first introduced by Gall et al. [17], is a representation of an "edge action" – the creation or deletion of an edge in the network. If we let $E_i$ represent the edges present in the network in round $i$, the work executed in round $i$ is $|\{(u,v) \in (E_i \setminus E_{i-1})\}| + |\{(u,v) \in (E_{i-1} \setminus E_i)\}|$. Gall et al. examined how limiting each node to only a constant amount of work in each synchronous round altered convergence time. They found convergence time may increase greatly when work per node is limited to a constant ($\mathcal{O}(n)$ to $\mathcal{O}(n^2)$ for the Linear network.

While the majority of the inter-node actions executed by some algorithms are in fact edge creation or deletion (such as with Linear and Skip+), some algorithms execute a large number of actions that exchange messages without creating an edge. In these cases, the measure of work seems arbitrary. For instance, an algorithm could send messages to gather the network's state, which could require a quadratic number of messages but no work. Furthermore, a node could add a linear number of edges in $\mathcal{O}(n)$ rounds and still execute only a constant amount of work per round, even though their maintenance overhead (e.g. heartbeat messages on each link) for the

$\mathcal{O}(n)$ edges is very high.

We present a new overlay network complexity measure for analyzing overlay network algorithm space requirements. Let $G$ be a graph with nodes $\lambda = (V, \mathtt{id}, \mathtt{rs})$, and let $\Delta_G$ be the maximum degree of nodes in $G$. For a self-stabilizing algorithm $\mathcal{A}$ executing on an overlay network $G = (V, E)$, 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 the *degree expansion of* $\mathcal{A}$ *on* $G$ as $\mathit{DegExp}_{\mathcal{A},G} = (\Delta_{\mathcal{A},G} / \max(\Delta_G, \Delta_{ON(\lambda)}))$. The degree expansion of $\mathcal{A}$ on $G$ measures the maximum degree any node $u \in V$ has during execution of the algorithm *relative to the maximum degree a node must have* – either their initial degree in network $G$, or their degree in the correct configuration $ON(\lambda)$. Let the *degree expansion of* $\mathcal{A}$ be $\mathit{DegExp}_{\mathcal{A}} = \max_{G \in \mathcal{G}}(\mathit{DegExp}_{\mathcal{A},G})$. Informally, the degree expansion of an overlay network algorithm is the largest any node degree in any initial network may grow "unnecessarily" relative to the initial and final configurations.

The degree expansion of a self-stabilizing overlay network algorithm is a more appropriate measure than work. In modern overlay networks, the cost of creating or deleting a link seems to be of similar magnitude to maintaining an existing link. Therefore, provided a node's neighborhood always remains "small", any operation – edge addition, edge deletion, exchanging messages – can be done quickly. Efficient overlay network creation should limit degree expansion to be polylogarithmic.

### 2.3.1 Modeling the Adversary

The measures listed above all measure an algorithm's "worst-case" performance. Specifically, they take the maximum value for each measure over all graphs $G \in \mathcal{G}$. There may be only a few configurations in $\mathcal{G}$, however, that realize these worst-case values. We often think of the initial network $G$ as being created by some *adversary*. The measures presented above all assume an omnipotent (capable of modifying any state variable) and omniscient (capable of reading any state variable) adversary that can connect nodes in any arbitrary fashion (provided they remain weakly-connected) and has full knowledge of every $\lambda = (V, \texttt{id}, \texttt{rs})$. This powerful adversary can always create a network that realizes the worst-case convergence time.

There are, however, weaker adversarial models. One common weaker adversary is the random-sequence-oblivious adversary. This adversary can weakly connect the nodes in any fashion, but only has knowledge of $V$ and `id`, not of `rs`. The complexity measures above are usually then given with some probability when using this adversary. This is due to the fact that the configuration realizing the worst-case performance may depend upon a particular distribution and arrangement of random sequences, which the adversary cannot control. Unless otherwise noted, however, the results here are all for the stronger adversary with knowledge of $\lambda$ (as are all prior results in the area).

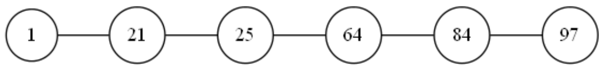


Figure 2.1: A 6-node Linear network.

## 2.4 Network Specifications

Throughout this work we will use several overlay networks frequently as examples. We present these networks here and reference them in later chapters.

### 2.4.1 The Linear Network

Presented in 2007 by Onus et al. [30], the Linear network is a "sorted" line graph. Specifically, every node keeps a link to its successor (if such a node exists) $succ(id_u) = \arg\min_{w \in V}(id_w > id_u)$ and to its predecessor (if it exists) $pred(id_u) = \arg\max_{w \in V}(id_w < id_u)$. The correct Linear network, then, has diameter $n - 1$. A Linear network consisting of 6 nodes is given in Figure 2.1.

The Linear network is locally checkable. Every node simply checks to see if it has at most two neighbors – one with a larger identifier, and one with a smaller identifier. All incorrect weakly-connected configurations violate at least one of these conditions – either a node has too many neighbors, or a node has more than one neighbor with larger or smaller identifiers.

### 2.4.2 Skip Graphs

The Skip graph, a distributed implementation of the Skip list data structure, was first introduced in 2003 by Aspnes and Shah [2]. The graph consists of a series of "levels", with the neighbors of a node $u$ at level $i$ being determined by both a node's random sequence $rs_u$ and the node's identifier $id_u$. To define Skip graphs, we introduce some additional notation adapted from Jacob et al. [23].

- $pre_i(u)$: for any node $u$ and nonnegative integer $i$, $pre_i(u)$ denotes the leftmost (most significant) $i$ bits of $rs_u$
- $pred(u, W)$: for any node $u$ and subset $W$ of nodes, $pred(u, W)$ is the node in the set $W$ with largest $id$ whose $id$ is less than $id_u$ (that is, node $u$'s predecessor from the set $W$). If no such node exists, $pred(u, W) = \perp$.
- $succ(u, W)$: for any node $u$ and subset $W$ of nodes, $succ(u, W)$ is the node in the set $W$ with smallest $id$ whose $id$ is more than $id_u$ (that is, node $u$'s successor from the set $W$). If no such node exists, $pred(u, W) = \perp$.

A Skip graph consists of levels $0, 1, 2, \ldots, L$, where $L = |rs|$. At level $i$, a node $u$ is neighbors with $pred(u, \{w | pre_i(w) = pre_i(u)\})$ and $succ(u, \{w | pre_i(w) = pre_i(u)\})$. It is often assumed that nodes are capable of extending their random sequences until every node's random sequence is unique. With this assumption, Skip graphs have logarithmic degree and diameter with high probability. A Skip graph is given in Figure 2.2 (nodes are ordered left-to-right by increasing identifiers, which are not listed).

It is impossible, however, to create a silent self-stabilizing algorithm for the Skip graph, since Skip graphs are *not* locally checkable. To see this, consider Figure 2.3, which is a faulty Skip graph with no detectors. The graph has a single fault – an edge should exist between the node with identifier 1 and 22. However, no node has

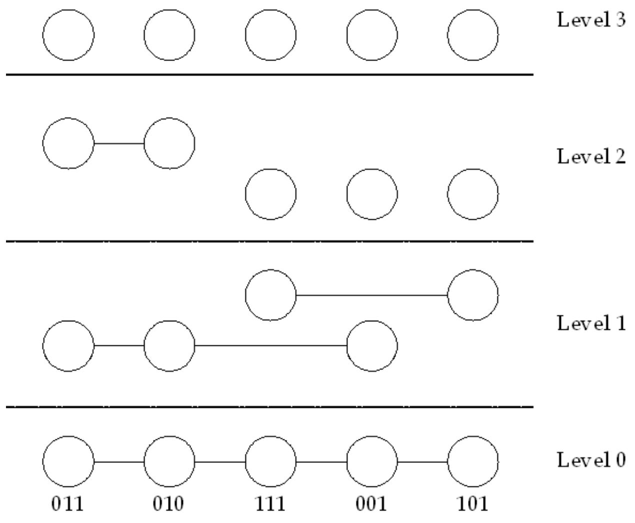


Figure 2.2: A Skip Graph.

knowledge of both 1 and 22, and therefore no node is a detector.

### 2.4.3 The Skip+ Network

Jacob et al. [23] augmented the Skip graph with additional edges to create a locally-checkable graph appropriately named the Skip+ graph. In addition to the edges of the Skip graph, every node in a Skip+ graph maintains edges in each level $i$ to "verify" its neighbor at level $i + 1$. Before presenting a formal definition, we introduce the following terms, again adapted from the work of Jacob et al. [23].

- For node $u$, nonnegative integer $i$, and $x \in \{0, 1\}$,

$$pred_i(u, x) = pred(u, \{w \mid pre_{i+1}(w) = pre_i(u) \cdot x\})$$

  In words, $pred_i(u, x)$ is the predecessor for $u$, selected from all nodes who have the same length-$i$ random sequence prefix as $u$ and whose $(i + 1)$ bit of their

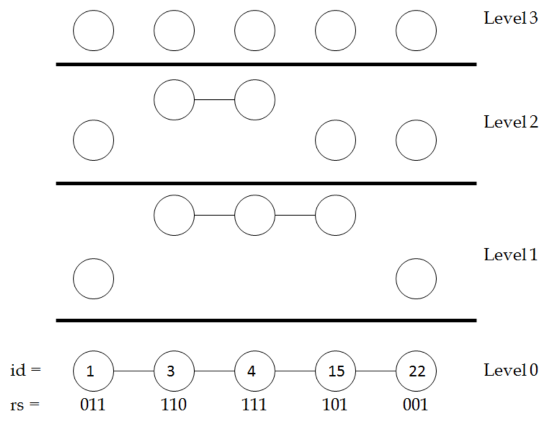


Figure 2.3: A faulty Skip configuration with no detectors.

random sequence is $x$.

- For node $u$, nonnegative integer $i$, and $x \in \{0, 1\}$,

$$succ_i(u, x) = succ(u, \{w \mid pre_{i+1}(w) = pre_i(u) \cdot x\})$$

- For node $u$ and nonnegative integer $i$,

$$low_i(u) = \min\{id_{pred_i(u,0)}, id_{pred_i(u,1)}\}$$

- For node $u$ and nonnegative integer $i$,

$$high_i(u) = \max\{id_{succ_i(u,0)}, id_{succ_i(u,1)}\}$$

- For node $u$ and nonnegative integer $i$,

$$range_i(u) = [low_i(u), high_i(u)]$$

Like Skip graphs, the Skip+ graph consists of levels $0, 1, \ldots, |\mathtt{rs}|$. In a level $i$, node $u$ has edges to all nodes $v$ such that $id_v \in range_i(u)$ and $pre_i(u) = pre_i(v)$. A legal Skip+ graph is shown in Fig. 2.4.

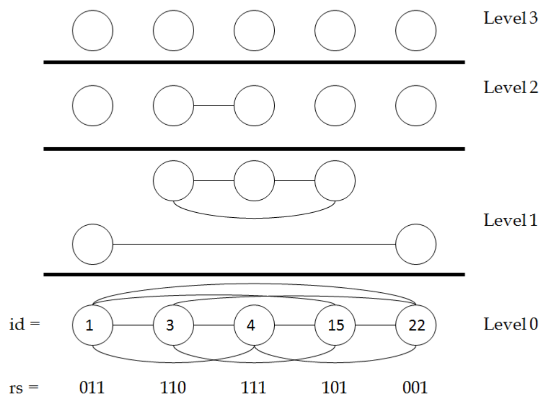


Figure 2.4: A Skip+ graph. Notice the Skip+ graph consists of all edges from the corresponding Skip graph and additional edges to make the graph locally checkable.

### 2.4.4 Chord

One of the most popular overlay networks for researchers is Chord, presented in 2001 by Stoica et al. [36]. The Chord network consists of $n$ nodes, each with a unique identifier from 0 to $(N-1)$ (clearly $n \leq N$). As with Linear, neighbors are determined by the successor function $succ(id)$, which returns the node $v$ with the smallest identifier greater than $id$. Unlike Linear, however, Chord is defined on a ring topology, so if no such node exists, $succ(id)$ returns the node $w$ with the smallest identifier.

Besides maintaining a link to its successor, every node $u$ also maintains several *fingers* to nodes spaced throughout the network. The $i$th finger of node $u$, $0 < i < \log N$, connects $u$ to $succ((id_u + 2^i - 1) \mod N)$. These fingers give the network logarithmic diameter. The base Chord ring for $N = 16$ is shown in Figure 2.5, along

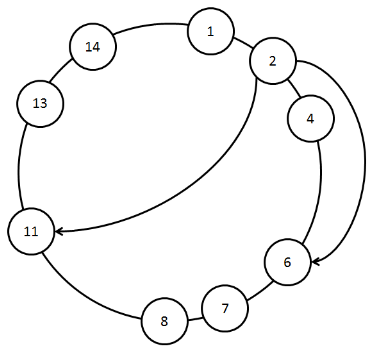


Figure 2.5: A small Chord network.

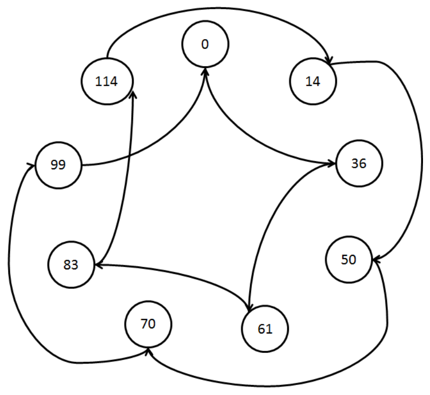


Figure 2.6: A faulty Chord configuration with no detectors.

with the fingers for node $u$, with $id_u = 2$. Notice that $succ(2) = succ(2+2-1) = 4$, and that $succ(2+8-1) = 11$, since nodes 3 and 10 are missing.

As pointed out in the original Chord work [36], the Chord network is not locally checkable. The authors offer an example similar to the one shown in Figure 2.6 (shown only with a node's immediate successor pointers). No node in the network can detect that the successor pointers are faulty, since no node has knowledge of a "better" successor for any neighboring node.

### 2.4.5 Re-Chord

The Re-Chord network [25] is a locally checkable version of the Chord network. Like Skip+ graphs, Re-Chord was created to enable the design of a silent self-stabilizing algorithm. In Re-Chord, the $n$ nodes of $V$ (called "real" nodes in the work) "simulate" zero or more "virtual" nodes. The Re-Chord network is defined over all nodes, real and virtual. Unlike Chord, nodes in Re-Chord are assigned real identifiers from $[0, 1)$ (for sufficiently large $N$ the approaches are equivalent). Every node (real and virtual) maintains links to their successor in the ring as with Chord. Every real node $u$ also maintains virtual nodes with identifiers $id_u + 1/2^i \mod 1$, for $0 < i \leq m$, where $m$ is the minimum value such that $u$ has no edge to a real node with identifier in the range $[id_u, id_u + 1/2^m]$. To ensure local checkability, nodes also label edges as either "un-marked" or "ring" edges, and only ring edges are allowed to "cross" 0 (that is, an edge $(u, v)$ with $id_u > id_v$ is a ring edge). Note the real nodes in Re-Chord have the same edges as Chord due to the presence of virtual nodes linking to their successors, meaning any operation possible on the Chord network is possible on the Re-Chord network.

# CHAPTER 3
# BUILDING SELF-STABILIZING OVERLAY NETWORKS WITH THE TRANSITIVE CLOSURE FRAMEWORK

Research in self-stabilizing overlay networks has mostly focused on specifics – creating a specific (locally checkable) network, creating a specific algorithm for building this network, and then providing specific convergence bounds for the given algorithm. This approach has several drawbacks. First, the specific algorithms are either complex and difficult to prove correct, or simple and for basic topologies only. This complexity also makes it hard to analyze the algorithms in terms of optimality – it is unclear from prior work what the best-case scenario is for overlay network convergence time. Finally, the details often time hide the techniques being used for defining the network and designing the algorithm.

To address these concerns, we present a generic framework for building self-stabilizing overlay networks called the *Transitive Closure Framework*. We also define a measure for overlay networks called the *detector diameter*, and provide a bound for optimal convergence time. Finally, we extend the Transitive Closure Framework to repair certain faults quickly.[1]

## 3.1 The Transitive Closure Framework

The *Transitive Closure Framework* (TCF) is a generic technique that can be used to create self-stabilizing algorithms for any locally checkable overlay network. It

[1]This chapter is derived from our work which appeared first as a brief announcement in 2010 [5], as a full paper in 2011 [6], and is currently under review for the journal *Theoretical Computer Science* [7].

is shown in Algorithm 3.1. The basic idea of the Transitive Closure Framework is that every node that detects the network is in a faulty configuration begins a transitive closure process on its neighborhood, placing all nodes at distance 2 from it into its neighborhood. After the network is completely connected, every node deletes its incident edges that are not part of a correct configuration, and in the next round the network has stabilized to the correct configuration.

As a "pre-processing" step of the algorithm, in every round nodes share their current state with all neighbors. This is equivalent to every node knowing its 2-neighborhood. We use $N^2(u)$ to denote all nodes that are at most distance 2 from $u$, and $\mathtt{id}_u^2$ and $\mathtt{rs}_u^2$ to denote the identifiers and random sequences of the nodes from $N^2(u)$. We use these constructs when defining the algorithm's components. Notice that TCF uses a predicate (called `Detect`) and a subroutine (called `Repair`) which are not implemented. These are generic constructs that need to be instantiated for each family of overlay network. The `Detect` predicate "signals" a node that the network is in an incorrect state and the transitive closure process should begin, while the `Repair` subroutine builds the correct network configuration. We define these constructs below.

**Definition 3.1.** *The* `Detect` *predicate at a node $u$ is a predicate defined over over $(N^2(u), \boldsymbol{id}_u^2, \boldsymbol{rs}_u^2), E_u$ and is false exactly when $N(w) = N_{ON(N^2(u), \boldsymbol{id}_u^2, \boldsymbol{rs}_u^2)}(w)$ for all $w \in N(u) \cup \{u\}$; otherwise the* `Detect` *predicate is true. A node $u$ is called a* detector *if the* `Detect` *predicate evaluates to true at node $u$.*

As an example, consider the `Detect` predicate for the Linear family of overlay networks presented in Section 2.4.1. The `Detect` predicate simply checks to see if the subgraph induced by a node $u$'s 2-neighborhood fails to induce a path of length at most 5, sorted by `ids`. If there is not such a path, `Detect` is set to true.

**Definition 3.2.** *Given $\lambda = (V, \mathtt{id}, \mathtt{rs})$, the* ***`Repair`*** *subroutine at node $v$ sets the neighborhood of $v$ to $N_{ON(\lambda)}(v)$ in one round.*

We now describe in more detail how the TCF algorithm from Algorithm 3.1 works. Whenever a node $u$ evaluates `Detect` to true, it sets its local variable $detect_u$ to true (Line 4). In the next round, all neighbors $v \in N(u)$ will set their $detect_v$ to true if they haven't already (Line 12). Every node with $detect_u = true$ will execute the transitive closure process (Line 11). This process is like "pointer jumping", and simply adds all $x \in N(w) : w \in N(u)$ to $N(u)$. Once the entire network is completely connected, nodes execute the `Repair` subroutine (Line 8) and the correct overlay network is built.

## 3.2 The Detector Diameter

The convergence time for the Transitive Closure Framework depends upon the distribution of detectors in the network. Notice that once all nodes become detectors, Algorithm 3.1 converges in $\mathcal{O}(\log n)$ rounds. We are interested, then, in how long it takes before all nodes in the network are detectors. In any faulty configuration, this depends upon the distance between a node and its closest detector. We formalize this concept next.

Let $\lambda = (V, \mathtt{id}, \mathtt{rs})$ be a set of nodes, and let $G = (V, E)$ be a weakly-connected

**Algorithm 3.1** Transitive Closure Framework

```
Algorithm for process $u$

Variables: neighborhood $N(u)$, Boolean $detect_u$

in each round do

1.  Send $N(u)$, $\texttt{id}_u$, and $\texttt{rs}_u$ to every neighbor $v \in N(u)$
2.  Receive $N(v)$, $\texttt{id}_v$, and $\texttt{rs}_v$ from each $v \in N(u)$
3.  Compute from this information: $\lambda^2 \leftarrow (N^2(u), \texttt{id}^2_u, \texttt{rs}^2_u)$ and
        $E_u \leftarrow \{(u,v)|v \in N(u)\} \cup \{(v,w)|v \in N(u)\}$
4.  $detect_u \leftarrow \texttt{Detect}(\lambda^2, E_u) \vee detect_u$
5.  Send $detect_u$ to every neighbor $v \in N(u)$
6.  Receive $detect_v$ from every neighbor $v \in N(u)$
7.  if $detect_u \wedge \forall v \in N(u) : (detect_v \wedge (\{N(v) \cup \{v\}\} = \{N(u) \cup \{u\}\}))$ then
8.      $N(u) \leftarrow \texttt{Repair}(N(u) \cup \{u\}, \texttt{id}_u, \texttt{rs}_u)$
9.      $detect_u \leftarrow false$
10. else if $detect_u \vee (\bigvee_{v \in N(u)} detect_v)$ then
11.     $N(u) \leftarrow N(u) \cup \{\bigcup_{v \in N(u)} N(v)\}$ //transitive closure
12.     $detect_u \leftarrow true$
13. fi

od
```

graph consisting of these nodes. Assume $G \neq ON(\lambda)$ for some family of overlay networks $ON$, and let $D \subseteq V$ be the set of detectors. The *detector diameter of* $G$ with respect to $\lambda$ is denoted $\mathit{DetDiam}_{ON(\lambda)}(G)$, and is the maximum distance in $G$ between any node in $V$ and its closest detector. Note that $\mathit{DetDiam}_{ON(\lambda)}(G)$ is also the maximum number of rounds required before all nodes in $G$ have $\mathit{detect}_u = \mathit{true}$ when executing the Transitive Closure Framework. The *detector diameter* $\mathit{DetDiam}_{ON}(n)$ of a family of overlay networks $ON$ is the maximum of $\mathit{DetDiam}_{ON(\lambda)}(G)$ over all $\lambda = (V, id, rs)$ with $|V| = n$ and all connected networks $G$ consisting of nodes from $\lambda$. Notice for networks that are *not* locally checkable, the detector diameter is $\infty$.

Using the detector diameter, we are then able to bound the running time of the Transitive Closure Framework given in Algorithm 3.1.

**Theorem 3.1.** *The Transitive Closure Framework presented in Algorithm 3.1 is a self-stabilizing algorithm for constructing any locally-checkable family of overlay networks* $ON$ *in at most* $\mathit{DetDiam}_{ON}(n) + \log(n) + 1$ *rounds.*

*Proof.* Consider some faulty (with respect to $ON$) configuration $G$. Since $ON$ is locally-checkable, there is at least one node $u$ in $G$ such that $u$ is a detector. Node $u$ will set $\mathit{detect}_u$ to $\mathit{true}$ (Line 4). In the next round, all neighbors $w \in N(u)$ will set $\mathit{detect}_w$ to $\mathit{true}$ (lines 10-12). In general, in every round the distance between a node $w$ and the closest node $x$ such that $\mathit{detect}_x = \mathit{true}$ decreases by one, until eventually all nodes are detectors. Since every node is at most distance $\mathit{DetDiam}_{ON}(n)$ from a detector in $G$, at most $\mathit{DetDiam}_{ON}(n)$ rounds are required. At this point, all nodes in the network are executing the transitive closure step (line 11).

When all nodes are executing the transitive closure step, the diameter of the network is halved in every round. In $\log n$ rounds, then, the diameter of the network is 1 (*i.e.* the network is a clique). Nodes will detect the network is a clique and all neighbors are detectors (line 7), at which point all nodes will simultaneously build the correct topology and clear their *detect* variable (lines 8-9).

Therefore, in at most $DetDiam_{ON}(n) + \log n + 1$ rounds, the correct network has been built from *any* arbitrary initial weakly-connected topology.

### 3.2.1 A Lower Bound for the Convergence Time

An interesting question is, given the detector diameter of a particular overlay network $ON$, how far off from optimal is the Transitive Closure Framework with respect to convergence time? We answer this question with the following theorem, which gives the first non-trivial lower bound on the convergence time for self-stabilizing overlay networks. Let $Diam_{ON}(n)$ be the maximum diameter of any member of $ON$ with at most $n$ nodes.

**Theorem 3.2.** *Let $ON$ denote any family of locally checkable overlay networks. Any silent self-stabilizing algorithm for constructing $ON$ requires, in the worst case, $\Omega(Diam_{ON}(n))$ time.*

*Proof.* Let $\lambda = (V, \texttt{id}, \texttt{rs})$ with $|V| = n$. Let $G = ON(\lambda)$ (that is, $G$ is the correct $ON$ network) and $d = Diam_{ON}(G)$. There exists a shortest path consisting of distinct nodes $p_0, p_1, \cdots, p_d$ in the network $G$. Let $V' = V \backslash \{p_0\}$ and $\texttt{id}'$ and $\texttt{rs}'$ be restrictions of $\texttt{id}$ and $\texttt{rs}$ (respectively) to $V'$, and let $\lambda' = (V', \texttt{id}', \texttt{rs}')$. Create a new network

$G' = ON(\lambda')$ (such a $G'$ must exist, as we have assumed $ON : \Lambda \to \mathcal{G}$ is defined on all members of $\Lambda$). Note that $G'$ is the correct $ON$ network with node $p_0$ removed. We prove the theorem holds by constructing a graph $G_F$ that requires $\Omega(Diam_{ON}(n))$ rounds to stabilize. To determine how to build $G_F$, we examine the distance between nodes $p_1$ and $p_d$ in $G'$.

Case 1: $dist_{G'}(p_1, p_d) > \frac{d}{2}$. To construct $G_F$, insert node $p_0$ as a neighbor to node $p_d$ in $G'$. The modified network is now faulty, as $p_0$ must be a neighbor of $p_1$ (and vice-versa) in the ideal configuration $ON(\lambda)$. Furthermore, only $p_d$ and its immediate neighbors have knowledge of node $p_0$, and these nodes are at least distance $\frac{d}{2} + 1$ away from node $p_1$, and node $p_1$ must change its local state to reach the correct configuration. Information concerning the existence of a fault due to the existence of $p_0$ can travel at most one hop per round, and no node on the path from $p_1$ to $p_d$ is a detector before receiving this information. Therefore, the self-stabilization time from such a state is at least $\frac{d}{2} + 1$ (the time required for node $p_1$ to be aware of node $p_0$'s existence).

Case 2: $dist_{G'}(p_1, p_d) \leq \frac{d}{2}$. In this case, $G_F$ is the graph resulting from removing node $p_0$ from $G$. Let $B(p_d, \frac{d}{2})$ be the subgraph induced by all nodes at distance at most $\frac{d}{2}$ from node $p_d$ in $G$, and let $B'(p_d, \frac{d}{2})$ be the subgraph induced by all nodes at distance at most $\frac{d}{2}$ from node $p_d$ in $G'$. Notice that $B(p_d, \frac{d}{2})$ does *not* include node $p_1$, while $B'(p_d, \frac{d}{2})$ *does* include $p_1$. Therefore, there must exist at least one node $w \in B(p_d, \frac{d}{2})$ whose neighborhood in $G'$ is different from its neighborhood in $G$. If this were not the case, then $B(p_d, \frac{d}{2}) = B'(p_d, \frac{d}{2})$, which would be addressed by Case 1 above. Notice that node $w$ is at least distance $\frac{d}{2}$ from $p_0$ in $G$, and that the correct neighborhood of $w$ depends upon the existence of node $p_0$ – that is, removal of $p_0$ from $G$ causes $w$ to change its neighborhood. However, only immediate neighbors of $p_0$ are aware of its removal, and information that the network is faulty can travel at most one hop per round. Therefore, it will require at least $\frac{d}{2}$ rounds before node $w$ is aware it should begin changing its neighborhood. Stabilization will require at least $\frac{d}{2}$ rounds in this case.

The implication of Theorem 3.2 is that for networks where the detector diameter is of the order of the diameter, the Transitive Closure Framework is nearly optimal – within at most an additive logarithmic factor of optimal. In fact, there may not exist a self-stabilizing algorithm that converges in sub-logarithmic time in our model, which would imply the Transitive Closure Framework is an optimal (with respect to convergence time) algorithm for any locally-checkable self-stabilizing overlay network family $ON$.

### 3.2.2 Example: The `Linear` Network

We can show easily that $DetDiam_{Linear}(n)$, the detector diameter for the Linear network, is $\mathcal{O}(n)$. Consider the network given in Figure 3.1. Notice that only nodes $(n-2)$, $(n-1)$ and 0 detect a fault in the network. Node $(n-2)$ is of distance $(n-3)$ from node 1, and therefore we have a detector diameter of $(n-3)$ (since Linear is locally checkable, the detector diameter must be finite, and clearly it cannot exceed

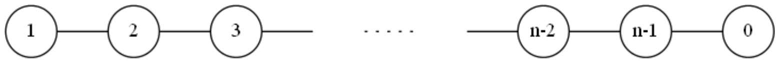


Figure 3.1: A faulty Linear network with $DetDiam_{Linear}(n) = \mathcal{O}(n)$.

$n$).

### 3.2.3 Example: The `Skip+` Network

To provide a non-trivial example of the Transitive Closure Framework, we consider the Skip+ graph overlay network family. To compare the performance of our Transitive Closure Framework with the optimal convergence of Skip+ graphs, we must prove the detector diameter of Skip+ graphs. We present this analysis below.

#### 3.2.3.1 The `Detect` Predicate and `Repair` Subroutine

The Transitive Closure Framework requires the existence of a `Detect` predicate and `Repair` procedure for the desired overlay network. `Detect` and `Repair` both follow easily from the definition of a Skip+ graph from Section 2.4.3. For the `Detect` predicate, each node computes its range using its 2-neighborhood, and then verifies that a node is a neighbor if and only if it is inside a node's range at level $i$. For the `Repair` subroutine, a node simply retains only those neighbors inside its range at level $i$, for all $i \in [0, |\texttt{rs}|]$.

#### 3.2.3.2 The Detector Diameter

While defining the `Detect` predicate and `Repair` subroutine is straightforward, analyzing the running time of the Transitive Closure Framework with these components is non-trivial. To find the convergence time of the Transitive Closure Framework with respect to Skip+ graphs, we compute the detector diameter for Skip+ graphs.

**Theorem 3.3.** *The detector diameter for the Skip+ network* $DetDiam_{Skip+}(n)$ *is* $|rs|+1$*, which is* $\mathcal{O}(\log n)$ *with high probability.*

*Proof.* Let $G_\rho$ be a subgraph of an arbitrary network $G$, induced by the set of nodes $\{v : pre_{|\rho|}(v) = \rho\}$. Furthermore, let $C_\rho$ be some connected component in $G_\rho$, $V_{C_\rho}$ be the nodes from $C_\rho$, and $\mathtt{id}_{C_\rho}$ and $\mathtt{rs}_{C_\rho}$ be the identifiers and random sequences of nodes in $V_{C_\rho}$. Notice that the maximum diameter of $Skip+(V_{C_\rho}, \mathtt{id}_\rho, \mathtt{rs}_\rho)$ is $L-|\rho|+1$, where $L = |u.rs|$ (each hop can bring a node at least one bit closer in random sequence to any other node). We will find the detector diameter by using induction on the length of $\rho$.

**Base Case:** $|\rho| = L$. In this case, $\forall v, v' \in C_\rho : v.rs = v'.rs$. Thus, either the nodes in $C_\rho$ are completely connected (and thus $C_\rho = Skip+(V_{C_\rho}, \mathtt{id}_\rho, \mathtt{rs}_\rho)$), or some node $u \in C_\rho$ has a neighbor $v \in C_\rho$, and $v$ has a neighbor $v' \in C_\rho$ such that $v'$ is not a neighbor of $u$. In this case, node $u$ ($v$) has $v'$ ($u$) in its 2-neighborhood, and will detect that a direct link should exist but does not. Therefore, $u$ and $v'$ will become detectors, and each node is at most $(L-L)+1 = 1$ hops away from a detector.

Let $\sigma \cdot x$ be a bit sequence consisting of $\sigma$ concatenated with the bit $x$, and $\sigma \cdot \bar{x}$ be bit sequence made up of $\sigma$ concatenated with the opposite of bit $x$.

**Induction Hypothesis:** Assume for all connected components in $G_{\sigma\cdot x}$ and $G_{\sigma\cdot\bar{x}}$, every node in a faulty connected component is within $L-(|\sigma|+1)+1 = L-|\sigma|$ steps of a detector.

**Inductive Step:** Consider the connected component $C_{\sigma \cdot x}$, which joins with 0 or more other connected components to form $C_\sigma$. We assume $C_\sigma$ does not match the ideal Skip+ graph corresponding to the nodes of $C_\sigma$, else the lemma is vacuously true. We examine the following cases, based upon how which components of $C_{\sigma \cdot x}$ are combined to form $C_\sigma$.

*Case 1: $C_{\sigma \cdot x} = C_\sigma$.* If $C_{\sigma \cdot x} = C_\sigma$, then all nodes in $C_\sigma$ are at most $L - |\sigma|$ from a detector by the induction hypothesis, and the claim holds.

*Case 2: $C_{\sigma \cdot x} \neq Skip{+}(V_{C_{\sigma \cdot x}}, \boldsymbol{id}_{C_{\sigma \cdot x}}, \boldsymbol{rs}_{C_{\sigma \cdot x}})$.* If $C_{\sigma \cdot x}$ was a faulty configuration, the claim again holds by the induction hypothesis, as all nodes in $C_{\sigma \cdot x}$ remain within $L - |\sigma|$ hops of a detector in $C_\sigma$.

*Case 3: $C_{\sigma \cdot x} \neq C_\sigma$ and $C_{\sigma \cdot x} = Skip{+}(V_{C_{\sigma \cdot x}}, \boldsymbol{id}_{C_{\sigma \cdot x}}, \boldsymbol{rs}_{C_{\sigma \cdot x}})$.* There must be some edge in $C_\sigma$ that was not present in $C_{\sigma \cdot x}$, or else $C_\sigma = C_{\sigma \cdot x}$. Furthermore, notice these new edges may only connect nodes in $C_{\sigma \cdot x}$ to nodes from $G_{\sigma \cdot \bar{x}}$ (otherwise, $C_\sigma$ would have been a single connected component in $G_{\sigma \cdot x}$).

Consider first the case where $C_{\sigma \cdot x}$ connects with exactly one other connected component $C_{\sigma \cdot \bar{x}}$ to form $C_\sigma$. If there exists a node in $C_{\sigma \cdot x}$ that is not connected to a node in $C_{\sigma \cdot \bar{x}}$, then there must exist a node $v \in C_{\sigma \cdot x}$ such that $v$ has no neighbor $x \in C_{\sigma \cdot \bar{x}}$ and $v$ has a neighbor $w \in C_{\sigma \cdot x}$ such that $w$ has a neighbor $x' \in C_{\sigma \cdot \bar{x}}$. Node $v$ will then have $x'$ inside it's calculated range at level $|\sigma|$, but no direct link to $x'$, and $v$ is a detector. By assumption, $v$ is within $L - |\sigma|$ of all other nodes in $C_{\sigma \cdot x}$. If all nodes in $C_{\sigma \cdot x}$ *are* connected to at least one node in $C_{\sigma \cdot \bar{x}}$, then either there is a detector in $C_{\sigma \cdot \bar{x}}$ that (by the induction hypothesis) is within $L - |\sigma| + 1$ of all nodes in $C_{\sigma\cdot x}$, or $C_{\sigma\cdot\bar{x}}$ was correct, in which case the diameter of the network is at most $L - |\sigma| + 1$, and our claim holds.

Next, consider where $C_{\sigma\cdot x}$ is connected to more than one component. As before, if any node in $C_{\sigma\cdot x}$ has no edge to a node from $G_{\sigma\cdot\bar{x}}$, then there must exist a node $v \in C_{\sigma\cdot x}$ without an edge to a node in $G_{\sigma\cdot\bar{x}}$, and $v$ must have a neighbor $w \in C_{\sigma\cdot x}$ such that $v$ has a neighbor from $G_{\sigma\cdot\bar{x}}$. Therefore, $v$ will be a detector, and our claim holds.

Consider where all nodes in $C_{\sigma\cdot x}$ are connected to at least one node in $G_{\sigma\cdot\bar{x}}$. If a node $v \in C_{\sigma\cdot x}$ has a link to nodes in two different connected components, say $C_{\sigma\cdot\bar{x}}$ and $C'_{\sigma\cdot\bar{x}}$, then $v$ will detect that these nodes should be connected but are not, and $v$ becomes a detector. If all nodes in $C_{\sigma\cdot x}$ are connected to at least one node from $G_{\sigma\cdot\bar{x}}$, and no node has connections to nodes from $C_{\sigma\cdot\bar{x}}$ and $C'_{\sigma\cdot\bar{x}}$, then there must exist two neighbors $v, w \in C_{\sigma\cdot x}$ such that $v$ and $w$ share no neighbors from $G_{\sigma\cdot\bar{x}}$. In this case, both $v$ and $w$ are detectors, and our claim holds.

To find the detector diameter, let $\rho$ be the empty string. Every node is at most distance $L + 1$ from a detector in a faulty configuration, and therefore $DetDiam_{Skip+}(n) = L + 1 = |rs| + 1$. Recall that the random sequence is often extended until every node has a unique random sequence, implying $L$ is $c \cdot \log n$ with high probability.

Combining Theorems 3.2 and 3.3 implies that *any* self-stabilizing algorithm for Skip+ networks requires $\Omega(\log n)$ rounds, and therefore the Transitive Closure Framework is an optimal (with respect to convergence time) self-stabilizing algorithm for Skip+ network construction. Notice that the original self-stabilizing Skip+ graph algorithm had a $\mathcal{O}(\log^2 n)$ convergence time.

## 3.3 Modification: Making Local Repairs

### 3.3.1 The Local Repair Framework

The main drawback of the Transitive Closure Framework is the required linear space (degree expansion) and logarithmic time for repairing *any* fault. Some faulty network configurations may be repairable using only a small amount of extra space and in only a few rounds (for instance, node joins and leaves), but the Transitive Closure Framework will always require $\Theta(n)$ degree during convergence (degree expansion remains linear). We introduce a simple extension to TCF that alleviates this limitation.

Key to improving upon the Transitive Closure Framework's space requirement is the ability to distinguish between configurations that can be repaired quickly without using TCF and configurations that cannot. Therefore, the high-level idea behind the approach is (i) defining local predicates for a subset of faulty configurations, (ii) showing these faulty configurations are repaired quickly, and (iii) showing that a node attempting to make a quick repair in a configuration that cannot be fixed with these local actions delays the start of the TCF for only a small number of rounds. We provide a framework for these repairs, which we call the *Local Repair Framework* (LRF). The LRF uses two new constructs which we define below.

**Definition 3.3.** *Let* $\texttt{L-Detect}_u$ *be a predicate evaluated over* $(\lambda^2, E_u)$ *at a node* *u.* ***L-Detect***$_u$ *is true when node u believes its neighborhood is part of a locally-repairable configuration – i.e. node u calculates using local information that executing the* ***L-Repair*** *subroutine will result in a correct configuration.*

**Definition 3.4.** *Let* ***L-Repair*** *be a subroutine with parameters node u, a set* $\lambda = (V, \boldsymbol{id}, \boldsymbol{rs})$*, and a set of edges E.* ***L-Repair*** *returns a neighborhood for node u.* ***L-Repair*** *satisfies the following two properties:*

- *Convergence : If* $N(u)$ *is set to* ***L-Repair***$(u, \lambda^2, E_u)$ *whenever* ***L-Detect***$_u$ *is true, after a finite number of rounds* ***L-Detect***$_u$ *becomes false.*
- *Connectivity : Setting* $N(u) =$ ***L-Repair***$(u, \lambda^2, E_u)$ *does not disconnect the network.*

Notice that a node that detects a fault and executes `L-Repair` will eventually reach a state where `L-Detect` is *false*. If the faulty configuration is a member of a particular set of faults (`LocalFaults`), executing `L-Repair` is guaranteed to converge to a legal configuration. If the faulty configuration is *not* a member of the set `LocalFaults`, then eventually `L-Detect` is false for a detector, and other repairs can be executed. Informally, this scenario is where a node thinks the configuration is repairable quickly, when in fact there may be more drastic faults present in the system. The maximum number of rounds `L-Repair` can be executed before `L-Detect` is is guaranteed to be false is called the *recovery time* of `LocalFault` $RT_{\texttt{L-Repair}}(n)$.

If every detector $u$ has $\texttt{L-Detect}_u = false$, the LRF reduces to the simple TCF – no local repairs are recognized ($RT_{\texttt{L-Repair}}(n) = 0$). Similarly, when

L-Detect$_u$ $= true$ for all detectors, the network configuration is stabilized using L-Repair in $RT_{\texttt{L-Repair}}(n)$ rounds (this is the "expected" behavior of L-Repair).

We present the Local Repair Framework in Algorithm 3.2.

**Algorithm 3.2** Local Repair Framework for Process $u$

```
Variables: neighborhood $N(u)$
in each round do
1. Send $N(u), \texttt{id}_u, \texttt{rs}_u$ to every neighbor $v \in N(u)$
2. Receive $N(v), \texttt{id}_v$, and $\texttt{rs}_v$ from each $v \in N(u)$
3. Compute from this information: $\lambda^2 := (N^2(u), \texttt{id}_u^2, \texttt{rs}_u^2)$ and
           $E_u := \{(u, v) | v \in N(u)\} \cup \{(v, w) | v \in N(u)\}$
4. if L-Detect$_u(\lambda^2, E_u)$ then
5.     $N(u) :=$ L-Repair$(u, \lambda^2, E_u)$;
6. else if $N^2(u) \neq ON(N^2(u))$ then
7.     Begin the Transitive Closure Process (Lines 4-13 of Algorithm 3.1);
8. fi
od
```

#### 3.3.1.1 Proof of the Local Repair Framework

We now provide a proof of correctness for the Local Repair Framework in Algorithm 3.2.

**Lemma 3.1.** *Starting from any configuration $G_\lambda$ with node set $\lambda = (V, \texttt{id}, \texttt{rs})$, where $\forall u \in V : \texttt{L-Detect}_u = false$, after at most $DetDiam_{ON}(n) + \log n + 1$ rounds the correct network $ON(\lambda)$ is constructed.*

*Proof.* Clearly if $G_\lambda = ON(\lambda)$, the lemma holds. If the network is faulty and $\texttt{L-Detect}_u$ is false for all nodes, then at least one node will initiate the Transitive Closure process. The network remains connected, as neither the Transitive Closure process nor the `L-Repair` subroutine disconnects the network (by definition). Also notice that once the Transitive Closure process is initiated, it continues until the correct network is built (as the network is locally checkable, at least one node will be a detector until the network is correct). The remainder of the proof follows from Theorem 3.1.

**Lemma 3.2.** *Starting from any configuration where $\texttt{L-Repair}_u = true$ for some detector, then in at most $RT_{\texttt{L-Repair}}(n)$ rounds either the correct overlay network is built or the Transitive Closure program is initiated.*

*Proof.* When $\texttt{L-Repair}_u = true$ for a detectors, the only neighborhood changes in the network occur due to line 5, which executes the `L-Repair` subroutine. By definition, after at most $RT_{\texttt{L-Repair}}(n)$ rounds the network is either correct, or the detector has started the Transitive Closure process (line 7).

**Theorem 3.4.** *Algorithm 3.2 is a self-stabilizing overlay network algorithm that can recover from* any *configuration in at most $RT_{\texttt{L-Repair}}(n) + DetDiam_{ON}(n) + \log n + 1$ rounds. Furthermore, a subset of configurations require at most $RT_{\texttt{L-Repair}}(n)$ rounds to repair.*

*Proof.* This follows easily from Lemmas 3.1 and 3.2.

### 3.3.2 Example of Local Repair: `Join` in `Skip+` Graphs

To demonstrate this technique, we consider one of the most common classes of faults: node joins. Specifically, we consider how to recover from node joins in Skip+ graphs.

As overlay network membership is often dynamic, accommodating nodes being added to the system is a commonly-addressed fault in overlay networks. A node that wishes to become a member of the overlay network begins the join process by connecting to a single node that is already a member of the current (correct) network. The goal of Join algorithms are to integrate the node into the correct network within some small amount of time. In this section, we instantiate the LRF by adding node Joins to Skip+ graphs.

### 3.3.3 The `L-Detect` Predicate and `L-Repair` Subroutine

The join procedure informally works as follows. A node that is joining is connected to an arbitrary node in the correct network. The joining node $u$ adds links to nodes with longer random sequence prefix matches than its current neighbors, until eventually it is neighbors with a node $v$ such that $v$ is the longest random sequence prefix match in the network. This is the highest level that $u$ will be a non-singleton in the network. Node $u$ then adds neighbors one level at a time beginning with this top level, with each level requiring one additional round. Once $u$ has added all neighbors at all levels, any extra links it created are deleted, and in the following round all neighbors can update their range (if necessary) and delete their additional edges.

We begin by defining the following predicates that are evaluated locally on each node $u$, which together form $\texttt{L-Detect}_u$. For ease of notation, let $pre(s_1, s_2)$ return the prefix match between strings $s_1$ and $s_2$, and $|pre(s_1, s_2)|$ be the length of the matching prefix. Recall that $E_u = \{(u, v)|v \in N(u)\} \cup \{(v, w)|v \in N(u)\}$, $N^2(u)$ is the set of the "at-most 2-hop neighbors" of $u$, and $\texttt{id}_u$ and $\texttt{rs}_u$ are the restrictions on $\texttt{id}$ and $\texttt{rs}$ from the set $N^2(u)$.

1. $SingleNodeFault_u(v) := [N_{\mathit{Skip+}(N^2(u),\texttt{id}_u,\texttt{rs}_u)}(u) \neq N(u)] \wedge$
   $[(E_u \setminus \{(v, x), \forall x\}) = \mathit{Skip+}(N^2(u) \setminus \{v\}, \texttt{id}_u, \texttt{rs}_u)]$
   - Node $u$ detects its neighborhood is incorrect. However, node $u$'s local neighborhood is consistent with the correct overlay network node $u$ would see with node $v$ removed.
2. $AllConnected_u := |N(u)| > 1 \implies \forall v \in N(u) : \exists w \neq v \in N(u)$ s.t. $w \in N(v)$
   - If a node $u$ has more than 1 neighbor, then for all neighbors $v \in N(u)$, there exists another neighbor $w \in N(u)$ (not equal to $v$) such that an edge exists between $v$ and $w$.
3. $LongerMatchExists_u := \exists w \in (N^2(u) \setminus N(u))$ s.t.

$\forall v \in N(u) : |pre(u.rs, w.rs)| > |pre(u.rs, v.rs)|$

- In node $u$'s 2-neighborhood, there exists a node $w$ who has a longer matching random sequence than any node currently in $u$'s immediate neighborhood.

4. $InitiatingJoin_u := LongerMatchExists_u \wedge AllConnected_u \wedge$

   $SingleNodeFault_u(u) \wedge$

   $[\forall v \in N(u) : \nexists w \in N(u) \text{ s.t. } |pre(u.rs, v.rs)| = |pre(u.rs, w.rs)|]$

   - For $u$, $LongerMatchExists_u$, $AllConnected_u$, and $SingleNodeFault_u(u)$ (see above) and no two nodes in $N(u)$ have the same length prefix match with $u$.

5. $CreatingJoinLinks_u := AllConnected_u \wedge \neg LongerMatchExists_u \wedge$

   $SingleNodeFault_u(u) \wedge (\exists w \in N^2(u) \text{ s.t. } (u, w) \in N_{Skip+(N^2(u), \mathtt{id}_u, \mathtt{rs}_u)}(u)$

   $\wedge w \notin N(u))$

   - $AllConnected_u$, $\neg LongerMatchExists_u$, and $SingleNodeFault_u(u)$ are true. Also, node $u$ has nodes in its 2-neighborhood that should be neighbors in the correct Skip+ graph, yet are not neighbors now.

6. $JoinCompleted_u := AllConnected_u \wedge SingleNodeFault_u(u) \wedge$

   $N_{Skip+(N^2(u), \mathtt{id}_u, \mathtt{rs}_u)}(u) \subset N(u)$

   - $AllConnected_u$ and $SingleNodeFault_u(u)$ is true for node $u$, and the neighbors of $u$ are a superset of the nodes $u$ should have in it's calculated ideal configuration.

7. $WaitForJoin_u := \exists v \in N(u) : SingleNodeFault_u(v) \wedge$

   $N_{\mathit{Skip+}(N^2(u),\texttt{id}_u,\texttt{rs}_u)}(v) \neq N(v)$

   - Node $u$ has a neighbor whose removal would make $N(u)$ correct, and $N(v)$ does not match the calculated neighorhood for $v$.

8. $JoinDetected_u := \exists v \in N(u) : SingleNodeFault_u(v) \wedge$

   $N_{\mathit{Skip+}(N^2(u),\texttt{id}_u,\texttt{rs}_u)}(v) = N(v) \wedge N_{\mathit{Skip+}(N^2(u),\texttt{id}_u,\texttt{rs}_u)}(u) \subset N(u)$

   - Node $u$ has a neighbor $v$ whose removal would make node $u$'s neighborhood correct, node $u$'s current neighborhood is currently a superset of $u$'s calculated neighborhood, and node $v$ has a correct neighborhood.

We use the predicates above to define `L-Detect`$_u$ next.

**Definition 3.5.** *For Join in Skip+ graphs, let* `L-Detect`$_u :=$ $InitiatingJoin_u \vee CreatingJoinLinks_u \vee JoinCompleted_u \vee WaitForJoin_u \vee$ $JoinDetected_u$

We define the subroutine `L-Repair` for Join in Skip+ graphs below. The high-level idea of the joining procedure from Algorithm 3.3 is simple. A joining node $u$ "traverses" the network (by adding edges), searching for the node with the longest prefix match with $u$'s random sequence (predicate $InitiatingJoin_u$ is *true*). Once this node is found, node $u$ adds the appropriate neighbors for each level (predicate $CreatingJoinLinks_u$). Once all neighbors have been added, node $u$ deletes the edges that it used to find the longest prefix match. Other nodes in the network do nothing if there is a single-node fault in the network ($WaitForJoin_u$) until that single node has successfully joined the network ($JoinDetected_u$), and then their neighborhoods are updated to incorporate the new node.

#### 3.3.3.1 An Example Join

To clarify the Join procedure, we present an example Join for the Skip+ graph from Fig. 2.4. Imagine a node $u$ with $u.id = 27$ and $u.rs = 110$ joins the network by connecting to node $w$, with $w.id = 1$. Both node $w$ and $u$ will detect their local neighborhoods would appear correct if node $u$ was removed ($SingleNodeFault_w(u)$ and $SingleNodeFault_u(u)$). Node $w$ evalutes $WaitForJoin_w$ to true and will not change its neighborhood (Line 10), while node $u$ evaluates $InitiatingJoin_u$ to true, since it has in its 2-neighborhood a node with a longer matching random sequence than any current neighbor (node $x$, with $x.id = 4$ and $x.rs = 111$). As a result, node $u$ will, in one round, add the edge $(u, x)$ (Line 2). In the next round, nodes $u$, $w$, and $x$ will detect their local neighborhoods would appear correct if $u$ was removed. Nodes $w$ and $x$ evaluate $WaitForJoin$ to true and do not change their neighborhoods (Line 10), while node $u$ evaluates $InitiatingJoin_u$ to true and adds node $y$ to its neighborhood (Line 2), with $y.id = 15$ and $y.rs = 110$. In the next round, nodes $u$, $w$, $x$, and $y$ detect the network would appear correct if $u$ was removed, and $u$ can no longer find a node with a longer matching prefix than $y$. The system state at this point is shown in Figure 3.2.

Now that node $u$ does not detect a node with a longer prefix match in its 2-neighborhood, it begins adding neighbors level-by-level ($CreatingJoinLinks_u$ eval-

**Algorithm 3.3** The `L-Repair` Subroutine for Join in Skip+ Graphs

**Parameters:** Node $u$, set $\lambda^2$, edges $E_u$

```
1.  if $InitiatingJoin_u$ then
        // add the longest random sequence match to the neighborhood
2.      return $N(u) \cup \{x : x \in (N^2(u) \setminus N(u)) \wedge$
                $|pre(u.rs, x.rs)| = \max_{t \in N^2(u) \setminus u}(|pre(u.rs, t.rs)|) \wedge$
                $(\forall y \in (N^2(u) \cap \{z : |pre(u.rs, z.rs)| = |pre(u.rs, x.rs)|\}) :$
                        $x.id \geq y.id)\};$
3.  else if $CreatingJoinLinks_u$ then
        // add links level-by-level
4.      return $N(u) \cup \{x : x \in N_{Skip+(N^2(u), \mathtt{id}_u, \mathtt{rs}_u)}(u) \wedge$
            $|pre(u.rs, x.rs)| = \max_{t \in N^2(u) \setminus N(u)}(|pre(u.rs, t.rs)|);$
5.  else if $JoinCompleted_u$ then
        // delete edges used to find longest matching random
        // sequence, making the network correct for u
6.      return $N_{Skip+(N^2(u), \mathtt{id}_u, \mathtt{rs}_u)}(u);$
7.  else if $JoinDetected_u$ then
8.      return $N_{Skip+(N^2(u), \mathtt{id}_u, \mathtt{rs}_u)}(u);$ // incorporate the joined node
9.  else // $WaitForJoin_u$
10.     return $N(u)$  // do nothing
11. fi
```

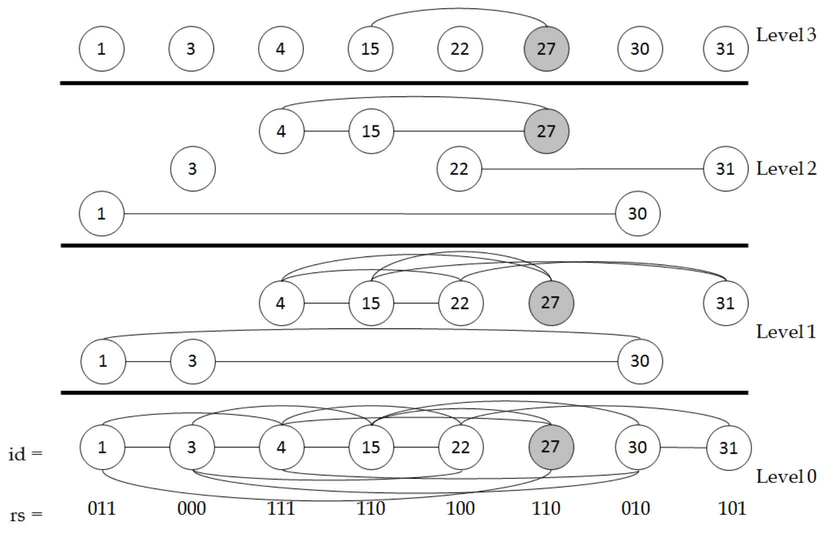


Figure 3.2: A Skip+ Join in progress. Node 27 is joining the network.

uates to true). Node $u$ already has the necessary links for levels 2 and 3 (created during the search for the longest match). Furthermore, all nodes that should be neighbors with $u$ in level 1 are in $u$'s 2-neighborhood. Therefore, node $u$ will create a link in the next round to nodes $a$ and $b$, with $a.id = 22$ and $b.id = 31$ (Line 4). In the subsequent round $u$ will add links to $c$ and $d$, with $c.id = 3$ and $d.id = 30$, filling in level 0. Neighbors of $u$ during this process will evaluate $WaitForJoin$ to true (as $u$'s neighborhood is not correct), and will not change their neighborhoods (Line 10).

Node $u$'s neighborhood is now a superset of its calculated correct neighborhood (the edge to node $w$, $w.id = 1$, is unnecessary). Node $u$, in the next round, will delete its edge to node $w$, since $JoinCompleted_u$ is true (Line 6). At this point the network is correct, and the Join procedure is complete.

### 3.3.4 Analysis of Skip+ Graph Join

We begin the analysis by proving the algorithm does not disconnect the network regardless of the configuration. From there, we simply analyze how the neighborhood of a node is modified by program actions, and show this can happen only a logarithmic number of times before `L-Detect` must be false.

**Lemma 3.3.** *(Connectivity)* ***`L-Repair`****, when starting from a weakly-connected topology, does not disconnect the network.*

*Proof.* Notice that only lines 6 and 8 may delete edges from the network. Therefore, we only need consider these two actions when examining connectivity. First, consider a node $u$ that deletes edges by executing line 6 of Algorithm 3.3. The predicate $JoinCompleted_u$ must be true. For all $v \in N(u)$, $JoinDetected_v$ is false, as $u$'s neighborhood contains too many neighbors, which all nodes $v$ detect. Therefore, only node $u$ will delete edges from its neighborhood when executing line 6. The predicate $AllConnected_u$ is true, meaning all of $u$'s neighbors are connected to each other. Therefore, since only node $u$ will delete a subset of its edges, and all $v \in N(u)$ are connected through a path that does not including $u$, the network remains connected when a node executes line 6.

Next, consider the case where node $u$ deletes edges from its neighborhood using line 8 from Algorithm 3.3. Notice that no node $v \in N(u)$ can delete an edge using line 6 (see prior paragraph). If an edge is deleted in line 8, then, it must be because the range of $u$ has shrunk due to the presence of some $w \in N(u)$ (since $N(u)$ is correct if $w$ is removed, and adding nodes can only shrink a calculated range). If

$w \in N(u)$ causes some $range_i(u)$ to shrink, then all nodes that were neighbors in level $i$ before are now at most distance 2 from $u$, and are reached by traveling through $w$. Node $w$ will not delete these edges in that round. Therefore, the network remains connected during execution of `L-Repair`. with some immediate neighbors becoming distance-2 neighbors through some node $w$.

**Lemma 3.4.** *(Convergence) The recovery time $RT_{Skip+Join}(n)$ for Algorithm 3.3 is $2 \cdot c \cdot \log n + 2$, where the random sequence $|rs|$ is of length $c \cdot \log n$.*

*Proof.* We consider two cases. First, imagine a single node $u$ is joining a legal Skip+ graph by creating a link to some node $x$ in the network. At this point, $InitiatingJoin_u$ may be true for at most $c \cdot \log n$ rounds, as in every round where $InitiatingJoin_u$ is true, $u$ adds an edge to a node $v$ such that $|pre(u.rs, v.rs)|$ is greater than $|pre(u.rs, w.rs)|$ for all $w \in N(u)$ (line 2). Once $InitiatingJoin_u$ is false, node $u$ has a neighbor $w$ such that $w$ has the longest matching random sequence prefix with $u$ in the network.

Once $InitiatingJoin_u$ is false, $CreatingJoinLinks_u$ can be true for at most $c \cdot \log n$ rounds. Assume a node $u$ is neighbors with some node $w$ such that $w$ has the longest random sequence prefix match of any node in the network, and let the length of this match be $i$. We calculate how many rounds are required before $u$ has the correct range at level $i$. When $CreatingJoinLinks_u$ is true, node $u$ executes line 4, which continues to add calculated neighbors to $N(u)$. After at most $c \cdot \log n - i + 1$ such additions, node $u$ has a neighbor $w'$ such that $w' \in range_i(u)$ and $u \in range_i(w')$, as the diameter of the subgraph induced by all nodes $x$ with $|pre(u.rs, x.rs)| = i$ is

$c \cdot \log n - i + 1$ (every hop is at least one random sequence bit closer to the destination). For every subsequent round $k$ where $CreatingJoinLinks_u$ remains true, $u$ adds links level-by-level – that is, $u$ adds links to its correct neighbors in level $i - k$. Note the correct neighbors for node $u$ at level $t - 1$ are in the correct 2-neighborhood of $u$ at level $t$. To see this, consider the nodes inside $u$'s (correct) range in level $(t - 1)$, which we shall denote $N_{t-1}(u)$. Note that all nodes in $N_{t-1}(u)$ that have a random sequence prefix match of length at least $t$ are immediate neighbors of $u$ in level $t$. Similarly, nodes in $N_{t-1}(u)$ that do not match at least $t$ bits of the random sequence prefix are at most distance 2, as they must be neighbors with at least one node whose random sequence differs in bit $t$, which then matches the first $t$ bits of $u$'s random sequence. This node must be at most distance 1 from $u$ in level $t$. Therefore, in $(c \cdot \log n - i + 1) + i - 1 = c \cdot \log n$ rounds, $CreatingJoinLinks_u$ is false, as node $u$ will be neighbors with all nodes in $N_{\mathit{Skip+}(N^2(u), \mathtt{id}_u, \mathtt{rs}_u)}(u)$.

Finally, in one round node $u$ will trim the unnecessary edges that were used to find the longest matching random sequence prefix (line 6). In one additional round, all nodes with which $u$ is joining with will trim their neighborhoods to be equal to their calculated ideal neighborhoods (line 8). At this point, the network is now in a correct Skip+ graph configuration. Therefore, $RT_{\mathtt{L\text{-}Repair}}(n) = (c \cdot \log n) + (c \cdot \log n) + 2 = 2 \cdot c \cdot \log n + 2$ rounds.

Next, consider the case where $\mathtt{L\text{-}Detect}_u$ is true for some $u$, but the network is not the result of a single node attempting to join. By similar arguments to the correct Join case, a node can only execute $2 \cdot c \cdot \log n + 2$ steps before it has built what it believes is the correct network. If at any point during this execution `L-Detect` is false for a node, but the node detects a faulty configuration, the Transitive Closure program is initiated.

**Theorem 3.5.** *The Local Repair Framework given in Algorithm 3.2, instantiated with the* ***`L-Repair`*** *subroutine from Algorithm 3.3 and predicate* ***`L-Repair`***$_u$ *from Definition 3.5, is a self-stabilizing algorithm for Skip+ graphs with convergence time for* any *configuration at most* $3 \cdot c \cdot \log n + \log n + 2$ *rounds. Joins occur in at most* $2 \cdot c \cdot \log n + 2$ *rounds.*

*Proof.* By Lemma 3.4, Joins require $2 \cdot c \cdot \log n + 2$ rounds. Since it will require at most $2 \cdot c \cdot \log n + 1$ rounds to initiate the Transitive Closure process, which will then require $\log n + c \cdot \log n + 1$ rounds to complete, any initial configuration requires at most $3 \cdot c \cdot \log n + \log n + 2$ rounds.

# CHAPTER 4
# THE AVATAR NETWORK

Self-stabilizing overlay networks to date have considered either (i) efficient (polylogarithmic) convergence time, or (ii) efficient (constant) degree growth. Clearly a scalable solution in terms of both time and space would be desirable.

Another shortcoming is that overlay network analysis has either been complicated and difficult to understand, or for simple topologies or algorithms that are not scalable with respect to time or space. In this chapter, we address these two concerns. Specifically, we present a self-stabilizing overlay network and prove that its convergence time and degree expansion are both polylogarithmic. The network definition is generic and can be used for other graphs, even those that are by themselves not locally checkable. Therefore, the network definition is a local checkability transformer. Furthermore, the algorithm has a modular design, making modifications and extensions for other topologies simple, as well as simplifying analysis.

## 4.1 Preliminaries

For computation, we use the synchronous shared memory model for overlay networks presented in Chapter 2. We add one new component to this model – a public random coin. Specifically, we use a model similar to the COMMON model presented by Newman [29]. In this model, all nodes have access to a random sequence $L$ of length $k$. Furthermore, the adversary has no knowledge of $L$ besides knowing its length $k$. Besides this shared random sequence, we assume a powerful adversary that can create an arbitrary weakly-connected initial state, including modifying any node's variables.

When analyzing the algorithm, we will need to describe future configurations resulting solely from executing the algorithm. For an algorithm $\mathcal{A}$, let $\mathcal{F}_{\mathcal{A}}$ be a function which maps a configuration $G_i$ to the set of all configurations that can result from executing $\mathcal{A}$ on $G_i$. $\mathcal{F}_{\mathcal{A}}(G_i)$ is a finite set for the algorithm, as the algorithm is self-stabilizing and silent (that is, the configuration reaches a point where no more changes are made).

As mentioned earlier, the stabilization process can be thought of as a walk through the state space of a system. In this chapter, we present an algorithm where this "walk" is both short (convergence time is low) and only goes through system states with (relatively) low degree. This work is the first of its kind to identify such a path.

## 4.2 The `Avatar` Network

In this section, we define the generic `Avatar` network, which embeds a guest topology onto the nodes of the overlay network. We also introduce an instantiation of this network based upon a binary search tree.

### 4.2.1 Specification

A *network embedding* [27] $\Phi$ maps the node set of a *guest network* $G_g = (V_g, E_g)$ onto the node set of a *host network* $G_h = (V_h, E_h)$. Thus an embedding is a mapping $\Phi : V_g \to V_h$. The *dilation* of $\Phi$ is defined as the maximum distance between any two nodes $\Phi(u), \Phi(v) \in V_h$ such that $(u, v) \in E_g$. Note that $\Phi$ can map more than one node in $V_g$ onto a single node in $V_h$.

Informally speaking, the `Avatar` network is a host network designed to realize a dilation-1 embedding for an appropriately-chosen guest network. Specifically, 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 $N$-node graph $F_N \in \mathcal{F}$; we assume the node set of $F_N$ is $[N]$. We shall call $\mathcal{F}$ a *full graph family*, capturing the notion that all graphs in the family have a "full" set of nodes (relative to the identifier space). For any $N \in \mathbb{N}$ and $V \subseteq [N]$, $\texttt{Avatar}_{\mathcal{F}}$ is a network with node set $V$ that realizes a dilation-1 embedding of $F_N \in \mathcal{F}$. The specific graph is defined below.

**Definition 4.1.** *Let $V \subseteq [N]$ be a subset $\{u_0, u_1, \ldots, u_{n-1}\}$, where $u_i < u_{i+1}$ for $0 \leq i < n-1$. Let the* range *of a node $u_i$, $range_{u_i} = [u_i, u_{i+1})$ for $0 < i < n-1$. Let $range_{u_0} = [0, u_1)$ and $range_{u_{n-1}} = [u_{n-1}, N)$. $\texttt{Avatar}_{\mathcal{F}}(N, V)$ is a graph with node set $V$ and edge set the union of the sets:*

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

It is easy to see there exists a dilation-1 embedding $\Phi$ of $F_N$ onto the network $\texttt{Avatar}_{\mathcal{F}}(N, V)$. Specifically, $\Phi$ maps a node $b \in V(F_N)$ onto the node $u_i$ when $b \in range_{u_i}$. Type (2) edges of $\texttt{Avatar}_{\mathcal{F}}(N, V)$ in the definition above are created to make sure that this embedding has dilation 1. Type (1) edges, $(u_i, u_{i+1})$, for $0 \leq i < n-1$, are also added in order to "linearize" the topology. A node $u_i \in V$ can locally determine its range, and therefore can locally determine the node set $\Phi^{-1}(u_i) \subseteq V(F_N)$ mapped to $u_i$.

### 4.2.2 Advantages of Embeddings

There are several advantages to defining overlay networks in terms of network embeddings. Concerning the design of algorithms, network embeddings allow a convenient abstraction, and this helps us in algorithm description and analysis in the remainder of the paper. However, embeddings also preserve several important properties. Specifically,

- The diameter of $\texttt{Avatar}_{\mathcal{F}}(N,V)$ is at most the diameter of $F_N \in \mathcal{F}$. Therefore, if $F_N$ has low diameter (e.g. $\mathcal{O}(\log N)$), then $\texttt{Avatar}_{\mathcal{F}}(N,V)$ has low diameter as well. In cases where $|V|$ and $N$ are relatively close, then $\texttt{Avatar}_{\mathcal{F}}(N,V)$ has low diameter with respect to $n$ as well as $N$ (e.g., if $N \leq n^c$ for some constant $c$, then $\mathcal{O}(\log N) = \mathcal{O}(\log n)$).
- The routing properties of $F_N$ are inherited by $\texttt{Avatar}_{\mathcal{F}}(N,V)$. Any routing algorithm created for $F_N$ can easily be simulated on $\texttt{Avatar}_{\mathcal{F}}(N,V)$. In fact, any algorithm created for $F_N$ can be simulated by nodes in $\texttt{Avatar}_{\mathcal{F}}(N,V)$.
- For some embeddings, the low degree of nodes in $F_N$ is preserved with the nodes in $\texttt{Avatar}_{\mathcal{F}}(N,V)$. That is, nodes in $F_N$ have low degree, and therefore nodes in $\texttt{Avatar}_{\mathcal{F}}(N,V)$ have low degree. However, this property does not hold for all graph families $\mathcal{F}$. For instance, consider a hypercube graph family, $\mathcal{H}$. While the degree of every node is logarithmic in a hypercube, a node $u_i$ in $\texttt{Avatar}_{\mathcal{H}}(N,V)$ may have degree which is linear in $N$ if $|\Phi^{-1}(u_i)|$ is large. There are two ways to address this concern. First, if we assume the identifiers of nodes in the host network are uniformly distributed, then $|\Phi^{-1}(u_i)|$ is, in

expectation, $N/n$. Even without this uniform distribution assumption, there are graph families where *regardless of the size of* $\Phi^{-1}(u_i)$, degrees remain low (e.g. logarithmic).

### 4.2.3 The Full Graph Family `Cbt`

We now describe an instance of the `Avatar` network where the full graph family is the family of *complete binary search trees.* We call this family of networks `Cbt`, and use this network throughout the remainder of the paper. We define `Cbt`$(N)$ recursively.

**Definition 4.2.** *For $a \leq b$, let* `Cbt`$[a,b]$ *be a binary tree rooted at* $r = \lfloor (b+a)/2 \rfloor$. *Node r's left subtree is* `Cbt`$[a, r-1]$*, and r's right subtree 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 b in* `Cbt`$[0, N-1]$ *be the distance from b to root* $\lfloor N - 1/2 \rfloor$.

Note there is exactly one topology satisfying these requirements for any $N \in \mathbb{N}$. Furthermore, every node $b \in$ `Cbt`$(N)$ can calculate this topology. Figure 4.1 gives an example of the `Cbt`$(15)$ graph.

#### 4.2.3.1 Maximum Degree of `Avatar`$_{\texttt{Cbt}}$

One potential drawback of creating a topology realizing an embedding with dilation of 1 is that a node $u_i$ may have a large $\Phi^{-1}(u_i)$, resulting in a high degree, *even if* no node in $F_N$ has a large degree. It is not hard to construct examples where the degree of a node $u_i$ is proportional to the size of $\Phi^{-1}(u_i)$. For instance, consider a simple full graph family $\mathcal{S}$, where every node $b_i$ in $\mathcal{S}_N$ has exactly one neighbor,

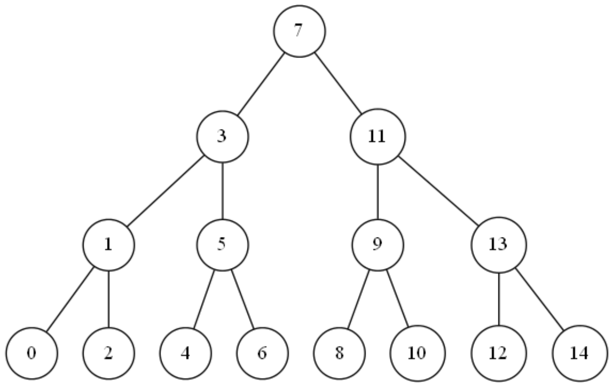


Figure 4.1: A $\texttt{Cbt}(15)$ graph.

node $b_k$, where $k = i + N/2 \mod N$. Imagine $\texttt{Avatar}_{\mathcal{S}}(N, V)$, where $u_0 = 0$ and $u_1 = N/4$ (there are no nodes with identifiers from 0 to $N/4$). The degree of $u_0$ in $\texttt{Avatar}_{\mathcal{S}}(N, V)$ may be $\mathcal{O}(N)$ – every node in $\Phi^{-1}(u_0)$ has a neighbor outside of $\Phi^{-1}(u_0)$.

Surprisingly, however, there are full graph families for which their embeddings have low degree *regardless of the distribution of identifiers of the host network.* The $\texttt{Cbt}$ network is one such network. We show a logarithmic bound on the degree of any node for the $\texttt{Avatar}_{\texttt{Cbt}}$ network next. This bound is important not only for scalable final configurations, but also for efficient convergence with the algorithm, as we shall show later.

**Lemma 4.1.** *Consider a node set $V \subseteq [N]$. The maximum degree of any node $u_i \in V$ in the $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ network is at most $2 \cdot \log N$.*

*Proof.* Consider the subset of nodes of $[N]$ mapped to node $u_i$, $\Phi^{-1}(u_i)$. Let $[N]_j$ be

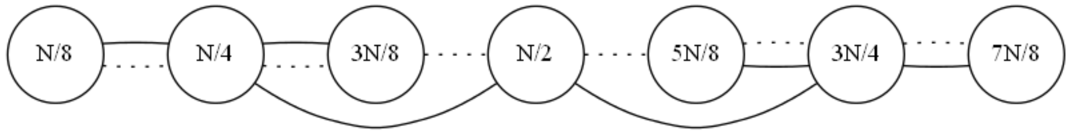


Figure 4.2: A linear representation of levels 0-2 of a complete binary search tree.

the set of all nodes at level $j$ of $\texttt{Cbt}(N)$. We show that, for any $0 \leq j \leq \log N + 1$, there are at most 2 nodes in $\Phi^{-1}(u_i) \cap [N]_j$ with neighbors outside of $\Phi^{-1}(u_i)$.

First, consider the edges in $\texttt{Cbt}(N)$. Let the *span of an edge* $(a, b) \in E[\texttt{Cbt}(N)]$ (with $a < b$) be all nodes from $[N]$ in the interval $(a, b)$, and let the *size of a span* be $b - a$. Let a *node segment* $S[a, b]$ be a contiguous set of nodes from $[N]$ (that is, $S = \{c : a \leq c \leq b\}$). Notice that, by definition of $\texttt{Cbt}(N)$, the spans of any two edges with the same span size are disjoint. Therefore, for any segment $S$, there can be at most two edges in $E(\texttt{Cbt}(N))$ with the same span size going from a node $b \in S$ to a node $b' \notin S$. Since there are at most $\log N$ span sizes in a complete binary search tree, there are at most $2 \cdot \log N$ edges from a node inside any segment $S[a, b]$ to a node outside the segment.

For example, consider Figure 4.2, which is a linear representation of the first two levels of $\texttt{Cbt}(N)$, where $N$ is a power of 2. Notice there are exactly 2 edges with spans of size $N/4$ (from $N/4$ to $N/2$ and $N/2$ to $3N/4$), and the span of the edges is disjoint. Similarly, there are 4 edges that span $N/8$ points ($(N/8, N/4)$, $(N/4, 3N/8)$, $(5N/8, 3N/4)$, $(3N/4, 7N/8)$), also with disjoint spans.

Notice in the $\texttt{Avatar}_{\texttt{Cbt}}$ network, $\Phi^{-1}(u_i)$ is a node segment from $[N]$. The only edges from nodes in $\Phi^{-1}(u_i)$ that require an edge in $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ are those from $b \in \Phi^{-1}(u_i)$ to $b' \notin \Phi^{-1}(u_i)$. As there are most $2 \cdot \log N$ such edges, the degree of any node $u_i \in \texttt{Avatar}_{\texttt{Cbt}}(N, V)$ is at most $2 \cdot \log N + 2$ (at most two edges of type (1)).

## 4.3 Local Checkability

As discussed in Chapter 2, silent self-stabilizing algorithms require the network to reach a configuration where the internal state of a node no longer changes. This requires nodes to determine when they can stop executing the actions that modify their internal state. Clearly an algorithm that becomes silent in an incorrect configuration cannot be self-stabilizing, and an algorithm that never stops modifying the internal state cannot be silent. Therefore, we need a mechanism to detect correct configurations and prevent internal state from changing. Local checkability is such a mechanism. Informally, local checkability is a distributed decision problem to decide if the network $G$ belongs to the family $\mathcal{F}$ – all nodes output true if the network is correct, and at least one node returns false if the network is incorrect.

We modify the definition of local checkability given in Chapter 2. Specifically, we introduce the use of proof labels, first presented by Göös and Suomela [20]. A *proof label* for node $u$ is a bit string $P(u) = \{0, 1\}^*$. Let $N_G(u)$ be the *closed neighborhood of $u$ in graph $G$*, which consists of all nodes at most distance 1 from $u$ in $G$, and let $P_G(u)$ be the proof labels associated with all nodes in $N_G(u)$. We now define local checkability using these concepts.

**Definition 4.3.** *Let $\mathcal{F}$ be a family of graphs. We say that $\mathcal{F}$ is* locally checkable *if there exists a function $\mathcal{A}_\mathcal{F}$ which takes as input $u_i$ and $(N_G(u_i), P_G(u_i))$ and returns a binary value $\mathcal{A}_\mathcal{F}(u_i, (N_G(u_i), P_G(u_i)))$ such that (i) if $G \notin \mathcal{F}$, then $\exists u_j \in V(G) : \mathcal{A}_\mathcal{F}(u_j, (N_G(u_j), P_G(u_j))) = 0$, and (ii) if $G \in \mathcal{F}$, then $\forall u_j \in V(G) : \mathcal{A}_\mathcal{F}(u_j, (N_G(u_j), P_G(u_j))) = 1$. We say that $\mathcal{A}_\mathcal{F}$ is a* verifier *for $\mathcal{F}$.*

Therefore, said another way, a family of graphs $\mathcal{F}$ is locally checkable if, using only their immediate neighborhood, (i) every node agrees the network is correct when $G \in \mathcal{F}$, and (ii) at least one node detects the network is incorrect if $G \notin \mathcal{F}$. While a *local algorithm* is traditionally allowed to access a constant-size $k$-neighborhood, for our discussion of overlay networks, we consider only $k = 1$.

For the $\texttt{Avatar}_\mathcal{F}$ family of networks, we assign proof labels based upon the ordering of nodes in $V$. Specifically, let the proof label $P_V(u_i)$ for a node $u_i \in V$ be defined as follows. For a set of nodes $V$, $P_V(u_i)$ $(u_i \in V)$ is a tuple $(u_{i-1}, u_{i+1})$, where $u_{i-1}$ is the *predecessor* for $u_i$ ($\perp$ for $u_0$), and $u_{i+1}$ is the *successor* of $u_i$ ($\perp$ for $u_{n-1}$). Using $P_V$, $\texttt{Avatar}_\texttt{F}$ is locally checkable for all full graph families $\mathcal{F}$. We prove this claim below.

**Theorem 4.1.** *Let $\mathcal{F}$ be a full graph family. For any $N \in \mathbb{N}$ and $V \subseteq [N]$, $\texttt{Avatar}_\mathcal{F}(N, V)$ is locally checkable with proof labels $P_V$.*

*Proof.* To prove this lemma, we describe verifier $\mathcal{A}_\mathcal{F}$. Notice that a verifier, given $u_i$ and $(N_G(u_i), P_{V,G}(u_i))$ can compute the range of all $u_j \in N_G(u_i)$. Therefore, $\mathcal{A}_\mathcal{F}$ can calculate $\Phi^{-1}(u_j)$ for all nodes $u_j \in N_G(u_i)$. It is easy to see that this information is sufficient to determine (i) if all type (1) and (2) edges are present, and (ii) if any additional edges are incident upon $u_i$. If there exists an edge incident upon $u_i$ such that the edge is not part of the realization of the dilation-1 embedding *or* is not $u_i$'s successor, then $\mathcal{A}_{\mathcal{F}}$ returns 0. Similarly, if there exists a node $b \in V(F_N)$ such that $b \in \Phi^{-1}(u_i)$, the edge $(b, b') \in E(F_N)$, and $u_i$ does not have an edge to a node $u_j$ with $b' \in \Phi^{-1}(u_j)$, then $\mathcal{A}_{\mathcal{F}} = 0$. Otherwise, $\mathcal{A}_{\mathcal{F}} = 1$. Therefore, $\texttt{Avatar}_{\mathcal{F}}$ is locally checkable for any full graph family $\mathcal{F}$.

### 4.3.1 Implementing Proof Labels in the Self-Stabilizing Model

While proof labels make $\texttt{Avatar}_{\mathcal{F}}$ locally checkable, their use is not entirely consistent with the self-stabilizing model. Specifically, we cannot assume the existence of an oracle to assign these proof labels, nor can we assume the adversary cannot corrupt them. Ideally, our verifier would be able to detect a fault if either (i) the network was faulty, or (ii) the proof labels were faulty. We show in this section that there exists a verifier for the conjunction of correct proof labels and topology – $\texttt{Avatar}_{\mathcal{F}}$ is locally checkable even if proof labels are able to be corrupted by an adversary.

Let $P.u$ be a *mutable* proof label for node $u$, and let $P_G[u]$ be the collection of $P.u$ from all nodes distance at most 1 from $u$ in $G$. Since $P.u$ is mutable, it may be corrupted by the adversary and may not always be equal to $P_V(u)$. We define next the notion of a locally checkability with mutable proof labels.

**Definition 4.4.** *Let $\mathcal{F}$ be a family of graphs. We say that $\mathcal{F}$ is* locally checkable with mutable proof labels *if there exists a function $\mathcal{A}_{\mathcal{F}}$ which takes as input $u_i$ and $(N_G(u_i), P_G[u_i])$ and returns a binary value $\mathcal{A}_{\mathcal{F}}(u_i, (N_G(u_i), P_G[u_i]))$ such that (i) if $G \notin \mathcal{F}$* or *there exists a node $u_k$ with $P_V(u_k) \neq P.u_k$, then $\exists u_j \in V(G)$ : $\mathcal{A}_{\mathcal{F}}(u_j, (N_G(u_j), P_G[u_j])) = 0$, and (ii) if $G \in \mathcal{F}$* and *$\forall u_k : P.u_k = P_V(u_k)$, then $\forall u_j \in V(G) : \mathcal{A}_{\mathcal{F}}(u_j, (N_G(u_j), P_G[u_j])) = 1$.*

**Theorem 4.2.** *The* `Avatar`$_{\mathcal{F}}$ *family of networks is locally checkable with mutable proof labels.*

*Proof.* Again, we describe the verifier $\mathcal{A}_{\mathcal{F}}$. First, notice that if node $u$ has $P.u[succ] = v$, and either $v \notin N(u)$ or $\exists w \in N(u) : v < w < u$, then $\mathcal{A}_{\mathcal{F}}$ will return 0. Similarly, if $P.u[pred] = v$ and either $v \notin N(u)$ or $\exists w \in N(u) : u < w < v$, $\mathcal{A}_{\mathcal{F}} = 0$. Therefore, either all proof labels are equal to the value of a correct neighbor or $\perp$ (when no $w < u$ or $u < w$ exists in $N(u)$), or $\mathcal{A}_{\mathcal{F}}$ returns 0.

To verify, then, that the values for $P.u[succ]$ and $P.u[pred]$ are correct is equivalent to checking if the graph induced by successors and predecessors forms the network family `Linear`. Since `Linear` is locally checkable, if the proof labels do not induce the `Linear` family, then there must exist at least one node $u$ such that $\mathcal{A}_{\mathcal{F}}$ returns 0. Therefore, if the mutable proof label $P.u$ does not match $P(u)$, there must exist at least one node $u$ for which $\mathcal{A}_{\mathcal{F}}(u, (N_G(u), P_G[u])) = 0$. The local checkability of the network when all $P.u = P(u)$ follows from Theorem 4.1.

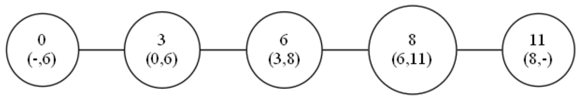


Figure 4.3: A faulty configuration. Node 0 has a faulty proof label, while node 6 is missing a neighbor.

#### 4.3.1.1 An Example Faulty Configuration

To see how mutable proof labels are locally checkable, consider Figure 4.3, which is a faulty network containing both faulty proof labels and faulty neighborhoods. First, notice that node 0 has a faulty proof label – its proof label indicates its successor is 6, while the actual successor should be 3. Node 0 can locally detect this. Secondly, notice that node 6 should have an edge to a node $u$ such that 11 from `Cbt` is embedded onto $u$. However, node 6 can detect that its neighbor 8 does *not* have 11 embedded onto it, and therefore node 6 is also a detector.

### 4.3.2 Proof Labels and Silent Stabilization

As mentioned earlier, a *silent* self-stabilizing algorithm, informally, is one which eventually ceases computation in a correct configuration. In the message-passing communication model, messages may continue to be exchanged to verify the system is in a legal state, but no other state should change at each node. Prior models have assumed that all of a node's state information is exchanged with all neighbors in every round. Even for networks where neighborhoods were small (logarithmic), sharing *all* state information can result in $\mathcal{O}(\log^3 n)$ bits begin sent in each round.

By explicitly noting a node's proof label, we give an easy way to determine what information nodes will share in a stable (correct) configuration. That is, if nodes constantly share only their proof labels, faulty configurations can still be detected, and nodes may share a much smaller amount of information. In our case, a node shares only $\mathcal{O}(\log N)$ bits per neighbor. For networks with neighborhoods of $\mathcal{O}(\log N)$ (such as $\texttt{Avatar}_{\texttt{Cbt}}$), we require only $\mathcal{O}(\log^2 N)$ bits to be sent per round in the correct ("silent") states. Once a node detects a fault, we allow (potentially) more information to be exchanged during convergence.

## 4.4 A Self-Stabilizing `Avatar` Network

In this section, we present the self-stabilizing algorithm for the $\texttt{Avatar}_{\texttt{Cbt}}$ network. Using several distributed computing techniques, including clustering, wave propagation, and randomized symmetry breaking, the algorithm achieves polylogarithmic convergence while limiting degree growth to at most a polylogarithmic factor from optimal.

### 4.4.1 Virtual and Real Nodes

Definition 4.1 gives a *structural* definition of the correct `Avatar` network. In this definition, the state of a node consists solely of (i) its neighborhood, and (ii) its proof label. The algorithm, however, requires every node in the network to maintain additional state, even in the correct `Avatar` configuration. We describe how we augment the state of every node below.

The algorithm uses the notion of virtual nodes (as done by Kniesburges et al. [25] and Trehan [37]). Virtual nodes are a convenient abstraction for the embedding of the guest network $F_N$ on the host network. Unlike an embedding (which is simply a mapping), virtual nodes are actual entities. Virtual nodes can execute actions and they have their own state, including their own neighborhoods. A node $a \in [N]$ from the guest network is simulated by a node $u \in V$. We call node $u \in V$ a *real node*, and say it *hosts* a *virtual node a* (denoted $host_a = u$). In almost all instances, the algorithms will assume they are executed on these virtual nodes.

We describe the implementation of these virtual nodes next. A real node $u$ hosts a set of virtual nodes, stored in the variable $Virtual_u$ – that is, virtual nodes are simply part of the state of real nodes. Every real node $u$ executes an independent program for each virtual node in $Virtual_u$, and each virtual node has its own state and neighborhood $N_b$, containing the virtual nodes $b$ is connected to. Each virtual node $b$ also has an associated identifier $id_b$. Notice that these identifiers need not be unique in a faulty state – there may be other virtual nodes with the same identifier. Virtual nodes also keep two variables representing their (mutable) proof labels, $succ_b$ and $pred_b$. Virtual nodes can also access the state of their host. Virtual node $b$ accesses its host's variable $var_0$ with the statement $host_b.var_0$. Finally, because `Cbt` is full and every node knows $N$, every virtual node from `Cbt`$(N)$ can calculate its (at most 3) correct neighbors in the complete binary search tree.

We now describe the legal state $\texttt{Avatar}_{\texttt{Cbt}}$ network *including* the virtual nodes. Assume the network configuration matches the legal configuration described in Definition 4.1. Every real node $u_i$ now also maintains a set of virtual nodes $Virtual_{u_i} =$

$\Phi^{-1}(u_i)$ such that, for each $b \in \mathit{Virtual}_{u_i}$, $N_b$ matches the neighborhood of $b$ in $\texttt{Cbt}(N)$. We can then think of a legal $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ configuration not as a network with node set $V$, but rather as a network of virtual nodes $[N]$, with each virtual node being simulated by a particular real node from $V$ (the assignment matches the embedding). Since the embedding is locally checkable, the assignment of virtual nodes is also locally checkable – e.g. a node can easily determine if $\mathit{Virtual}_u$ matches $\Phi^{-1}(u)$.

Since virtual nodes are state, and an adversary can corrupt state in the self-stabilizing model, initial configurations may consist of an arbitrary number of virtual nodes. The algorithms will add and delete virtual nodes, and execute actions on these virtual nodes, until eventually the network consists of exactly $[N]$ virtual nodes assigned to real nodes as described above. Thinking of the network not as a collection of real nodes, but rather as a collection of *independent virtual nodes*, will simplify the design and analysis of the algorithms.

#### 4.4.1.1 Virtual Edges

A natural question when using virtual nodes is how are edges handled in the network – are edges between real nodes or virtual nodes? We shall assume that edges exist between virtual nodes. We say an edge $(a, b)$ is a *virtual edge* if $a$ and $b$ are virtual nodes. A virtual edge $(a, b)$ in the guest network $G_g$ is realized by an edge $(\mathit{host}_a, \mathit{host}_b)$ in the host $G_h$ – that is, $(a, b) \in G_g \Rightarrow (\mathit{host}_a, \mathit{host}_b) \in G_h$. We shall assume that the initial configuration from the adversary includes these virtual edges.

Since an adversary can modify both the topology and the local states of nodes, it can easily make a real edge $(u, v)$ in the real network, and also have nodes $u$ and $v$ believe this edge is the realization of some set $E_{((u,v)}$ in the virtual network.

Two virtual nodes can create an edge between them if their hosts are at most distance 2 from each other (a superset of the condition where two virtual nodes are distance 2 from each other). This edge creation corresponds with an edge being created in the host network, or an existing real edge realizing another virtual edge. Therefore, a node $c$ creating the virtual edge $(a, b)$ corresponds to $host_c$ informing $host_a$ and $host_b$ that an edge should be created between virtual node $a$ and virtual node $b$. If $host_a$ and $host_b$ are already connected, no changes occur in the real network – only the local state of $host_a$ and $host_b$ is updated. If $host_a$ and $host_b$ are not connected, the edge $(host_a, host_b)$ is added to the real network *and* the virtual edge $(a, b)$ is added. Finally, when virtual nodes $a$ and $b$ are hosted by the same real node $u$, edges can be created and shared between nodes $a$ and $b$ by program actions in a single round.

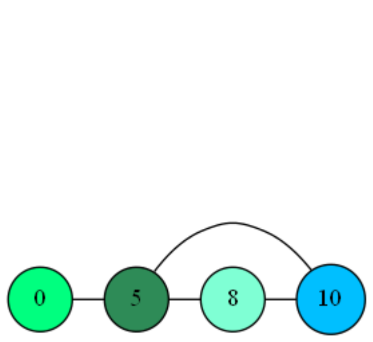
Since each virtual node knows its virtual neighborhood, and a real node can read the state of its hosted virtual nodes (in fact, virtual nodes *are* state, but for purposes of abstraction, we think of them as separate entities), every real node can determine which virtual nodes are using any incident edge. There may be multiple virtual edges realized by a single real edge. If the virtual edge $(a, b)$ is deleted, the edge $(host_a, host_b)$ is either deleted (if no other virtual edge is realized by it), or no change in the real network topology is made (only changes to the states of $a$ and $b$).

Notice that, in the self-stabilizing model, the adversary may arbitrarily modify the local state of any node, meaning there may exist edges incident upon a real node $u$ that are not incident upon any virtual node hosted by $u$. These faulty configurations are easily detectable and repairable by $u$ – $u$ will detect an incident real edge that is not realized by any hosted virtual node, and can assign this edge to a hosted virtual node.

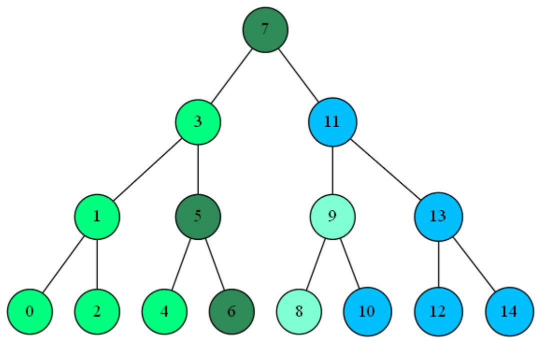
#### 4.4.1.2 Two Examples

Figure 4.4 contains an example of (i) the real nodes of a $\texttt{Avatar}_{\texttt{Cbt}}$ network, and (ii) the virtual nodes corresponding to these real nodes. The colors of the $N$ virtual nodes correspond with the host of the nodes – each virtual node with the same color is hosted by the same real node. Each virtual node operates independently, regardless of the host – for instance, an action may "exchange" messages between nodes 0 and 1 in a single round, even though 0 and 1 are hosted by the same real node. Again, our algorithms will think about this system not as a collection of $n$ real nodes, but rather as a collection of $N$ *independent* virtual nodes.

An adversary may create configurations where there are an arbitrary number of virtual nodes for each real node, and these virtual nodes are "connected" in an arbitrary fashion. To see an example of this, consider Figure 4.5. Here, the boxes represent the real nodes, while the nodes inside the boxes represent the adversarially-assigned virtual nodes. As the figure shows, some real edges in the network do not have corresponding virtual edges (the edge between real nodes 3 and 25). Further-



(a) The real nodes of $\texttt{Avatar}_{\texttt{Cbt}}(15, V)$



(b) Virtual nodes for $\texttt{Avatar}_{\texttt{Cbt}}(15, V)$. Colors represent hosts.

Figure 4.4: A legal $\texttt{Avatar}_{\texttt{Cbt}}(15, V)$ network, for $V = \{0, 5, 8, 10\}$

more, every real node is hosting an arbitrary number of virtual nodes, and these virtual nodes may or may not have incident edges. The algorithm starts from any initial state, including the one shown in Figure 4.5, creates and deletes virtual nodes and edges (and therefore modifies the underlying real network), and converges to a state where the virtual network (and therefore, the real network) match a correct configuration like shown in Figure 4.4.

### 4.4.2 Summary of the Algorithm

The self-stabilizing algorithm for $\texttt{Avatar}_{\texttt{Cbt}}$ uses a technique similar to the technique used in the algorithm for distributed minimum-weight spanning tree construction by Gallager, Humblet, and Spira (1983) [18] – the network is organized into disjoint clusters, each with a leader which coordinates the merging of clusters. After a short amount of time, only a single cluster remains, and the system is in a correct configuration.

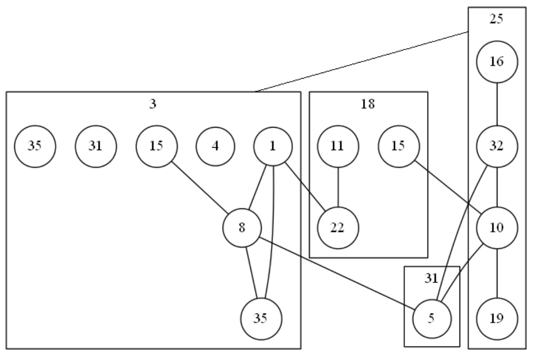


Figure 4.5: An arbitrary initial configuration of 4 real nodes and their internal state with virtual nodes.

We divide the algorithm into three components. First, we divide the network into clusters. Note that, in the self-stabilizing model, there is no guarantee that all nodes are members of a cluster when the program begins. There may be arbitrary initial configurations where nodes are not members of a cluster, and yet are unable to detect this. Therefore, we present a process we call *reset*, which guarantees that, regardless of the initial configuration, after a short amount of time every node in the network is a member of a cluster. This critical component simplifies the design and analysis of the remaining modules of the algorithm.

Once the network consists entirely of clusters, we repeat two steps: matching clusters together, and then merging these matched pairs (we limit merges to pairs of clusters only, which we shall explain later). Note that matching for any graph is a difficult problem to solve efficiently without using randomization. Therefore, we use a randomized symmetry-breaking algorithm to quickly compute a matching between clusters. There are topologies, however, where a maximal matching is small, perhaps even of constant size. In these cases, convergence may be linear. In the overlay network model, however, we can add edges to these topologies to ensure that we can create a matching that is sufficiently large ($\mathcal{O}(n)$), and therefore guarantee fast convergence. The concern with this approach is that too many edges may be added, causes nodes to have large degrees. The algorithm randomly assigns leaders, and allows only leaders to add edges to the network. As we shall see, this is sufficient to ensure (i) degrees remain small in the network, and (ii) matchings are large enough to allow efficient convergence.

Once two clusters have been matched, the algorithm merges them together into a single cluster. Merging is done in a systematic fashion so as to ensure that, regardless of the initial topology of each cluster, degrees remain small. The first step of a merge is a pre-processing step, which deletes all edges between two clusters (with the exception of a single special edge). Next, two clusters merge together, building the correct network level-by-level (with regards to the `Cbt` network). Once the clusters have merged, the process of matching and merging repeats.

Note that while our implementation is for an embedding of the `Cbt` network family, this technique is easy to extend to other topologies. To implement $\texttt{Avatar}_{\mathcal{F}}$ for a network $\mathcal{F} \neq \texttt{Cbt}$, one can simply modify these three components.

### 4.4.3 Defining a Cluster: The Subtree

Before describing the algorithm, we must define precisely the notion of a cluster. There are several important properties we wish to maintain with our clusters. First, since nodes in a cluster must communicate with the cluster leader, we would like a mechanism to exist for efficient intra-cluster communication. Furthermore, we would like an efficient way for two clusters to be merged together into a single cluster. Finally, if the network consists of a single cluster, the correct $\texttt{Avatar}_{\texttt{Cbt}}$ network should not be too difficult to create.

These requirements lead us to the cluster for the $\texttt{Avatar}_{\texttt{Cbt}}$ network, which we call a subtree. We define subtrees below.

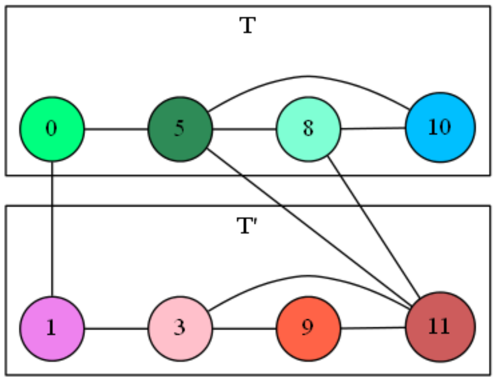
**Definition 4.5.** *Let $G$ be a graph with node set $V$. A* subtree *is set of nodes $V' \subseteq V$ in graph $G$ such that $G[V']$, the subgraph of $G$ induced by $V'$, is $\textit{Avatar}_{\textit{Cbt}}(N, V')$, and the* virtual nodes *hosted by nodes in $V'$ form the correct $\textit{Cbt}(N)$ network, and each virtual node $b$ is hosted by $u_i$, where $b \in \Phi^{-1}(u_i)$ in $\textit{Avatar}_{\textit{Cbt}}(N, V')$.*

One can think of a subtree as an embedding of $\texttt{Cbt}(N)$ onto $V' \subset V$. Any algorithm which runs on $\texttt{Cbt}(N)$ can be executed not only on $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$, but also on any subtree $\texttt{Avatar}_{\texttt{Cbt}}(N, V')$. Furthermore, since our algorithm uses virtual nodes, which can be created and deleted with program actions, we can think of each subtree as a set of virtual nodes $[N]$. The goal of the algorithm is take the at-most $n$ subtrees of the network, each consisting of virtual nodes $[N]$, and merge them together until eventually the set $V$ is a subtree. By definition, the network is correct at this point. We use the term *subtree* to help highlight the fact that our clusters are actually $N$ node `Cbt` networks (when considering virtual nodes), and not simply a grouping of real nodes.

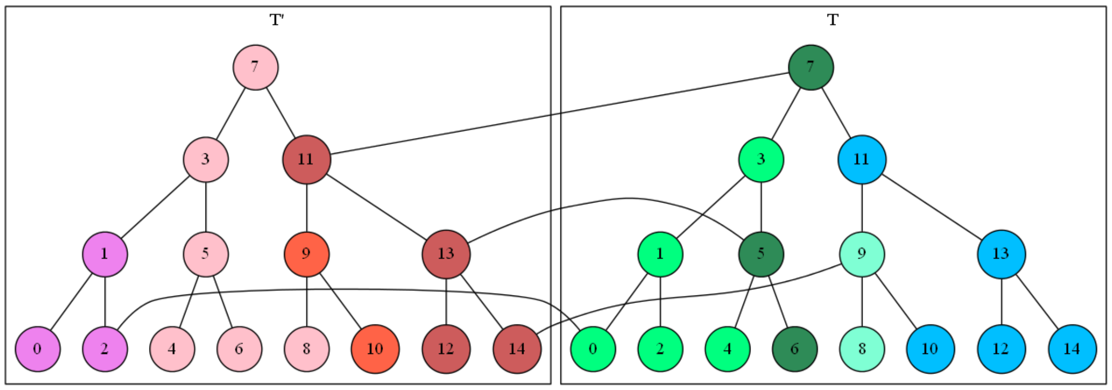
To clarify the notion of a subtree using both its real and virtual nodes, consider Figure 4.6. Figure 4.6(a) contains a graph $G$ with two subtrees – one consisting of nodes in various shades of green, and one consisting of nodes shaded red. The virtual nodes for these two subtrees is given in Figure 4.6(b). Each of the two subtrees is represented as its own $\texttt{Cbt}(N)$ graph. An edge in $G$ corresponds to a connection between nodes in one of the two $\texttt{Cbt}(N)$ subtrees.

Notice that it may be possible to partition $V$ into clusters several different ways. Trivially, the set $V' = \{u\}$ is a subtree. To design an algorithm to merge clusters, however, requires nodes to identify themselves as members of only one cluster. To do this, the algorithm uses the variable $tree_u$ for each $u \in V$, which we call the *tree identifier.* The tree identifier is the identifier of node $u_i$, where $u_i \in V'$ and $\lfloor N/2 \rfloor \in \Phi^{-1}(u_i)$ for $\texttt{Avatar}_{\texttt{Cbt}}(N, V')$ – that is, the tree identifier for subtree $T$ is the host of the root of the virtual subtree. Note that the virtual nodes of a subtree also have access to the tree identifier.

Finally, we present an intra-subtree communication mechanism in Section 4.4.7. This communication mechanism can guarantee that either (i) a message from the root is correctly delivered to all nodes in the subtree, and the root receives feedback from all nodes, or (ii) some node detects a fault. If the communication mechanism is not faulty, we shall say it is *consistent* (more on this in Section 4.4.7). A subtree in the algorithm should (i) have the correct real and virtual node structure,



(a) The real nodes of subtrees $T$ (top) and $T'$ (bottom)



(b) Virtual nodes for $T$ (right) and $T'$ (left)

Figure 4.6: Two subtrees $T$ and $T'$, and the virtual nodes hosted by nodes from $T$ and $T'$. Colors relate a virtual node to its host. Notice a single real edge may correspond with multiple virtual edges. For instance, the edge $(5, 11)$ in the real network corresponds to $(5_T, 13_{T'})$ and $(7_T, 11_{T'})$ in the virtual network.

(ii) have the correct tree identifier, and (iii) have a consistent communication state. We formalize this notion in the following definition.

**Definition 4.6.** *A set of nodes $T$ is called a* proper subtree *when, for each $b \in T_{\mathcal{C}bt(N)}$, (i) $b$ is neighbors with all calculated subtree neighbors, (ii) all calculated subtree neighbors have the same tree identifier as $b$, (iii) the PFC state of $b$ is consistent, (iv) every real node $host_b$ hosts all virtual nodes between itself and its successor in $T$, and (v) all non-subtree neighbors have a different tree identifier.*

The algorithms below will assume that the network consists entirely of proper subtrees. We explain later how we ensure this, and prove it is ensured in the analysis section.

### 4.4.4 Merging Subtrees

The key component of the algorithm is to merge together subtrees until eventually all nodes $u \in V$ are members of the same subtree. In the following section, we describe how subtrees are merged. In this section, assume that the initial configuration $G_0$ contains only subtrees. We discuss later how to relax this assumption.

We also assume the existence of a communication mechanism which the root of the subtree can use to (i) send information to all nodes in the subtree, and (ii) collect some information from nodes in the subtree. We describe this intra-subtree communication mechanism later.

#### 4.4.4.1 Merging Subtrees

The merge algorithm is based upon the observation that virtual subtrees $T_{\texttt{Cbt}(N)}$ and $T'_{\texttt{Cbt}(N)}$ each have $N$ nodes. If $T$ and $T'$ merge, the resulting subtree $T''$ will host $N$ virtual nodes, as well. Therefore, if $T$ and $T'$ merge, there must be $N$ virtual nodes from $T_{\texttt{Cbt}(N)} \cup T'_{\texttt{Cbt}(N)}$ that are deleted to form $T''_{\texttt{Cbt}}$. Merging two subtrees, each with $N$ nodes, into a single subtree with $N$ nodes involves *resolving* which virtual nodes will remain after the merge and which will be deleted. That is, there are two real nodes $u \in T$ and $u' \in T'$ such that both $u$ and $u'$ are simulating a virtual node $b$. When $T$ and $T'$ merge, either node $u$ or $u'$ will delete its virtual node $b$. The merging algorithm simply does an orderly comparison between virtual nodes from each subtree to determine the "best" virtual node for each. Informally, the "best" virtual node is the virtual node whose host's identifier is closest to the identifier of the virtual node. Formally, we define the relation $\prec$ as follows: for two virtual nodes $a_i$ and $a_j$, with $id_{a_i} = id_{a_j}$, $a_i \prec a_j$ if and only if either $id_{a_j} \leq host_{a_j} < host_{a_i}$ or $host_{a_j} \leq id_{a_j} < host_{a_i}$. When $a_i \prec a_j$, node $host_{a_j}$ is a "better" host than $host_{a_i}$ for a virtual node with identifier $id_{a_j}$, and therefore virtual node $a_i$ should be deleted.

A subtree $T$ may participate in at most one merge at any particular time. If subtree $T$ is merging with subtree $T'$, we say that $T'$ $(T)$ is the *merge partner* of $T$ $(T')$. We represent the merge partner of subtree $T$ with the variable $partner_a = T'$, for all $a \in T$. Subtree $T$ begins the merge algorithm once the root of $T$, $r_T$, is connected to the root of $T'$, $r_{T'}$, $partner_{r_T} = tree_{T'}$, and $partner_{r_{T'}} = tree_T$ (we say that $T$ has been assigned merge partner $T'$). We define the state of a subtree with regards to its merge partner below.

**Definition 4.7.** *Let $T$ be a proper subtree. $T$ is an* unmatched *proper subtree if and only if the root of $T$ has not been assigned a merge partner. Similarly, $T$ is a* matched *proper subtree if and only if the root of $T$ has been assigned a merge partner.*

Once $T$ ($T'$) is a matched proper subtree, $T$ ($T'$) executes a preprocessing step to (i) notify all subtree nodes that a merge with subtree $T'$ ($T$) is about to occur, and (ii) to delete all edges between nodes in $T$ and $T'$, excepting the edge connecting $r_T$ and $r_{T'}$. To begin the preprocessing, node $r_T$ communicates the identity of merge partner $T'$ to all nodes in subtree $T$. Node $r_T$ also communicates the value of the shared random sequence $L$ with all nodes in $T$. When a node $u \in T$ receives this information, it updates its $partner_u$ variable, as well as a variable $rs_u$, representing the received random sequence value. This random sequence value is used as a "randomized connectivity testing" procedure to delete the additional edges between nodes in $T$ and $T'$ without disconnecting the network. If a node $u$ is not the root node and detects an edge $(u, v)$ such that (i) $partner_u = tree_v$, (ii) $partner_v = tree_u$, and (iii) $rs_u = rs_v = L$, then $u$ deletes the edge $(u, v)$. We call such an edge a *matched edge between $T$ and $T'$*. Notice that, with some probability dependent upon the length of $L$, the network remains connected. If two neighboring nodes $u \in T$ and $v \in T'$ have a matched edge between them, then with some probability there must exist another path between $u$ and $v$ which will not be disconnected (specifically a path through $r_T$ and $r_{T'}$), and therefore the edge $(u, v)$ can be removed without partitioning the network.

Once this pre-processing has completed, subtrees $T$ and $T'$ are now connected with a single edge between their roots. The resolution process can begin. The resolution algorithm is designed to be executed upon the virtual nodes in $T_{\texttt{Cbt}(N)}$ and $T'_{\texttt{Cbt}(N)}$. We think of the resolution process as determining which $N$ virtual nodes from $T_{\texttt{Cbt}(N)}$ should remain, and which should be deleted, when forming the new subtree $T'' = T \cup T'$. The algorithm begins with the virtual root nodes (which are connected), and works recursively. For any two virtual nodes $c_0 \in T_{\texttt{Cbt}(N)}$ and $c_1 \in T'_{\texttt{Cbt}(N)}$ with identical identifiers, the resolution procedure simply compares the two virtual nodes to determine which should remain. Without loss of generality, assume that $c_0 \prec c_1$. We say that $c_1$ is the *winner* of the resolution process, and $c_0$ is the *loser*. The loser node is "replaced" by the winner node (more on this procedure in a moment), and the children of the loser node are copied to the winner node. Next, the winner connects its left child with the left child received from the loser, and connects its right child with the right child from the loser, and the resolution process is recursed concurrently on these two subtrees. This allows logarithmic (in $N$) running time, since all resolutions for virtual nodes at the same level of the tree complete in the same round. Figure 4.7 contains an example of this merging process on the virtual representation of subtrees $T$ and $T'$. Once this procedure reaches the leaves, of $T_{\texttt{Cbt}(N)}$ and $T'_{\texttt{Cbt}(N)}$, a new subtree $T'' = T \cup T'$ has been formed. Nodes in $T''$ are informed of their new subtree identifier, which is either $T$ or $T'$.

The merge process is given in Algorithm 4.1.

Notice that during the merge of $T$ and $T'$, nodes from $T$ and $T'$ may no longer

**Algorithm 4.1** The Merge Algorithm

```
    **Precondition:** $T$ and $T'$ are merge partners with connected roots.
1.  $root_T$ ($root_{T'}$) notifies $T$ ($T'$) of (i) merge partner $T'$ ($T$), and
        (ii) value of the shared random sequence.
2.  Remove all *matched edges between* $T$ *and* $T'$.
3.  $ResolveSubtree(root_T, root_{T'})$
4.  Once *ResolveSubtree* completes at leaves,
        inform nodes in new subtree $T'' = T \cup T'$ about new subtree identifier

$ResolveSubtree(a, b)$ : for $a \in T_{\texttt{Cbt}(N)}, b \in T'_{\texttt{Cbt}(N)}$
        // without loss of generality, assume $a \prec b$
1.      $ReplaceNode(a, b)$
        // Node $b$ is now connected to children of $a$.
        // Let $l_a$ ($r_a$) be the left (right) child of $a$,
        // and $l_b$ ($r_b$) be the left (right) child of $b$.
2.      Create edges $(l_a, l_b)$ and $(r_a, r_b)$;
3.      Concurrently execute $ResolveSubtree(l_a, l_b)$ and $ResolveSubtree(r_a, r_b)$
```

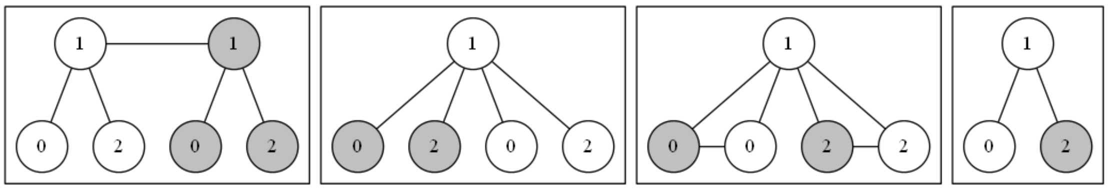


Figure 4.7: The steps of the resolution process between two trees $T$ and $T'$.

form a proper subtree. However, every node in the set $T \cup T'$ can determine that a merge is occurring – every node knows the identity of its subtree's merge partner, and can therefore detect when a parent or child has a wrong subtree identifier (in $T_{\texttt{Cbt}(N)}$), or when a successor has an incorrect subtree identifier (in $T$).

Finally, we discuss the way in which a loser node is "replaced" with the winner node. A simple method would be for the "loser" node to (i) create edges from its two children to the winner node, and (ii) delete itself. The resolution process would then continue on to the next level. Notice, however, that this procedure, while only causing a degree increase of 2 in the *virtual* network, causes a large degree in the underlying *real* network. To see this, imagine a subtree $T$ consisting of a single real node $u$ ($u$ is hosting $N$ virtual nodes). When the last (full) level of the tree resolves, $u$ will be hosting $N/2$ virtual nodes, each of which is connected to a virtual node from subtree $T'$. If each of these virtual nodes is hosted by a different real node, node $u$ will have a degree of $N/2$ in the real network. Therefore, the simple node replacement strategy using only our virtual node abstraction may result in high node degree. To avoid high degree in the real network, we must slightly violate the virtual node abstraction and take into account the underlying real network. We describe how this is done next.

Suppose a real node $u$ is executing the merge procedure for a hosted virtual node $b$, and suppose that $b$ is the loser of a resolution step with winner $b'$. The host of virtual node $b'$ must be a better successor (or predecessor) for node $u$ than it currently has. Therefore, node $u$ can not only delete its virtual node $b$, but it can also delete all virtual nodes from its new successor (or predecessor) and its old successor (or 0, if $u$ updated its predecessor). Of course, simply deleting these virtual nodes (and their incident edges) may lead to network disconnection. Therefore, a real node $u$, before deleting these virtual nodes, "copies" these nodes to its new-found successor (or predecessor) (note that the operation of copying virtual nodes can be compressed into the sending a range and the (at most $2 \cdot (\log N + 1)$) links to nodes outside that range). Since incident non-subtree edges do not affect the correctness of the new subtree $T''$, these incident non-subtree edges are copied to the virtual node with identifier equal to $host_b$, as done before (e.g. for every deleted virtual node hosted by 14, any edges not realizing a subtree edge are "copied" to virtual node 14). The resolution process in the virtual network, then, corresponds to the process of updating the real network to be a correct subtree, starting with the successor pointers. Informally, degrees remain low because (i) a node $u$ is set as the new successor of at most one node per level during a merge, (ii) there are $\mathcal{O}(\log N)$ edges transferred as the result of setting this new successor, and (iii) there are $\mathcal{O}(\log N)$ levels, resulting in polylogarithmic degree increases in the real network. We provide a full proof in the analysis section. We give the *ReplaceNode* procedure in Algorithm 4.2.

**Algorithm 4.2** The *ReplaceNode* Procedure

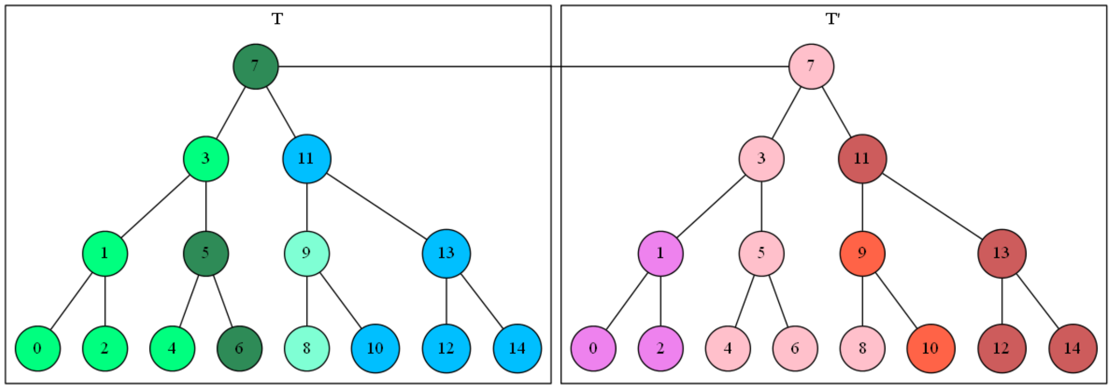
```
ReplaceNode(c, d) :
1.  if $partner_a \neq tree_b \vee partner_b \neq tree_a \vee rs_a \neq L \vee rs_b \neq L$ then
2.      Reset hosts of nodes $c$ and $d$, ending the merge process.
3.  fi
4.  if $host_d < succ_{host_c}$ then
5.      $succ_{host_c} = host_d$
6.      $LostNodes_c = \{b : host_b = host_c \wedge b > succ_{host_c}\}$
7.  else if $pred_{host_c} = \perp \wedge host_d < host_c$ then
8.      $pred_{host_c} = host_d$;
9.      $LostNodes_c = \{b : host_b = host_c \wedge host_d < b < host_c\}$
10. else // No successor pointer is updated
11.     Connect subtree children of $c$ to $d$; Delete node $c$
12. fi
    // Let $h$ be virtual node hosted by $host_c$ with $id_h = host_c$
13. for each $a \in LostNodes_c$ do
14.     Copy edges $(a, e) : tree_e \notin \{tree_c, tree_d\}$ to node $h$; Delete $(a, e)$;
15.     Copy node $a$ (including neighbors) to $host_d$; Delete $a$ from $Virtual_{host_c}$
16. od
```

#### 4.4.4.1.1 An Example Merge

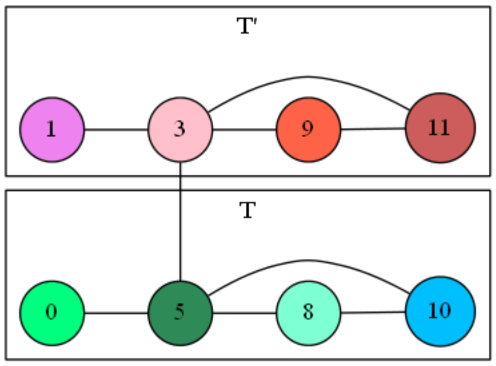
We now present an example merge of two subtrees $T$ and $T'$. We shall show both the real network during this merge, and the virtual $\texttt{Cbt}(N)$ networks to make clear how the abstraction updates the network in an orderly fashion. Assume subtrees $T$ and $T'$ prepare for a merge by (i) connecting the roots of $T_{\texttt{Cbt}}$ and $T'_{\texttt{Cbt}}$, and (ii) removing any additional edges between $T$ and $T'$. This configuration is shown in Figure 4.8. For the remainder of this example, we will use $b_T$ to denote a virtual node that was originally hosted by a member of subtree $T$.

The merge process begins with virtual nodes $7_T$ and $7_{T'}$ resolving. The winner of this resolution is $7_T$ (hosted by $5 \in T$), while the loser is $7_{T'}$ (hosted by $3 \in T'$). Real node 3 (i) updates its successor from 9 to 5, (ii) creates links from virtual nodes $3_{T'}$ and $11_{T'}$ to $7_T$, (iii) deletes virtual node $7_{T'}$, and (iv) transfers virtual nodes $\{5, 6, 8\}_{T'}$ to 5. Real node 5 is now hosting two virtual nodes with identifiers 5 and 6 – one each from $T_{\texttt{Cbt}}$ and $T'_{\texttt{Cbt}}$. This configuration is shown in Figure 4.9. In one additional round, virtual node $7_T$ will connect its two left children and two right children – $3_T$ and $3_{T'}$, and $11_T$ and $11_{T'}$, and the next level is ready to merge.

Next, virtual nodes $3_T$ and $3_{T'}$ resolve, as do $11_T$ and $11_{T'}$. For $3_T$ and $3_{T'}$, the winner is $3_{T'}$. Real node 0 (i) updates its successor to 3, (ii) connects children $1_T$ and $5_T$ to winner $3_{T'}$, (iii) deletes virtual node $3_T$, and (iv) send virtual node $4_T$ to real node 3. Similarly, $11_{T'}$ is the winner, so 10 (i) updates its successor to 11, (ii) connects children $9_T$ and $13_T$ to $11_{T'}$, (iii) deletes virtual node $11_T$, and (iv) sends virtual nodes $\{12, 13, 14\}_T$ to 11. This configuration is given in Figure 4.10. In the

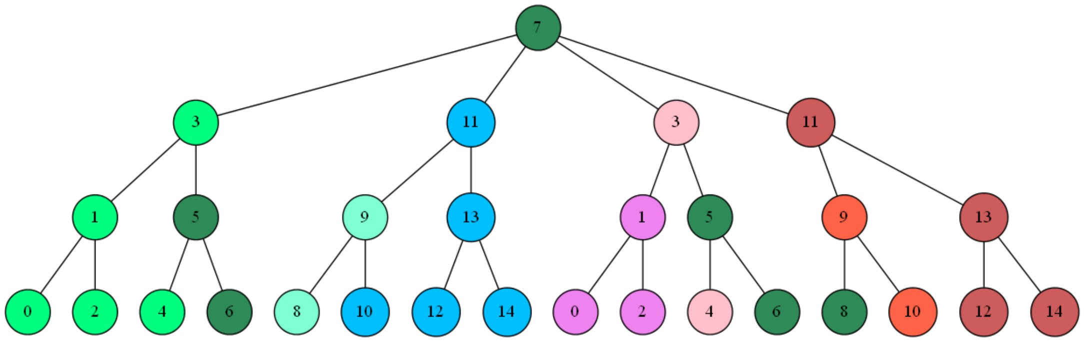


(a) The virtual nodes from $T$ and $T'$. Colors denote the hosts of each virtual node. Note that after preprocessing, only a single edge connects $T$ and $T'$.

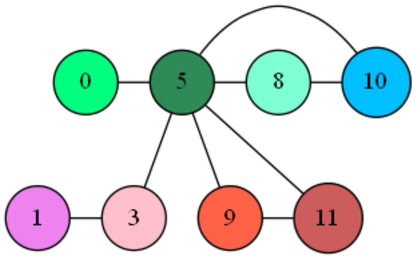


(b) The real nodes of subtrees $T$ and $T'$. Note each real edge may realize multiple virtual edges between virtual nodes, as shown in Part 4.8(a).

Figure 4.8: Subtrees $T$ and $T'$ are merge partners and have completed the preprocessing step.



(a) The virtual nodes after one stage of resolution. Virtual node $7_{T'}$ has been deleted, and several virtual nodes in $T'$ are now hosted by node 5 from $T$. In the next round, $7_T$ will add edges $(3_T, 3_{T'})$ and $(11_T, 11_{T'})$.



(b) The real nodes of subtrees $T$ and $T'$ after the first level has been merged. 3 has changed its successor from 9 to 5. In the next round, 0 and 3 are connected (as the result of the addition of virtual edge $(3_T, 3_{T'})$), as are 11 and 10 (as the result of added virtual edge $(11_T, 11_{T'})$).

Figure 4.9: The first level of `Cbt`$(N)$ has resolved during a merge.

next round, the winners connect their new children, preparing for the next resolution.

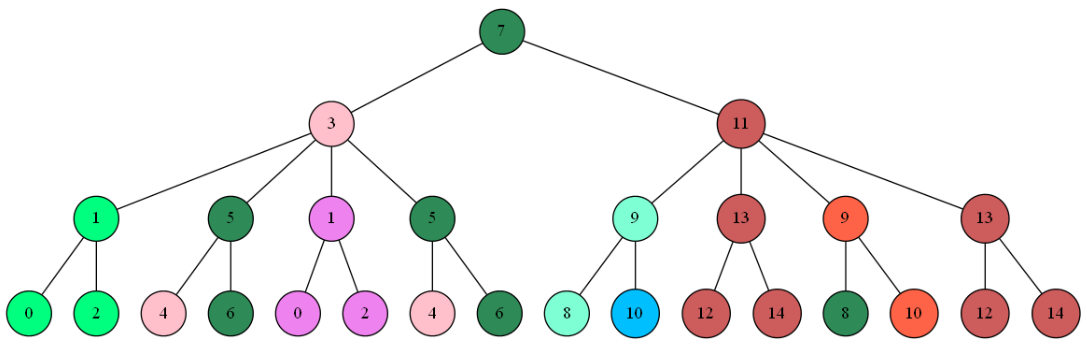
In the next round, virtual nodes 1, 5, 9, and 13 are resolved. The winners of the resolution are $1_{T'}$, $5_T$, $9_{T'}$, and $13_{T'}$. The steps are repeated as given above, resulting in the configuration shown in Figure 4.11.

Finally, the last level resolves. Notice the only resolutions that involve updating successors are between virtual nodes 0 and 8 – all other virtual nodes have the same host, and no successor updates are required. These final two nodes resolve, with $0_T$ and $8_T$ winning. Afterwards, the new subtree $T''$ exists, given in Figure 4.12.

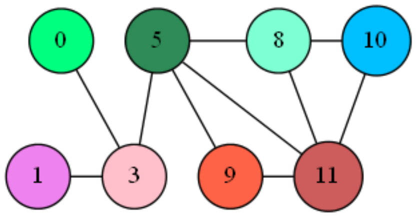
#### 4.4.4.2 Selecting Merge Partners

As mentioned earlier, every time a node $u$ becomes the new successor a node $v$, the real degree of node $u$ may increase by $\mathcal{O}(\log N)$. If a subtree is merging with $k$ subtrees at the same time, it is possible for node $u$ to become the new successor for $k$ other nodes concurrently. Therefore, to limit degree increase during merging, we limit merges to occurring only between pairs of subtrees. This ensures degree increases due to merges remains polylogarithmic regardless of a node's initial degree.

There are two challenges for assigning merge partners to subtrees. The first challenge is in finding a matching of subtrees. Imagine a graph where every node represents an entire subtree in the network, and edges between nodes represent edges between subtrees. Our goal is to find a large matching on this *subtree graph*. Finding a matching is a non-trivial problem, and is quite difficult to do quickly without using randomization. We use a randomized symmetry-breaking technique to facilitate the

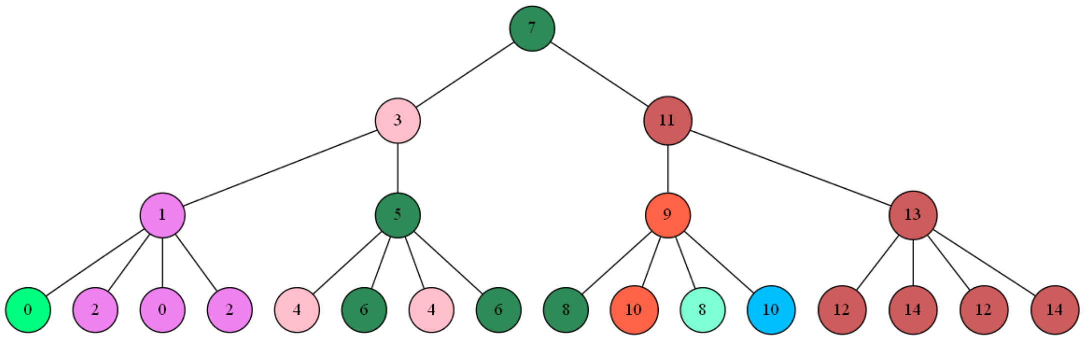


(a) The virtual nodes after two levels have been resolved. In the next round, the edges $(1_T, 1_{T'})$, $(5_T, 5_{T'})$, $(9_T, 9_{T'})$, and $(13_T, 13_{T'})$ will be added.

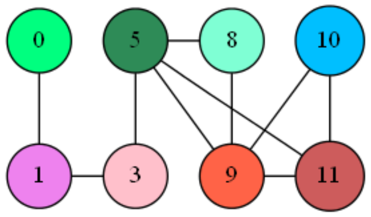


(b) The real nodes of subtrees $T$ and $T'$ after two levels have resolved during a merge. 0 has updated its successor from 5 to 3, while 10 creates a successor pointer to 11. In the next round, nodes 0 and 1 will be connected (from the virtual edge $(1_T, 1_{T'})$), as will nodes 8 and 9 (from virtual edge $(9_T, 9_{T'})$).

Figure 4.10: Subtrees $T$ and $T'$ after merging two levels.

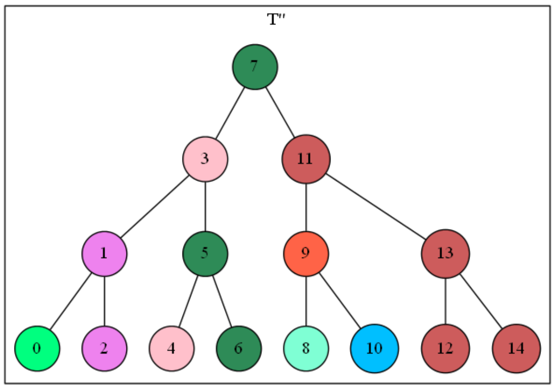


(a) The virtual nodes after three levels have been resolved. After one round, all leaves with the same identifier will be connected (e.g. $(0_T, 0'_T)$ will be added).

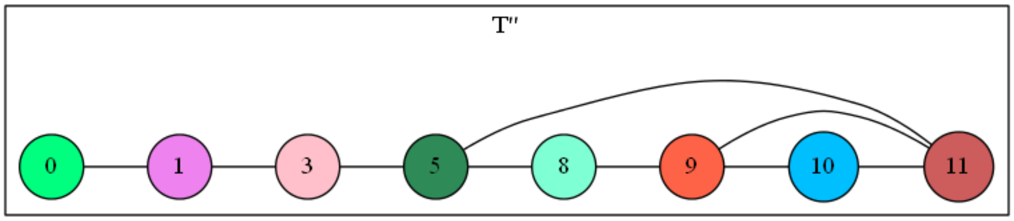


(b) The real nodes of subtrees $T$ and $T'$ after three levels have resolved during merge. Node 0 again updates its successor, now to node 1, while 8 has new successor 9. Note the connection of virtual nodes that occurs in the next round does not add any edge to this network.

Figure 4.11: Subtrees $T$ and $T'$ after three levels have been resolved.



(a) The virtual nodes after merging is complete.



(b) The real network after merging is complete. Node 9 added node 10 as a successor in the last resolution. Node 5 deletes virtual node 8 during resolution, causing the deletion of link to 9.

Figure 4.12: Subtrees $T$ and $T'$ merged together into $T''$.

matching. However, we cannot simply rely on standard matching techniques. There are topologies where even a *maximum* matching consists of only a small number of nodes. For instance, a star topology has a maximum matching of a single pair. If the algorithm operates by merging only a single pair of subtrees at a time, convergence time will clearly be at least linear.

The second challenge, then, is how to handle instances where even a maximum matching is insufficient for fast convergence. Note that, while a matching on the subtree graph may be small, we may be able to create a large matching in the square of the subtree graph – that is, the graph resulting from connecting all nodes in the subtree graph that are distance at most two. Notice in the overlay network model, we can create these edges between nodes at distance 2 in a single round. Therefore, we can transform configurations where even a maximum matching is small into configurations where a large number of nodes (subtrees) are able to be matched quickly. Of course, if too many edges are added to the network, we may no longer be able bound the degrees of nodes during convergence. Therefore, the algorithm designates some subtrees as "special" subtrees, and allows these subtrees to add edges to the network. By using a randomized symmetry-breaking technique, we can ensure there are sufficient number of these "special" subtrees to ensure large matchings are found quickly, while at the same time preventing the degree of any node from growing too large during convergence. We describe the algorithm next.

The algorithm works by focusing not on assigning merge partners directly, but by assigning subtrees a *role*, which can be either *leader*, *short follower*, or *long* *follower* (this role is assigned either randomly or deterministically, depending upon the configuration, as we show later). Consider first a *follower* subtree $T$. Nodes in a follower subtree search their neighborhood for a potential leader – a node from a leader subtree (how long nodes in $T$ search for a leader depends upon whether it $T$ is a short or long follower). The root of $T$ selects from the set of potential leaders a subtree $T'$, informs all nodes of $T$ that $T'$ is the leader, and connects to a node from $T'$. Subtree $T$ then waits for $T'$ to assign a merge partner $S$ (which will occur in a short amount of time, as we show in the analysis section). Note that, since a subtree can have at most one leader, only a single extra edge will be added to find $T$ a merge partner. $T$ and $S$ merge using the merge algorithm from above, and the selection process then repeats. If a follower subtree $T$ does *not* find a leader, it simply randomly selects a new role (selecting leader with probability 1/2, short follower with probability 1/4, and long follower with probability 1/4).

Nodes in a *leader* subtree $T$ announce to their neighbors that they are *open* – that is, they can be selected as a potential leader. After a short amount of time, $T$ stops being an open leader, and no new followers are added. Nodes in $T$ then assign all followers a merge partner. This is done by pairing together all followers. If there are an odd number of followers, one follower $S$ will merge with the leader subtree $T$. If there are an even number of followers, subtree $T$, after connecting all followers, simply randomly selects a new role (with probabilities given above) and repeats the selection procedure.

Notice that subtree $T$ is guaranteed to merge if (i) $T$ is a follower, and (ii) $T$

selects a leader. However, there are configurations where a subtree $T$ is either (i) a leader, or (ii) a follower, and has only follower neighbors. Since a leader remains a leader for only a short amount of time, the former case is handled by the fact that $T$ (if it does not find a merge partner) will select a new role in a short amount of time, resulting in a constant probability of $T$ being assigned a merge partner quickly. In the latter case, note that with constant probability $T$ is a long follower and has a neighbor $T'$ that is a short follower. In a short amount of time, then, $T'$ has at least a constant probability of becoming a leader, resulting in $T$ being assigned a merge partner quickly. We prove these claims more formally in the analysis section.

We now describe in more detail an implementation of this selection procedure. When a subtree sees that it is unmatched and the network is faulty, it selects (by way of the root node) one of three roles – short follower, long follower, or leader. Consider first the case where $r_T$, the root of subtree $T$, selects the role of leader. Node $r_T$ informs all nodes in $T$ of the new role, at which point nodes in $T$ are *open leaders*. If open leader $u$ has a neighbor $v$ such that $v$ is a follower, $v$ may tentatively select $u$ as its leader. After all nodes have been notified that $T$ is a leader, the root directs nodes in $T$ to assign merge partners to all followers. This is done with the procedure *ConnectFollowers*, which we describe later (in Section 4.4.7.1). The procedure *ConnectFollowers* assigns a merge partner to every follower of $T$ – if $T$ had an odd number of followers, $T$ is also assigned a merge partner. Therefore, after the leader algorithm has executed, either the subtree $T$ has been assigned a merge partner, or $r_T$ re-selects randomly a new role and continues program execution. The selection procedure for leaders is given in Algorithm 4.3.

**Algorithm 4.3** The Selection Algorithm for a Leader

```
   // Suppose the root $r_T$ of $T$ has selected the leader role
1. Node $r_T$ informs all nodes in $T$ of their role as leader.
2. Node $r_T$ initiates the ConnectFollowers procedure.
3. if $T$ was not assigned a merge partner during ConnectFollowers then
4.    $r_T$ randomly selects a new role
5. fi
```

Next, consider the case where root node $r_T$ selects one of the follower roles. As mentioned previously, there are two types of followers: short followers and long followers, named after how long they search for a leader. After selecting a follower role, the root $r_T$ informs all nodes in $T$ of the role, at which point every node $u \in T$ begins searching for a leader. The root periodically *polls* the subtree to see if anyone has found a leader. That is, the root queries all nodes in the subtree to see if they have found a leader. If a subtree node *has* found a leader, it informs the root of it. This is done using the communication mechanism described in Section 4.4.7. If $T$ is a short follower, the root polls the subtree at most twice, while if $T$ is a long follower, the root polls the subtree at most 12 times. The root no longer checks for a leader once one has been found.

A node in follower subtree $T$ selects a *potential leader* $b$ if either (i) $b$ is currently a leader and is open, or (ii) $b$ has already been assigned a merge partner. Nodes in $T$ inform the root when a potential leader is found, and the root node selects one. The identity of this selected node, called the *leader*, is shared with $T$. Furthermore, an edge to the leader is forwarded from some node in $T$ to the root $r_T$. If no potential leaders are found before the search expires, the root randomly selects a new role and continues execution. The selection procedure for followers is given in Algorithm 4.4.

Finally, we discuss how the role of a subtree $T$ is set. If $T$ was a follower and found no leader, or was a leader and was not assigned a merge partner, then $T$ randomly selects another role and repeats the procedure. *RandomRole* is a random variable that is assigned the value *short follower* with probability 1/4, *long follower* with probability 1/4, and *leader* with probability 1/2. If $T$ is completing a merge and discovers it has at least one tree $T'$ that is following $T$, then $T$ will become a leader.

### 4.4.5 Terminating the Computation

While the $\texttt{Avatar}_{\texttt{Cbt}}$ network is locally checkable, the algorithm presented above requires some nodes that detect a fault to execute no repair action. For instance, imagine two subtrees with a single edge connecting them, incident upon nodes $c \in T$ and $d \in T'$. Note that only $c$ and $d$ know the network is faulty, but neither will modify the edge $(c, d)$ until "instructed" to do so by our communication mechanism. This *delayed repair* mechanism is a key component of the algorithm, as it allows us to coordinate repairs to limit degree increases (e.g. only a select few neighbors of any

**Algorithm 4.4** The Selection Algorithm for a Follower

```
      // Assume the root $r_T$ of $T$ has selected a follower role (short or long).
1.    Root $r_T$ communicates the new role to nodes in $T$.
2.    **if** $T$ is a *short follower* **then**
3.        Root $r_T$ sets $pollCount = 2$
4.    **else** $T$ is a *long follower*:
5.        Root $r_T$ sets $pollCount = 12$
6.    **fi**
7.    **while** $pollCount > 0$ **and** no potential leader is found **do**
8.        Root $r_T$ queries subtree $T$ for a potential leader.
9.        **if** $u \in T$ found a potential leader $v \in T'$ **then** inform $r_T$ of $v$ **fi**
10.       $pollCount = pollCount - 1$;
11.   **od**
12.   **if** a potential leader is returned to $r_T$ **then**
13.       Root node $r_T$ selects one potential leader $v \in T'$, informs nodes in $T$ of $v$
14.       Nodes in $T$ forward any edge to $v \in T'$ to the root.
15.   **else** $r_T$ randomly selects a new role *RandomRole* **fi**
```

node $u$ may increase the degree of $u$ at any particular point in time).

The problem with this delayed repair, however, is that the only detector in the network may be waiting for the "command" to execute a repair action, but this command may never arrive (since no other node is a detector). To rectify this problem is quite simple – every node keeps (and shares) a single bit representing whether or not the configuration is correct. Let $correct_c$ be this bit. If either node $c$ detects a fault, or $c$ has a neighbor $b$ with $correct_b = 0$, then $correct_c = 0$. Since the network is locally checkable, all incorrect configurations have at least one node with this bit set correctly. Therefore, a faulty configuration will always have an enabled action. We include this $correct_c$ bit in the proof label of each node.

Finally, this bit must be set to 1 once the correct configuration is built. To do this, we can simply include in every invocation of the efficient communication mechanism a "fault detected" bit, which allows the root to easily check if any node in its subtree detects a fault in its immediate neighborhood. If a root discovers that no node in its subtree detects a fault, then the configuration is correct and the root can instruct them to set their $correct_c$ bit to 1.

### 4.4.6 Initializing the Subtrees

The algorithms presented thus far are intended to be executed on proper subtrees. Due to the arbitrary initial configuration, however, there may exist nodes in the network that do not belong to a subtree. We present an algorithm that ensures, after few rounds, the network consists entirely of subtrees and merging nodes (see Section

4.4.4). This ensures that the algorithms given above execute as intended even in a self-stabilizing setting.

This initialization process is called *reset*. Informally, a node executes a reset when it detects a configuration that it does not know how to recover from. Specifically, if a node $u$ detects (i) it is not part of a proper subtree, and (ii) it is not executing a merge, then $u$ executes a reset, becoming a subtree of size 1. Node $u$ can then resumes executing the algorithms given above.

**Algorithm 4.5** The Reset Algorithm

```
1.  if Reset fault detected and did not reset in previous round then
2.      $Virtual_u = [N]$
3.      $succ_u = \perp$; $pred_u = \perp$;
4.      for $a \in Virtual_u$ do
5.          Connect $a$ to Cbt neighbors from $Virtual_u$
```

### 4.4.7 Intra-Subtree Communication

To allow the cluster leaders to coordinate merges efficiently, we need an efficient method for communicating amongst nodes of the same cluster (subtree). The algorithms use wave propagation for communication, a natural choice for the tree topology. Specifically, we use *propagation of information with feedback and clean-*

*ing* (*PFC*) [10] for communication. This communication procedure is executed on the virtual subtree $T_{\texttt{Cbt}(N)}$. 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 information, and (iii) makes all nodes ready to execute another *PFC* wave. We assume this procedure is used whenever the root needs to inform nodes of the subtree of some information, or nodes in the subtree wish to communicate some information to the root. Note that nodes keep a state variable for each part of the *PFC* wave. These variables were used to ensure the communication mechanism is snap-stabilizing. For our purposes, they ensure that communication between the root and all other nodes occurs as expected regardless of the initial configuration – either there is a message "in transit", or at least one node detects a faulty configuration. This is important so that, for instance, no node is waiting for feedback that will not occur.

Notice that, while the original *PFC* algorithm was snap-stabilizing, we do not require this property. We choose instead to handle faulty configurations with the reset action discussed earlier.

#### 4.4.7.1 Implementing *ConnectFollowers*

To implement the *ConnectFollowers* procedure, we can use the *PFC* communication mechanism. It works as follows. First, a *PFC* wave is initiated by the root, and when nodes receive the propagate portion of the wave, they set their role to *ClosedLeader*, preventing any neighbors from selecting them as potential leaders. Once

**Algorithm 4.6** Subroutine for $PFC(I, F)$ on virtual subtree $T_{\texttt{Cbt}(N)}$

**Variable for Node** $a$**:** $\mathit{PFCState}_a$

```
1.  when root node root satisfies $\mathit{PFCState}_{\mathit{Children}(\mathit{root})} = \mathit{PFCState}_{\mathit{root}} = \mathit{Clean}$ then
2.      $\mathit{root}_T$ initiates $PFC$ wave by setting $\mathit{PFCState}_{\mathit{root}} = \mathit{Propagate}(I)$
3.      Each node $a$ executes the following:
4.          if $\mathit{PFCState}_a = \mathit{Clean} \wedge \mathit{PFCState}_{\mathit{Parent}(a)} = \mathit{Propagate}(I) \wedge$
              $\mathit{PFCState}_{\mathit{Children}(a)} = \mathit{Clean}$ then
5.              $\mathit{PFCState}_a = \mathit{Propagate}(I)$
6.          else if $\mathit{PFCState}_a = \mathit{Propagate}(I) \wedge \mathit{PFCState}_{\mathit{Parent}(a)} = \mathit{Propagate}(I) \wedge$
              $\mathit{PFCState}_{\mathit{Children}(a)} = \mathit{Feedback}(F)$ then
7.              $\mathit{PFCState}_a = \mathit{Feedback}(F')$
8.          else if $\mathit{PFCState}_a = \mathit{Feedback}(F) \wedge \mathit{PFCState}_{\mathit{Parent}(a)} = \mathit{Feedback}(F) \wedge$
              $\mathit{PFCState}_{\mathit{Children}(a)} = \mathit{Clean}$ then
9.              $\mathit{PFCState}_a = \mathit{Clean}$
10.         fi
11. fi
```

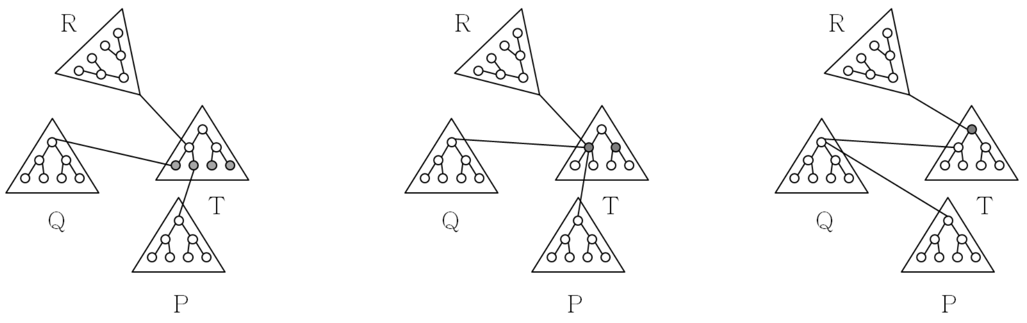


Figure 4.13: Example of the *ConnectFollowers* Procedure. Subtrees $Q$ and $P$ are assigned merge partners in the second round (middle figure), while $R$ is assigned merge partner $T$ in the third round (right-most figure).

the feedback wave begins, a node $b$ only sends the feedback wave up the tree if it has no neighboring node $c$ that is a potential follower of $b$. This means the feedback wave may be delayed slightly while neighbors decide to either (i) follow a node from $T$, or (ii) follow a different subtree $T'$. Notice, however, that this delay is short-lived, since no new potential followers are added.

After a node has no more neighboring potential followers, it assigns merge partners to all followers. If $b$ has an even number of followers, each is paired with one other follower. If $b$ has an odd number of followers, all but one of these followers are paired up, and the last follower is forwarded to the parent of $b$, who will continue the process. If the root of the subtree receives an odd number of followers, it pairs up all followers and itself. We give an example of this procedure in Figure 4.13.

We require a slightly modified *PFC* wave for the *ConnectFollowers* procedure.

We show this modification with the use of “Feedback” actions, which we assume a node executes before transitioning from the propagation to feedback state. We call this the *feedback action for node a*. We give the *ConnectFollowers* procedure in Algorithm 4.7.

**Algorithm 4.7** Subroutine *ConnectFollowers*

```
1.  Execute $PFC(\mathit{ConnectFollowers}, \perp)$:
2.      **Feedback Action for** $a$**:**
3.          **while** $\exists b \in N_a : \mathit{role}_b = \mathit{PotentialFollower}(a) \vee$
                $(\mathit{role}_b = \mathit{Follower}(a) \wedge b \neq \mathit{root})$ **do** skip; **od**
4.          Order the $k$ followers in $N_a$ by tree identifiers $b_0, b_1, b_2, \ldots, b_{k-1}$
5.          **for** $i = 0, 2, 4, \ldots, \lfloor k/2 \rfloor$ **do**
6.              Create edge $(b_i, b_{i+1})$; Set merge partner of $b_i$ to $b_{i+1}$ and vice versa
7.              Delete edge $(a, b_{i+1})$
8.          **od**
9.          **if** $k \mod 2 \neq 0$ **then**
10.             Create edge $(\mathit{parent}_a, b_{k-1})$ and delete edge $(a, b_{k-1})$
11.         **fi**
```

### 4.4.8 Summary

Figure 4.14 provides a transition diagram for the subtrees in the algorithm. To begin, assume a subtree $T$ randomly selects a role (node *Random Role Selection*). Suppose the subtree $T$ selects the role of leader. All nodes in $T$ become open leaders (node *Open Lead*), and can acquire followers. After a short amount of time, all followers of $T$ are connected (node *Connecting Followers*). $T$ may have been assigned a merge partner during this process, in which case it participates in a merge (node *Merging*). If $T$ was not assigned a merge partner, it simply randomly selects a new role and the process repeats.

Suppose a subtree $T$ selects the follower role. All nodes in $T$ begin searching for a neighboring open leader (node *Searching for Leader*). If at least one leader was found, the root of $T$ selects one neighboring leader, $T'$, informs all nodes of the leader, connects to a node from $T'$, and waits to be assigned a merge partner (node *Waiting for Partner*). After a short amount of time, leader $T'$ assigns $T$ a merge partner $S$, and $T$ and $S$ merge (node *Merging*).

Finally, after subtrees $T$ and $S$ complete a merge to form $R$, nodes check for any neighboring subtrees that have selected $R$ as a leader. If there is such a neighbor, $R$ assumes the role of leader (node *Open Leader*). If there is no such neighbor, $R$ randomly selects a new role.

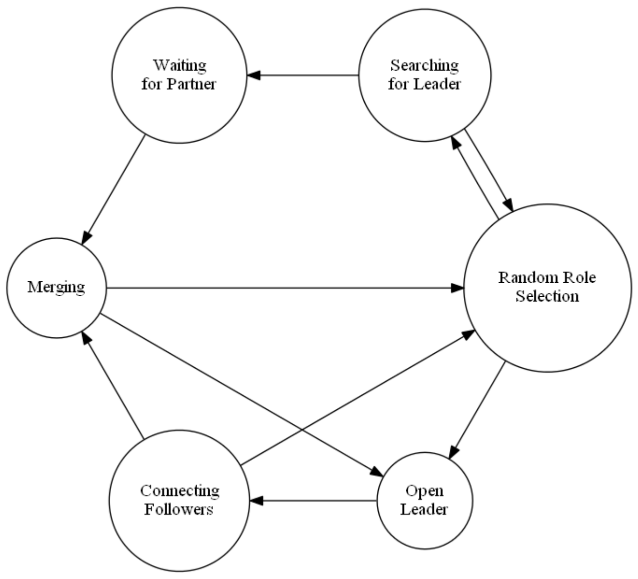


Figure 4.14: The states of a subtree in the algorithm.

## 4.5 Analysis

The self-stabilizing algorithm for the $\texttt{Avatar}_{\texttt{Cbt}}$ network presented in Section 4.4 converges in a polylogarithmic number of rounds in expectation, and has a degree expansion that is at most polylogarithmic. We prove these claims in the following section. Unless otherwise noted, all subtrees mentioned are *virtual subtrees*.

### 4.5.1 Convergence Time

We begin our analysis of the running time by showing that every node is part of a cluster (the subtree) in a short amount of time. We then show that a subtree merges with another subtree in a short amount of time (with constant probability), quickly reaching a configuration with a single subtree.

Before discussing the running time of reset and merge, we prove the following lemma concerning the running time of a *PFC* wave on a subtree $T$.

**Lemma 4.2.** *If the root of a subtree $T$ initiates the PFC$(I, F)$ wave without propagate or feedback actions, the PFC wave is complete (all nodes receive the information, the root receives the feedback, and nodes are ready for another PFC wave) in $2 \cdot (\log N + 1) + 2$ rounds.*

*Proof.* After the root initiates the *PFC*$(I, F)$ wave, in every round the information $I$ moves from level $k$ to $k+1$ until reaching a leaf. As the subtree has at most $\log N + 1$ levels, after at most $\log N + 1$ rounds, all nodes have received the propagation wave. Upon receiving the propagation wave, leaves set their *PFC* states to *Feedback* to begin the feedback wave. Again, in every round the feedback wave moves one level closer to the root, yielding at most $\log N + 1$ rounds before the root has received the feedback wave. Consider the transition to *PFC* state *Clean*. If a leaf node $b$ sets its *PFC* state to *Feedback*, in one round the parent of $b$ will set its state to *Feedback*, and in the second round, $b$ will set its state to *Clean*. The process then repeats for the parent of $b$. In general, two rounds after a node $b$ is in state *Feedback*, it transitions to state *Clean*. Therefore, $2 \cdot (\log N + 1) + 2$ rounds after the root initiates a *PFC* wave, all nodes receive the propagation wave, return the feedback wave, and move back to a clean state, ready for another *PFC* wave.

#### 4.5.1.1 Subtree Initialization

We begin by showing that all nodes are members of a proper subtree in a short amount of time, and then show this implies no further reset faults are executed. First, notice that no node in a proper subtree has a reset fault enabled.

**Definition 4.8.** *Let $T$ be a proper subtree. $T$ is a* proper clean subtree *if and only if all nodes in $T$ have a PFC state of Clean.*

**Lemma 4.3.** *Let node $b$ be a member of a proper subtree $T$. Node $b$ can only execute a reset action if $T$ begins the merging process from Algorithm 4.1.*

*Proof.* To begin, notice that no reset is executed when the subtree edges of $T$ remain and the *PFC* state remains consistent. Since program actions cannot cause a consistent *PFC* state to be inconsistent, a merge is the only action that can modify the configuration of a node $b$ such that $b$ executes a reset.

**Lemma 4.4.** *Let node $r_T$ detect locally it is the root of a proper subtree $T$ ($r_T$ has a consistent PFC state, and has two children with appropriate identifiers and tree identifiers). If $r_T$ initiates a $PFC(I, F)$ wave and later receives the corresponding feedback wave, then $r_T$ is the root of a proper subtree.*

*Proof.* Every node will only continue to forward the propagate and feedback waves if (i) they have the appropriate subtree neighbors, and (ii) their *PFC* states are consistent. If either of these conditions are violated (e.g. $T$ is not a proper subtree), then there exists at least one node which will execute a reset, making it no longer a part of the “subtree” $T$. Therefore, the *PFC* wave cannot complete when $r_T$ is not the root of a proper subtree.

**Lemma 4.5.** *If node $b$ is a member of a proper unmatched clean subtree $T$ in configuration $G_i$, then $b$ will never execute a reset in any configuration $G_j \in \mathcal{F}(G_i)$.*

*Proof.* By Lemma 4.3, only a merge can cause node $b$ to execute a reset fault. We show that any merge $b$ participates in must be between two proper subtrees, and therefore completes correctly (Lemma 4.16).

Suppose the root of $T$ has been matched with the root of another subtree $T'$. $T$ cannot begin modifying its subtree edges for the merge until both $T$ and $T'$ have successfully completed the $PFC(Prep(T, T'), \perp)$ wave. Suppose $T'$ was not a proper subtree. In this case, $T'$ would not successfully complete the $PFC$ wave (Lemma 4.4), and $T$ would not begin a merge with $T'$. Therefore, if $T$ and $T'$ merge together, both must be proper subtrees, implying the merge completes successfully and either $b$ has been deleted, or $b$ is again a member of a clean proper subtree $T''$ (see Lemma 4.16).

**Lemma 4.6.** *Consider a node $b$ that is not a member of a proper subtree in configuration $G_i$. In $\mathcal{O}(\log N)$ rounds, $b$ is a member of a proper unmatched clean subtree.*

*Proof.* If $b$ is not a member of a proper subtree and detects a reset fault in $G_i$, our lemma holds.

Consider the case where $b$ is not a member of a proper subtree but has no reset fault. If $b$'s neighborhood is either missing its parent or children, or has a parent or child with non-matching tree identifiers and $b$ does not detect a reset fault, then

$b$'s state must be consistent with performing a merge between trees $T$ and $T'$ from Algorithm 4.1. In 2 rounds, $b$ either has the correct subtree neighbors, each with tree identifier of either $T$ or $T'$, and has passed the merge process on to its children, or $b$ has executed a reset. The children of $b$ now either execute a reset, or are in a state consistent with the merge process, and we repeat the argument. As there are $\log N + 1$ levels, after $2 \cdot (\log N + 1)$ rounds either a node has reset due to this merging, or the merge is complete. If a node has reset in round $i$, its parent will reset in round $i + 1$, its parent's parent will reset in $i + 2$, and so on. After at most $\log N + 1$ rounds, $b$ resets and becomes part of an unmatched clean proper subtree.

If $b$ does not detect locally that it is not a member of proper subtree $T$, then there must exist a node $c$ within distance $2 \cdot \log N$ such that every node $p_i$ on a path from $b$ to $c$ believes it is part of the same subtree as $b$, and node $c$ detects an incorrect subtree neighborhood or inconsistent *PFC* state. As with the above argument, either $c$ detects a reset fault immediately, or $c$ is participating in a merge. In either case, after $\mathcal{O}(\log N)$ rounds, $c$ has either reset, or $c$ is a member of a proper unmatched clean subtree resulting from a successful merge.

Combining Lemmas 4.5 and 4.6 gives us the following lemma.

**Lemma 4.7.** *No node executes a reset action after $\mathcal{O}(\log N)$ rounds.*

We call a configuration $G_i$ a *reset-free configuration* if and only if no reset actions are executed in any configuration $G_j \in \mathcal{F}(G_i)$. We provide the following assertion to simplify the remaining proofs. Notice that this assertion will hold true after a logarithmic number of rounds (as shown above).

**Assertion 1.** *The proofs of Section 4.5.1.2 assume a* reset-free configuration.

#### 4.5.1.2 Merging

The next step of the analysis is to show that subtrees merge often enough to quickly reach a configuration with a single subtree. We show a proper subtree has constant probability of being matched with a merge partner within a logarithmic number of rounds. Once a subtree $T$ is assigned a merge partner $T'$, the merge completes within $4 \cdot \log N$ rounds, resulting in a single $N$-node subtree $T''$, with $V(T'') \subset V(T) \cup V(T')$.

First, we analyze the time required for a node $b \in T$ that has selected some neighboring subtree $T'$ as a potential follower for the root of $T$ to be connected to a node in a leader subtree $T'$.

**Lemma 4.8.** *Let $b \in T$ be a follower that has selected a neighbor $c \in T'$ as a potential leader. In at most $5 \cdot (\log N + 1) + 6$ rounds, the root of $T$ has an edge to some leader subtree $T''$, and all nodes in $T$ know this leader.*

*Proof.* When $b$ detects $c$ becomes a potential leader, $b$ marks $c$ as a potential leader immediately, regardless of the *PFC* state. After at most $2 \cdot (\log N + 1) + 2$ rounds, a feedback wave will reach $b$, at which point $b$ will forward the information about its potential leader. In an additional $(\log N + 1) + 2$ rounds, the *PFC* wave completes, at which point the root of $T$ has at least one potential leader (which may or may not be $c$) returned to it. The root of $T$ will select one returned leader and execute the leader-inform *PFC* wave. In $\log N + 1$ rounds, all nodes know the identity of their leader and its subtree. Subtree $T$'s selected leader $c'$ will be forwarded up the tree during the feedback wave, reaching the root in an additional $(\log N + 1) + 2$ rounds.

**Lemma 4.9.** *Let $r_T$ be the root of subtree $T$. If $r_T$ selects the role of Leader, within $9 \cdot (\log N + 1) + 10$ rounds either $T$ has been paired with a merge partner, or $T$ randomly selects a new role. Furthermore, all followers of $T$ have been assigned a merge partner.*

*Proof.* First, note that $PFC(Lead, \perp)$ requires $2 \cdot (\log N + 1) + 2$ rounds to complete (Lemma 4.2). The $PFC(ConnectFollowers, \perp)$ wave requires at most $7 \cdot (\log N + 1) + 8$ rounds. To see this, notice that the feedback action for this wave cannot advance past a node $b \in T$ until $b$ has no neighbors that are potential followers and all followers are root nodes. Let $T'$ be a follower subtree that has selected $T$ as a potential leader. By Lemma 4.8, after at most $5 \cdot (\log N + 1) + 6$ rounds, all potential followers of $b$ are either no longer following $b$, or are connected with a root to $b$.

Notice that the *total* wait for all nodes in $T$ is $5 \cdot (\log N + 1) + 6$, since all nodes in $T$ have a role of *ClosedLead* and are not assigned any additional potential followers. Therefore, the feedback wave can be delayed at most $5 \cdot (\log N + 1) + 6$ rounds, leading to a total $7 \cdot (\log N + 1) + 8$ rounds for the $PFC(ConnectFollowers, \perp)$ wave. Upon completion of the $PFC(ConnectFollowers, \perp)$ wave, all followers of $T$ have been assigned a merge partner. If there were an odd number of followers, $T$ has also been assigned a merge partner, else $T$ will randomly re-select a role.

**Lemma 4.10.** *Let $T$ be a short follower subtree. In at most $4 \cdot (\log N + 1) + 4$ rounds,* *either $T$ has selected a leader, or $T$ randomly re-selects a role.*

*Proof.* A short follower polls its subtree for a leader at most twice, each requiring $2 \cdot (\log N + 1) + 2$ rounds (Lemma 4.2). If a leader is not returned, $T$ will randomly select a new role. If at least one leader is returned, $T$ selects it.

**Lemma 4.11.** *Let $T$ be a long follower subtree. In at most $24 \cdot (\log N + 1) + 24$ rounds, either $T$ has selected a leader, or $T$ randomly re-selects a role.*

*Proof.* By similar argument to Lemma 4.10, a long follower polls its subtree at most 12 times, each requiring $2 \cdot (\log N + 1) + 2$ rounds (Lemma 4.2). If a leader is not returned, $T$ will randomly select a new role. If at least one leader is returned, $T$ selects it.

**Lemma 4.12.** *Let $T$ be a follower subtree that has returned a leader after a PFC search wave of Algorithm 4.4. After at most $16 \cdot (\log N + 1) + 16$ rounds, $T$ has a merge partner.*

*Proof.* Let $r_T$ be the root node of $T$. Node $r_T$ selects a returned leader $T'$ and, in $2 \cdot (\log N + 1) + 2$ additional rounds, all nodes in $T$ have been informed of leader $T'$ and $r_T$ has an edge to a node $b$ from $T'$.

After $root_T$ has an edge to a node $b \in T'$, $root_T$ waits to be assigned a merge partner. Suppose $T'$ had the role of leader when selected by $T$. By Lemma 4.9, after at most $9 \cdot (\log N + 1) + 10$ rounds, $T$ will be assigned a merge partner.

Suppose $T'$ was executing a merge when selected by $T$. $T$ will be assigned a merge partner when (i) $T'$ finishes its merge, and (iii) $T'$ finishes executing Algorithm

4.3. If $T'$ was merging, it completes in at most $5 \cdot (\log N + 1) + 4$ rounds (Lemma 4.16). The resulting subtree $T''$ will be a leader, and will require at most $9 \cdot (\log N + 1) + 10$ rounds before all followers have been assigned a merge partner (Lemma 4.9).

Since the initial configuration is set by the adversary, it can be difficult to make probabilistic claims when dealing with the initial configuration. For instance, the adversary could assign all subtrees the role of long follower, in which case the probability that a merge happens over $24 \cdot (\log N + 1) + 24$ rounds is 0. Notice, however, that after a short amount of time, regardless of the initial configuration, subtrees are guaranteed to have randomly selected their current roles. Therefore, we ignore the first $24 \cdot (\log N + 1) + 24$ rounds of execution in the following lemmas.

**Definition 4.9.** *Let $G_0$ be the initial network configuration. We define $\mathcal{F}_\Delta(G_0)$ to be the set of future configurations reached after $\Delta = 26 \cdot (\log N + 1) + 26$ rounds of program execution from the initial configuration.*

**Lemma 4.13.** *Let $T$ be a follower subtree in configuration $G_i \in \mathcal{F}_\Delta(G_0)$. With probability at least $1/2$, $T$ either randomly selects a new role or has found a leader in $4 \cdot (\log N + 1)$ rounds.*

*Proof.* Subtree $T$ must have randomly selected its follower role in $G_i$, as no subtree can be a follower for longer than $24 \cdot (\log N + 1) + 24$ rounds. Given that $T$ is a follower, with probability $1/2$, $T$ must have been a short follower, and therefore either $T$ finds a leader or selects a new role after $4 \cdot (\log N + 1) + 4$ rounds (Lemma 4.10).

**Lemma 4.14.** *Consider configuration $G_i \in \mathcal{F}_\Delta(G_0)$. With probability at least $1/4$, every node in subtree $T$ will have been a potential leader for at least one round over the next $21 \cdot (\log N + 1) + 20$ rounds.*

*Proof.* Consider the possible roles and states of any subtree $T$. If $T$ is currently participating in a merge or is an *OpenLeader*, then the lemma holds.

Suppose $T$ is a follower in configuration $G_i$. By Lemma 4.13, with probability at least $1/2$, $T$ will either find a leader or randomly select another role after $4 \cdot (\log N + 1) + 4$ rounds. If $T$ finds a leader, after an additional $16 \cdot (\log N + 1) + 16$ rounds (Lemma 4.12), $T$ is assigned a merge partner, and after at most $\log N + 1$ additional rounds, all nodes in $T$ are potential leaders. If $T$ randomly selects another role, with probability $1/2$ $T$ selects the leader role, and all nodes are potential leaders after at most $\log N + 1$ additional rounds.

Suppose a node $b \in T$ is a closed leader ($role_b = ClosedLeader$). After at most $9 \cdot (\log N + 1) + 10$ rounds (Lemma 4.9), either the root of $T$ is assigned a merge partner and $b$ becomes a potential leader after an additional $\log N + 1$ rounds, *or* the root of $T$ is not assigned a merge partner and selects a new role at random. With probability $1/2$, then, $b$ becomes a potential leader after an additional $\log N + 1$ rounds.

**Lemma 4.15.** *Every subtree $T$ in configuration $G_i \in \mathcal{F}_\Delta(G_0)$ has probability at least $1/16$ of being assigned a merge partner over $64 \cdot (\log N + 1) + 64$ rounds.*

*Proof.* After at most $24 \cdot (\log N + 1) + 24$ rounds, if $T$ has not been assigned a merge partner, $T$ will re-select its role. With probability $1/4$, $T$ will be a long follower and be searching for a leader for $24(\log N + 1) + 24$ rounds. By Lemma 4.14, a neighboring subtree $T'$ has probability at least $1/4$ of being a potential leader during this time. Therefore, $T$ has probability at least $1/16$ of selecting a leader within $48 \cdot (\log N + 1) + 48$ rounds, which will result in $T$ being assigned a merge partner after at most an additional $16 \cdot (\log N + 1) + 16$ rounds (Lemma 4.12).

**Lemma 4.16.** *Let $T$ and $T'$ be proper subtrees, and let the merge partner of $T$ ($T'$) be $T'$ ($T$). Assume the root $r_T$ of $T$ and the root $r_{T'}$ of $T'$ are connected. In $5 \cdot (\log N + 1) + 4$ rounds, $T$ and $T'$ have merged together into a single proper unmatched clean subtree $T''$, containing exactly $N$ nodes.*

*Proof.* The first step of the merge procedure is to execute the *PFC*(*Prep*) wave, which requires $2 \cdot (\log N + 1) + 2$ rounds (Lemma 4.2). Consider an invocation of the procedure *ResolveSubtree*$(a, b)$. Let $a$ be from subtree $T$, $b$ be from subtree $T'$, and without loss of generality let $a \prec b$. *ReplaceNode*$(a, b)$ requires only 1 round, and results in the children of $a$ being connected to node $b$. In the next round, $b$ will connect its children with the children from $a$, which requires 1 round. *ResolveSubtree* is then executed concurrently for nodes from level $i + 1$. Therefore, the running time starting from level $i$ is $T(i) = 2 + T(i + 1)$. Since there are $\log N + 1$ levels, we have $T(0) = \sum_{i=0}^{\log N} 2 = 2 \cdot (\log N + 1)$. After the resolution process reaches the leaves, the final feedback travels up the tree, requiring an additional $\log N + 1$ rounds, plus 2 rounds for cleaning.

**Lemma 4.17.** *With probability at least* $(1 - N/2^k)$ *(where* $k = |L|$ *and* $k \geq \log N$*), the algorithms from Section 4.4 do not disconnect the network.*

*Proof.* First, notice that deletions (of edges and nodes) that occur due to proper subtrees $T$ and $T'$ merging do not disconnect the network – the only edges deleted are those between nodes in $T \cup T'$, and these nodes will form a proper subtree $T''$ after the merge. The only way in which the network can be disconnected is if the adversary creates an initial configuration such that a node $b$ believes it is either merging, or preparing for a merge, and thus deletes an edge to a node $c$. Notice that for any edge $(b, c)$ to be deleted, both $b$ and $c$ must have the same value for their random sequence, and this value must match the shared random string $L$. While the adversary can enforce the first condition, they are unable to guarantee the second. Instead, for any particular pair of nodes $a$ and $b$, the adversary has probability $1/2^k$ of setting the random sequences of $a$ and $b$ to match $L$. An adversary can have up to $N/2$ different "guesses" in any initial configuration. Therefore, the probability that the network is disconnected is at most $N/2^{k+1}$ (for $k \geq \log N$).

**Theorem 4.3.** *The algorithms given in Section 4.4 form a self-stabilizing algorithm for the* $\texttt{Avatar}_{\texttt{Cbt}}$ *network, with expected convergence time* $\mathcal{O}(\log^2 N)$*.*

*Proof.* We prove this theorem by noting that each time a merge occurs, the number of subtrees in the network decreases by 1. Every subtree has a constant probability of merging over a logarithmic-size time span. Therefore, if there are $n \leq N$ subtrees in the network, the expected number of subtrees after $\mathcal{O}(\log N)$ rounds is at most

$n/c$. After $\mathcal{O}(\log N)$ such segments, the expected number of remaining subtrees is 1. When there is only a single proper subtree, it is the correct $\texttt{Avatar}_{\texttt{Cbt}}$ network.

### 4.5.2 Degree Expansion

In this section, we analyze the degree expansion of the algorithm. To do so, we consider first how a node's degree may grow from program actions before the node is part of a proper subtree. Next, we consider how a node's degree may grow after it becomes a member of a proper subtree. First, we prevent the following corollary, which is a result of Lemma 4.1.

**Corollary 1.** *Consider a node $u$ hosting a set of nodes $Virtual_u$ such that all $b \in Virtual_u$ belong to the same proper subtree $T$. Node $u$ has at most $2 \cdot \log N$ virtual nodes with neighbors in subtree $T$ that are not hosted by $u$.*

We define the set of actions a node may execute that can increase the degree of a real node $u$.

**Definition 4.10.** *Let a* degree-increasing action *of a virtual node $b$ be any action that adds a node $c$ to the neighborhood of a node $b' \in N_b$ such that $b'$ is not hosted by $host_b$. Specifically, the degree-increasing actions are:*

1. *(Selection for Leaders): edges added from the connecting and forwarding of followers during the $PFC(ConnectFollowers, \bot)$ wave of Algorithm 4.3*
2. *(Selection for Followers): forwarding an edge to a leader after the root has selected a leader in Algorithm 4.4*
3. *(Merge): resolution and virtual node transfer actions during Algorithm 4.1*

Notice that transferring all non-subtree edges from a loser node is not a degree-increasing action, as the edges are "virtual transfers" between two virtual nodes hosted by the same real node.

**Lemma 4.18.** *Let $u$ be a real node in configuration $G_i$. The maximum number of real nodes $u$ will add to any neighbor $v$'s neighborhood in a single round is $2 \cdot \log N$.*

*Proof.* We consider the degree-increasing actions. Notice that a real node will detect a reset fault if it hosts two virtual nodes $b$ and $b'$ such that $b$ and $b'$ are executing different algorithms – for example, if $b$ is merging while $b'$ is executing a selection for leaders, host $u$ will reset and not forward any neighbors.

Consider the selection algorithms for both leaders and followers as executed on a virtual node $b$. Node $b$ can only increase the degree of its parent or of a follower. Consider the degree increase $b$ causes to its parent. Node $b$ may give its parent a single edge to a follower or a leader. Since $u$ hosts at most $2 \cdot \log N$ virtual nodes with subtree neighbors not hosted by $u$, and each of these neighbors can increase the degree of a node by at most 1, node $u$ can only increase the degree of a subtree neighbor when executing selection for leaders and followers by at most $2 \log N$.

Next, consider how virtual node $b$ may increase the degree of a follower with Algorithm 4.3. Every follower $b'$ of $b$ may have one additional edge added by $b$. Notice, however, that every follower $b'$ must have a unique host – if not, this host would detect a reset fault. Therefore, $u$ can increase the degree of a real node $v$ by at most 1 when connecting followers in Algorithm 4.3.

Consider the merge actions of virtual nodes hosted by $u$. Again, node $u$ must have all virtual nodes executing the merge algorithm, else $u$ resets. In a given round, $u$ may update its successor and give up all hosted virtual nodes in a particular range to its new successor $v$. The virtual nodes in this range can have at most $2 \cdot \log N$ real neighbors.

**Lemma 4.19.** *Let $u$ be a real node in some configuration $G_i$. The degree expansion of $u$ before all virtual nodes hosted by $u$ are members of a proper clean unmatched subtree is $\mathcal{O}(\log^2 N)$.*

*Proof.* By Lemma 4.18, the largest number of nodes any node $v$ will give to node $u$ in a single round is $2 \cdot \log N$. Furthermore, in order for $u$ to receive $2 \cdot \log N$ nodes from a neighbor $v$, $u$ must host a virtual node whose merge partner is equal to the subtree of the virtual node of $v$. If $u$ detects virtual nodes without matching tree identifiers, it executes a reset. If $u$ detects nodes from the same subtree but different levels being connected to a neighbor attempting to merge, $u$ executes a reset. Therefore, after the first round, at most $2 \cdot \log N$ nodes can be added to $u$'s neighborhood in a single round. Since, by Lemma 4.6, all nodes hosted by $u$ are members of a proper clean unmatched subtree in $\mathcal{O}(\log N)$ rounds, the degree expansion of $u$ before all nodes hosted by $u$ are members of the same proper clean unmatched subtree is $\mathcal{O}(\log^2 N)$.

In the initial configuration, it is possible for all neighbors of $u$ to give $u$ $2 \cdot \log N$ neighbors. In this case, the degree expansion is limited to $\mathcal{O}(\log N)$, since $u$ will reset immediately after receiving these neighbors.

**Lemma 4.20.** *Let $u$ be a real node such that all virtual nodes hosted by $u$ are members* *of a proper clean unmatched subtree in configuration $G_i$. Let u's degree in $G_i$ be $\Delta_u$. Node u's degree will be at most $\Delta_u + 2 \cdot \log N \cdot (\log N + 1) + 2 \cdot \log N \cdot T(lead))$ until the algorithm terminates, where $T(lead)$ is the number of times the virtual nodes hosted by u participate in the leader selection procedure of Algorithm 4.3.*

*Proof.* We consider the three degree-increasing actions that a proper clean unmatched subtree $T$ from configuration $G_i$ may execute. Consider first the follower selection procedure from Algorithm 4.4. The degree can increase only from node $b$ adding the neighbor *leader* from subtree $T'$ to the neighborhood of $parent_b$. Furthermore, this degree increase of one is temporary – a node deletes an edge to *leader* after forwarding it, and once the root has the edge, it eventually becomes part of a merge, and either the root of $T$ or the root of $T'$ is deleted.

Next, consider the selection procedure for leaders in Algorithm 4.3. During the $PFC(ConnectFollowers, \perp)$ wave, a virtual node $b$ may receive at most a single neighbor from each child, and after an additional round will retain at most 1 of these edges. Since a real node $u$ can host at most $2 \cdot \log N$ nodes with children from another host, the degree increase each time a node $u$ participates in the selection procedure for leaders is at most $2 \cdot \log N$.

Finally, consider the degree increase from the merge algorithm. Assume $b$ and $b'$ are nodes in level $i$ in $T$ and $T'$ (respectively), and suppose $b$ and $b'$ resolve. Without loss of generality, let $b' \prec b$. The degree of $b$ can only increase by 2 (the children of $b'$). The degree of $host_b$ may increase if $host_{b'}$ copies some of its virtual nodes to $host_b$. By Corollary 1, any node $v$ can have at most $2 \cdot \log N$ real neighbors inside subtree $T'$. Furthermore, there can be exactly one node $b' \in T'$ that updates its successor to $host_b$ at level $i$. As there are $\log N + 1$ levels, the maximum degree increase during a merge is $2 \cdot \log N \cdot (\log N + 1)$. Notice that, unlike the degree increase from the selection procedure for leaders, the degree increase from merges is not additive – the largest node $u$'s degree can be as the result of intra-subtree edges is $2 \cdot \log N$, regardless of how many merges $u$ participates in. Therefore, while a node's degree may temporarily grow during a merge to $2 \cdot \log N \cdot (\log N + 1)$, after the merge is complete, $u$'s intra-subtree degree is at most $2 \cdot \log N$.

**Theorem 4.4.** *The degree expansion of the* $\texttt{Avatar}_{\texttt{Cbt}}$ *algorithm is* $\mathcal{O}(\log^2 N)$ *in expectation.*

*Proof.* By Lemma 4.19, a node's degree can increase by at most a $\mathcal{O}(\log^2 N)$ factor from $u$'s initial degree before all nodes hosted by $u$ are in a proper clean unmatched subtree. By Lemma 4.20, the degree of $u$ can increase during execution by at most $2{\cdot}\log N{\cdot}(\log N{+}1){+}2{\cdot}\log N{\cdot}T(\mathit{lead})$, regardless of the initial degree of $u$. Since a node may begin execution with degree 1, and have degree 1 in the correct configuration, this additive degree increase may be equal to the node's degree expansion. Notice each leader selection procedure requires $\mathcal{O}(\log N)$ rounds, and the network converges in $\mathcal{O}(\log^2 N)$ rounds in expectation (Theorem 4.3). Therefore, $T(\mathit{lead})$ is $\mathcal{O}(\log N)$ in expectation, leading to a degree expansion of $\mathcal{O}(\log^2 N)$ in expectation.

# CHAPTER 5
# SCAFFOLD NETWORKS

In this chapter, we investigate how $\texttt{Avatar}_{\texttt{Cbt}}$ can be used as an intermediate step towards building the `Chord` network. We present an algorithm which starts with the $\texttt{Avatar}_{\texttt{Cbt}}$ network and adds edges in a systematic fashion to create the $\texttt{Avatar}_{\texttt{Chord}}$ network. We show that this process increases the running time of the self-stabilizing algorithm from Chapter 4 by only $\mathcal{O}(\log^2 N)$ rounds, and increases the degree expansion by only $\Delta_0$, where $\Delta_0$ is the maximum degree of a node in the initial configuration.

## 5.1 Defining Scaffold Networks

One of the major barriers to both the design and analysis of self-stabilizing overlay networks is the level of complexity. When constructing a large building, a common approach is to erect a scaffold temporarily, and use this scaffold to build the more complex permanent structure. Presumably, this scaffolding is easier to build than the final structure, but at the same time it makes work significantly easier. Applying this idea to overlay networks, we use a simple self-stabilizing overlay network $ON_s$ to build a more complex topology $ON_f$. We call this approach *network scaffolding.* The network $ON_s$ is called the *scaffold network*, and the desired topology $ON_f$ is called the *final network*.

More formally, the network scaffolding approach is to design an algorithm $\mathcal{A}$ which executes on a family of overlay networks $ON_s$ such that $\mathcal{A}$ eventually creates the overlay network family $ON_f$. Algorithm $\mathcal{A}$ uses the edges of a simpler network to create the edges for the more complex network. The challenges here are two-fold. First, attempts to build $ON_f$ should not slow down the building of $ON_s$ (if necessary), nor should they increase the degree too much. Secondly, nodes should be able to determine when to execute $\mathcal{A}$, and when to execute the algorithm for the scaffold network.

### 5.1.1 Example: `Linear` to `Re-Chord`

While the explicit definition of this approach is new, the idea of network scaffolding is not. Consider the self-stabilizing `Re-Chord` network [25]. The basic idea of the authors' algorithm is (i) build a `Linear` network, and then (ii) create the edges necessary for the `Chord` network using the `Linear` network. `Re-Chord` can be thought of as a `Linear` scaffold network with the `Chord` final network.

The drawbacks of using `Linear` as a scaffold network are that (i) `Linear` requires $\mathcal{O}(n)$ rounds to converge, and (ii) once converged `Linear` has diameter $n - 1$, meaning routing on the scaffold network is slow. These limitations result in a $\mathcal{O}(n \cdot \log n)$ running time for `Re-Chord` (`Re-Chord` uses $n$ real nodes and $\log n$ virtual nodes). Clearly, then, using `Linear` as a scaffold network may simplify analysis, but it comes at the cost of scalable convergence time.

## 5.2 Avatar as a Scaffold for Chord

In this section, we discuss how to use $\texttt{Avatar}_{\texttt{Cbt}}$ as a scaffold for creating the $\texttt{Avatar}_{\texttt{Chord}}$ network. The algorithm uses the fact that `Chord` edges can be created inductively. That is, assuming all fingers from 0 to $k$ are present, the $k+1$ finger can be created in a single round. Specifically, if node $b$ is the $(i-1)$ finger of $c_0$, and $c_1$ is the $(i-1)$ finger of $b$, the $i$th finger of $c_0$ is $c_1$. The algorithm begins by correctly building finger 0, then recursively adds the first finger, then the second, and so on.

### 5.2.1 Defining $\texttt{Chord}(N)$

We begin by defining the full graph family $\texttt{Chord}$.

**Definition 5.1.** *For any $N \in \mathbb{N}$, let $\texttt{Chord}(N)$ be a graph with nodes $[N]$ and edge set defined as follows. For every node $i$, $0 \leq i < N$, add to the edge set $(i, j)$, where $j = (i + 2^k) \mod N$, $0 \leq j < \log N - 1$. When $j = (i + 2^k) \mod N$, we say that $j$ is the $k$th finger of $i$.*

Our goal is to create a graph $G$ with node set $V \subseteq [N]$ such that (i) the edges of the $\texttt{Avatar}_{\texttt{Cbt}}$ network are a subset of the edges of $G$, and (ii) the edges of $\texttt{Avatar}_{\texttt{Chord}}$ are also a subset of the edges of $G$. Notice that, when $N$ is a power of 2, the edges of $\texttt{Chord}(N)$ are a superset of the edges of $\texttt{Cbt}(N)$. That is, the edges of $\texttt{Cbt}(N)$ are present in $\texttt{Chord}(N)$. Therefore, for $N \in \{2^i | i = 0, 1, \ldots\}$ and $V \subseteq [N]$, the edges of $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ are a subset of the edges of $\texttt{Avatar}_{\texttt{Chord}}(N, V)$. For the remainder of this chapter, we assume $N$ is a power of 2.

#### 5.2.1.1 Original Chord vs. Embedded Chord

The $\texttt{Avatar}_{\texttt{Chord}}(N, V)$ network does not match the original $\texttt{Chord}$ network composed of node set $V$. In our embedding, a virtual node $b$ is hosted by a real node $u$ such that $b$ is between $u$ and its successor. That is, a node $u$ hosts all virtual nodes between $u$ and $succ_u$. This implies that the $k$th finger $f$ of a node $b$ is an edge in the real network between $host_b$ and $v$, where $v = \max_{w \in V}(w \leq f)$. In the original `Chord` network, however, the $k$th finger $f$ of node $u \in V$ is to $v = \min_{w \in V}(w \geq f)$.

If we require $\texttt{Avatar}_{\texttt{Chord}}$ to have the edges as the original `Chord` network, we can build $\texttt{Avatar}_{\texttt{Chord}}$ with the algorithm presented here, and then use the successor pointers of $\texttt{Avatar}_{\texttt{Chord}}$ to forward the $\texttt{Chord}(N)$ edges. This action can be coordinated through a *PFC* wave, for instance. Assuming node identifiers are uniformly distributed, there is no significant increase in a node's degree or "responsible range" (the range of object identifiers mapped to a node $u$) between our mapping and the original `Chord` mapping.

### 5.2.2 Avatar: A Chord Scaffold

Taking advantage of the network embedding approach, we design the algorithm to be executed upon the virtual nodes of $\texttt{Cbt}(N)$. Assume at the moment that the network is a correct $\texttt{Avatar}_{\texttt{Cbt}}$ network and all nodes are executing the algorithm to build $\texttt{Avatar}_{\texttt{Chord}}$. The algorithm begins with the root of $\texttt{Cbt}(N)$ initiating a *PFC* wave which connects each virtual node with its 0th finger. Notice that, with the exception of one node, the edges in the real network realizing every virtual node's 0th finger are already present. For any virtual node $b \neq N-1$, the 0th finger of $b$ is either (i) a virtual node with the same host as $b$, or (ii) a virtual node which is hosted by the successor of $host_b$. Virtual nodes 0 and $N-1$ forward an edge to themselves up the tree on the feedback wave, and the root of the tree connects them, thus completing every virtual node's 0th finger. The root then executes $\log N - 1$ additional *PFC* waves, with wave $k$ correctly adding the $k$th finger for all nodes. After $\mathcal{O}(\log^2 N)$ rounds, $\texttt{Avatar}_{\texttt{Chord}}$ is built.

The algorithm requires nodes to know whether to execute the algorithm to build $\texttt{Avatar}_{\texttt{Cbt}}$ or the algorithm to build $\texttt{Avatar}_{\texttt{Chord}}$ from $\texttt{Avatar}_{\texttt{Cbt}}$. We do so by having every real node $u$ maintain two bits – $chord_u$ and $cbt_u$. When $cbt_u = 0$, a node is executing the algorithm to build the $\texttt{Avatar}_{\texttt{Cbt}}$ network. If $cbt_u = 1$ and $chord_u = 0$, then Algorithm 5.1 is executed (shown below). We discuss the setting of these bits next.

## 5.2.3 Algorithm Selection

Algorithm 5.1 assumes it begins execution from the correct $\texttt{Avatar}_{\texttt{Cbt}}$ network. However, in a self-stabilizing setting, this may not be the case. Therefore, we need a way to correctly determine which algorithm a node should execute. As mentioned previously, this is done by each node keeping two bits, $cbt_u$ and $chord_u$, which a node shares with its neighbors. Informally, if a node detects its state is consistent with executing Algorithm 5.1, it continues executing Algorithm 5.1. If it is not consistent, it reverts to the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm from Chapter 4.

Determining which algorithm to execute requires a node to determine if the configuration will stabilize to a legal $\texttt{Avatar}_{\texttt{Chord}}$ configuration simply by executing Algorithm 5.1, or if the network needs to restore the legal $\texttt{Avatar}_{\texttt{Cbt}}$ network first. We define a subset of states under which Algorithm 5.1 will converge, and then define a predicate which nodes can use to determine if the network is in one of these states.

**Algorithm 5.1** Algorithm for `Chord`$(N)$ from `Cbt`$(N)$

```
     // Execute when $cbt_u = 1$ and $chord_u = 0$
     // If $cbt_u = 0$, then execute the Avatar_{Cbt} algorithm from Chapter 4
1.   Tree $T$ executes a $PFChord(MakeFinger(0), \perp)$ wave:
2.      Propagate Action for $a$: $LastWave_a = 0$
3.      Feedback Action for $a$:
           // Let $b$ be the 0th finger of $a$.
4.         if $LastWave_a = LastWave_b = 0$ then
5.            Create the virtual edge $(a, b)$
6.            Forward an edge to node 0 or $N - 1$ (if present) to parent
7.         else $cbt_u = 0$ (where $u$ is $host_a$) fi
8.   for $k = 1, 2, \ldots, \log N - 1$ do
9.      Tree $T$ executes a $PFChord(MakeFinger(k), \perp)$ wave:
10.        Propagate Action for $a$: $LastWave_a = k$
11.        Feedback Action for $a$:
              // Let $b_0, b_1$ be the $k - 1$ fingers of $a$.
12.           if $LastWave_a = LastWave_{b_0} = LastWave_{b_1} = k$ then
13.              Create edge $(b_0, b_1)$, the $k$th finger of $b_0$.
14.           else $cbt_u = 0$ (where $u$ is $host_a$) fi
15.  od
```

**Definition 5.2.** *A graph $G$ with node set $V$ is said to be in a* scaffolded `Chord` configuration *if $G \in \mathcal{F}_{\mathcal{A}}(\texttt{Avatar}_{\texttt{Cbt}}(N, V))$, where $\mathcal{A}$ is Algorithm 5.1. In other words, $G$ is reachable by executing Algorithm 5.1 on a correct $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ network.*

Informally, to determine if the configuration is a scaffolded `Chord` configuration, every virtual node simply checks to see if its neighborhood is a superset of `Cbt`$(N)$ but a subset of `Chord`$(N)$, and the first $k$ fingers from `Chord`$(N)$ are present. We define the predicate a node can use for this operation below.

**Definition 5.3.** *Let $scaffolded_b$ be a predicate defined over the local state of a virtual node $b$, as well as the state of nodes $b' \in N(b)$. The value of $scaffolded_b$ is the conjunction of the following conditions.*

1. *Node $b$ has all neighbors from $\texttt{Cbt}(N)$, each with the proper host and tree identifier.*
2. *Node $b$ has last executed the $k$th feedback wave of a $PFC(MakeFinger(k), \perp)$ wave.*
3. *All neighbors of $b$ have either all $k$ fingers present, or $k + 1$ fingers (if a child has just processed a feedback wave), or $k - 1$ (if parent has not yet processed the current feedback wave).*
4. *Node $b$'s parent has last executed the $k$th feedback wave, and has the first $k$ $\texttt{Chord}(N)$ fingers, or $k - 1$ fingers if $b$ has just completed the feedback transition and $b$'s parent has not.*

This predicate is used to set the $chord_u$ and $cbt_u$ bits as follows. If a fault is detected and $\mathit{scaffolded}_u = \mathit{false}$, then $u = \mathit{host}_b$ sets $\mathit{cbt}_u = 0$. Furthermore, if any neighbor has a different value for either bit $\mathit{cbt}_u$ or $\mathit{chord}_u$, then both bits are set to 0. We will show in a moment that this procedure is sufficient to ensure the correct algorithm is executed within a short amount of time (i.e. if the configuration is not a scaffolded `Chord` configuration, then all nodes begin executing the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm from Chapter 4 quickly). Notice that, once the correct configuration is built, nodes can execute a final *PFC* wave to set $\mathit{chord}_u = 1$. If any node detects *any* fault during this process, it simply sets both bits to 0. Since $\texttt{Avatar}_{\texttt{Chord}}$ is locally checkable, at least one node will not complete this $\mathit{chord}_u = 1$ *PFC* wave, and the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm will begin.

### 5.2.4 Analysis

#### 5.2.4.1 Convergence Time

The convergence time analysis of the algorithm begins by showing that, after $\mathcal{O}(\log N)$ rounds, if the configuration is neither the correct $\texttt{Avatar}_{\texttt{Chord}}$ network nor a scaffolded `Chord` configuration, then all nodes are executing the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm from Chapter 4. During these $\mathcal{O}(\log N)$ rounds, the degree of a node can at most double from its initial degree. Finally, after the correct $\texttt{Avatar}_{\texttt{Cbt}}$ network is built, the $\texttt{Avatar}_{\texttt{Chord}}$ network is built in $\mathcal{O}(\log N)$ rounds. We prove each of these claims below.

**Lemma 5.1.** *Let $G_i$ be a configuration with node set $V \subseteq [N]$. Suppose $G_i \neq$ $\texttt{Avatar}_{\texttt{Chord}}(N, V)$, and $G_i$ is not a scaffolded `Chord` configuration. Any node $u \in G_i$*

*is distance at most* $2 \cdot (\log N + 1)$ *from a node* $v$ *with* $cbt_v = 0$*.*

*Proof.* Imagine the subgraph induced by all nodes at distance at most $2 \cdot (\log N + 1)$ from any node $u$. If this ball does not contain all the nodes in the network, then there exists at least one node $v$ in this ball such that $v$ has a neighbor that it would not have in the correct $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ configuration, and $v$ detects this extra neighbor. Therefore, $v$ would set $cbt_v = 0$. Furthermore, the real nodes in this ball must be hosting exactly $N$ virtual nodes, with each virtual node hosted by the correct real node with the correct tree identifier, else at least one node detects an incorrect embedding (Chapter 4, Theorem 4.1). Finally, the *PFC* state of these $N$ virtual nodes must be consistent, else a faulty configuration that is not a scaffolded `Chord` configuration is detected, and $cbt_u = 0$.

Assume the $N$ virtual nodes in this ball realize a $\texttt{Cbt}(N)$ network with some additional edges, and the *PFC* state is consistent. If the network is a correct $\texttt{Avatar}_{\texttt{Chord}}$ configuration, no node detects a fault, as $\texttt{Avatar}_{\mathcal{F}}$ is locally checkable for any full graph family. A virtual node can easily determine if it has the first $k$ $\texttt{Chord}(N)$ fingers by examining its set of neighbors. Virtual nodes can also verify that the last *PFC* feedback wave processed was for $k$ (potentially $k - 1$ if finger $k$ is a member of $\texttt{Cbt}(N)$). If a $b$ node detects it has a different number of correct `Chord` fingers as a neighbor, it must either be currently processing a *PFC* wave and about to receive the feedback wave (if the children have an extra finger) or pass the feedback wave to its parent (if its parent has one less correct `Chord` finger). If neither of these conditions are true, then $b$ sets $scaffolded_b = 0$. Therefore, if all nodes do not have their first $k$

$\texttt{Chord}$ fingers correct, or all nodes are not in a *PFC* wave to add their $k$th finger to the already-correct set of $k-1$ fingers, at least one node must set $\mathit{scaffolded}_b = 0$, which results in $\mathit{cbt}_u = 0$ (where $u = \mathit{host}_b$).

**Lemma 5.2.** *Suppose configuration $G_i$ is not the $\texttt{Avatar}_{\texttt{Chord}}$ configuration, and $G_i$ is not a scaffolded $\texttt{Chord}$ configuration. In at most $2 \cdot (\log N + 1)$ rounds, all nodes are executing the self-stabilizing $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm from Chapter 4.*

*Proof.* By Lemma 5.1, if $G$ is not $\texttt{Avatar}_{\texttt{Chord}}(N, V)$ nor a scaffolded $\texttt{Chord}$ network, then every node $u$ has a node $v$ within distance at most $2 \cdot (\log N + 1)$ with $\mathit{cbt}_v = 0$. In every round, neighbors of nodes with $\mathit{cbt}_v = 0$ set their own $\mathit{cbt}_u = 0$. In at most $2 \cdot (\log N + 1)$ rounds, every node $u$ has $\mathit{cbt}_u = 0$, thus every node is executing the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm.

**Lemma 5.3.** *Let $G_0$ be the correct $\texttt{Avatar}_{\texttt{Cbt}}(N, V)$ configuration, and let each node $u \in V$ have $\mathit{chord}_u = 0$ and $\mathit{cbt}_u = 1$. In $\mathcal{O}(\log^2 N)$ rounds, the configuration converges to $\texttt{Avatar}_{\texttt{Chord}}(N, V)$.*

*Proof.* We prove this by induction on the number of correct fingers. In $G_0$, the root of the virtual $\texttt{Cbt}$ will begin the first *PFC* wave to add the 0th fingers. For every virtual node $b \neq N-1$, the 0th finger is either (i) a virtual node hosted by $u = \mathit{host}_b$, or (ii) a virtual node hosted by $v$ such that $v = \mathit{succ}_u$. In either case, the real edge realizing this virtual edge already exists, and creating this 0th finger is simply a matter of a host updating a virtual node's local state. For node $N-1$, the first *PFC* wave forwards to the root node an edge to virtual nodes 0 and $N-1$, and the root will connect these two nodes. This requires $2 \cdot (\log N + 1)$ rounds. At this point, all nodes have the correct 0th finger.

Assume finger $i$ has been created by the $PFC(Chord(i), \perp)$ wave. The root will execute the $PFC(Chord(i+1), \perp)$ wave. Let virtual node $b$ receive the feedback wave. By the inductive hypothesis, node $b$ has links to virtual nodes $c_0$ and $c_1$, where $b$ is finger $i$ of $c_0$, and $c_1$ is finger $i$ of $b$. Node $b$ connects $c_0$ and $c_1$ in one round, thereby creating finger $(i+1)$ for $c_0$. After $2 \cdot (\log N + 1)$ rounds, all nodes have added their $i + 1$ finger. As there are $\log N - 1$ fingers, the total convergence time to reach the $\texttt{Avatar}_{\texttt{Chord}}$ network from $G_0$ is $\mathcal{O}(\log^2 N)$.

Using Theorem 4.3 and the above lemmas gives us the following result.

**Theorem 5.1.** *Algorithm 5.1 combined with the self-stabilizing algorithm from Chapter 4 is a self-stabilizing algorithm for the $\texttt{Avatar}_{\texttt{Chord}}$ network with convergence time $\mathcal{O}(\log^2 N)$ in expectation.*

#### 5.2.4.2 Degree Expansion

Besides converging to a correct configuration quickly, the algorithm maintains low degree expansion. We prove this next.

**Lemma 5.4.** *Let $\Delta_i$ be the maximum degree of a real node $v$ in configuration $G_i$. Suppose $G_i$ is neither an $\texttt{Avatar}_{\texttt{Chord}}$ configuration nor a scaffolded $\texttt{Chord}$ configuration. The degree of any node $u \in V$ is multiplied by at most $\Delta_i$ before $u$ begins executing the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm.*

*Proof.* Consider a virtual node $b$ hosted by $u$ in configuration $G_i$. Notice that each virtual neighbor $b'$ of $b$ can add at most one node to the neighborhood of $b$ per $2 \cdot (\log N + 1)$ rounds when executing Algorithm 5.1, as $b'$ attempts to add the $k$th finger to $b$ (at which point $b'$ must wait for another $PFC$ wave). Furthermore, by Lemma 5.2, after at most $2 \cdot (\log N + 1)$ rounds, all nodes in $G_i$ have $cbt_u = 0$. Therefore, in the worst case a node $u$ may become immediate neighbors with all real nodes at distance 2 from $u$ in $G_i$, resulting in a degree expansion of at most $\Delta_i$.

Combining the lemma above with Theorem 4.4 from Chapter 4 gives us the following theorem.

**Theorem 5.2.** *Algorithm 5.1 builds the* $\texttt{Avatar}_{\texttt{Chord}}$ *network with degree expansion of* $\mathcal{O}(\Delta_0 \cdot \log^2 N)$ *in expectation, where* $\Delta_0$ *is the maximum degree of a real node in the initial configuration.*

*Proof.* By Lemma 5.4, the degree expansion from Algorithm 5.1 is at most $\Delta_0$ before the $\texttt{Avatar}_{\texttt{Cbt}}$ algorithm from Chapter 4 is executed. By Theorem 4.4, this algorithm has a degree expansion of $\mathcal{O}(\log^2 N)$ in expectation. When executing from a scaffolded `Chord` configuraiton, the only edge added by Algorithm 5.1 that is not an edge in the final configuration is the forwarding of an edge to node 0 and $N-1$. Therefore, once the correct $\texttt{Avatar}_{\texttt{Cbt}}$ network is built, Algorithm 5.1 builds $\texttt{Avatar}_{\texttt{Chord}}$ with degree expansion of at most 2. Therefore, from any configuration Algorithm 5.1 may increase degrees by a factor of at most $\max(2, \Delta_0) = \Delta_0$ (a connected graph must have $\Delta_i \geq 2$ for $n > 2$), the algorithm from Chapter 4 may increase degrees by $c \cdot \log^2 N$, giving us $\mathcal{O}(\Delta_0 \cdot \log^2 N)$ degree expansion.

# CHAPTER 6
# CONCLUSIONS

Self-stabilization is an elegant approach for maintaining the topology of an overlay network in the presence of faults. In this thesis, we presented several self-stabilizing algorithms for overlay network creation. We also presented a non-trivial lower bound on the convergence time of self-stabilizing overlay networks, and demonstrated a systematic approach for overlay network design that is easily extensible. This thesis is a good starting point for future overlay network research.

## 6.1 Summary of Contributions

### 6.1.1 The Transitive Closure Framework

In Chapter 3, we presented the *Transitive Closure Framework*, which is a transformer that makes any locally checkable overlay network self-stabilizing. We proved a bound on the convergence time of self-stabilizing overlay networks and used this bound to show that the Transitive Closure Framework provides near-optimal convergence time for a class of overlay networks which includes `Linear` and `Skip+`. We then extended the framework to repair a subset of faults quickly with the Local Repair Framework. We demonstrated the Local Repair Framework by implementing a node join procedure for Skip+ graphs that executed in logarithmic time.

### 6.1.2 The Avatar Network

In Chapter 4, we presented a self-stabilizing overlay network algorithm which converges in a polylogarithmic number of rounds and limits degree expansion to at most polylogarithmic. To do this, we first defined a family of overlay networks based upon the embedding of *any* full graph family. We extended the notion of local checkability to include proof labels, which gives a better measure of the state nodes must exchange in the "correct" configuration to be locally checkable. We then showed the graph definition was locally checkable using only a small number of bits per proof label.

Using the full graph family `Cbt`, a graph based upon the binary search tree, we designed a self-stabilizing algorithm which converges in $\mathcal{O}(\log^2 N)$ rounds in expectation, while limiting the expected degree expansion of a node to $\mathcal{O}(\log^2 N)$. This result is the first to achieve efficient time *and* space convergence, and holds even for a powerful adversary that can modify a node's state arbitrarily. The algorithm uses a modular design which is easy to understand, and which suggests a simple extension to build other topologies efficiently.

### 6.1.3 Scaffolding Networks

Finally, in Chapter 5, we introduced a technique for using the $\texttt{Avatar}_{\texttt{Cbt}}$ network to build the $\texttt{Avatar}_{\texttt{Chord}}$ network. Specifically, we showed (i) how to build the $\texttt{Avatar}_{\texttt{Chord}}$ network when starting from a legal $\texttt{Avatar}_{\texttt{Cbt}}$ network, and (ii) how nodes in the network can quickly determine whether to execute the scaffolding algorithm to build $\texttt{Avatar}_{\texttt{Chord}}$, or whether to execute the algorithm from Chapter 4 to build the $\texttt{Avatar}_{\texttt{Cbt}}$ network. This combination results in a self-stabilizing algorithm for $\texttt{Avatar}_{\texttt{Chord}}$ network with a running time of $\mathcal{O}(\log^2 N)$ rounds in expectation and an expected degree expansion of $\mathcal{O}(\Delta \cdot \log^2 N)$, where $\Delta$ is the maximimum degree taken over all nodes in the initial configuration. This algorithm is interesting in two respects: first, it is the only efficient self-stabilizing algorithm for creating the Chord network; secondly, it demonstrates the power of the algorithm from Chapter 4 – efficient extensions are simple to design and analyze.

## 6.2 Future Work

Several open problems are highlighted by this thesis. First, the efficient algorithm for the $\texttt{Avatar}_{\texttt{Cbt}}$ network from Chapter 4 requires a shared random sequence $L$ to avoid network disconnection. A natural open problem is to investigate if this shared randomness is required for efficient convergence. An efficient solution without shared randomness and a powerful adversary may not exist, since an edge cannot be deleted without first determining if it is a cut edge. Since local states can be arbitrarily corrupted in the self-stabilizing model, local states cannot be used to determine if an edge is a cut edge (without the use of shared randomness, as we did). Therefore, edges cannot be explicitly deleted, but rather must be *delegated* towards the endpoint of the edge. That is, the edge $(u, v)$ can become $(u', v)$ such that $u'$ is on the path from $u$ to $v$. Eventually, the edge becomes $(v, v)$, at which point it can safely be deleted. However, an adversary may be able to construct examples where many edges must traverse a single node during this process. In these cases, it would seem that either the degrees grow quite large, or the convergence time slows considerably. Further work is needed to either (i) find a solution where shared randomness is not required, or (ii) prove such a result is impossible.

Second, future work could create an algorithm to efficiently create $\texttt{Avatar}_{\mathcal{F}}$ for *any* full graph family $\mathcal{F}$. The approach to building an efficient $\texttt{Avatar}_{\texttt{Cbt}}$ network was generic and modular, and seems extensible to other networks. One need only (i) define a cluster, (ii) discuss efficient communication in this cluster, and (iii) define a merging algorithm. How such an extension would perform with other clusters and other communication mechanisms remains to be seen. While bounding the convergence time seems simple to prove using similar techniques to our implementation, understanding how the degree expansion is changed by the new modules is non-trivial. Ideally, we would be able to define a measure analogous to the detector diameter, only for degree expansion instead of convergence time.

Third, this work ignores the underlying implementation of overlay links and assumes that each logical link is equivalent. One can think of a model where each link has a cost associated with it. Our work has assumed a uniform cost model with unlimited budgets – every node could send the same amount of information over every incident edge, and all edges were equal. In practice, however, many logical links may be realized by the same physical link, which imposes bandwidth constraints on the amount of *total* information a node can share in a particular round. Furthermore, the physical links of intermediate nodes (nodes not part of the network itself, but required for routing) may have their own capacity constraints, which may make logical links heterogeneous – some logical links may have a larger capacity than others, and some may be slower than others. Understanding the role of physical links in logical networks is a hard. In fact, some measures of this connectivity are not even efficiently computable with a centralized algorithm [16]. In the future, we would like to determine how a non-uniform cost model of communication can be integrated into the self-stabilizing overlay network model.

Finally, this work has always assumed the existence of a single correct configuration for any set of nodes $V$. One could relax this requirement to allow *semi-structured overlay network* – networks where a correct configuration is one satisfying predicate $P$, and there are multiple such configurations for any node set $V$. Open problems include not only finding interesting predicates, but also proving what kinds of predicates can be locally verified (or verified with the minimum amount of global knowledge). Future work can (i) investigate what predicates can be guaranteed in the self-stabilizing overlay network model, and (ii) design efficient algorithms to satisfy these predicates.

## REFERENCES

[1] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[2] James Aspnes and Gauri Shah. Skip graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 384–393, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[3] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 326–339, London, UK, 1994. Springer-Verlag.

[4] Andrew Berns, Anurag Dasgupta, and Sukumar Ghosh. Brief announcement: optimal self-stabilizing multi-token ring: a randomized solution. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 302–303, New York, NY, USA, 2009. ACM.

[5] Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Brief announcement: a framework for building self-stabilizing overlay networks. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 398–399, New York, NY, USA, 2010. ACM.

[6] Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 62–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. *Theoretical Computer Science*, 2012. Under Review.

[8] Inc. BitTorrent. http://www.bittorrent.com/ - bittorrent - delivering the world's content, 2012.

[9] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks (extended abstract). In *In Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85. IEEE Computer Society Press, 1999.

[10] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In *Workshop on Self-stabilizing Systems*, ICDCS '99, pages 78–85, Washington, DC, USA, 1999. IEEE Computer Society.

[11] Curt Cramer and Thomas Fuhrmann. Self-stabilizing ring networks on connected graphs. Technical report, University of Karlsruhe (TH), 2005.

[12] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *CoRR*, abs/0802.1123, 2008.

[13] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[14] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997(4), December 1997.

[15] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theor. Comput. Sci.*, 410(6-7):514–532, 2009.

[16] Yuval Emek, Pierre Fraigniaud, Amos Korman, Shay Kutten, and David Peleg. Notions of connectivity in overlay networks. In *Proceedings of the 19th international conference on Structural Information and Communication Complexity*, SIROCCO'12, pages 25–35, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] Dominik Gall, Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Tubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In Alejandro Lpez-Ortiz, editor, *LATIN 2010: Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 294–305. Springer Berlin / Heidelberg, 2010.

[18] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[19] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 45–54, New York, NY, USA, 1996. ACM.

[20] Mika Göös and Jukka Suomela. Locally checkable proofs. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 159–168, New York, NY, USA, 2011. ACM.

[21] Ted Herman. Self-stabilization: randomness to reduce space. *Distrib. Comput.*, 6(2):95–98, 1992.

[22] Amos Israeli and Marc Jalfon. Self-stabilizing ring orientation. In *Proceedings of the 4th international workshop on Distributed algorithms*, pages 1–14, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[23] Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 131–140, New York, NY, USA, 2009. ACM.

[24] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distrib. Comput.*, 7(1):17–26, 1993.

[25] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: a self-stabilizing chord overlay network. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 235–244, New York, NY, USA, 2011. ACM.

[26] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *Proceedings of the 4th international conference on Peer-to-Peer Systems*, IPTPS'05, pages 13–23, Berlin, Heidelberg, 2005. Springer-Verlag.

[27] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[28] T. Nakata. On the expected time for Herman's probabilistic self-stabilizing algorithm. *Theoretical Computer Science*, 349(3):475–483, 2005.

[29] Ilan Newman. Private vs. common random bits in communication complexity. *Inf. Process. Lett.*, 39(2):67–71, July 1991.

[30] Melih Onus, Andréa W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *ALENEX*. SIAM, 2007.

[31] David Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, 2000.

[32] Andréa Richa, Christian Scheideler, and Phillip Stevens. Self-stabilizing de bruijn networks. In *Proceedings of the 13th international conference on Stabilization,*

*safety, and security of distributed systems*, SSS'11, pages 416–430, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 99–, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology p2p systems. Technical report, Department of Computer Science, N.C. State University, 2005.

[35] Skype. http://www.skype.com/ - free skype internet calls, 2012.

[36] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[37] Amitabh Trehan. Self-healing using virtual structures. *CoRR*, abs/1202.2466, 2012.