A Distributed Polylogarithmic Time Algorithm for Self-Stabilizing Skip Graphs*

Riko Jacob
Dept. Computer Science
Technische Universität
München
D-85748 Garching bei
München, Germany
jacob@in.tum.de

Andrea Richa
Department of Computer
Science and Engineering
Arizona State University
Tempe, Arizona
AZ 85287-8809, USA
aricha@asu.edu

Christian Scheideler
Department of
Computer Science
University of Paderborn
D-33102 Paderborn
Germany
scheideler@upb.de

Stefan Schmid
Dept. Computer Science
Technische Universität
München
D-85748 Garching bei
München, Germany
schmiste@in.tum.de

Hanjo Täubig
Dept. Computer Science
Technische Universität
München
D-85748 Garching bei
München, Germany
taeubig@in.tum.de

ABSTRACT

Peer-to-peer systems rely on scalable overlay networks that enable efficient routing between its members. Hypercubic topologies facilitate such operations while each node only needs to connect to a small number of other nodes. In contrast to static communication networks, peer-to-peer networks allow nodes to adapt their neighbor set over time in order to react to join and leave events and failures. This paper shows how to maintain such networks in a robust manner. Concretely, we present a distributed and self-stabilizing algorithm that constructs a (variant of the) skip graph in polylogarithmic time from *any* initial state in which the overlay network is still weakly connected. This is an exponential improvement compared to previously known self-stabilizing algorithms for overlay networks. In addition, individual joins and leaves are handled locally and require little work.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

1. INTRODUCTION

Peer-to-peer computing is one of the most intriguing networking paradigms of the last decade. Numerous Internet applications make use of peer-to-peer technology, including file sharing, streaming and gaming tools. A distinguishing feature of these networks is that they typically have an open clientele, allowing machines to join and leave at any time and concurrently. If no countermeasures are taken, the dynamic membership changes can degenerate the network, rendering central operations such as routing inefficient. In an effort to gain deeper insights into (and deal with) these dynamics, researchers have studied different approaches. In theory, the dominating approach has been to make sure that an overlay network never leaves a certain set of legal states so that at any time, information can be efficiently exchanged between its members. This is mostly achieved through redundancy in the overlay network topology, which can significantly increase its maintenance overhead, so the rate of changes such networks can sustain might be rather limited. However, a high change rate can happen due to heavy churn or join-leave attacks. Also, partitions of the underlying physical network or DoS attacks may push the overlay network into some illegal state. In this case, the overlay network may get into a state in which it is highly vulnerable to further changes, so proper recovery mechanisms are needed to get it back into a legal state *as quickly as possible*. Some results are known in this direction, but most of the proposed protocols only manage to recover the network from a restricted class of illegal states (e.g., [2, 5, 34]). Those few results known for truly self-stabilizing networks either just show eventual self-stabilization (e.g., [12]) or do not provide sublinear bounds on the convergence time (e.g., [19, 31]). Our work is the first that demonstrates that sublinear, in fact, polylogarithmic recovery time is possible. More precisely, we present a self-stabilizing algorithm for a proper extension of the skip graph [3] (as the original skip graph is not locally checkable). Skip graphs are very useful for scalable overlay networks. They have logarithmic diameter and degree and constant expansion w.h.p. [4]. Also, like in hypercubic networks, no extra routing tables have to be maintained for fast, low congestion routing. Before we delve into the details of our solution, we discuss related work and present our model.

1.1 Related Work

There is a large body of literature on how to maintain peer-to-peer networks efficiently, e.g., [3, 8, 10, 18, 22, 28, 30, 32, 34]. Recently, the first structured overlay networks have also found their way into the practical world; for instance, the Kademlia [29] over-

*This work was supported by NSF award CCF-0830704 and by the DFG-Project SCHE 1592/1-1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'09, August 10–12, 2009, Calgary, Alberta, Canada.
Copyright 2009 ACM 978-1-60558-396-9/09/08 ...$10.00.

lay is used in the popular *Kad network* which can be accessed with *eMule* clients. An interesting and flexible overlay structure are *skip graphs* [3, 7, 20, 21, 22]. These networks are based on the classical skip list data structure and allow for efficient, low-congestion routing while requiring a small node degree only. Due to the typically very dynamic nature of peer-to-peer systems, there is a need to maintain the overlay topology or—in case of catastrophic events—recover it from arbitrary connected states. While many results are already known on how to keep an overlay network in a legal state, not much is known about self-stabilizing overlay networks.

In the field of self-stabilization, researchers are interested in algorithms that are guaranteed to eventually converge to a desirable system state from *any* initial configuration. The idea of self-stabilization in distributed computing first appeared in a classical paper by E.W. Dijkstra in 1974 [15] in which he looked at the problem of self-stabilization in a token ring. Since Dijkstra's paper, self-stabilization has been studied in many contexts, including communication protocols, graph theory problems, termination detection, clock synchronization, and fault containment. For a survey see, e.g., [11, 16, 24].

Also general techniques for self-stabilization have been considered. Awerbuch and Varghese [9] showed that every local algorithm can be made self-stabilizing if all nodes keep a log of the state transitions until the current state. Since then several other methods have emerged including various local and global checking and correction techniques [6, 13, 25, 35, 36]. Also so-called time-adaptive techniques [23, 26, 27] as well as local stabilizers [1] have been presented which can recover any distributed algorithm in $O(f)$ time depending only on the number f of faults. This, however, does not hold any more if faults include changes in the topology. In this case, a single fault may require the involvement of all nodes in the system and is therefore quite expensive to fix. Thus, people have also looked at so-called superstabilizing protocols, which are protocols that can handle a single topology change as well as arbitrary state faults with low overhead (e.g., [17]).

Interestingly, though much attention has been given to self-stabilizing distributed computing, even in the context of dynamic networks, the problem of designing self-stabilizing networks has only been given very little attention. The general techniques mentioned above are not applicable here as they have not been designed to actively perform local topology changes (network changes are only considered as faults or dynamics not under the control of the algorithm). Even though logging techniques such as [9] to convert non-self-stabilizing algorithms into self-stabilizing algorithms can also be applied to self-stabilizing networks, they usually need some non-local knowledge of the network (such as its size) to bound the state space which can make self-stabilization very expensive. Our goal instead was to find dedicated, much more light-weight algorithms for self-stabilizing networks.

Some preliminary work in this direction has already been done. In the technical report of the Chord network [34], protocols are described which allow the topology to recover from certain degenerate states. Similarly, it is also known how to repair skip graphs from certain degenerate states [3] but the problem of recovering them from an arbitrary connected state has remained open. This is not surprising as the neighborhood information alone is not sufficient for the Chord network as well as skip graphs to locally verify the correctness of the topology. Hence, additional information would be needed, which significantly complicates the self-stabilization process.

In order to recover scalable overlays from any initial graph, researchers have started with simple non-scalable line and ring networks. The *Iterative Successor Pointer Rewiring Protocol* [14] and the *Ring Network* [33] organize the nodes in a sorted ring. However, the runtime of both protocols is rather large. Aspnes et al. [2] describe an asynchronous protocol which turns an initially weakly connected graph into a sorted list. Unfortunately, their algorithm is not self-stabilizing. In a follow-up paper [5], a self-stabilizing algorithm is given for the case that nodes initially have out-degree 1. In [31], Onus et al. present a local-control strategy called *linearization* for converting an arbitrary connected graph into a sorted list. However, the algorithm is only studied in a synchronous environment, and the strategy may need a linear number of communication rounds. Clouser et al. [12] formulated a variant of the linearization technique for asynchronous systems in order to design a self-stabilizing skip list. Gall et al. [19] combined the ideas from [12, 31] and introduced a model that captures the parallel time complexity of a distributed execution that avoids bottlenecks and contention. Two algorithms are presented together with an analysis of their distributed runtime in different settings. No sublinear time bounds are shown there either.

To the best of our knowledge, this is the first paper to describe a self-stabilizing algorithm for a scalable overlay network (in our case, skip graphs) in sublinear time. In fact, the skip graph construction terminates in a polylogarithmic number of communication rounds. In addition to being able to recover quickly from an arbitrary connected state, we also show that when the network forms the desired topology, our algorithm efficiently supports join and leave events, which incur only a polylogarithmic amount of work to fix. This means (in contrast to considering a completely new starting situation and recovering the structure in polylogarithmic time) that only a small part of the nodes are involved in repairing the overlay topology.

1.2 Model

We represent an overlay network as a directed graph $G = (V, E)$, where $|V| = n$. Each node is assumed to have a unique identifier (or short: *ID*) $v.id \in U$ that is immutable, where U is the (ordered) universe of all identifiers. At any time, each node can inspect its own state and the state of its current neighbors. Beyond that, a node does not know anything, including the current size n of the overlay network. Only local topology changes are allowed, i.e., a node may decide to cut a link to a neighbor or ask two of its neighbors to establish a link. The view and the influence of a node are essentially local. The decisions to cut or establish links are controlled through *actions* (which we will also call *rules*). An action has the form `label` : `guard` $\rightarrow$ `commands` where `guard` is a Boolean predicate over the state of the executing node and its neighbors and `commands` is a sequence of commands that may affect the state of the executing node or request a new edge between two neighbors. This is done via an $insert(v, w)$ request by which a node asks its neighbor v to establish an edge to neighbor w. An action is called *enabled* if and only if its guard is true.

For simplicity, we assume that time proceeds in rounds, and all requests generated in round i are delivered simultaneously at the beginning of round $i + 1$. In other words, we assume the standard synchronous message-passing model with the restriction that a node can only communicate with nodes that it has currently links to. In each round, all actions that are enabled are executed by the nodes. If two actions executed by the same node are in conflict with each other, any one of them may win and the other is discarded. Our goal is to minimize the number of rounds needed in the worst case (over all initial states in which the network is weakly connected) until the overlay network has reached its desired structure. We make this a bit more precise by defining what we mean by self-stabilization.

When using the synchronous message-passing model, the global state of the system at the beginning of each round is well-defined. A *computation* is a sequence of states such that for each state s_i at the beginning of round i, the next state s_{i+1} is obtained after executing all actions that were fired in round i. In our context, we call a distributed algorithm *self-stabilizing* if from *any* initial state (from which a legal state is still reachable) it eventually reaches a legal

state in which no more actions are enabled, i.e., once the overlay network reaches its desired topology, it does not change anymore. Our goal will be to find a self-stabilizing algorithm that needs as few rounds as possible for this.

1.3 Our Contributions

We present a variant of the skip graph, called SKIP$^+$, that can be locally checked for the correct structure. For this graph, we present a distributed self-stabilizing algorithm that arrives at SKIP$^+$ for *any* initial state in which the nodes are weakly connected in $O(\log^2 n)$ rounds. This is an exponential improvement over all previous results on the number of communication rounds needed to arrive at a scalable overlay network. We also show that a single join event (i.e., a new node connects to an arbitrary node in the system) or leave event (i.e., a node just leaves without prior notice) can be handled by our algorithm with polylogarithmic work, demonstrating that our algorithm is not just useful for the worst case but also for the case where the overlay network is already forming the desired topology (which is the standard case in the literature).

1.4 Paper Organization

In the rest of this paper we present and analyze our self-stabilizing algorithm for SKIP$^+$ graphs. The paper ends with a conclusion.

2. ALGORITHM

We first introduce the skip graph SKIP$^+$ we want to construct and then present our algorithm ALG$^+$.

2.1 The SKIP$^+$ Graph

We start with the definition of the skip graph. In skip graphs, the identity of a node v consists of two components: $v.id$, a unique but otherwise arbitrarily chosen identifier, and $v.rs$, a (pseudo-)random bit string of sufficient length that was uniformly chosen at random when the node entered the system. Both parts are assumed to be immutable.

For a node v and a subset $W \subseteq V$ of nodes define the *predecessor of* v *in* W $\text{pred}(v, W)$ to be the node $u \in W$ such that $u.id = \max\{w.id \mid w \in W \text{ and } w.id < v.id\}$. By the assumption that no two nodes share the same id, this is well defined. If such a u does not exist, set $\text{pred}(v, W) := \bot$ and define $\bot.id = -\infty$. Similarly define the *successor of* v *in* W $\text{succ}(v, W)$ to be $u \in W$ such that $u.id = \min\{w.id \mid w \in W \text{ and } w.id > v.id\}$, or if this does not exist $\text{succ}(v, U) := \top$ and define $\top.id = \infty$. Here, $-\infty$ and $+\infty$ are resp. the lowest and largest elements in the identifier space U not allowed as identifiers of real nodes.

The definitions needed for the ideal skip graph are marked by a superscript $*$ to distinguish them from analogous definitions used in the algorithm, which are all based on the current local views of the nodes.

For any $i \geq 0$, let $pre_i(v)$ denote the first i bits of $v.rs$ (i.e., the prefix of $v.rs$ of length i) and $v.rs[i]$ represent the ith bit of $v.rs$. Now define the *level-i predecessor* of v by $\text{pred}^*_i(v) := \text{pred}(v, \{w \mid pre_i(w) = pre_i(v)\})$, and the *level-$i$ successor* of v by $\text{succ}^*_i(v) := \text{succ}(v, \{w \mid pre_i(w) = pre_i(v)\})$.

DEFINITION 2.1 (SKIP GRAPH). *Assume we are given a set of nodes together with associated IDs and random strings. In the corresponding skip graph, every node v is connected exactly to $\text{pred}^*_i(v)$ and $\text{succ}^*_i(v)$ for every $i \geq 0$ (except for the case of $\bot$ and $\top$).*

Given unique node identifiers, the skip graph is uniquely defined. It is not difficult to see that the skip graph has logarithmic diameter and maximum degree and allows hypercubic-style routing between any pair of nodes in $O(\log n)$ time, w.h.p. However, the nodes cannot locally verify the correctness of the skip graph topology. Therefore, we propose a slight extension of the skip graph that we call SKIP$^+$.

The definition of SKIP$^+$ requires us to also define (extended) predecessors and successors on level i with a specific value in the next bit. For any $i \geq 0$ and $x \in \{0, 1\}$ define $\text{pred}^*_i(v, x) = \text{pred}(v, \{w \mid pre_{i+1}(w) = pre_i(v) \circ x\})$ and similarly $\text{succ}^*_i(v, x)$ (where operator $\circ$ means concatenation of bit strings). Let $low^*_i(v) = \min\{\text{pred}^*_i(v, 0).id, \text{pred}^*_i(v, 1).id\}$ and $high^*_i(v) = \max\{\text{succ}^*_i(v, 0).id, \text{succ}^*_i(v, 1).id\}$, and let $v.range^*[i] \subseteq U$ be defined as $[low^*_i(v), high^*_i(v)]$. With this, the SKIP$^+$ graph has the neighbor set $N^*_i(v)$ of v at level i as the set of all nodes w with $pre_i(w) = pre_i(v)$ and $w.id \in v.range^*[i]$.

DEFINITION 2.2 (SKIP$^+$GRAPH). *Assume we are given a set of nodes together with associated IDs and random strings. In the corresponding SKIP$^+$ graph every node v is connected to exactly the nodes in $N^*_i(v)$ for all $i \geq 0$, i.e., $N(v) = \bigcup_{i \geq 0} N^*_i(v)$.*

Figure 1 illustrates the connections in SKIP$^+$. The white (resp. black) nodes in the figure illustrate the nodes v at level i for which $v.rs[i+1] = 0$ (resp. $v.rs[i+1] = 1$). The total sorted order of the nodes according to their identifiers is shown at the bottom; the SKIP$^+$ structure at level 0 is depicted in the middle, whereas the top part of the figure (top two connected components of black and white nodes) correspond to level 1. Note that skip graph edges of level $i+1$ appear in the SKIP$^+$ graph already on level i.

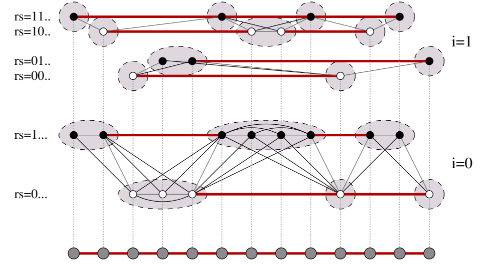

Figure 1: Visualization of SKIP$^+$ connections.

Since the skip graph is a subgraph of SKIP$^+$ (on the same set of nodes), SKIP$^+$ has a logarithmic diameter and constant expansion. Also, it is not too difficult to see that the maximum degree remains logarithmic w.h.p.

DEFINITION 2.3 (HEIGHT H). *The* height H_G *of a SKIP$^+$ graph G is defined as the maximal number of non-trivial levels. A non-trivial level is a level which consists of more than one node, that is, $H_G = \max\{|\rho| : |V_\rho| \geq 2\}$ where $V_\rho = \{v \in V \mid \rho = pre_{|\rho|}(v)\}$.*

It is straightforward to show that H_{SKIP^+} is in $O(\log n)$ w.h.p.

The goal is to establish the SKIP$^+$ graph as the target topology using bi-directed edges (this is why the edges in Figure 1 are undirected), but during the construction, the network has to deal with directed edges.

2.2 The ALG$^+$ Algorithm

In the description and analysis of our algorithm, we will make use of the following definitions.

DEFINITION 2.4 (GRAPH G_ρ). *Given any directed graph $G = (V, E)$ currently formed by the nodes and any prefix ρ, the graph $G_\rho = (V_\rho, E_\rho)$ is a subgraph of G where $E_\rho = \{(u, v) \in E : u, v \in V_\rho\}$. The nodes in V_ρ are called* ρ-nodes *and the edges in E_ρ are called* ρ-edges.

DEFINITION 2.5 (CONNECTED ρ-COMPONENT). *Given a prefix ρ, we will refer to a weakly connected component of nodes in G_ρ as a* connected ρ-component*. A pair of nodes in such a component is called* ρ-connected*.*

For any node v let $N(v)$ be its *current* outgoing neighborhood and $v.range[i]$ be its current range at level i, which is based on its *current view* of $\mathrm{pred}_i(v,x)$ and $\mathrm{succ}_i(v,x)$, where $\mathrm{pred}_i(v,x)$ is the node such that $\mathrm{pred}_i(v,x).id = \max\{w.id \mid w \in N(v)$ and $w.id < v.id$ and $pre_{i+1}(w) = pre_i(v) \circ x\}$ and $\mathrm{succ}_i(v,x)$ is the node such that $\mathrm{succ}_i(v,x).id = \min\{w.id \mid w \in N(v)$ and $w.id > v.id$ and $pre_{i+1}(w) = pre_i(v) \circ x\}$. For each level i, $v.range[i] \supseteq v.range^*[i]$, i.e, the current range will always be a superset of the desired range in the target topology (as defined in SKIP$^+$). We will see that as long as no faults or adversarial changes happen during the self-stabilization process, ALG$^+$ monotonically converges to the desired ranges for every i.

ALG$^+$ distinguishes between *stable* edges and *temporary* edges. Node v considers an edge (v,w) to be temporary if from v's point of view (v,w) does not belong to SKIP$^+$ and so v will try to forward it to some of its neighbors for which the edge would be more relevant. Otherwise, v considers (v,w) to be a stable edge and will make sure that the connection is bi-directed, i.e., it will propose (w,v) to w. There is a binary flag $v.F(w)$ for each neighbor w that states whether the edge to it is stable. The flag turns out to be important when a stable edge destabilizes (i.e., converts into a temporary edge) because this triggers the introduction of several temporary edges that are needed for our proofs to go through. The other conversion, from temporary to stable, essentially boils down to introducing also the other direction of the edge. More details will be given later.

The intuition behind the ALG$^+$algorithm is as follows. The algorithm has two main phases: The first phase proceeds in a *bottom-up* (i.e., from level 0 upwards) fashion, forming connected ρ-components for every prefix ρ. This will be accomplished by letting each node v find another node w of the opposite color, i.e., such that $pre_i(w) = pre_i(v)$ and $v.rs[i+1] \neq w.rs[i+1]$ for all levels $i \geq 0$ (we will call w a *buddy* of v). We can show that once all nodes in V_ρ have formed a single connected component and have found a buddy, then connecting all nodes which are at most three hops away in the ρ-component results in a single connected $\rho0$- and $\rho1$-component. This will be accomplished by Rules 1 (where new nodes in the range of a node are discovered and where ranges may be refined) and Rules 3 (where an efficient variation of a local transitive closure is performed) below.

Once the connected ρ-components are formed, the second phase of the algorithm will form a sorted list out of each ρ-component. This is accomplished in a *top-down* fashion by merging the two already sorted $\rho0$- and $\rho1$-components into a sorted ρ-component until all nodes in the bottom level form a sorted list.

Of course, this "division into phases"-intuition oversimplifies what is really going on in our algorithm. Whereas for the sake of simplicity, we can think of the execution of the phases of the algorithm as being perfectly synchronized, with all nodes waiting for the connected components at level i to converge before the components at level $i+1$ are formed in the first phase, and with the sorting of the components at level i only starting after we have successfully sorted the components at level $i+1$ in the second phase (and of course with the second phase only kicking in after the first phase is completed), all actions in our algorithm may be enabled *at any time*, causing the phases to be intertwined in the real execution. Hence, the main challenge in this paper is to show that nevertheless the actions transform any initially connected graph into SKIP$^+$ in $O(\log^2 n)$ rounds.

Now we are ready to present ALG$^+$. The *local state* of a node v consists of $v.id$, $v.rs$, $N(v)$ (which imply the edges and ranges of v) and its flags. We assume that every node v knows (besides its own local state) the current local state of all nodes $w \in N(v)$. Hence, the actions of a node v may be based on any local state information of v or its neighbors. Recall that we assume the synchronous message-passing model. At the beginning of a round i, every node receives all the requests to establish an edge that were generated in the previous round. After a preprocessing step in which each node updates its neighborhood and the state information about its neighborhood, a set of three types of actions is processed, which we also call rules here. For readability, we will present the rules in words, but transforming them into the formal terminology of our model is straightforward. We note that the preprocessing step is separated from the actions to ensure a deterministic state transition in the synchronous message-passing model. In an asynchronous model, the preprocessing step would be continuously performed in the background at any time. For each node u we do the following in a round:

Preprocessing.

First, a node u processes all $insert(u,v)$ requests from the previous round where $v \in V \setminus (\{u\} \cup N(u))$ (the others would be thrown away, but our algorithm avoids issuing such requests in the first place). This is done by adding v to $N(u)$ and setting its flag $u.F(v)$ to 0 (temporary). Then u makes sure that its state is valid, i.e., the flags carry binary values and $N(u)$ is a set of nodes $v \neq u$ that are all alive (otherwise, u removes that node from $N(u)$). Now u determines for every i its current predecessors $\mathrm{pred}_i(u,0)$ and $\mathrm{pred}_i(u,1)$ and its current successors $\mathrm{succ}_i(u,0)$ and $\mathrm{succ}_i(u,1)$ (within $N(u)$). This allows u to update its range information. The updated local state is exchanged between the nodes so that the rules below are based on up-to-date information.

DEFINITION 2.6 (STABLE EDGES). *Every edge (u,v) is considered* stable*, if*

- *its endpoints mutually fall on each other's range at some level $pre_i(u) = pre_i(v)$, i.e., $v.id \in u.range[i]$ and $u.id \in v.range[i]$, for some i. In this case (u,v) is* defined *to be stable on all levels $j \geq i$ with $pre_j(u) = pre_j(v)$.* Or

- *$v = \mathrm{pred}_i(u,x)$ or $u = \mathrm{pred}_i(v,x)$, or $v = \mathrm{succ}_i(u,x)$ or $u = \mathrm{succ}_i(v,x)$, for some level i and bit $x \in \{0,1\}$. In this case (u,v) is stable only on level i.*

This second kind of stable edges is needed to stay in touch with a "buddy" (see below), in order to forward temporary edges. Our algorithm guarantees that once a node has a buddy to the left (or to the right), it will always have such a buddy in the future.

DEFINITION 2.7 (LEVEL OF TEMPORARY EDGE). *We define the level of a temporary edge (u,v) as the length of the longest common prefix of u and v.*

All of the rules below are only activated if the resulting action changes the graph or the state, i.e., if the to be inserted edge does not already exist or the flag changes its value.

Rule 1a: Create Reverse Edges.

For every stable edge (u,v), u sets $F(v) = 1$ (if it has not already done so) and initiates an $insert(v,u)$ request.

Rule 1b and Rule 1c: Introduce Stable Edges.

For every stable neighbor v (the edge (u, v) is considered stable as defined in Preprocessing) of a node u, for every $i \geq 0$ and every node $w \in N(u), w \neq v$ with $pre_i(v) = pre_i(w)$ and $w.id \in v.range[i]$, node u initiates $insert(v, w)$ (Rule 1b) and $insert(w, v)$ (Rule 1c).

Rule 2: Forward Temporary Edges.

Every temporary edge (u, v) is forwarded to a stable neighbor of u that has the largest common prefix with $v.rs$. (Such an edge exists because otherwise (u, v) would be a stable edge.)

Rule 3a: Introduce All.

For all nodes $u \in V$ whose set of stable neighbors is different from the previous round, u initiates $insert(v, w)$ for all neighbors w of u. (In particular, if an edge destabilizes, both incident nodes will introduce their neighbors.)

Rule 3b: Linearize.

For every level i, u identifies the stable neighbors $v_1, \ldots, v_k$ with $v_1.id < v_2.id < \ldots < v_k.id$ that have exactly the first i bits in common with $u.rs$ and initiates $insert(v_1, v_2)$, $insert(v_2, v_3)$,..., $insert(v_{k-1}, v_k)$ for them.

3. ANALYSIS

We first analyze the bottom-up phase and then tackle the top-down phase.

3.1 Bottom-up Phase

LEMMA 3.1. *Consider any bit string $\rho \in \{0, 1\}^*$. Suppose that nodes a and b are ρ-connected at time t_0. Then a and b are also ρ-connected at any time $t \geq t_0$.*

PROOF. We prove the lemma by induction over the time steps. Consider any edge $e = (u, v)$ (temporary or stable) at time $t \geq t_0$ with $u, v \in V_\rho$. The only rule that may remove this edge is Rule 2, all other rules only create edges. If the edge e is forwarded by Rule 2 to a node w, node w must have a shared prefix with u that extends ρ, and all three nodes u, v, w remain connected in G_ρ at the next time step. □

In the following lemma, an edge (u, v) is said to be *to the right* (resp. *left*) if $u.id < v.id$ (resp. $u.id > v.id$).

LEMMA 3.2. *Assume a node u has a stable ρ-edge to the right (left) at time t_0. Then at any time $t > t_0$, node u will have a stable ρ-edge to the right (left).*

PROOF. By induction over time. A stable ρ-edge (u, v) only destabilizes because the $|\rho|$-range of u has become smaller, and there now is another ρ-node w stably connected to u, and w is between u and v. □

A *ρ-buddy* of a node u is a stable neighbor v (the edge (u, v) exists and is considered stable) with $pre_{|\rho|}(u) = pre_{|\rho|}(v) = \rho$ and $u.rs[|\rho| + 1] \neq v.rs[|\rho| + 1]$. Our algorithm ensures that once a node has a left (or right) buddy at time t, then it will also have a left (or right) buddy at all times $t' \geq t$. In the following, the V in *V-linked* does not refer to a set V but to the V-shaped situation of two nodes being linked indirectly via precisely two edges to a third node.

DEFINITION 3.3 (σ-V-LINK). *Consider any $\rho \in \{0, 1\}^*$ and $x \in \{0, 1\}$. Assume there are two nodes u, v with prefix $\rho x = \sigma$ and one node w with prefix $\rho\bar{x}$. If $u, v \in N(w)$, we say that u and v are* σ-V-linked *via w.*

LEMMA 3.4. *Assume that u and v are σ-V-linked via w at time t. Then, at time $t + 1$ the nodes u and v are σ-connected, i.e., in the same connected component of G_σ.*

PROOF. Assume (w.l.o.g.) that u, v have label $\sigma = \rho 1$, and w has label $\rho 0$. By Rule 3b, all stable $\rho 1$-neighbors of w are in the same $\rho 1$-component C at time $t + 1$. If (w, u) is a temporary edge, this is forwarded to one of the stable $\rho 1$ neighbors of w, and hence u is also in the $\rho 1$-component C. By the same reasoning v is also in C at time $t + 1$, which proves the lemma. □

LEMMA 3.5. *Consider any $\rho \in \{0, 1\}^*$ and $x \in \{0, 1\}$. Let node a with label ρx be a ρ-buddy of u at time t and b a ρ-buddy of u at time $t' \geq t$. Then a and b are in the same ρx-component at time $t' + 1$.*

PROOF. If than $t = t'$, this follows from Lemma 3.4. Otherwise the proof is an induction over time. Let $a = a_t, a_{t+1}, \ldots, a_{t'} = b$ be ρ-buddies of u at the respective times. With Lemma 3.1, it suffices to show that a_{i-1} and a_i are in the same ρx-component at time $i + 1$. Because no rule deletes or forwards stable edges, the edge (u, a_{i-1}) exists at time i and Lemma 3.4 can be applied. □

Note that the σ in "σ-V-linked" refers to the nodes that are linked, not the node that is providing the link. Similarly, we define a bridge by the prefix of the nodes that are connected by this bridge, not by the nodes that provide the bridge.

DEFINITION 3.6 (TEMPORARY/STABLE BRIDGE). *Consider two nodes a, b with prefix $\sigma = \rho x$ that are in different connected components of G_σ. Then, for $k \geq 1$, a and b are connected by a* (σ, k)-bridge *if there is a node c with stable edge (c, a) and a node d with stable edge (d, b), where c, d have prefix $\rho\bar{x}$ and have the edge (c, d) at level $|\rho| + k$ (i.e., $pre_{|\rho|+k}(c) = pre_{|\rho|+k}(d)$). If (c, d) is stable, this is a* stable bridge*; if (c, d) is temporary, it is a* temporary bridge.

The following definition is needed to show connectedness moving up the levels by the rules. It is central to the bottom-up proof that eventually for all prefixes ρ we have that G_ρ consists of only one connected component. In the process where the connectedness of $G_{\rho 0}$ follows from G_ρ being connected, the bridges play an important role. First, as long as a bridge is temporary, the level increases in every step, but then, once it becomes a stable bridge, the level of the used bridges decreases.

DEFINITION 3.7 ((σ, k)-PRE-COMPONENT). *Two nodes a and b, both with prefix $\sigma = \rho x$ for some $x \in \{0, 1\}$, are (directly)* (σ, k)-pre-connected *if (1) G_ρ is connected, if (2) every node in $G_{\rho x}$ knows at least one node from $G_{\rho\bar{x}}$ and vice versa,[1] and if (3) there is an edge (irrespective of this being temporary or stable), between a and b, or if they are σ-V-linked, or if there exists a stable (σ, k')-bridge with $k' \leq k$ between them. The transitive closure of this (undirected) relation defines the* (σ, k)-pre-component.

[1]Throughout this paper, $\bar{\cdot}$ will denote the logical negation.

Requirements (1) and (2) will be made precise in the subsequent analysis and especially Lemma 3.14 where the connectivity of G_ρ is proved by induction. Note that for $k = 0$, the σ-links and the $(\sigma, 0)$-pre-component (but without stable bridges) yield the transitive hull of the σ-V-links. Figure 2 illustrates the situation.

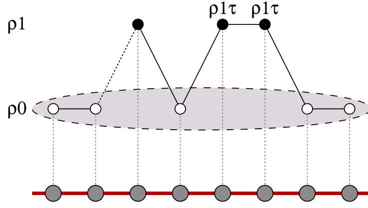

Figure 2: The shaded nodes belong to the same $(\rho 0, k)$-pre-component. In this figure, $|\tau| = k - 1$, temporary edges are dashed and stable edges are solid.

LEMMA 3.8. *Assume nodes a, b are in the same (σ, k)-pre-component at time t_0, $k \geq 1$. Then a and b are in the same (σ, k)-pre-component at any time $t > t_0$.*

PROOF. Assume w.l.o.g. $\sigma = \rho 1$. By induction over time, and the definition of the pre-component being a transitive closure, it is sufficient to argue that nodes a, b with prefix $\rho 1$ that are directly (σ, k)-pre-connected at time t are in the same (σ, k)-pre-component at time $t + 1$.

If a, b are directly linked, we can employ Lemma 3.1, and the case that they are σ-V-linked is due to Lemma 3.4. The remaining (and only interesting) case is that a and b have a (σ, k)-bridge via a stable edge $e = (u, v)$ (then there are also stable edges (u, a) and (v, b)) at time t. Let a' and b' be σ-nodes that are stable neighbors of u and v respectively. Then a and a' are σ-V-linked and are in the same σ-component at time $t+1$ by Lemma 3.4. The same reasoning shows that b and b' are in the same σ-component at time $t + 1$. If e remains stable, a' and b' have a (σ, k)-stable bridge also at time $t + 1$. Otherwise e destabilizes and by Rule 3a a temporary edge (v, a') or (u, b') is present at time $t + 1$, so a' and b' are σ-V-linked via v or u, and hence in the same (σ, k)-pre-component at time $t + 1$. □

LEMMA 3.9. *Assume that nodes a, b are in the same (σ, k)-pre-component at time t, $k > 1$. Then, a and b are σ-connected at time $t + 4$.*

PROOF. If a and b are in the same (σ, k)-pre-component (assume again that $\sigma = \rho x$ for some $x \in \{0, 1\}$), there must exist a path $a = p_1 \rightsquigarrow p_2 \rightsquigarrow \ldots \rightsquigarrow p_\ell = b$ where p_j is connected to p_{j+1} either directly, via a σ-V-link, or via a stable (σ, k)-bridge. We only allow a bridge between p_j and p_{j+1} if these two nodes are not in the same $(\sigma, 0)$-pre-component.

Consider this path and assume p_j and p_{j+1} are connected by a bridge over an edge $e = (u, v)$ of length λ (w.r.t. node identifiers, u and v having prefixes of the form $\rho\overline{x}\tau y$ where $|\tau| = k - 1$ and $y \in \{0, 1\}$). Then, whenever possible, we replace e by two edges via an intermediate node w if for the lengths it holds that $|(u, w)| < \lambda$ and $|(w, v)| < \lambda$. Thus, in our path, a new node p_*—w's buddy—is inserted, maintaining our "path property". Note that this process terminates (the number of "turning points"—the local extrema of the identifiers along the path—does not increase).

Consider a bridge edge (u, v) on this path with shortest bridges. First, consider the case that (u, v) is *unilaterally* stable because of the nearest neighborhoods. That is, w.l.o.g. assume v is u's nearest neighbor but for some level, u is not in v's range. Then, it holds that v must know two closer nodes than u (due to the range definition), one of which will be proposed to u in the next round (hence (u, v) will no longer be stable), triggering Rule 3a at $t + 2$, from which the claim follows.

From now on it remains to consider the case where u and v mutually include each other in their ranges on some level. First assume that (u, v) is stable on a level $> |\sigma|$—the other case where (u, v) is stable on level $|\sigma|$ is treated later. Let us refer to the lowest level where (u, v) is stable as $i+1$, that is, where $pre_{i+1}(u) = pre_{i+1}(v)$ but either $u.id \notin v.range[i]$ or $v.id \notin u.range[i]$ (or both, cf Definition 2.6), so $|\sigma| + k \geq i + 1 > |\sigma|$. W.l.o.g., assume that $u.id \notin v.range[i]$. Due to the definition of the ranges, v must have both a stable predecessor (relatively to the current neighborhood) with last bit 0 and a predecessor with last bit 1 that lie between u and v: $w_0 = \mathrm{pred}_i(v, 0)$ and $w_1 = \mathrm{pred}_i(v, 1)$. Let $w \in \{w_0, w_1\}$ such that $pre_{i+1}(u) = pre_{i+1}(w) = pre_{i+1}(v)$. See Figure 3.

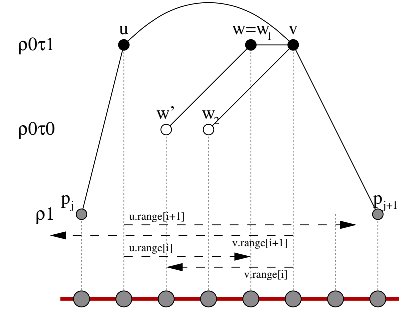

Figure 3: Situation for $i + 1 > |\sigma|$ in proof of Lemma 3.9 at time t. Here we use $x = 0$.

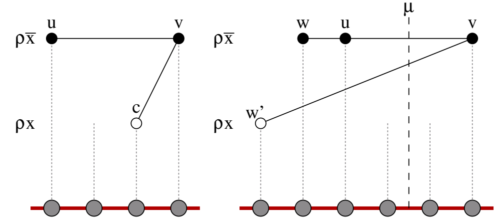

Figure 4: The two situations for $i + 1 = |\sigma|$ in proof of Lemma 3.9.

However, $(w, u) \notin E$ at time t, due to our selection of the shortest bridge path: (u, w) and (w, v) would imply a path (see your "path property" described before) with shorter bridges. Thus, at time $t + 1$, (u, w) is created by Rule 1b. Now, we will show that this triggers u to fire Rule 3a at $t + 2$, either because (u, w) becomes stable or because (u, v) becomes instable. (Note that (u, v) must be stable at time t, since the edge forms a bridge.) Observe that it is not possible that (u, v) remains stable and (u, w) is instable, since both are within u's range on level $i + 1$, as u and w have a common prefix $\rho = pre_{i+1}(w)$. Thus, at time $t + 1$, Rule 3a is triggered at u, and thus, at $t + 2$, a and b are σ-V-linked and connected in round $t + 3$.

Observe that if w has a buddy w' closer to u, see Figure 3, the same arguments apply, as w' will be introduced to v, which triggers Rule 3a at v.

It remains to study the case where (u, v) is stable on level $i = |\sigma|$ (see Figure 4). Assume u is stably connected to p_j and v is stably connected to p_{j+1} ($u, v \in G_{\rho\overline{x}}$). We distinguish two cases: Either there is a node $c \in G_{\rho x}$ between u and v or not. First, assume there is such a node c, and w.l.o.g., assume c is stably connected to v on

level $|\rho|$. (Observe that due to our path selection, $c \notin G_{\rho\overline{x}}$, and hence it holds that $c \in G_{\rho x}$.) We will show that in this case, a new connection (u, c) is proposed. Thus, there cannot be both edges (u, c) and (c, v), otherwise the bridge would not be necessary, as the nodes must be in the same $(\sigma, 0)$-pre-component. Therefore, either (u, v) destabilizes at time $t+1$, triggering Rule 3a at $t+2$, or (u, c) or (c, v) must be proposed according to Rule 1b. By the same reasoning as above, the claim follows for time $t+4$ in this case.

If there is no such node c, consider the buddy of u or v closest to position $\mu = (u.id + v.id)/2$. That is, let B denote the set of all buddies of u and v on level i, i.e., $B = (N(u) \cup N(v)) \cap G_{\rho x}$. Let $w \in B$ be the buddy which minimizes $|\mu - w|$. Without loss of generality, assume w is the buddy of u. Now let w' be the buddy of v which is located on the same side of μ as w. Observe that $v.range[i]$ is defined by w' due to our assumption that c does not exist and since w' is further away from μ and hence also from v. Therefore, $w \in v.range[i]$. On the other hand, if no such w' exists, then $v.range[i]$ is not bounded, and the claim follows trivially. Thus, a and b are σ-V-linked in round $t+2$. Therefore, the claim also follows for the case $i+1 = |\sigma|$, which concludes the proof. □

LEMMA 3.10. *A temporary edge (u, v) of level ℓ at time t is either transformed into a stable edge or forwarded and changed by this into a temporary edge of level at least $\ell + 1$.*

PROOF. Let ρ be the common prefix of u and v with $|\rho| = \ell$, and assume w.l.o.g. that u has label $\rho 0$ and v has label $\rho 1$. If (u, v) is temporary, it either becomes stable by Rule 1a or there must be a stable $\rho 1$-node $w \in N(u)$ between u and v In this case, Rule 2 replaces (u, v) with (w, v) for such a w, which is a temporary edge of level at least $\ell + 1$. □

The following lemma follows from the previous one because no temporary edge has a level higher than the height H.

LEMMA 3.11. *Every temporary edge becomes stable after H time steps.*

Next, the case of temporary bridges is investigated.

LEMMA 3.12. *Assume two ρx-nodes a, b for some $x \in \{0, 1\}$ are directly connected by a temporary ρx-bridge, i.e., there is a temporary edge (u, v) and stable edges (u, a) and (v, b). Then, for $k = H - |\rho|$, a and b are $(\rho x, k)$-pre-connected at time $t + k$.*

PROOF. W.l.o.g. let $x = 1$. By the rules of temporary edges, the destination v never changes. Let b_i be a ρ-buddy of v, i.e., the stable edge (v, b_i) exists at time $t+i$, $b_0 = b$. At time $t+i+1$, there is at least the temporary edge (v, b_i), and b_i and b_{i+1} are $\rho 1$-V-linked, and hence, by Lemma 3.8, b_i is $(\rho 1, k)$-pre-connected with b also at time $t + k$.

Let u_i be the starting point of the temporary edge at time $t+i$, i.e. (u_i, v) be the temporary edge, and let a_i (with $a_0 = a$) be some stable neighbor of u_i at that time, i.e., the stable edge (u_i, a_i) exists at time $t+i$. At time $t+i$ the stable edge (u_i, u_{i+1}) exists (because the temporary edge was forwarded along this edge). Further, let c_i be some stable neighbor of u_{i+1} at time $t+i$. If this does not exist, set $c_i = a_i$, and note that at time $t+i+1$ the stable edge (u_{i+1}, a_i) must exist because of Rule 1b. By definition, a_i and c_i have a stable $(\rho 1, k')$-bridge for $k' \leq k$ and are hence $(\rho 1, k)$-pre-connected. At time $t+i+1$ at least the temporary edge (u_{i+1}, c_i) exists, and hence c_i and a_{i+1} are $\rho 1$-V-linked, so by Lemma 3.8 a_i is $(\rho 1, k)$-pre-connected with a at time $t + k$.

By Lemma 3.10 the level of (u_i, v) is at least $|\rho| + i$. Hence for some $j \leq k$ the stable edge (u_j, v) exists at level smaller than H at time $t+j$. Hence, at time $t+j$ the nodes a_j and b_j are $(\rho 1, k)$-pre-connected, and we can conclude from Lemma 3.8 that a and b are $(\rho 1, k)$-pre-connected at time $t + k$. □

LEMMA 3.13. *Consider any bit string $\rho \in \{0, 1\}^*$. Suppose that G_ρ is weakly connected at time t_0. Then every node $u \in V_\rho$ will have a neighbor in $V_{\rho 0}$ and $V_{\rho 1}$ within $O(\log n)$ rounds.*

PROOF. Consider any node u that does not have a neighbor in $V_{\rho 0}$ or $V_{\rho 1}$. In this case, $u.range[i] = U$ for $i = |\rho|$, which implies together with Rule 1b and Rule 1c that every node $v \in N(u) \cap V_\rho$ with missing ρ-buddy will introduce every node $w \in N(v) \cap V_\rho$ to u. Suppose w.l.o.g. that u is still missing a node in $V_{\rho 0}$. Then the neighbor introduction of Rule 1b and Rule 1c implies that the distance of u in G_ρ to the closest node in $V_{\rho 0}$ is cut in half in each round. Thus, it takes at most $O(\log n)$ rounds into a node in $V_{\rho 0}$ is a direct neighbor of u, which implies the lemma. □

Lemma 3.13 can also be proved by observing that in every round, due to the "pointer doubling" operations, the diameter of the connected component formed by nodes without a buddy is cut in half.

LEMMA 3.14. *Assume G_ρ is connected at time t. Then, at time $t + (H - |\rho|) + O(\log n)$ the graph $G_{\rho 0}$ is connected and so is $G_{\rho 1}$.*

PROOF. Define $k = H - |\rho|$. At time $t_0 = t + O(\log n)$, by Lemma 3.13, every $\rho 0$ node has a stable connection to a $\rho 1$ node and vice versa. Let a and b be two $\rho 1$ nodes. Because G_ρ is connected at time t, it is so at time t_0, and there is a path $a = u_1, u_2, \ldots, u_m = b$ in G_ρ at time t_0. If u_i is a $\rho 0$ node, define v_i to be one of its $\rho 1$ buddies, otherwise set $v_i = u_i$. Then, by definition, at time t_0 the nodes v_i and v_{i+1} are either directly connected, or connected by a temporary $(\rho 1, k)$-bridge. Hence, by Lemma 3.12 (or Lemma 3.1), at time $t_0 + k$ the nodes v_i and v_{i+1} are in the same $(\rho 1, k)$-pre-component, and by Lemma 3.9, Lemma 3.8, and Lemma 3.1 v_i and v_{i+1} are in the same connected component of $G_{\rho 1}$ at time $t_0 + k + O(1)$. With this and the symmetric argument for $G_{\rho 0}$ the claim of the lemma follows. □

3.2 Top-down Phase

In the previous section, it has been shown that in $O(\log^2 n)$ rounds, all nodes sharing a given prefix ρ have discovered each other and are connected (via other nodes with prefix ρ). In addition, each node is connected—on each level—to at least one "buddy" having the opposite last bit of the corresponding prefix. We now prove that after these properties have been achieved, the final SKIP$^+$topology is established in only $O(\log n)$ rounds.

The analysis of the top-down phase is done by induction overloading. For our induction step, we need the concept of *finished* levels.

DEFINITION 3.15 (i-FINISHED). *We say that a graph is i-finished if and only if*

1. *$\forall \rho$ with $|\rho| = i$, it holds that G_ρ contains all edges of the SKIP$^+$-graph as stable edges;*
2. *$\forall u \in V$ let $\rho = pre_i(u)$ and $\forall j < i$: if v is a level j right-buddy of u (current closest node on the right with $pre_j(v) = pre_j(u)$ and $v.rs[j+1] \neq u.rs[j+1]$), then for all $\forall w$ with $pre_i(w) = \rho$ (i.e., $w \in G_\rho$) and w lying between u and v, i.e., $w.id \in [u.id, v.id]$, it holds that u and w are connected by a stable edge. If there is no such buddy v, then u is connected to all nodes to the right of u in G_ρ; and similarly to the left.*

Figure 5 shows an example.

Observe that after the bottom-up phase, the "top label" is finished trivially: This label forms a graph G_ρ with more than one node, whereas $G_{\rho 1}$ and $G_{\rho 0}$ are trivial, i.e., consist of a single node. Clearly, for a top label ρ the graph G_ρ consists of precisely two nodes. In addition, once the graph G_ρ is connected, it contains all edges of the SKIP$^+$-graph as stable edges.

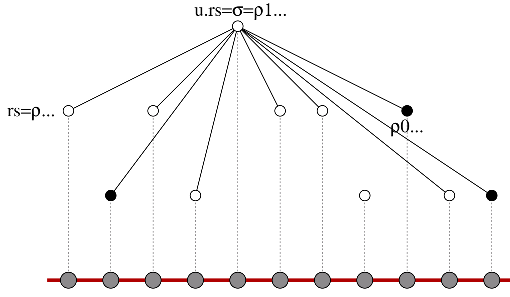

Figure 5: Visualization of the i-finished concept: node u is connected to all nodes having prefix ρ between its buddies on level i. Note that the distances to the buddies may not decrease monotonously towards lower levels!

The following reasoning shows that the levels will finish one after the other starting from the highest level (hence this phase's name), where each level takes constant time only.

LEMMA 3.16. *Assume at time t the graph is i-finished. Then, at time $t+3$, the graph is $(i-1)$-finished*

PROOF. Figure 6 illustrates the situation. We consider a node u. We will show that at time $t+1$, u knows its closest level-i neighbor w in the direction of the old buddy (that must exist due to Lemma 3.14 of the bottom-up phase). Having established this, it follows directly that at time $t+2$, w will inform u about all other neighbors in the desired region (Rule 1b and Rule 1c). At time $t+3$, node u will be informed about its neighbors on the opposite side of the level $i-1$ interval by the corresponding buddy, establishing our induction invariant.

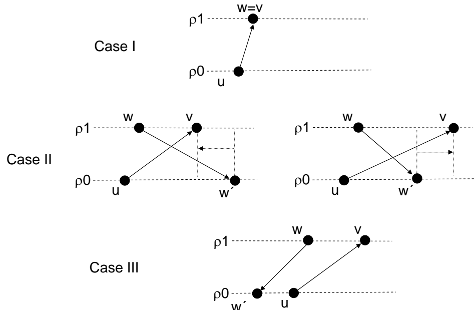

Figure 6: Proof of Lemma 3.16.

In order to prove that u knows its closest neighbor w at time $t+1$, we distinguish three cases. From the bottom-up phase, we know that u already has a buddy on one side. Without loss of generality, assume this buddy is on the right of u. Let this buddy node be denoted by v.

Case I: If v is already the closest node to the right, the claim holds trivially ($v = w$).

We know that w also has a buddy, which can either be on the right (Case II) or on the left (Case III) of w (and hence also u). Let w's buddy (take any if w has two) be denoted by w'.

Case II: Assume w' is also on the right of w. We distinguish two cases: Either v is left of w' or right of w'. If v is left of w', by our induction hypothesis, w must have a stable edge to node v. By Rule 1b and Rule 1c, v will introduce w to u at $t+1$ (edge (w, u)), and the claim follows. The case where v is right of w' is analogue: the roles of v and w' are simply switched.

Case III: Now assume w has a buddy w' on the left. We distinguish two cases. If u has another buddy u' on the left as well, the same arguments as in Case II show that either u' or w' will introduce the corresponding neighbors at time $t+1$. If, on the other hand, u does not have a buddy on the left, then by our induction hypothesis, w' must have a stable edge to u, over which u is introduced to w in the next round as well. □

Finally, observe that after all levels are finished, all non-SKIP$^+$ edges must be temporary. They will be forwarded towards the highest level and disappear in logarithmic time (in H rounds). Therefore, we conclude that the top-down phase takes logarithmic time only.

3.3 Combining Bottom-up and Top-down

From the previous discussion, we can draw the following conclusions. From Lemma 3.14, by summing up over all levels, it follows that the bottom-up phase lasts for at most $O(\log^2 n)$ rounds w.h.p. The subsequent top-down phase takes time $O(\log n)$ (cf Lemma 3.16) w.h.p. Thus, we have derived the following theorem.

THEOREM 3.17. *Given an arbitrary weakly connected graph, the self-stabilizing algorithm ALG$^+$ constructs SKIP$^+$ in $O(\log^2 n)$ rounds, w.h.p.*

4. NODE JOIN/LEAVE

In this section we study the amount of work it takes to handle a node departure (leave) or the inclusion of a new node (join). We show that once we reach a valid SKIP$^+$ graph, our algorithm can efficiently support any single join or leave operation in $O(\log^4 n)$ work and $O(\log n)$ rounds w.h.p. For proving this result, we first need to bound the degree of a node in SKIP$^+$.

LEMMA 4.1. *The degree of a node v in SKIP$^+$ is $O(\log n)$ w.h.p.*

PROOF. Recall the definition of a SKIP$^+$ graph and consider any node v. For any level i, let the random variable X_i^R be defined as

$$X_i^R = \max\{|\{w \in N_i^*(v) \mid w.id > v.id\}| - 1,\ 0\}.$$

In order to bound the probability that $X_i^R = k$ for some $k > 0$, we consider three cases. If v does not have $k+1$ nodes in $N_i^*(v)$ to the right of it, then $\Pr[X_i^R = k] = 0$. If v has exactly $k+1$ nodes in $N_i^*(v)$ to the right of it, then $X_i^R = k$ if and only if for the k closest successors w of v in $N_i^*(v)$ it holds that $w.rs[i+1] = \mathrm{succ}_i^*(v).rs[i+1]$, so $\Pr[X_i^R = k] = 1/2^{k-1}$. Finally, if v has more than $k+1$ nodes in $N_i^*(v)$ to the right of it, then $X_i^R = k$ if and only if the closest k successors w of v in level i have the property that $w.rs[i+1] = \mathrm{succ}_i^*(v).rs[i+1]$ and the $(k+1)$th closest successor of v, w', satisfies $w'.rs[i+1] \neq \mathrm{succ}_i^*(v).rs[i+1]$, so $\Pr[X_i^R = k] = 1/2^k$. In any case, $\Pr[X_i^R = k] \leq 1/2^{k-1}$ for any $k > 0$, and this probability bound holds independently of the other levels. Hence, for $X_R = \sum_{i=1}^{H} X_i^R$, where $H = \Theta(\log n)$ is an upper bound on the number of levels that holds w.h.p., it follows that

$$\Pr[X_R = d] \leq \binom{d}{H} \frac{1}{2^{d-H}}.$$

If $d = c \cdot H$, we get

$$\binom{d}{H} \frac{1}{2^{d-H}} \leq \frac{(ec)^H}{2^{cH-H}} \leq \frac{1}{n^{c'}}$$

for some constant c' that can be made arbitrarily large if the constant c is sufficiently large. Hence, the number of v's neighbors to the right is at most $O(\log n)$ w.h.p. A similar argument applies to the left neighbors of v, which implies the claim. □

THEOREM 4.2. *When a node v leaves the system, it takes $O(\log n)$ rounds of the algorithm and $O(\log^4 n)$ total work w.h.p. for the graph to converge back to a valid SKIP$^+$ structure.*

PROOF. Certainly, only the nodes that were directly connected to node v will need to adapt their current set of neighbors upon the departure of v (since the departure of v could not possibly alter the neighborhoods or ranges of other nodes, given that v is directly connected to all nodes in its range for all levels). By Lemma 4.1, the size of the entire neighborhood of node v (across all levels) is $O(\log n)$ w.h.p., so only $O(\log n)$ nodes need to change their neighborhood.

In order to show that these $O(\log n)$ nodes can quickly adapt their neighborhoods, we distinguish between several cases for every level i. In these cases, let V_l (resp. V_r) be the set of all left (resp. right) neighbors $w \in N_i^*(v)$ with $w.rs[i+1] = v.rs[i+1]$ and let W_l (resp. W_r) be the set of all left (resp. right) neighbors $w \in N_i^*(v)$ with $w.rs[i+1] \neq v.rs[i+1]$. Certainly, $V_l \cup V_r \cup W_l \cup W_r = N_i^*(v)$. Let v_l, v_r, w_l and w_r be the closest neighbors in V_l, V_r, W_l and W_r to v. Let us assume for now that v_l, v_r, w_l and w_r exist.

Case 1: $w_l.id < v_l.id < v_r.id < w_r.id$. In this case, all neighborhoods are correct once v has been removed, so no further edges are needed by the nodes.

Case 2: $v_l.id < w_l.id < v_r.id < w_r.id$. In this case, all nodes in $\{v_l\} \cup W_l \setminus \{w_l\}$ have to learn about v_r and vice versa to update the neighborhoods. The other nodes just have to remove v from their neighborhood. Since w_l has edges to all nodes in $\{v_l, v_r\} \cup W_l$, a single round of applying Rule 3a suffices to update all neighborhoods correctly.

Case 3: $w_l.id < v_l.id < w_r.id < v_r.id$. This case is just the reverse of Case 2.

Case 4: $v_l.id < w_l.id < w_r.id < v_r.id$. In this case, all nodes in $\{v_l\} \cup W_l$ have to learn about $W_r \cup \{v_r\}$ and vice versa. Since w_l knows $\{v_l\} \cup W_l \cup \{w_r\}$ and w_r knows $\{w_l\} \cup W_r \cup \{v_r\}$, after one round of applying Rule 3a, all nodes in $\{v_l\} \cup W_l$ learn about w_r and all nodes in $W_r \cup \{v_r\}$ learn about w_l. Since the stable neighborhoods of w_l and w_r just got updated, w_l and w_r will trigger another round of "introduce all" by Rule 3a, son nodes in $\{v_l \cup W_l \cup W_r \cup \{v_r\}$ will have updated their neighborhoods by the second round.

The other cases when some of the nodes v_l, v_r, w_l and w_r do not exist are very similar and dropped here. Hence, after at most two rounds, all (stable) neighborhoods have been updated. Since only $O(\log n)$ nodes need to change their neighborhood, and each of these nodes inserts at most $O(\log^2 n)$ edges due to Rule 3a in each round, at most $O(\log^3 n)$ edges are inserted in total. These either merge with stable edges, become a new stable edge or become a temporary edge. Each of the temporary edges will need at most $O(\log n)$ applications of Rule 2 until it merges with a stable edge. Hence, altogether the time is bounded by $O(\log n)$ and the work is bounded by $O(\log^4 n)$. □

THEOREM 4.3. *Assume a new node v joins the system by establishing an edge to a node u which is currently in SKIP$^+$. It will take $O(\log n)$ rounds of the algorithm and $O(\log^4 n)$ total work w.h.p. for the graph to converge back to a valid SKIP$^+$ structure.*

PROOF. Upon learning about node u, node v immediately considers edge (v, u) as stable (since u is currently the only predecessor or successor node v knows). That will prompt the insertion of edge (u, v) by Rule 1a. If u considers (u, v) as a temporary edge, then it forwards the edge via Rule 2 to a node u' with a longer prefix match with v than u. This leads, after at most H steps, to a stable edge (w, v). Till then, v keeps inserting the edge (u, v) in each round (as it considers u to be a stable neighbor), so there will be a string of temporary edges moving upwards from u. Besides Rule 2, no other rule will be applied at this point by the old nodes in SKIP$^+$.

Suppose that w is the first node that considers the edge (w, v) to be stable. Let i be the maximum level such that $pre_i(v) = pre_i(w)$. Let V_l (resp. V_r) be the set of all nodes to the left (resp. right) of v in SKIP$^+$ of maximum cardinality so that for all $w' \in V_l$ (resp. $w' \in V_r$), $pre_{i+1}(w') = pre_{i+1}(v)$ and there is no node w'' in between the nodes of V_l (resp. V_r) with maximum common prefix equal to i with v. Moreover, let W_l (resp. W_r) be the set of all nodes to the left (resp. right) of v of maximum cardinality so that for all $w' \in W_l$ (resp. $w' \in W_r$), the maximum common prefix with v is equal to i and there is no node w'' in between the nodes of W_l (resp. W_r) with maximum common prefix more than i with v. Let v_l, v_r, w_l and w_r be the closest neighbors in V_l, V_r, W_l and W_r to v. Let us assume for now that v_l, v_r, w_l and w_r exist. Recall that w considers v to be a stable neighbor. Suppose w.l.o.g. that v is to the right of w. We distinguish between the following cases.

Case 1: $w_l.id < v_l.id < v_r.id < w_r.id$. In this case, all nodes in $\{w_l\} \cup V_l \cup V_r \cup \{w_r\}$ have to connect to v and vice versa, and besides these, no other connections are needed to fully integrate v into level i. Since $w = w_l$ and w therefore knows all nodes in $V_l \cup V_r \cup \{w_r\}$ by the SKIP$^+$ definition, one round of applying Rule 3a (which is caused by (w, v)) suffices to fully integrate node v into level i.

Case 2: $v_l.id < w_l.id < v_r.id < w_r.id$. In this case, all nodes in $\{v_l\} \cup W_l \cup V_r \cup \{w_r\}$ have to learn about v and vice versa to fully integrate v into level i. Since w is a node in W_l, w has links to all nodes in $\{v_l\} \cup W_l \cup V_r \cup \{w_r\}$, so again one round of applying Rule 3a suffices to fully integrate node v into level i.

Case 3: $w_l.id < v_l.id < w_r.id < v_r.id$. In this case, all nodes in $\{w_l\} \cup V_l \cup W_r \cup \{v_r\}$ have to learn about v and vice versa to fully integrate v into level i. Since $w = w_l$, w has links to all nodes in $V_l \cup \{w_r\}$. Hence, one round of Rule 3a introduces v to the nodes in $V_l \cup \{w_r\}$ and vice versa. Afterwards, w_r will apply Rule 3a since its stable neighborhood changed due to v, so w_r will introduce v to $W_r \cup \{v_r\}$ and vice versa, which completes the integration of v into level i.

Case 4: $v_l.id < w_l.id < w_r.id < v_r.id$. In this case, all nodes in $\{v_l\} \cup W_l \cup W_r \cup \{v_r\}$ have to learn about v and vice versa to fully integrate v into level i. As w is any node in W_l, w knows about $\{v_l\} \cup W_l \cup W_r \cup \{v_r\}$, so one round of applying Rule 3a suffices to fully integrate node v into level i.

The remaining cases in which v_l, v_r or w_r do not exist are similar and dropped here. Hence, it takes at most two rounds to fully integrate v into level i.

Once v is fully integrated into a level i, it knows the closest predecessor and successor w in SKIP$^+$with maximum prefix match at least $i+1$ (if it exists). Since each of these nodes will consider v to be a stable neighbor in level $i+1$, we can use similar case distinctions as above to show that v will be fully integrated into level $i+1$ in at most two further rounds. Node v also knows its closest predecessor and successor w in SKIP$^+$with maximum prefix match at least i. Since each of these nodes will consider v to be a stable neighbor in level $i-1$, we can also use similar case distinctions as above to show that v will be fully integrated into level $i-1$ in at most two further rounds. Using these arguments inductively implies that v will be fully integrated into the SKIP$^+$ graph in $O(\log n)$ time. It remains to bound the work. The first part (creating and forwarding temporary edges),[2] just consumes $O(\log^2 n)$ work. Each time a node destabilizes, $O(\log^2 n)$ new edges are created. Certainly, only

[2]By "first part" we mean the part where v contacts u until a first node is met that considers the edge to v to be stable. The sequence of nodes involved here will continuously forward temporary edges upwards until v ceases to insert the edge (u, v). This happens once v is not a nearest neighbor of u anymore for some level.

nodes that will consider v to be their stable neighbor (and vice versa) will destabilize, and we know from Lemma 4.1 that the degree of v in SKIP$^+$ will be $O(\log n)$ in the end w.h.p. Hence, altogether at most $O(\log^3 n)$ new edges are created. These either merge with stable edges, become a new stable edge or become a temporary edge. Each of the temporary edges will need at most $O(\log n)$ applications of Rule 2 until it merges with a stable edge. This yields the claim. □

5. CONCLUSION

This paper described the first self-stabilizing algorithm that quickly establishes a scalable peer-to-peer topology out of any state in which this is in principle possible. Our work opens many important directions for future research. In particular, so far, we do not have a polylogarithmic bound on the enabled actions per node and round. Hence, we want to explore the corresponding complexities further and come up with the necessary algorithmic modifications.

6. REFERENCES

[1] Y. Afek and S. Dolev. Local stabilizer. *Journal of Parallel and Distributed Computing*, 62(5):745–765, 2002.

[2] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. Fast construction of overlay networks. In *Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2005.

[3] J. Aspnes and G. Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, Nov. 2007.

[4] J. Aspnes and U. Wieder. The expansion and mixing time of skip graphs with applications. In *Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 126–134, 2005.

[5] J. Aspnes and Y. Wu. $O(\log n)$-time overlay network construction from graphs with out-degree 1. In *Proc. International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *LNCS*, pages 286–300, 2007.

[6] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self stabilization by local checking and correction. In *Proc. of the 32nd IEEE Symp. on Foundations of Computer Science (FOCS)*, 1991.

[7] B. Awerbuch and C. Scheideler. Peer-to-peer systems for prefix search. In *Proc. of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

[8] B. Awerbuch and C. Scheideler. The hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. 15th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 318–327, 2004.

[9] B. Awerbuch and G. Varghese. Distributed program checking: A paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 258–267, 1991.

[10] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting. In *Proc. 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 170–179, 2004.

[11] J. Brzezinski and M. Szychowiak. Self-stabilization in distributed systems - a short survey. *Foundations of Computing and Decision Sciences*, 25(1), 2000.

[12] T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list. In *Proc. 10th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2008.

[13] A. Costello and G. Varghese. Self-stabilization by window washing. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 35–44, 1996.

[14] C. Cramer and T. Fuhrmann. Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe, 2005.

[15] E. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

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

[17] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 4:1–40, 1997.

[18] P. Druschel and A. Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[19] D. Gall, R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. Modeling scalability in distributed self-stabilization: The case of graph linearization. Technical Report TUM-I0835, Technische Universität München, Computer Science Dept., Nov. 2008.

[20] M. T. Goodrich, M. J. Nelson, and J. Z. Sun. The rainbow skip graph: A fault-tolerant constant-degree distributed data structure. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 384–393, 2006.

[21] N. Harvey and J. Munro. Deterministic SkipNet. *Inf. Process. Lett.*, 90(4):205–208, 2004.

[22] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 113–126, 2003.

[23] T. Herman. Observations on time adaptive self-stabilization. Technical Report TR 97-07, University of Iowa, 1997.

[24] T. Herman. Self-stabilization bibliography: Access guide. University of Iowa, December 2002. See ftp://ftp.cs.uiowa.edu/pub/selfstab/bibliography/.

[25] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[26] S. Kutten and B. Patt-Shamir. Time-adaptive self stabilization. In *Proc. of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 149–158, 1997.

[27] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 20–27, 1995.

[28] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. 21st PODC*, pages 183–192, 2002.

[29] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65, 2002.

[30] M. Naor and U. Wieder. Novel architectures for P2P applications: the continuous-discrete approach. *ACM Transactions on Algorithms (TALG)*, 3, 2007.

[31] M. Onus, A. Richa, and C. Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proc. 9th ALENEX*, 2007.

[32] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, 2001.

[33] A. Shaker and D. S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Proc. 5th IEEE International Conference on Peer-to-Peer Computing*, pages 39–46, 2005.

[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report

MIT-LCS-TR-819, MIT, 2001.

[35] G. Varghese. *Self-Stabilization by local checking and correction*. PhD thesis, MIT, 1992.

[36] G. Varghese. Self stabilization by counter flushing. In *Proc. of the 13th ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.