MULTISKIPGRAPH: A Self-stabilizing Overlay Network that Maintains Monotonic Searchability*

Linghui Luo, Christian Scheideler, Thim Strothmann
Department of Computer Science
Paderborn University, Germany
{*linghui.luo, scheideler, thim.strothmann*}*@upb.de*

***Abstract*—Self-stabilizing overlay networks have the advantage of being able to recover from illegal states and faults. However, the majority of these networks cannot give any guarantees on their functionality while the recovery process is going on. We are especially interested in *searchability*, i.e., the functionality that search messages for a specific node are answered successfully if a node exists in the network. In this paper we investigate overlay networks that ensure the maintenance of *monotonic* searchability while the self-stabilization is going on. More precisely, once a search message from node u to another node v is successfully delivered, all future search messages from u to v succeed as well. We extend the existing research by focusing on skip graphs and present a solution for two scenarios: (i) the goal topology is a super graph of the perfect skip graph and (ii) the goal topology is exactly the perfect skip graph.**

***Index Terms*—Overlay networks, self-stabilization, search**

I. INTRODUCTION

In this paper, we continue the research started in [1] and investigate protocols for self-stabilizing overlay networks that guarantee the *monotonic* preservation of a characteristic that is called *searchability*. This property captures the idea that once a search message from a node u to another node v is successfully delivered, all future search messages from u to v succeed as well. Searching is not only one of the most fundamental tasks in overlay networks, but our notion of searchability also captures the desired feature that we can successfully route messages to a target node once a single search message has successfully reached the target, i.e., we preserve routing paths while stabilization of the overlay topology is still in progress. Conversely, if a network does not maintain searchability, it cannot maintain simple functionalities while stabilizing messages are not reaching their desired target nodes.

The first results for monotonic searchability focus on specific simple topologies (e.g., the line in [1]). The follow-up paper [2] presents a universal approach that can be used to transform existing self-stabilizing protocols (that fulfill certain requirements) in order to get a protocol that maintains monotonic searchability. The main drawback of this universal approach is the fact that protocols for certain topologies cannot be transformed since they violate one of the requirements needed for the transformation. For example, protocols which use random decisions during topology construction (e.g., the small-world protocol [3]) cannot be transformed by the generic approach. Another wide class of topologies which violate the requirements of the universal approach are graphs that make heavy use of fast routing paths by having shortcuts that change over time in the construction phase, e.g., the chord network [4] or skip graphs [5]. We bridge the latter gap by solving the problem of monotonic searchability for a topology that uses shortcut edges on top of a list in order to achieve a logarithmic diameter: the (perfect) skip graph [6].

We investigate monotonic searchability for the perfect skip graph in two scenarios: (i) classical self-stabilization as introduced by Dijkstra [7] (i.e., the desired final topology has to be the skip graph) and (ii) *relaxed* self-stabilization (i.e., the skip graph has to be a *subtopology* of the final topology). From a self-stabilization point of view the second scenario is easier to achieve than the first, i.e., protocols and their proofs are easier to design. However, we can achieve monotonic searchability in both cases. To the best of our knowledge, even though the notion of relaxed self-stabilization is not new, in general we are the first to exploit this idea in topological self-stabilization. Our protocol for the classical scenario shows that monotonic searchability can be maintained even in cases where the generic approach is not applicable (since its requirements are not fulfilled). Moreover, our protocol for the relaxed scenario shows that the cost of maintaining monotonic searchability can be mitigated by allowing more edges in the final topology. More precisely, the classical scenario incurs a lot of overhead and requires an elaborate and slow procedure to search for nodes, whereas the relaxed scenario achieves searchability more efficiently (in terms of overhead, complexity of the protocols as well as the proofs). Additionally, simulations show that the constructed topology in the relaxed scenario does not generate a lot of *topology overhead*, i.e., the average degree growth is polylogarithmic. Due to space constraints, we focus on our result in the relaxed scenario and describe the changes for the classical scenario. A detailed version of this paper can be found in [8]

A. Model

We model a distributed system as a directed graph $G = (V, E)$. Each peer is represented by a node $v \in V$. Each node $v \in V$ has a unique reference and a unique identifier $v.id \in \mathbb{N}$ (called *ID*). A node v is on the left (right) side of a node u if $v.id < u.id$ ($v.id > u.id$). For two nodes

*This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

u and v we define the *identifier distance* (or short distance) $d(u.id, v.id)$ as the number of nodes in the system whose IDs are in the interval $(u.id, v.id]$ if $u.id < v.id$ (or $(v.id, u.id]$ if $u.id > v.id$). Additionally, each node v maintains local protocol-based variables and has a *channel* $v.ch$, which is a system-based variable that contains incoming messages. The message capacity of a channel is unbounded and messages never get lost. If a node u has the reference of some other node v, u can send a message m to v by putting m into $v.ch$. When a node u processes a message m, then m is removed from $u.ch$. We assume for simplicity that there are no references to non-existing nodes in our system. *Failure detectors* would solve this scenario, but this is out of scope for this paper, since the problem of guaranteeing monotonic searchability is already non-trivial if all references point to existing nodes.

We distinguish between two different types of *actions*: The first type is used for standard procedures and has the form $\langle label \rangle(\langle parameters \rangle) : \langle command \rangle$, where $label$ is the name of that action, $parameters$ defines the set of parameters and $command$ defines the statements that are executed when calling that action. It may be called locally or remotely, i.e., every message that is sent to a node has the form $\langle label \rangle(\langle parameters \rangle)$. The second action type has the form $\langle label \rangle : (\langle guard \rangle) \longrightarrow \langle command \rangle$, where $label$ and $command$ are defined as above and $guard$ is a predicate over local variables. An action for some node u may only be executed if its guard is $true$. An action whose guard is simply $true$ is called TIMEOUT action, which is thus called periodically.

We define the *system state* to be an assignment of values to every node's variables and messages to each channel. A *computation* is an infinite sequence of system states, where the state s_{i+1} can be reached from its previous state s_i by executing an action that is enabled in s_i. We call the first state of a given computation the *initial state*. Given a computation $s_1, s_2, s_3, \ldots$, a *computation suffix* is a subsequence of the computation that is obtained by removing s_1 and finitely many subsequent states. We assume *fair message receipt*, i.e., every message of the form $\langle label \rangle(\langle parameters \rangle)$ that is contained in some channel, is eventually processed. Furthermore, we assume *weakly fair action execution*, meaning that if an action is enabled in all but finitely many states of a computation, then this action is executed infinitely often (the TIMEOUT action as an example for this). We place no bounds on message propagation delay or relative node execution speed, i.e., we allow fully asynchronous computations and non-FIFO message delivery. Our protocol does not manipulate node identifiers and thus only operates on them in *compare-store-send* mode, i.e., we are only allowed to compare node IDs to each other, store them in a node's local memory or send them in a message.

Concerning G, there is a directed edge $(u, v) \in E$, if u stores a reference to v in its local memory or if there is a message in $u.ch$ carrying the reference of v. In the former case, we call that edge *explicit* and in the latter case we call that edge *implicit*. We use $G_e = (V, E_e)$ to denote the subgraph of G that only contains explicit edges. In order for our distributed

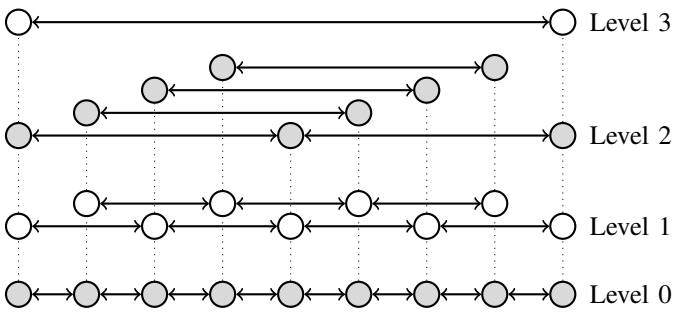


Figure 1. A perfect skip graph with 9 nodes and $maxLevel = 3$

algorithms to work, we require the directed graph $G = (V, E)$ to stay weakly connected throughout a computation. A directed graph $G = (V, E)$ is *weakly connected*, if the undirected version of G, namely $G' = (V, E')$ is connected, i.e., for every two nodes $u, v \in V$ there is a path from u to v in G'. Once there are multiple weakly connected components in G, these components cannot be connected to each other anymore in our scenario [9].

B. Problem Statement

We are interested in the formation and maintenance of a *perfect skip graph* topology for the nodes in the distributed system. A perfect skip graph is a deterministic version of skip graph [6] in which each node has a neighbor on level i if the distance between these two nodes is equal to 2^i. Each node can have at most $\lfloor \log(n-1) \rfloor + 1$ levels (We denote $\lfloor \log(n-1) \rfloor$ with $maxLevel$), where n is the total number of nodes. An example is illustrated in Figure 1.

We say the system is in a *legitimate (stable) state*, if the nodes and the explicit edges form the perfect skip graph and there are no *corrupted messages* in the system. An arbitrary message m is called *corrupted* if the existence of m violates a predefined message invariant (see proof of Theorem 2 for details). Intuitively speaking a message is corrupted, if it violates certain properties that capture the essence of searchability in our topology. A system state s is called *admissible* if there are no corrupted messages in s.

A protocol is *self-stabilizing* if it satisfies the following two properties: (i) **Convergence:** Starting from an arbitrary system state, the protocol is guaranteed to arrive at a legitimate state and (ii) **Closure:** Starting from a legitimate state, the protocol remains in legitimate states thereafter.

Besides classical topological self-stabilization we also investigate a weaker form stabilization that we call *relaxed* self-stabilization. Intuitively, this means that the topology of the network in a legitimate state is allowed to be a supertopology of the desired topology, i.e., in legitimate states G_e contains at least the edges of the perfect skip graph.

An important concept in overlay networks is searching, since nodes have to initiate search requests to interact with each other. A search request can be interpreted as a SEARCH(u, $destID$) message where u is the initiating node

and $destID$ is the ID of the node we are searching for, which will be routed along G_e. A self-stabilizing protocol satisfies *monotonic searchability* according to some search protocol R if it holds for any pair of nodes u and w that once a SEARCH(u, $w.id$) request initiated by u at time t succeeds, any SEARCH(u, $w.id$) requests initiated by u at time $t' > t$ will succeed. A protocol *admissibly satisfies* monotonic searchability, if (i) it satisfies monotonic searchability in computations in which every state is admissible and (ii) starting from any initial state, there is a computation suffix in which every state is admissible. Since all known results in the area consider protocols that admissibly satisfy monotonic searchability, we drop the word *admissibly* to enhance the readability of statements.

C. Related Work

The idea of self-stabilization was introduced by E.W. Dijkstra in 1974 [7], in which he investigated the problem of self-stabilization in a token ring. In order to recover certain network topologies from any weakly connected state, researchers started with simple line and ring networks (e.g., [10], [11]). Over the years more and more topologies were considered, ranging from skip lists and skip graphs [9], [5], to expanders [12], and small-world graphs [3]. Also a universal algorithm for topological self-stabilization is known [13].

In the last 20 years many approaches have been investigated that focus on maintaining safety properties during the convergence phase (of self-stabilization), e.g., snap-stabilization [14], [15], super-stabilization [16], safe convergence [17] and self-stabilization with service guarantee [18]. Closest to our work is the notion of *monotonic convergence* by Yamauchi and Tixeuil [19]. A self-stabilizing protocol is monotonically converging if every change done by a node p makes the system approach a legitimate state and if every node changes its output only once. The authors investigate monotonically converging protocols for different classical distributed problems (e.g., leader election and vertex coloring) and focus on the amount of non-local information that is needed to solve them.

Research on monotonic searchability was initiated in [1], in which the authors proved that it is impossible to satisfy monotonic searchability if corrupted messages are present. In addition, they presented a self-stabilizing protocol for the line that is able to satisfy monotonic searchability. This work is complemented in the subsequent paper of the same authors [2] in which they investigate a universal approach for monotonic searchability. The base for their approach is a set of primitives for manipulating overlay edges that allows maintenance of searchability, a transformation technique such that existing self-stabilizing protocols use these primitives only and a generic routing protocol. However, adapting their protocol to specific topologies comes at the cost of convergence times and additional message overhead. A very recent publication investigates monotonic searchability for high-dimensional networks based on a quad-tree construction [20].

D. Our Contribution

Our major contributions are as follows:

1) We propose a novel self-stabilizing protocol MULTISKIPGRAPH and a corresponding search strategy that greedily makes use of shortcut edges in the topology (see Section II). MULTISKIPGRAPH is a solution for the relaxed self-stabilization problem for the perfect skip graph topology.
2) In addition, we show how to extend the MULTISKIPGRAPH protocol to solve the classic self-stabilization problem for the perfect skip graph topology: the MULTISKIPGRAPH* protocol (see Section IV). To maintain monotonic searchability, we present a new search protocol called SLOWGREEDYSEARCH, which combines a greedy forwarding strategy and a backtracking algorithm. To the best of our knowledge, MULTISKIPGRAPH* is the first self-stabilizing and monotonic searchability satisfying protocol for the perfect skip graph.
3) Finally, we compare our two approaches experimentally in simulations (see Section V).

We do have to note that all present protocols do not consider *node departures* from an overlay network. There is a different line of work that considers this self-stabilizing scenario (e.g., [21], [22]). It was shown in [1] that it is possible to construct a self-stabilizing algorithm that (i) stabilizes to the line topology, (ii) handles node departures while maintaining connectivity and (iii) maintains monotonic searchability. However, even for such a simple topology the protocols are hard to follow (since they aim to achieve a multitude of goals) and do not provide many algorithmic insights (except for the fact that such a combination is indeed possible). Thus, we opted for a version of the problem which puts its focus on the maintenance of searchability and leave the combination with a corresponding node departure protocol for future research.

The rest of the paper is structured as follows: In Section II we present our MULTISKIPGRAPH protocol together with the corresponding search protocol. We prove the corresponding correctness (in terms of self-stabilization and monotonic searchabilty) in Section III. Afterwards we sketch the necessary changes to transform MULTISKIPGRAPH into MULTISKIPGRAPH* in Section IV. We conclude the paper by evaluating both protocols experimentally in Section V.

II. THE MULTISKIPGRAPH PROTOCOL

We now introduce our MULTISKIPGRAPH protocol presented in Algorithm 1 that solves the relaxed self-stabilization and monotonic searchability problems for perfect skip graphs. Since higher levels of a perfect skip graph are built on the top of lower levels, it is natural to stabilize the level-0 list first.

To stabilize the level-0 list, we reuse the classic linearization protocol of [23], in which each node only keeps the reference of a single left and right neighbor. If a node u with a current right neighbor w receives a reference of node v with $u.id < v.id$, node u either saves v as its new right neighbor if v is closer to u than w and delegates w to v, or v is

Variables and Constants

id: the unique identifier of the current node
$self$: the reference of the current node
$maxLevel$: the predefined maximal level of the perfect skip graph
LeftLevel[i]: the left level-i neighbor
RightLevel[i]: the right level-i neighbor
LeftUnknown: the set of left neighbors which are not assigned to any level
RightUnknown: the set of right neighbors which are not assigned to any level
Left: the set of all left neighbors, i.e., the union of *LeftLevel*[i]s and *LeftUnknown*
Right: the set of all right neighbors, i.e., the union of *RightLevel*[i]s and *RightUnknown*
$v.level$: the level of a neighbor v stored by the current node
$Waiting$: the set to store $destID$ of each SEARCH($self, destID$) message initiated by the current node
$WaitingFor[destID]$: the set of all SEARCH($self, destID$) messages initiated by the current node
seq: the sequence number counter for search messages
$seqs[destID]$: it stores the sequence number of the latest initiated SEARCH($self, destID$) messages by the current node

Action TIMEOUT()

```
                    // The self-stabilizing part;
// Let v_1.id < v_2.id < ... < v_n.id;
for i ← 1 to n − 1 do
    for v_i, v_{i+1} ∈ Left: send INTRODUCE(v_i) to v_{i+1};
// Let w_1.id < w_2.id < ... < w_m.id;
for i ← 1 to m − 1 do
    for w_i, w_{i+1} ∈ Right: send INTRODUCE(w_{i+1}) to w_i;
send INTRODUCE(self) to v_n;
send INTRODUCE(self) to w_1;
for i ← 0 to maxLevel − 1 do
    if LeftLevel[i] ≠ ⊥ ∧ RightLevel[i] ≠ ⊥ then
        send INTROLEVELNODE(LeftLevel[i], i + 1) to RightLevel[i];
        send INTROLEVELNODE(RightLevel[i], i + 1) to LeftLevel[i];
                    // The HybridSearch part;
for destID ∈ Waiting do
    send GREEDYPROBE(self, destID, seq) to self;
    send GENERICPROBE(self, destID, {self}, seq) to self;
```

Action INTRODUCE(v)

```
if v.id ≠ id then
    if v.id < id then
        if LeftLevel[0] = ⊥ then
            if v ∈ LeftUnknown then
                LeftUnknown ← LeftUnknown \ {v};
            LeftLevel[0] ← v;
        else
            w ← LeftLevel[0];
            if v.id ≠ w.id then
                if v.id > w.id then
                    LeftUnknown ← LeftUnknown ∪ {w};
                    if v ∈ LeftUnknown then
                        LeftUnknown ← LeftUnknown \ {v};
                    LeftLevel[0] ← v;
                else
                    x ← arg max {u.id | u.id < v.id ∧ u ∈ Left};
                    y ← arg min {u.id | u.id > v.id ∧ u ∈ Left};
                    send INTRODUCE(v) to x if x ≠ ⊥;
                    send INTRODUCE(v) to y if y ≠ ⊥;
    else
        // Analogous to the previous case.
```

Action INTROLEVELNODE(v, i)

```
if v.id ≠ id then
    if i > 0 then
        doIntro ← false;
        if v.id < id then
            if Left ≠ ∅ then
                for j ← 0 to i-1 do
                    if LeftLevel[j] = ⊥ then
                        doIntro ← true;
                        break;
            else
                doIntro ← true;
            if doIntro then
                INTRODUCE(v);
            else
                w ← LeftLevel[i];
                if w ≠ ⊥ ∧ w ≠ v then
                    LeftUnknown ← LeftUnknown ∪ {w};
                if v ∈ Left then
                    if v ∈ LeftUnknown then
                        LeftUnknown ← LeftUnknown \ {v};
                    else
                        LeftLevel[v.level] ← ⊥;
                    LeftLevel[i] ← v;
        else
            // Analogous to the previous case.
    else
        INTRODUCE(v);
```

Action GREEDYPROBE(src, $destID$, seq)

```
if src ∉ Left ∪ Right then
    INTRODUCE(src);
if destID = id then
    send PROBESUCCESS(destID, seq, self) to src;
else
    if destID < id then
        if Left ≠ ∅ then
            v ← arg min {u.id | u ∈ Left ∧ u.id ≥ destID};
            send GREEDYPROBE(src, destID, seq) to v;
    else
        if Right ≠ ∅ then
            v ← arg max {u.id | u ∈ Right ∧ u.id ≤ destID};
            send GREEDYPROBE(src, destID, seq) to v;
```

Action GENERICPROBE(src, $destID$, $Next$, seq)

```
if src ∉ Left ∪ Right then
    INTRODUCE(src);
for ∀w ∈ Next ∧ w ∉ Left ∪ Right do
    INTRODUCE(w);
if destID = id then
    send PROBESUCCESS(destID, seq, self) to src;
else
    Remove ← {w | w ∈ Next ∧ d(w.id, destID) ≥ d(id, destID)};
    Next ← Next \ Remove;
    if destID < id then
        Next ← Next ∪ {w | w ∈ Left ∧ w.id ≥ destID};
    else
        Next ← Next ∪ {w | w ∈ Right ∧ w.id ≤ destID};
    if Next = ∅ then
        send PROBEFAIL(destID, seq) to src;
    else
        v ← arg max {d(u.id, destID) | u ∈ Next};
        send GENERICPROBE(src, destID, Next, seq) to v;
```

Algorithm 1: The MULTISKIPGRAPH protocol

not saved by u but delegated to w. Here, *delegation* means that a node reference is sent in a message to another node and not kept in the local memory afterwards. However, it was proven in [2] that this delegation hinders maintaining monotonic searchability. Thus, in our protocol each node does not remove the references of its neighbors when delegating, but only marks them as *unknown* (i.e., it adds them into the unknown sets *LeftUnknown* or *RightUnknown*). Neighbors of a node u in an unknown set are those, for which u cannot determine which level they belong to in the current state. All local variables and constants are listed in Algorithm 1.

Searching works similarly to [1], whenever a node u wants to initiate a search message, it calls the INITIATENEWSEARCH($destID$) function. In this function, instead of sending a SEARCH(u, $destID$) message m directly, node u stores m into $u.WaitingFor[destID]$ and periodically initiates a probing process for m in the TIMEOUT() action. Node u only sends m when it gets a positive answer to a probe.

The TIMEOUT() action is called periodically and it is divided into a self-stabilizing part and a *HybridSearch* part. To build the level-0 list fast, each node introduces itself to its closest neighbors and its left and right neighbors linearly by sending INTRODUCE messages in the self-stabilizing part. If a node u has a and b as neighbors on level i in a perfect skip graph, then a and b should be neighbors on level $i+1$. Thus, each node sends INTROLEVELNODE messages with the corresponding level number to its left and right neighbors on each level (see Line 8-10). In the HybridSearch part, each node initiates the HybridSearch probing process by sending the messages GREEDYPROBE() and GENERICPROBE() to itself.

We now explain how INTRODUCE and INTROLEVELNODE messages stabilize the perfect skip graph. Consider a node u and a node v with $v.id < u.id$ (the other case is analogous). Whenever node u receives an INTRODUCE(v) message, the action INTRODUCE(v) is triggered. In this action, node u either keeps v locally or introduces it to its neighbors that are closest to v. Node u keeps v locally, if u has no left level-0 neighbor, or it is closer to v than its current left level-0 neighbor w. In the latter case, w is inserted into the unknown set. INTROLEVELNODE(v, i) messages are used to stabilize the higher levels. In action INTROLEVELNODE(v, i), higher level edges will only be created when the lower level edges have already been established (checks in Line 6-9). If all level-j neighbors with $j < i$ exist, the current node u sets v as its new level-i neighbor and marks the old one as unknown if it exists. If $i \leq 0$ (can only happen in the initial state) or some level-j neighbor with $j < i$ does not exist, the action will be handled as the INTRODUCE(v) action.

The GREEDYPROBE() messages are forwarded in a greedy manner among nodes. Consider a node u that receives a GREEDYPROBE(src, $destID$, seq) message with a node reference src, two numbers $destID$ and seq. The corresponding action works as follows: (i) to maintain the weak connectivity to src, u calls INTRODUCE(src) at first and (ii) if u is the target node, i.e., $destID = u.id$, a PROBESUCCESS($destID$, seq, u) message is sent to src as a positive answer. Otherwise, u forwards the GREEDYPROBE(src, $destID$, seq) message to its neighbor that is closest to the target node.

The GENERICPROBE() messages are forwarded in a progressive manner according to the generic search protocol in [2]. Each GENERICPROBE() message has a set of nodes, called $Next$, which contains the nodes this message will visit in the future. Whenever a GENERICPROBE(src, $destID$, $Next$, seq) message is at a node u with $u.id < destID$ (for $u.id > destID$ it is analogous), u first removes itself and nodes with smaller IDs than itself from $Next$. Then it adds all its right neighbors to $Next$ and forwards this message to a node with minimal ID in $Next$. If u is the target node, a PROBESUCCESS($destID$, seq, u) message is sent back to src. If $Next$ is empty, a PROBEFAIL($destID$, seq) message is sent. GENERICPROBE messages are used as a fallback for cases, in which a path from src to u exists, but it cannot be found greedily.

When a node u receives a PROBESUCCESS($destID$, seq, $dest$) message, it checks if seq is at least as big as the locally stored sequence number for $destID$. If so, u sends all SEARCH(u, $destID$) messages which are waiting in the set $WaitingFor[destID]$ to the target node $dest$. Otherwise, it is a positive answer for the batch of already delivered or discarded SEARCH(u, $destID$) messages and u only executes INTRODUCE($dest$) to preserve the weak connectivity. If u receives a PROBEFAIL($destID$, seq) message which indicates the failed probe, it drops all waiting SEARCH(u, $destID$) messages. The pseudocode of INITIATENEWSEARCH, PROBESUCCESS and PROBEFAIL are similar to the ones in [2].

III. PROOFS OF MULTISKIPGRAPH

Theorem 1 *The* MULTISKIPGRAPH *protocol is a relaxed self-stabilizing solution to the perfect skip graph topology.*

The proof consists of Lemma 1, 2, 3 and 4.

Lemma 1 *If a computation of* MULTISKIPGRAPH *starts from a state in which G is weakly connected, then G remains weakly connected in each subsequent state.*

Proof: Consider arbitrary nodes $u, v \in V$ s.t. there is a path from u to v in G. If the path consists of explicit edges only, then it will always exist, since no explicit edge is removed in MULTISKIPGRAPH. If the path contains an implicit edge (a, b), then there is a message in $a.ch$ carrying the reference of b. When a processes the message (regardless of its type), in our protocol a either keeps b locally or introduces it to one of its neighbors c, i.e., the implicit edge (a, b) is either replaced by an explicit edge (a, b) or a path $(a, c), (c, b)$. Thus, the weak connectivity is always preserved. ■

We define the *potential function* of a node u for a level i as $\phi(u, i) := d(u.id, pred(u, i).id) + d(u.id, succ(u, i).id)$, i.e., the identifier distance between the predecessor and successor of node u on level i. If u has no predecessor (or successor) on level i, $d(u.id, pred(u, i).id)$ (or $d(u.id, succ(u, i).id)$) is

replaced by a constant D which is bigger than the maximal distance between two nodes in the line topology. The *level-i subgraph* of a graph $G = (V, E)$ is defined as $G_i := (V, E_i)$, where E_i contains the level-i edges of all nodes. The *potential function* of a level-i subgraph G_i is defined as $\Phi(G_i) = \sum_{u \in V} \phi(u, i)$. According to our protocol $\phi(u, 0)$ never increases for any node u and neither does $\Phi(G_0)$. Obviously, $\Phi(G_0)$ is minimal if the level-0 subgraph G_0 is the line topology. Thus, for Lemma 2 it is sufficient to show that if $\Phi(G_0) \neq \Phi_{min}(G_0)$, $\Phi(G_0)$ will decrease to $\Phi_{min}(G_0)$ (the value of the line topology) in finite time. For convenience, we denote $G^t = (V, E^t)$ as the directed graph at time t.

Lemma 2 *Any computation of* MULTISKIPGRAPH *starting from a state in which G is weakly connected contains a state in which the level-0 subgraph G_0 is the line topology.*

Proof: We prove the statement by contradiction. Assume there is a time t such that for all $t' > t$: $\Phi(G_0^{t'}) = \Phi(G_0^t) \neq \Phi_{min}(G_0)$, i.e., the level-0 subgraph $G_0^t = (V, E_0^t)$ at time t and afterwards is not the line topology. We define a *connected (line) component* $C_i := (V_i, F_i)$, s.t. $V_i \subseteq V$, $F_i \subseteq E_0^t$ and C_i is the line topology over nodes in V_i. Decompose G_0^t into disjoint connected components $C_1, C_2, ..., C_k$, s.t. $\bigcup_{i \in \{1,...,k\}} V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. For $j > i$, the nodes in C_j all have greater IDs than nodes in C_i. Figure 2 illustrates this decomposition. According to Lemma 1 G^t is weakly connected, so there are edges between the connected components. Consider two neighboring components C_i and C_j with the property that $\exists (u, v) \in E^t : u \in V_i, v \in V_j$ or vice-versa and $i < j$. If there are multiple edges with that property, pick edge (u, v) such that $d(u.id, v.id)$ is minimal. W.l.o.g. that $u \in V_i$ and $v \in V_j$. We consider the following two cases of the edge (u, v):

- (u, v) is an explicit edge. According to our protocol node u introduces itself to v periodically in TIMEOUT() by sending INTRODUCE(u) messages. Under the fair message receipt assumption node v will receive the INTRODUCE(u) message in finite time. In the corresponding INTRODUCE action, v will either add u as it's new level-0 neighbor or delegate it to its neighbor which is closer to u. In the former case, $\phi(v, 0)$ decreases. In the latter case, this INTRODUCE(u) message can be delegated further until a node x_m stops delegating it. Consider the delegation path $(x_1, x_2, ..., x_m)$ of this INTRODUCE(u) message with $x_1 = v$, then it must satisfy that (1) $x_1.id > x_2.id > ... > x_m.id > u.id$, (2) $d(x_i.id, x_{i+1}.id) < d(v.id, u.id)$ holds for all $i = 1, ..., m-1$, and (3) this delegation path only contains explicit edges. These properties imply that all nodes on the delegation path must be in C_j, otherwise it violates our assumption that $d(u.id, v.id)$ is minimal among all edges between C_i and C_j. Thus, x_m is in C_j and $d(x_m.id, u.id) < d(v.id, u.id)$. Node x_m must add u as its left level-0 neighbor when it receives the INTRODUCE(u) message and $\phi(x_m, 0)$ decreases. If it is not the case, it means that x_m has a left level-0 neighbor w with $d(x_m.id, w.id) < d(x_m.id, u.id) < d(v.id, u.id)$. Such a node w can not be in C_j, since otherwise the delegation path does not end in x_m. Moreover, w can not be in C_i since it violates the assumption that $d(u.id, v.id)$ is minimal, i.e., w does not exist.
- (u, v) is an implicit edge, i.e., there is a message m in $u.ch$ carrying the reference of v. When u processes m, it will either add v as its right neighbor or treat it as an INTRODUCE(v) message for all possible types of m. In the former case, (u, v) becomes explicit and $\phi(u, 0)$ decreases. In the latter case, when processing INTRODUCE(v), node u either keeps v as it as its neighbor or delegates it. This is analogous to the scenario in which (u, v) is an explicit edge, i.e., a delegation path $(x_1, x_2, ..., x_m)$ exists and $\phi(x_m, 0)$ will decrease.

We have proven that there exists a node y such that $\phi(y, 0)$ decreases. This implies that $\Phi(G_0^{t'})$ will decrease, which is a contradiction to our initial assumption. ■

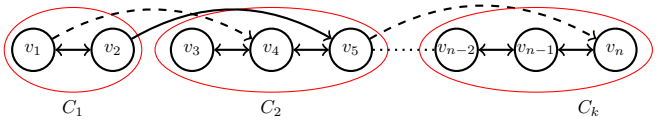


Figure 2. Disjoint connected components on level 0 (dashed edges are implicit edges)

Lemma 3 *If a computation of* MULTISKIPGRAPH *contains a state in which the level-0 subgraph G_0 is the line topology, then G_0remains the line topology.*

Proof: Since G_0 is the line topology, $\Phi(G_0) = \Phi_{min}(G_0)$ holds. According to our protocol $\Phi(G_0)$ can not increase and it can not decrease anymore once it reaches $\Phi_{min}(G_0)$. Thus, it will always stay as the line topology. ■

Lemma 4 *Any computation of* MULTISKIPGRAPH *starting from a state in which G is weakly connected contains a computation suffix in which G_e is a super graph of the perfect skip graph topology.*

Proof: It is sufficient to prove that there is a computation suffix in which the level-i subgraph of G_e is the level-i subgraph of the perfect skip graph, i.e., the level-i subgraph is stable. We prove this by induction.
Basis: The case $i = 0$ is proven in Lemma 2.
Inductive step: $i \rightarrow i + 1$ for $i \in \{0, ..., maxLevel - 1\}$.
Consider the state where level i is stable and an arbitrary node u whose left level-i neighbor is v and right level-i neighbor is w. In TIMEOUT(), node u introduces v and w to each other by sending INTROLEVELNODE(v, $i + 1$) to w and INTROLEVELNODE(w, $i + 1$) to v periodically. The INTROLEVELNODE messages are only sent in TIMEOUT(), i.e., once w and v are stable level-i neighbors of u, only node u (and no other node) sends INTROLEVELNODE(v, $i + 1$)

to w and INTROLEVELNODE(w, $i+1$) to v. Under the fair message receipt assumption, there will be a state s in which all other INTROLEVELNODE messages in $w.ch$ and $v.ch$ for level $i+1$ which are not INTROLEVELNODE(v, $i+1$) and INTROLEVELNODE(w, $i+1$) are processed. Afterwards, node w will only receive INTROLEVELNODE(v, $i+1$) and node v will only get INTROLEVELNODE(w, $i+1$) for level $i+1$. According to our protocol, node w will have v as its stable left level-$(i+1)$ neighbor and v will have w as its stable right level-$(i+1)$ neighbor. Since level-i is stable and $\phi(u,i) = 2^i$, $\phi(u,i+1) = 2 \cdot \phi(u,i) = 2^{i+1}$ must hold. Consider all nodes on level i that have predecessor and successor like u, their level-i neighbors will become stable level-$i+1$ neighbors analogously. In such state, the identifier distance between neighboring nodes on level $i+1$ is 2^{i+1}, which satisfies the property of the level-$i+1$ subgraph in the perfect skip graph. ■

Theorem 2 *The* MULTISKIPGRAPH *protocol satisfies monotonic searchability according to HybridSearch.*

The proof of Theorem 2 consists of Lemma 6, 7, 8 and 9. Due to space constraints, omitted proofs can be found in [8]. We define the *reachable set* of a node u towards a target node w with $w.id = destID$ as $R(u, destID) := \{u\} \cup \{v \in V |$There is a directed path P_v in G_e from node u to node v s.t. for each explicit edge (a,b) in P_v it holds that $d(a.id, distID) > d(b.id, distID)$.$\}$. The *reachable set* of a set U towards a target node w with $w.id = destID$ is defined as $R(U, destID) := \cup_{u \in U} R(u, destID)$. Since no explicit edges are removed in MULTISKIPGRAPH, i.e., the reachability between every two nodes is always preserved, the following lemma holds.

Lemma 5 *For arbitrary nodes u and v, if $v \in R(u, destID)$ in state s, then $v \in R(u, destID)$ holds in every state $s' > s$.*

We know that adding edges will not violate the monotonic searchability and explicit edges are never removed in MULTISKIPGRAPH, thus it is sufficient to consider only the messages used in the HybridSearch part when checking if a state is admissible. We define a system state as admissible if the following message invariants for HybridSearch hold:

1. If there is a GREEDYPROBE(src, $destID$, seq) in $u.ch$, then $u \in R(src, destID)$.
2. If there is a GENERICPROBE(src, $destID$, $Next$, seq) in $u.ch$, then
   a. $u \in Next$ and $\forall v \in Next \setminus \{u\} : d(v.id, destID) \leq d(u.id, destID)$;
   b. $R(Next, destID) \subseteq R(src, destID)$;
   c. If a node w exists with $w.id = destID$ and $w \notin R(Next, destID)$, then for every admissible state with $src.seqs[destID] < seq$ it holds that $w \notin R(src, destID)$.
3. If there is a PROBESUCCESS($destID$, seq, $dest$) in $u.ch$, then $dest.id = destID$ and $dest \in R(u, destID)$.
4. If there is a PROBEFAIL($destID$, seq) in $u.ch$ and a node w with $w.id = destID$ exists, then $w \notin R(u, destID)$ holds for every admissible state with $u.seqs[destID] < seq$.
5. If there is a SEARCH(v, $destID$) in $u.ch$, then $u.id = destID$ and $u \in R(v, destID)$.

Lemma 6 *If a computation of* MULTISKIPGRAPH *contains an admissible state, then every subsequent state is admissible.*

Lemma 7 *In every computation of* MULTISKIPGRAPH*, there is an admissible state and after that all states are admissible.*

Lemma 8 *If there is a* GENERICPROBE(src, $destID$, $Next$, seq) *message in $u.ch$ with $u.id < destID$ and there exists a node w with $w.id = destID$ and $w \in R(u, destID)$, then a* GENERICPROBE(src, $destID$, $Next'$, seq) *message will be in $w.ch$ eventually.*

Lemma 9 *The* MULTISKIPGRAPH *protocol guarantees monotonic searchability according to HybridSearch in every computation suffix starting in an admissible state.*

Proof: We prove this lemma by contradiction. Consider two SEARCH(u, $destID$) messages m and m' created in admissible states at time t and t' with $t < t'$ s.t. m is delivered successfully but m' is not. Let seq_1, seq_2 be the sequence numbers for m and m'. The sequence number increases monotonically. Let w be the target node with $w.id = destID$.

If m' is created when m is still in $u.WaitingFor(destID)$, then the protocol will handle both messages the same since they belong to the same batch, i.e., m' will be delivered successfully as well. The assumption is violated.

If m' is created when m is already sent by node u, then $seq_2 \geq seq_1$. Since m' is not delivered successfully, there are two possibilities: (1) u receives a PROBEFAIL($destID$, seq) with $seq \geq u.seqs[destID] \geq seq_2$ or (2) u receives no PROBESUCCESS($destID$, seq, w) message with $seq \geq u.seqs[destID]$.

Case (1): The invariant of PROBEFAIL($destID$, seq) holds for every admissible state with $u.seqs[destID] < seq$, including the state when m is delivered where the sequence number is $seq_1 \leq seq_2$, that is $w \notin R(u, destID)$. This is contradictory to the invariant of m which is $w \in R(u, destID)$.

Case (2): m is delivered successfully and invariant of m holds, thus $w \in R(u, destID)$. u sends GENERICPROBE(u, $destID$, $\{u\}$, seq) to itself with $seq \geq u.seqs[destID]$ periodically for m' in TIMEOUT(). According to Lemma 8, a GENERICPROBE(u, $destID$, $Next'$, seq) message will eventually arrive at w. When this happens, w sends a PROBESUCCESS($destID$, seq, w) message to u. Node u will receive this message and (2) is violated. ■

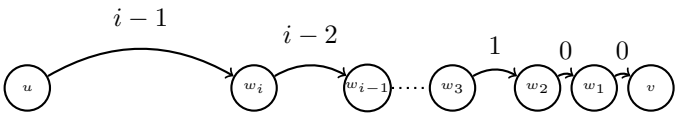


Figure 3. A deterministic search path

IV. The MultiSkipGraph* Protocol

We now introduce the MultiSkipGraph* protocol which stabilizes the system to the perfect skip graph topology. Due to space constraints, we describe only the main differences to MultiSkipGraph. More details can be found in [8].

In MultiSkipGraph* every node safely delegates neighbors which are in the unknown sets similarly as the Build-List+ protocol in [1]. In Timeout(), a node u safely delegates its neighbor $v \in$ *RightUnknown* (or *LeftUnknown*) by sending SafeIntroduce(v, u) to a neighbor w which is closest to v. When w receives SafeIntroduce(v, u), it will add v to w.*RightUnknown* if it did not know v before. Afterwards, node w sends a SafeDeletion(v) message back to node u. Node u removes v from its neighborhood only when it receives a SafeDeletion(v) message. This way, the explicit edge (u, v) is replaced by the explicit edges (u, w) and (w, v) so that node v is always reachable from node u.

Additionally, whenever a node u receives a IntroLevelNode(v, i) message, it does not add v immediately as its level-i neighbor, but first initiates a probing process. A ProbeLevelNode() message will be forwarded to u's neighbor w_i on level $i-1$, then w_i's neighbor w_{i-1} on level $i-2$, ..., w_3's neighbor w_2 on level 1, w_2's neighbor w_1 on level 0, and w_1's neighbor w_0 on level 0. We call this forwarding path the corresponding *deterministic search path* (see Figure 3). In a perfect skip graph, if u and v are level-i neighbors, the deterministic search path between them must exist and this message will eventually arrive at v (i.e., $w_0 = v$). Node u adds v as its level-i neighbor only when such a path exists. This reduces the creation of illegitimate edges in the topology.

Finally, we use a search protocol called SlowGreedySearch. SlowGreedySearch tries to skip nodes as much as possible similar to the greedy search protocol, but still keeps nodes discovered in the probing process in the set $Next$. Consider a node u with right neighbors $v_1, ..., v_n$ with ascending IDs. The path from u to a target node w with $w.id > u.id$ has to use a node in $v_1, ..., v_n$. The SlowGreedySearch protocol first forwards the probe message SlowGreedyProbe (see the corresponding action in Algorithm 2) along the path via node v_n but keeps other nodes $v_1, ..., v_{n-1}$ in memory (in the set $Next$). If there is no path that leads from v_n to the target node w, then the protocol tries the next farthest node v_{n-1}.

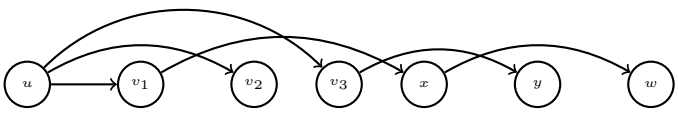


Figure 4. An example ($u.id < v_1.id < ... < y.id < w.id$)

Action SlowGreedyProbe(src, $destID$, $Prev$, $Next$, seq)

```
1  if src ∉ Left ∪ Right then
2  |  Introduce(src);
3  for ∀w ∈ Prev ∪ Next ∧ w ∉ Left ∪ Right do
4  |  Introduce(w);
5  if destID == id then
6  |  send ProbeSuccess(destID, seq, self) to src;
7  else
8  |  if destID < id then
9  |  |  N ← {w ∈ Left | w.id ≥ destID ∧ w ∉ Prev};
10 |  else
11 |  |  N ← {w ∈ Right | w.id ≤ destID ∧ w ∉ Prev};
12 |  Next ← Next ∪ N \ {self};
13 |  Prev ← Prev ∪ {self};
14 |  if Next == ∅ then
15 |  |  send ProbeFail(destID, seq) to src;
16 |  else
17 |  |  v ← arg min {d(u.id, destID) | u ∈ Next};
18 |  |  send SlowGreedyProbe(src, destID, Prev, Next,
   |  |    seq) to v;
```

Algorithm 2: The SlowGreedyProbe action

If this also fails, it tries v_{n-2} and so on until $Next = \emptyset$. By using this backtracking approach, a path to w will be found if it exists. To avoid a ping-pong effect which may cause an infinite loop, the protocol use a set $Prev$ to keep track of all visited nodes. Only unvisited nodes will be inserted into the set $Next$. For example, the forwarding path for a SlowGreedyProbe message from node u to the target node w in Figure 4 should be $u \to v_3 \to y \to v_2 \to v_1 \to x \to w$, in which the message is forwarded backwards twice (i.e., $y \to v_2$ and $v_2 \to v_1$).

V. Evaluation

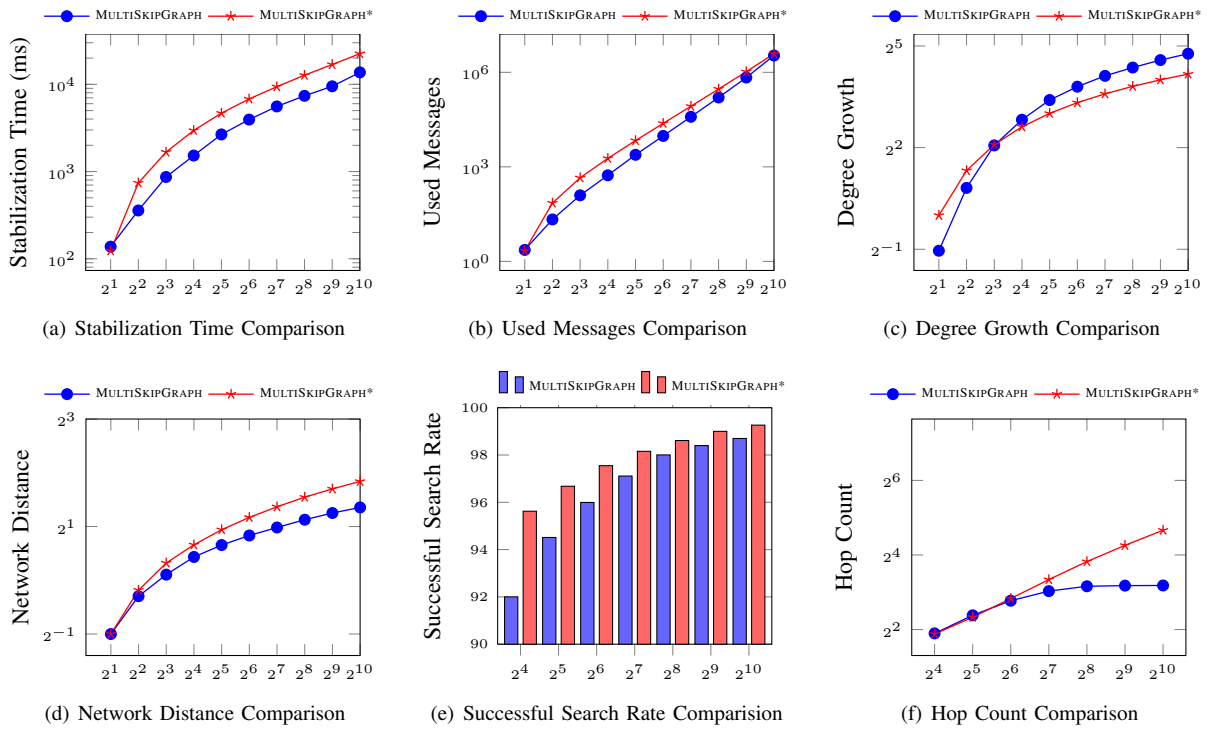
A. Experimental Design

To compare MultiSkipGraph and MultiSkipGraph*, we implemented a simulator[1] in Java which can simulate self-stabilizing overlay networks. In the following we introduce the design decisions we made for the simulation.

a) Asynchronous System: we simulate an asynchronous system by using the multi-threading mechanism in Java, i.e., each node is a thread which runs the self-stabilizing protocol locally. Moreover, message delivery is not in FIFO. Whenever a message m is created in the simulation, a transmission delay t (smaller than a predefined maximum value) is randomly generated, and m will be received by its target node after time t.

b) Initial Graphs: to compare the two protocols, both protocols have to operate on the same initial graphs. We use scale-free graphs as initial graphs for our experiments, because networks in the real world usually self-organize into scale-free graphs – a subset of power law graphs [24], [25], [26], [27]. To generate the scale-free graphs we chose the Barabási-Albert model [24]. Self-stabilization usually requires that an initial graph is weakly connected, which is satisfied by scale-free graphs generated from the Barabási-Albert model. Once a scale-free graph is generated, each edge is randomly assigned to be explicit or implicit.

[1] Available on https://github.com/linghui2016/MultiSkipGraph.



(a) Stabilization Time Comparison (b) Used Messages Comparison (c) Degree Growth Comparison

(d) Network Distance Comparison (e) Successful Search Rate Comparision (f) Hop Count Comparison

Figure 5. Comparisons between MULTISKIPGRAPH and MULTISKIPGRAPH*

c) Termination: The simulations terminate whenever the system achieves the desired topology. For the MULTISKIPGRAPH protocol the desired topology is any graph that contains the perfect skip graph as a subgraph and for the MULTISKIPGRAPH* protocol exactly the perfect skip graph. Because each node has only a local view, a central controller is used to check if the system is stabilized every 200 ms.

d) Metrics: In the following we introduce the metrics we measured in the simulation:

- **Stabilization Time**: The duration of the self-stabilization process starting from an initial graph.
- **Used Messages**: The number of messages used in the self-stabilization process starting from an initial graph.
- **Degree Growth**: The average difference of node degree in the convergent graph and the initial graph. The node degree is defined as the number of explicit edges starting from this node.
- **Network Distance**: The average length of the shortest path between two nodes in the network.
- **Successful Search Rate**: The number of successful search requests divided by the total number of search requests during the self-stabilization process.
- **Hop Count**: The number of intermediate nodes a probe message for searching passes between source and target.

e) Configuration: We conducted scalability experiments with network sizes (i.e., number of nodes) from $2^1(2)$ to $2^{10}(1024)$ increasing by powers of 2. Unfortunately, our implementation of the simulator did not scale well for larger networks. For instance, during the simulation of network size 2^{11} the memory consumption and CPU utilization of the testing computer were already close to 100%.

Given a network size n, the simulator generates a weakly connected scale-free graph with parameter 2 (this is the maximum number of edges to be added in each step according to the Barabási-Albert model) so that the average degree of the generated graph is not bigger than 2. With this configuration we wanted to see how efficient the protocols are stabilizing their respective topologies if the initial graph is low-connected. For every network size we conducted 100 experiments and in each experiment the simulator generates a new initial graph and executes both protocols on this same graph one after another: The measured values are an average value of the 100 experiments for each network size.

To evaluate how efficient the search algorithms perform, we simulated a scenario that randomly generated batches of search requests from time to time during the self-stabilization process. After some exploration tests we chose 10 searches per 100 ms for our experiments.

The experiments were done on a standard computer with a six-core processor (3.30 GHz) and 8 GB RAM.

B. Evaluation Results

a) Comparison in Stabilization: The log-log plots in Figure 5 (a)-(d) illustrate the results from experiments without generating search requests. Figure 5 (a) shows the comparison of stabilization time between the two protocols. Both curves show asymptotically similar results: the greater the network size is, the longer time required for self-stabilization. MULTISKIPGRAPH* generally requires more time for stabilization

than MULTISKIPGRAPH and both curves show a polylogarithmic tendency with growing network size. Similar to the stabilization time, MULTISKIPGRAPH shows a advantage (i.e., fewer messages) over MULTISKIPGRAPH* as show in Figure 5 (b). However, both curves behave almost linear.

Figure 5 (c) shows the comparison in degree growth between the two protocols. The average degree growth of MULTISKIPGRAPH is up to two times larger as of MULTISKIPGRAPH* due to the fact that it never removes edges, which means more local storage for the edges is required in the convergent state. However, these extra of MULTISKIPGRAPH can be beneficial for searching. As shown in Figure 5 (d), the MULTISKIPGRAPH has shorter network distances than MULTISKIPGRAPH*, which is an indicator for shorter search paths in the topology.

Consequently, MULTISKIPGRAPH outperforms MULTISKIPGRAPH* in terms of stabilization time, used messages and network distances, which is traded off by a higher local memory overhead. We believe that MULTISKIPGRAPH may have more potential in real-world distributed systems. For instance, it may bring the system to a convergent state even much faster than MULTISKIPGRAPH*, since the transmission time for probing the deterministic search path in MULTISKIPGRAPH* can be more costly for larger network sizes and since the computation usually starts from a graph in which most parts are already stabilized.

b) Comparison in Searchability: Figure 5 (e) and (f) are results from experiments with search requests during the self-stabilization process. Figure 5 (e) shows the comparison in successful searches between MULTISKIPGRAPH with its search protocol HybridSearch and MULTISKIPGRAPH* with SlowGreedySearch. Both protocols prove to be efficient in our experiments with high successful search rates ($\geq 92\%$) for all network sizes. MULTISKIPGRAPH* with SlowGreedySearch shows a better performance than HybridSearch. However, the difference decreases with increasing network size.

Figure 5 (f) shows the average hop count for successful search requests. In contrast to the successful search rate, MULTISKIPGRAPH* performs worse than MULTISKIPGRAPH since each search request requires more hops on average. While the MULTISKIPGRAPH protocol requires a logarithmic number of hops on average in each probing process for search requests, the curve of MULTISKIPGRAPH* appears to be linear.

These results show a trade-off between the two protocols. If one wants to optimize the number of successfully delivered search requests, MULTISKIPGRAPH* is a preferable choice. However, if one desires shorter routing path, MULTISKIPGRAPH should be chosen.

REFERENCES

[1] C. Scheideler, A. Setzer, and T. Strothmann, "Towards establishing monotonic searchability in self-stabilizing data structures," in *19th International Conference on Principles of Distributed Systems*, 2015.

[2] ——, "Towards a universal approach for monotonic searchability in self-stabilizing overlay networks," in *30th International Symposium on Distributed Computing*, 2016, pp. 71–84.

[3] S. Kniesburges, A. Koutsopoulos, and C. Scheideler, "A self-stabilization process for small-world networks," in *26th IEEE International Parallel and Distributed Processing Symposium*, 2012, pp. 1261–1271.

[4] ——, "Re-chord: A self-stabilizing chord overlay network," *Theory of Computing Systems*, vol. 55, pp. 591–612, 2014.

[5] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig, "Skip$^+$: A self-stabilizing skip graph," *Journal of the ACM*, vol. 61, pp. 36:1–36:26, 2014.

[6] J. Aspnes and G. Shah, "Skip graphs," *ACM Transactions on Algorithms*, vol. 3, 2007.

[7] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, pp. 643–644, 1974.

[8] L. Luo, C. Scheideler, and T. Strothmann, "MultiSkipGraph: A Self-stabilizing Overlay Network that Maintains Monotonic Searchability." [Online]. Available: https://github.com/linghui2016/MultiSkipGraph/blob/master/paper/multiSkipGraph_verbose.pdf

[9] R. M. Nor, M. Nesterenko, and C. Scheideler, "Corona: A stabilizing deterministic message-passing skip list," *Theoretical Computer Science*, vol. 512, pp. 119–129, 2013.

[10] D. Gall, R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig, "A note on the parallel runtime of self-stabilizing graph linearization," *Theory of Computing Systems*, vol. 55, pp. 110–135, 2014.

[11] A. Shaker and D. S. Reeves, "Self-stabilizing structured ring topology p2p systems," in *5th IEEE International Conference on Peer-to-Peer Computing*, 2005, pp. 39–46.

[12] S. Dolev and N. Tzachar, "Spanders: Distributed spanning expanders," *Science of Computer Programming*, vol. 78, pp. 544–555, 2013.

[13] A. Berns, S. Ghosh, and S. V. Pemmaraju, "Building self-stabilizing overlay networks with the transitive closure framework," *Theoretical Computer Science*, vol. 512, pp. 2–14, 2013.

[14] A. Bui, A. K. Datta, F. Petit, and V. Villain, "Snap-stabilization and pif in tree networks," *Distributed Computing*, vol. 20, pp. 3–19, 2007.

[15] S. Delaët, S. Devismes, M. Nesterenko, and S. Tixeuil, "Snap-stabilization in message-passing systems," *Journal of Parallel and Distributed Computing*, vol. 70, pp. 1220–1230, 2010.

[16] S. Dolev and T. Herman, "Superstabilizing protocols for dynamic distributed systems," *Chicago Journal of Theoretical Computer Science*, vol. 1997, 1997.

[17] H. Kakugawa and T. Masuzawa, "A self-stabilizing minimal dominating set algorithm with safe convergence," in *20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[18] C. Johnen and F. Mekhaldi, "Robust self-stabilizing construction of bounded size weight-based clusters," in *16th International Euro-Par Conference*, 2010, pp. 535–546.

[19] Y. Yamauchi and S. Tixeuil, "Monotonic stabilization," in *14th International Conference on Principles of Distributed Systems*, 2010, pp. 475–490.

[20] M. Feldmann, C. Kolb, and C. Scheideler, "Self-stabilizing overlays for high-dimensional monotonic searchability," in *20th International Symposium Stabilization, Safety, and Security of Distributed Systems SSS 2018*, 2018, pp. 16–31.

[21] A. Koutsopoulos, C. Scheideler, and T. Strothmann, "Towards a universal approach for the finite departure problem in overlay networks," in *17th International Symposium Stabilization, Safety, and Security of Distributed Systems*, 2015, pp. 201–216.

[22] D. Foreback, M. Nesterenko, and S. Tixeuil, "Infinite unlimited churn (short paper)," in *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2016, pp. 148–153.

[23] M. Onus, A. W. Richa, and C. Scheideler, "Linearization: Locally self-stabilizing sorting in graphs," in *9th Workshop on Algorithm Engineering and Experiments*, 2007.

[24] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, pp. 509–512, 1999.

[25] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999, pp. 251–262.

[26] A. Clauset, C. Shalizi, and M. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.

[27] M. Ripeanu and I. T. Foster, "Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01, 2002, pp. 85–93.