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 its 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[1], 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. Conversly, if a network does not maintain searchability, it cannot maintain simple functionalities while stabilizing messages are not reaching their desired target nodes. The first result 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 of [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 exemplifies 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 in Section IV.

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

[1]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. One could use *failure detectors* to solve this scenario, but this is not within the scope of 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.

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 of 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 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 (see [8]).

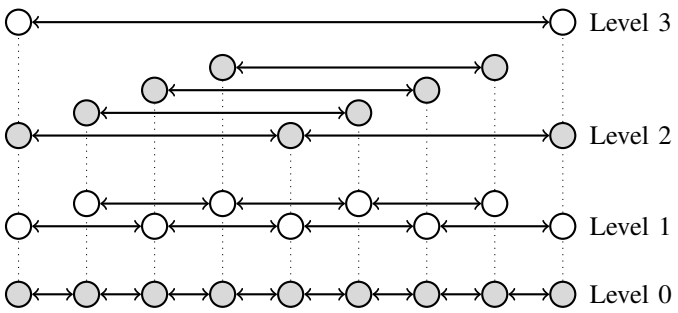


Figure 1. A perfect skip graph with 9 nodes and $maxLevel = 3$

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 [9] 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 in a system is called *corrupted* if the existence of m violates a predefined message invariant (see proof of Theorem 2). 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 in order 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 routing 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 [8], [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 theses 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 makes greedily 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). In order 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 to 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 are not considering *node departures* from an overlay network. There is a different line of work that considers this self-stabilizing scenario (see e.g., [21], [22]). It was shown in [1] that it possible to construct a self-stabilizing algorithm that a) stabilizes to the line topology, b) handles node departures while maintaining connectivity and c) maintains monotonic searchability. However, even for such a simple topology the protocols are hard to follow (due to the bloated nature of the problem) 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 changes necessary in order to transform MULTISKIPGRAPH into MULTISKIPGRAPH*. 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 not saved by u but delegated to w. Here, *delegation* means that a node reference is sent in a message to another node

Variables and Constants

id: the unique identifier of the current node
$self$: the reference of the current node
$maxLevel$: the pre-defined 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

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. 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. 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 it's 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 it's 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: (1) to maintain the weak connectivity to src, u calls INTRODUCE(src) at first and (2) 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] and can be found in Appendix A.

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 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. ■

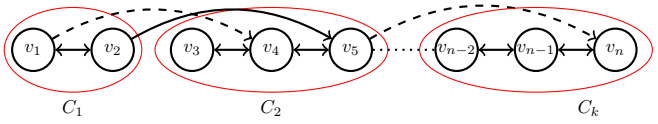


Figure 2. Disjoint connected components on level 0 (dashed edges are implicit edges)

Lemma 3 *If a computation of* MULTISKIPGRAPH *contains a state in which the level-0 subgraph G_0 is the line topology, then G_0 stays being 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 and 9. 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 won't 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.*

See proof in Appendix B.

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

See proof in Appendix B.

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.*

See proof in Appendix B.

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. ■

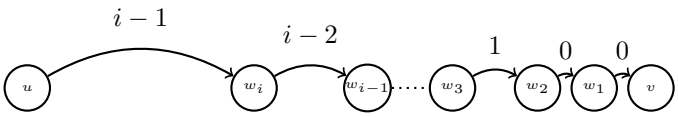


Figure 3. A deterministic search path

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 are in Appendix C and D.

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 doesn't 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 doesn't 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}. 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 $v_2 \to v_1$).

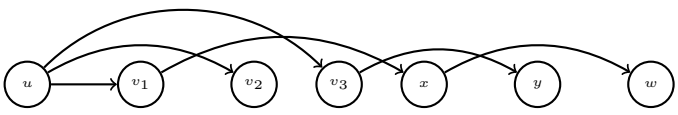


Figure 4. An example ($u.id < v_1.id < ... < y.id < w.id$)

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 of MultiSkipGraph*

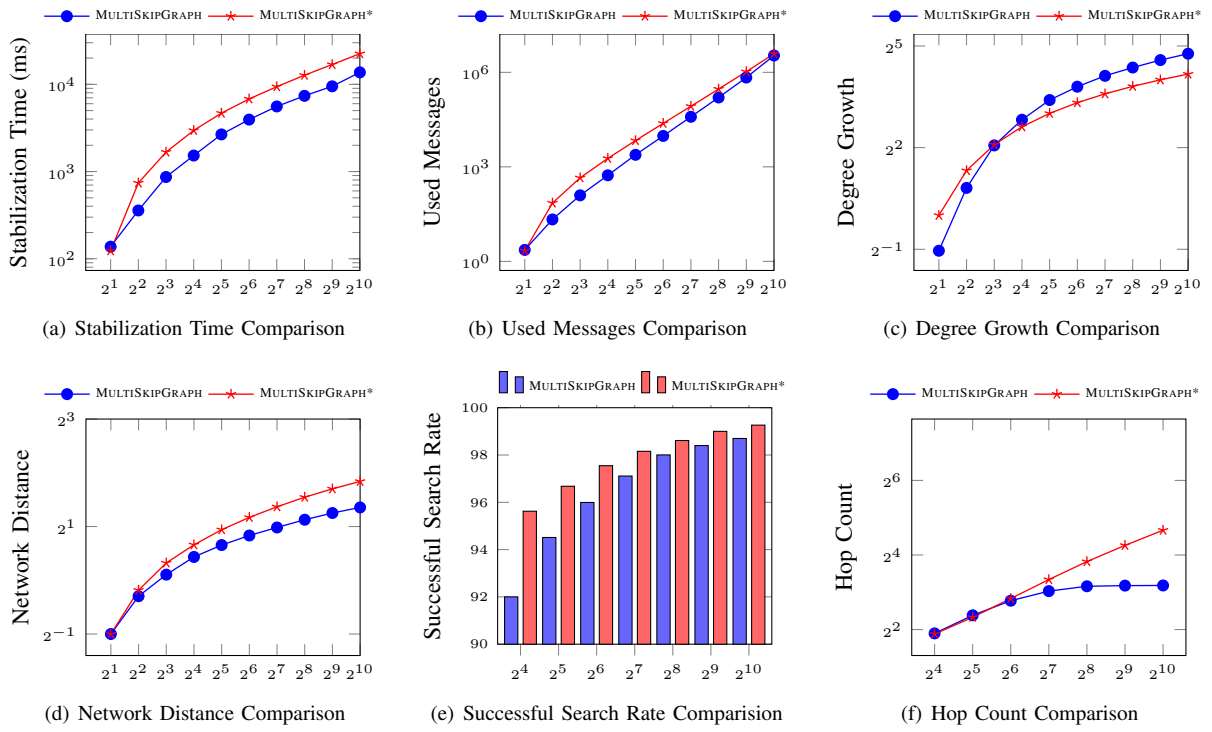
V. Evaluation

A. Experimental Design

To compare MultiSkipGraph and MultiSkipGraph*, we implemented a simulator 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 pre-defined 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.



(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 terminates 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 for 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 a source node and a target node.

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 initail 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 bacthes 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 slight advantage (i.e., less messages) over MULTISKIPGRAPH* as show in Figure 5 (b). However, the difference between the two curves is very small and both curves behave sublinear.

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 the most parts are already stabilized.

b) Comparison in Searchability: Figure 5 (d)-(f) are results from experiments with search requests during the self-stabilization process. Figure 5 (d) shows the comparison in successful searches between MULTISKIPGRAPH with its search algorithm 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 HYBRIDSEARC. However, the difference decreases with increasing network size.

Figure 5 (f) show 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," in *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003, pp. 384–393.

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

[8] 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.

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

[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 *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Proceedings*, 2018, p. to appear.

[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 *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, 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*, vol. 286, no. 5439, 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, ser. SIGCOMM '99. New York, NY, USA: ACM, 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. London, UK, UK: Springer-Verlag, 2002, pp. 85–93.

APPENDIX

A. Additional Pseudocode of MULTISKIPGRAPH

Function INITIATENEWSEARCH($destID$)
Create new message $m = \text{SEARCH}(self, destID)$;
if $WaitingFor[destID] = \emptyset$ **then**
  $seq \leftarrow seq + 1$;
  $seqs[destID] \leftarrow seq$;
$WaitingFor[destID] \leftarrow WaitingFor[destID] \cup \{m\}$;
if $destID \notin Waiting$ **then**
  $Waiting \leftarrow Waiting \cup \{destID\}$;

Action PROBESUCCESS($destID$, seq, $dest$)
if $seq \geq seqs[destID]$ **then**
  send all $m \in WaitingFor[destID]$ to $dest$;
  $WaitingFor[destID] \leftarrow \emptyset$;
  $Waiting \leftarrow Waiting \setminus \{destID\}$;
INTRODUCE($dest$);

Action PROBEFAIL($destID$, seq)
if $seq \geq seqs[destID]$ **then**
  $WaitingFor[destID] \leftarrow \emptyset$;
  $Waiting \leftarrow Waiting \setminus \{destID\}$;

B. Additional Proofs of MULTISKIPGRAPH

Proof of Lemma 6.

Proof: We prove this lemma by contradiction. Assume there is an admissible state s_1 in which all HybridSearch message invariants hold and in the subsequent state s_2 in which one of the invariants doesn't hold. This can only happen when a new message used in HybridSearch is sent in state s_1. Consider the case that $src.id < destID$ and the other case is analog, we will make a case distinction over five possibilities:

Invariant 1 is violated in state s_2.

Assume a node x sends a GREEDYPROBE() message to a node u in state s_1. This can happen in two cases: x sends GREEDYPROBE(x, $destID$, seq) to itself in action TIMEOUT() or x receives a GREEDYPROBE(src, $destID$, seq) message and sends a GREEDYPROBE(src, $destID$, seq) message in the corresponding action.

(i) x sends a GREEDYPROBE(x, $destID$, seq) message in TIMEOUT() to itself in state s_1. Because $x \in R(x, destID)$ holds for every x in every state according to the definition of the reachable set of node x and this is the invariant of this message, the invariant of this message holds in state s_2.

(ii) x sends a GREEDYPROBE(src, $destID$, seq) message m' to a node u in state s_1. According to the GREEDYSEARCH() action node x does that when it receives a GREEDYPROBE(src, $destID$, seq) message m in state s_1 and $x.id \neq destID$. Moreover, node u is a right neighbor of x with $u.id \leq destID$ according to the action and the assumption $src.id < destID$. State s_1 is admissible, thus the invariant of m must hold and that is $x \in R(src, destID)$. $u \in x.Right$, $u.id \leq destID$ and $x \in R(src, destID)$ imply that $u \in R(src, destID)$ should hold. Thus, the invariant of m' must hold in state s_2.

Invariant 2 is violated in state s_2.

Assume a node x sends a GENERICPROBE() message to a node u in state s_1. This can happen in two cases: x sends a GENERICPROBE(x, $destID$, $Next$, seq) message to itself in action TIMEOUT() or x receives a GENERICPROBE(src, $destID$, $Next$, seq) message and sends a GENERICPROBE(src, $destID$, $Next'$, seq) message in the corresponding action.

(i) x sends a GENERICPROBE(x, $destID$, $Next$, seq) message m in TIMEOUT() to itself in state s_1 and $Next = \{x\}$.
- Invariant 2a holds in state s_2 since $u = x$ and $Next = \{x\}$;
- Invariant 2b holds in state s_2 since $src = x$ and $Next = \{x\}$;
- Invariant 2c holds in state s_2 and the proof is as follows:
Assume the invariant 2c doesn't hold in state s_2, then there exists a node w with $w.id = destID$ and $w \notin R(Next, destID) = R(\{x\}, destID)$ and there is an admissible state s_0 with $x.seqs[destID] < seq$ and $w \in R(x, destID)$.
Because $x.seqs[destID]$ increases monotonically and $x.seqs[destID] \geq seq$ in state s_1, so s_0 must be a earlier state than s_1. Since $w \in R(x, destID)$ is true in state s_0 and MULTISKIPGRAPH removes no edges in any state, $w \in R(x, destID)$ must be true in all states latter than s_0, including state s_2. This is contradictory to the assumption that $w \notin R(\{x\}, destID)$ in state s_2. Therefore, the invariant 2c must hold in state s_2.

(ii) x sends a GENERICPROBE(src, $destID$, $Next'$, seq) message m' to a node u in state s_1 by receiving a GENERICPROBE(src, $destID$, $Next$, seq) message m.
- Invariant 2a holds in state s_2 and the proof is as follows:
According to the assumption $src.id < destID$ only right neighbors of node x which are in set $A := \{v | v \in x.Right \wedge v.id \leq destID\}$ are added to $Next$ in the GENERICPROBE() action, thus $Next' = Next \cup A \setminus \{x\}$. $u \in Next'$ holds since x only sends m' to a node in $Next'$. $\forall v \in Next'$ it holds that $d(v.id, destID) \leq d(u.id, destID)$, since u is a node in $Next'$ which has maximal distance to the target node.
Thus, $\forall v \in Next' \setminus \{u\} : d(v.id, destID) \leq d(u.id, destID)$.
- Invariant 2b holds in state s_2 and the proof is as follows:
The invariants 2a and 2b hold for message m received by x, so $x \in Next$ and $R(Next, destID) \subseteq R(src, destID)$ hold. $R(A, destID) \subseteq R(x, destID) \subseteq R(src, destID)$ holds since $A \subseteq x.Right$.

$R(A, destID) \subseteq R(src, destID)$, $R(Next, destID) \subseteq R(src, destID)$ and $Next' = Next \cup A \setminus \{x\}$ imply $R(Next', destID) \subseteq R(src, destID)$.
- Invariant 2c holds in state s_2 and the proof is as follows:
Assume the invariant 2c doesn't hold, then there exists a node w with $w.id = destID$ and $w \notin R(Next', destID)$ in state s_2 and there is a previous admissible state s_0 with $src.seqs[destID] < seq$ and $w \in R(src, destID)$.
Obviously $w.id \neq x.id$, otherwise x sends a PROBESUCCESS() message rather than a GENERICSEARCH() message.
$w \notin R(Next', destID)$ in state s_2 implies $w \notin R(Next, destID)$ in state s_1, since x is the only node in $R(Next, destID)$ but not in $R(Next', destID)$ and $x.id \neq w.id$.
Because the invariant 2c for message m holds and $w \notin R(Next, destID)$, then for every admissible state with $src.seqs[destID] < seq$ it holds that $w \notin R(src, destID)$. This is contradictory to the assumption.
Thus, the invariant 2c for message m' holds in state s_2.

Invariant 3 is violated in state s_2.
Assume a node x sends a PROBESUCCESS($destID$, seq, $dest$) message to a node u in state s_1. This can happen in action GENERICPROBE() and GREEDYPROBE() when x receives a GENERICPROBE(src, $destID$, $Next$, seq) or GREEDYPROBE(src, $destID$, seq) message with $x.id = destID$.
In both cases x sends PROBESUCCESS($destID$, seq, x) to node u. The invariant of this message holds because $x.id = destID$ implies $x \in R(u, destID)$.

Invariant 4 is violated in state s_2.
Assume a node x sends a PROBEFAIL($destID$, seq) message to a node u in state s_1. This can only happen in action GENERICPROBE() and it means a GENERICPROBE(src, $destID$, $Next$, seq) message m has arrived at x with $x.id \neq destID$, $u = src$ and $Next = \{x\}$ and there is no node $y \in x.Right$ with $y.id \leq destID$. If there is no node $w = destID$ exists, then the invariant of this message holds for sure. Otherwise, the invariant 2c of the message m implies the invariant of the PROBEFAIL($destID$, seq) message holds since $u = src$ and $w \notin R(Next, destID)$.

Invariant 5 is violated in state s_2.
Assume a node x sends a SEARCH(x, $destID$) message to a node u in state s_1. This can only happen when x gets a PROBESUCCESS($destID$, seq, $dest$) message. The invariant of the PROBESUCCESS($destID$, seq, $dest$) message implies directly the invariant of the SEARCH(x, $destID$) message.

We have proven that for all cases s_2 must be an admissible state, which is contradictory to the assumption. Thus, once an admissible state is reached, the system stays admissible according to the MULTISKIPGRAPH protocol. ■

Proof of Lemma 7.

Proof: We will show that the corrupt messages will disappear at some time point t and no corrupt messages will be created from the protocol after t. We will analyze messages used in HybridSearch one by one.

A new GREEDYPROBE() message can be created in the TIMEOUT() action of a node u or in the GREEDYPROBE() action when node u receives a GREEDYPROBE() message. First to be mentioned that as proven in Lemma 6 the GREEDYPROBE() messages created in the TIMEOUT() action can not be corrupt. For the latter case, if the GREEDYPROBE(src, $destID$, seq) message m received by node u is a valid (not corrupt) message whose invariant holds. Thus, $u \in R(src, destID)$ and the new created GREEDYPROBE(src, $destID$, seq) message m' is only sent to a neighbor v of u. Since v is a neighbor of u, then $v \in R(u, destID) \subseteq R(src, destID)$. So the invariant of m' holds. Therefore, valid GREEDYPROBE messages only cause valid GREEDYPROBE. If there is an arbitrary corrupt GREEDYPROBE() message m received by a node u, then it can cause at most one other corrupt GREEDYPROBE() message m' to be sent to a right neighbor v of u, and this corrupt message m' can cause another one sent to a right neighbor of v and this goes on. Since there are finitely many nodes, only finitely many corrupt GREEDYPROBE() messages will be created which are initially caused by m and all these GREEDYPROBE() messages will eventually disappear after reception by nodes.

Starting from the state s_0 in which all corrupt GREEDYPROBE() messages disappeared, a new GenericProbe() message can be created in the TIMEOUT() action of a node u or in the GENERICPROBE() action when node u receives a GENERICPROBE() message. As proven in Lemma 6 the GENERICPROBE() message created in TIMEOUT() can not be corrupt. If u received a valid GENERICPROBE it is also proven in Lemma 6 the new created GENERICPROBE can not violate its invariant. If u received a corrupt GENERICPROBE(src, $destID$, $Next$, seq) message m, it will only causes one new corrupt message GENERICPROBE(src, $destID$, $Next'$, seq) m', where the difference between $Next$ and $Next'$ are only u and right(left) neighbors of u. Since there are finite nodes on the right(left) side of u, only finite corrupt GENERICPROBE messages will be created in the subsequent states after u received m. They will also eventually disappear after reception by nodes.

Starting from the state s_1 in which all corrupt GREEDYPROBE() and GENERICPROBE() messages disappeared, a new PROBESUCCESS($destID$, seq, $dest$) message m can be created in action GREEDYPROBE() or GENERICPROBE() when u receives a valid GREEDYPROBE(src, $destID$, seq) or GENERICPROBE(src, $destID$, $Next$, seq) message with $u.id = destID$ and $dest = u$. Since $u \in R(u, destID)$, the invariant of m holds. Notice that any PROBESUCCESS() message can not cause a new PROBESUCCESS message, thus all corrupt PROBESUCCESS() messages will disappear after reception by nodes.

Starting from the state s_2 in which all corrupt

GREEDYPROBE(), GENERICPROBE() and PROBESUCCESS() messages disappeared, a PROBEFAIL($destID$, seq) message m can only by created in action GENERICPROBE() when a node u receives a GENERICPROBE(src, $destID$, $Next$, seq) message. Since state s_2 contains no corrupt GENERICPROBE() message and as proven in Lemma 6 PROBEFAIL() message caused by a valid GENERICPROBE() message is also valid, so the invariant of m must also hold.

Starting from the state s_3 in which all corrupt GREEDYPROBE(), GENERICPROBE(), PROBESUCCESS() and PROBEFAIL() messages disappeared, then a SEARCH(v, $destID$) message m can only be sent to a node u in a PROBESUCCESS() action in node v when v receives a PROBESUCCESS($destID$, seq, $dest$) message. This PROBESUCCESS($destID$, seq, $dest$) is not corrupt, which implies $u = dest$ and $u \in R(v, destID)$. Thus, the invariant of m holds. m will disappear when it is received by u.

Consider the state s_4 in which all corrupt GREEDYPROBE(), GENERICPROBE(), PROBESUCCESS(), PROBEFAIL() and SEARCH() messages disappeared, then all HybridSearch message invariants hold in s_4 and no corrupt message will be produced by the protocol. s_4 is therefore an admissible state and after s_4 all states stay admissible. ■

Proof of Lemma 8

Proof: Node u receives GENERICPROBE(src, $destID$, $Next$, seq) and it will change $Next$ in the GENERICPROBE() action to $Next_1$ such that it satisfies $R(Next_1, destID) = R(u, destID) \setminus \{u\}$. Since $w \in R(u, destID)$ and $w \neq u$, then $w \in R(Next_1, destID)$ holds.

Node u sends a GENERICPROBE(src, $destID$, $Next_1$, seq) message to a node $v_1 \in Next_1$ which has the minimal id. So for node v_1 there is a GENERICPROBE(src, $destID$, $Next_1$, seq) message in its channel with $w \in R(Next_1, destID)$ and v_1 will receive it under the fair message receipt assumption.
- If $v_1 = w$, the proof is done.
- If $v_1 \neq w$, then v_1 will change $Next_1$ to $Next_2$ such that $R(Next_2, destID) = R(Next_1, destID) \setminus \{v_1\} \cup \{x | x \in v_1.Right \wedge x.id \leq destID\}$ in the action GENERICPROBE(). Since $w \in R(Next_1, destID)$ and the only node in $Next_1$ which $Next_2$ doesn't contain is v_1 and v_1 has minimal ID in $Next_1$, $w \in R(Next_2, destID)$ must hold.

To summarize what happened to node $v_1 \neq w$ as follows: node v_1 receives a GENERICPROBE(src, $destID$, $Next_1$, seq) message with $w \in R(Next_1, destID)$ and it sends a GENERICPROBE(src, $destID$, $Next_2$, seq) message with $w \in R(Next_2, destID)$ to a node in $Next_2$. In other words, the reachability to the target node is preserved by the GENERICPROBE() action. This will happen to every node v_i with $v_i.id < destID$ afterwards in the generic probing process which receives a GENERICPROBE(src, $destID$, $Next_i$, seq) message with $w \in R(Next_i, destID)$. Assume $v_n \neq w$ is the last node receives a GENERICPROBE(src, $destID$, $Next_n$, seq) message with $w \in R(Next_n, destID)$ in the generic probing process for $destID$, then v_n has to send a GENERICPROBE(src, $destID$, $Next'$, seq) to a node in $Next_n$ and this node must be w, otherwise v_n is not the last node that gets such a message. So w will eventually get a GENERICPROBE(src, $destID$, $Next'$, seq) message. ■

C. Pseudocode of MULTISKIPGRAPH*

See the MULTISKIPGRAPH* protocol in Algorithm 3. Note that INITIATENEWSEARCH($destID$), INTRODUCE(v), PROBESUCCESS($destID$, seq, $dest$) and PROBEFAIL($destID$, seq) are the same ones as in the MULTISKIPGRAPH protocol.

D. Proofs of MULTISKIPGRAPH*

Theorem 3 *The* MULTISKIPGRAPH* *protocol is a self-stabilizing solution to the perfect skip graph topology.*

The proof of this theorem consists of Lemma 10,Lemma 11,Lemma 12 and Lemma 13.

Lemma 10 *If a computation of the* MULTISKIPGRAPH* *protocol starts from a state in which G is weakly connected, then G remains weakly connected in every subsequent state.*

Proof: The proof is analogous to the proof of Lemma 1. In every action of the MULTISKIPGRAPH* protocol whenever a message with a reference of a node v is received by a node u then either v is added to the set $u.Left$ or $u.Right$ or a new message is created with v as a parameter and sent to a node $w \in u.Left \cup u.Right$. Even in the SAFEDELETION(v) action, although the explicit edge (u, v) is removed by u, node u does INTRODUCE(v) which creates a new message including the reference of v and sends this message to a neighbor. Thus, the implicit edge (u, v) is replaced by a path (u, w, v) and the weak connectivity is always preserved. ■

Lemma 11 *For any computation of the* MULTISKIPGRAPH* *protocol starting from any state in which G is weakly connected, there is a state in which the level-0 subgraph G_0 is the line topology.*

Proof: The proof is analogous to Lemma 2, since only INTRODUCE(v) messages are used for establishing the level 0. In every action of the MULTISKIPGRAPH* protocol whenever a message with a reference to a node v is received by a node u, either v is added to the set $u.Left$ or $u.Right$ or the INTRODUCE(v) action is triggered. In both cases $\Phi(G_0)$ decreases. ■

Lemma 12 *If a computation of the* MULTISKIPGRAPH* *protocol contains a state in which the level-0 subgraph G_0 is the line topology, then G_0 will always be the line topology afterwards.*

Proof: The proof is exactly the same as the proof for Lemma 3. ■

Lemma 13 *For any computation of the* MULTISKIPGRAPH* *protocol starting from any state in which G is weakly connected, there is a computation suffix in which G_e is exactly*

```
Action TIMEOUT()
                    // The self-stabilizing part;
forall v ∈ LeftUnknown do
    x ← arg min {x.id | x ∈ Left ∧ x.id > v.id};
    if x ≠ ⊥ then
        send SAFEINTRODUCE(v, self) to x;
forall w ∈ RightUnknown do
    x ← arg max {x.id | x ∈ Right ∧ x.id < w.id};
    if x ≠ ⊥ then
        send SAFEINTRODUCE(w, self) to x;
// Left := {v_1, ..., v_n} with 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};
// Right := {w_1, ..., w_m} with 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 SlowGreedySearch part;
for destID ∈ Waiting do
    send SLOWGREEDYPROBE(self, destID, ⊥, ∅, {self})seq to self;

Action SAFEINTRODUCE(v, src)
if src.id ≠ id ∧ src ∉ Left ∪ Right then
    INTRODUCE(src);
if v.id ≠ id then
    if v.id < id then
        if v ∉ Left then
            LeftUnknown ← LeftUnknown ∪ {v};
        send SAFEDELETION(v) to src;
    else
        // Analogous to the previous case.

Action SAFEDELETION(v)
if v.id ≠ id then
    if v.id < id then
        if v ∈ LeftUnknown then
            LeftUnknown ← LeftUnknown \ {v};
        INTRODUCE(v);
    else
        // Analogous to the previous case.

Action INTROLEVELNODE(v, i)
if v.id ≠ id then
    if i > 0 then
        doIntro ← true;
        if v.id < id then
            if Left ≠ ∅ then
                for j ← 0 to i-1 do
                    if LeftLevel[j] == ⊥ then
                        doIntro ← false;
                        break;
            else
                doIntro ← false;
            if doIntro==false then
                INTRODUCE(v);
            else
                next ← LeftLevel[i − 1];
                send PROBELEVELNODE(self, v, i, i − 1) to next;
        else
            // Analogous to the previous case.
    else
        INTRODUCE(v);

Action PROBELEVELNODE(src, dest, level, i)
if src ∉ Left ∪ Right then
    INTRODUCE(src);
if dest ∉ Left ∪ Right then
    INTRODUCE(dest);
if level > 0 then
    if v.id == id ∧ i == −1 then
        send LEVELNODESUCC(dest, level) to src;
    else
        if v.id < id then
            if i > 0 then
                if LeftLevel[i − 1] ≠ ⊥ then
                    next ← LeftLevel[i − 1];
                    send PROBELEVELNODE(self, v, level, i − 1) to
                      next;
            if i == 0 then
                if LeftLevel[i] ≠ ⊥ then
                    next ← LeftLevel[i];
                    send PROBELEVELNODE(self, v, level, i − 1) to
                      next;
        else
            // Analogous to the previous case.

Action LEVELNODESUCC(v, i)
if v.id ≠ id then
    if i > 0 then
        doIntro ← true;
        if v.id < id then
            if Left ≠ ∅ then
                for j ← 0 to i-1 do
                    if LeftLevel[j] == ⊥ then
                        doIntro ← false;
                        break;
            else
                doIntro ← false;
            if doIntro==false 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;
                x ← arg min {x.id | x ∈ Left ∧ x.id > v.id};
                send SAFEINTRODUCE(w, self) to x;
        else
            // Analogous to the previous case.
    else
        INTRODUCE(v);
```

Algorithm 3: The MULTISKIPGRAPH* protocol

the perfect skip graph topology and no new explicit edges will ever be created again.

Proof: The proof can be split into three parts:

1. There is a computation suffix in which G_e is a supergraph of the perfect skip graph topology.
2. There is a computation suffix in which all temporary edges (neighbors stored in the unknown sets) are removed and G_e is exactly the perfect skip graph.
3. Once G_e is exactly the perfect skip graph, no explicit edge will be created again.

Part 1:
The proof is analogous to the proof for Lemma 4. We omit it here.

Part 2:
Starting from the state s_1 in which G_e is a supergraph of the perfect skip graph topology, the level neighbors of each node won't change any more according to the protocol. Consider the left most node u that has at least one temporary neighbor stored in u.*RightUnknown*. Consider the left most neighbor v in u.*RightUnknown*, then a node $x \in u$.*Right* with $x.id < v.id$ exists and x must be a level neighbor of node u. If x doesn't exist, then v is the closest neighbor in u.*Right* which would mean v is the level-0 neighbor of u and this is contradictory to the assumption that $v \in u$.*RightUnknown*. When x exists and $x.id < v.id$, since v is the left most neighbor in u.*RightUnknown*, then x must be a level neighbor of u.

In action TIMEOUT() node u sends SAFEINTRODUCE(v, u) to node x. Node x receives this message, will add v to x.*RightUnknown* if $v \notin x$.*Right* and sends SAFEDELETION(v) back to u, node u receives this message, will remove v from u.*RightUnknown*. Once this edge is removed it won't be created again, since a temporary edge (u, v) can only be created when u receives a SAFEINTRODUCE(v, y) message sent by a node y whose id is smaller than u and $v \in y$.*RightUnknown*. However, such a node y doesn't exits since u is the most left node that has temporary neighbors.

After u removes v from u.*RightUnknown*, consider the next most left neighbor in u.*RightUnknown* and node u will remove it analogously. So eventually node u will remove all nodes in u.*RightUnknown* and these temporary edges won't be created again. After node u removes all its temporary edges, then consider the next most left node that has at least one temporary right neighbors and analogously it will also remove these temporary edges. Hence, there will be a state in which all nodes remove their temporary right neighbors.

Starting from the state s_2 in which all nodes have no temporary right neighbors and consider the most right node that has at least one temporary neighbor stored in u.*LeftUnknown* and the rest of the proof is analog to the deletion of temporary right neighbors. Thus, there will be a state all nodes also remove their temporary left neighbors. Once all temporary edges are removed, G_e will be exactly the perfect skip graph since those temporary edges won't be created again.

Part 3:
Starting from the state s_3 in which G_e is exactly the perfect skip graph, no explicit edge will be created again since they can only be created in the following actions:

- INTRODUCE() action only creates a new explicit edge when the level 0 is not stable. Since G_e is exactly the perfect skip graph, so level 0 is stable and no edge will be created in this action.
- LEVELNODESUCC() action only creates a new explicit edge if a level i is not stable. Since G_e is exactly the perfect skip graph, so all levels are stable and no edge will be created in this action.
- SAFEINTRODUCE() action is triggered by reception of a SAFEINTRODUCE() message and SAFEINTRODUCE() message can only be sent when there is a temporary edge. However, all $Unknown$ sets of each node are empty since G_e is exactly the perfect skip graph. So no SAFEINTRODUCE() messages will be created and the SAFEINTRODUCE() action will not be called.

■

Theorem 4 *The* MULTISKIPGRAPH* *protocol satisfies monotonic searchability according to SlowGreedySearch.*

The proof structure for the theorem is analogous to the proof for the MULTISKIPGRAPH protocol. This theorem is proven by using Lemma 15, Lemma 16 and Lemma 17. However, the message invariants are different here since removing an explicit edge is allowed in the MULTISKIPGRAPH* protocol and the search protocol is different. Because delegating an edge may affect the monotonic searchability, it's sufficient to consider the messages which are related to that operation in the MULTISKIPGRAPH* protocol, namely the SAFEINTRODUCE() and SAFEDELETION messages. We introduce the following message invariants of SlowGreedySearch:

Invariant 1: If there is a SAFEINTRODUCE(v, src) message with $src \neq \perp$ in $u.ch$, then $v \neq src$ and $u \in R(src)$ (or $u \in L(src)$).

Invariant 2: If there is a SAFEDELETION(v) message in $u.ch$, then there is a node $w \neq v$ with $w \in u.Right$ and $v \in R(w)$ (or $w \in u.Left$ and $v \in L(w)$).

Invariant 3: If there is a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message in $u.ch$, then:

a. $Prev \cap Next = \emptyset$;
b. $u \in Next$ and $\forall v \in Next \setminus \{u\} : d(v.id, destID) \geq d(u.id, destID)$;
c. $R(Next, destID) \subseteq R(src, destID)$;
d. If a node w exists with $w.id = destID$ and $w \in R(Next, destID)$, then there exists at least one path from a node in $Next$ to w without traversing the nodes in $Prev$ and for every edge (a, b) on that path $d(a.id, destID) > d(b.id, destID)$ holds.
e. 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 $w \notin R(src, destID)$.

Invariant 4: If there is a PROBESUCCESS($destID$, seq, $dest$) message in $u.ch$, then $dest.id = destID$ and $dest \in R(u, destID)$.

Invariant 5: If there is a PROBEFAIL($destID$, seq) message in $u.ch$, then if a node w exists with $w.id = destID$, then for every admissible state with $u.seqs[destID] < seq$ it holds that $w \notin R(u, destID)$.

Invariant 6: If there is a SEARCH(v, $destID$) message in $u.ch$, then $u.id = destID$ and $u \in R(v, destID)$.

Note that the messages with invariants are called **SlowGreedySearch Messages**. Invariant 4, 5, 6 are exactly the same as those defined for the HybridSearch in MULTISKIPGRAPH. Invariant 1 and 2 ensure the deletion of an edge only happens when the reachability between the two nodes of that edge is preserved by an alternative path. Invariant 3b, 3c and 3e are as same as the Invariant 2 for the HybridSearch. Invariant 3a and 3d indicate that all nodes in the set $Prev$ are visited on the forwarding path of the probe message and if the target node is reachable from nodes in $Next$, then there exists at least one path from a node in set $Next$ to the target node without revisiting the nodes in $Prev$.

The following lemma shows that the reachability between arbitrary two nodes is preserved by the MULTISKIPGRAPH* protocol.

Lemma 14 *If there is a state s in which the first two SlowGreedySearch message invariants hold and for arbitrary nodes u and v, if $v \in R(u, destID)$, then $v \in R(u, destID)$ holds in every state $s' > s$.*

Proof: The only action that delegates away an explicit edge (x, y) stored in $x.Right$ for arbitrary nodes x and y is the SAFEDELETION() action when $x.id < y.id$. Therefore, consider an arbitrary SAFEDELETION(y) action executed by node x. Because the first two invariants (invariant 1 and 2) hold, then there is a node $w \neq y$ with $w \in x.Right$ and $y \in R(w)$ according to invariant 2. Thus, after node y is removed from $x.Right$ the property $y \in R(x)$ still holds. Analogous to the case $x.id > y.id$, $y \in L(x)$ holds after y is removed from $x.Left$. Thus, the reachability from node x to node y is preserved and this implies the lemma. ∎

Lemma 15 *If in a computation of the MULTISKIPGRAPH* protocol there is an admissible state, then every state afterwards is admissible.*

Proof: **Proof by contradiction:**
Assume there is an admissible state s_1 in which all SlowGreedySearch message invariants hold and in the subsequent state s_2 in which one of the invariants doesn't hold. This can only happen when a new SlowGreedySearch message is sent in state s_1. Consider the case that $src.id < destID$ and the other case is analog. We make a case distinction over the six possibilities:

Invariant 1 is violated in state s_2.
Assume a node x sends a SAFEINTRODUCE() message to a node u in state s_1. Assume $v.id > x.id$ (for the other case it is analog), then this can only happen in the TIMEOUT() action: x sends a SAFEINTRODUCE(v, x) message to a node $u \in x.Right$ where $u.id < v.id$ and $v \in x.RightUnknown \subseteq x.Right$. Thus, $u \in R(x)$ and the invariant holds in state s_2.

Invariant 2 is violated in state s_2.
Assume a node x sends a SAFEDELETION(v) message to a node u in state s_1. Assume $v.id > x.id$ (for the other case it is analog), then this can only happen in the SAFEINTRODUCE() action when x receives a SAFEINTRODUCE(v, u) message, in this action x adds v to $x.Right$ and sends a SAFEDELETION(v) message to node u. Since state s_1 is admissible, which means the invariant of the SAFEINTRODUCE(v, u) must hold and that is $v \neq u$ and $x \in R(u)$. This implies that there must be a node w in $u.Right$ such that $x \in R(w)$ or $x = w$. Node v was added to $x.Right$ in the action which means $v \in R(x)$, together with $x \in R(w)$ or $x = w$ implies $v \in R(w)$ hold for node $w \in u.Right$. Thus, the invariant holds in state s_2.

Invariant 3 is violated in state s_2.
Assume a node x sends a SLOWGREEDYPROBE() message to a node u in state s_1. This can only happen in two cases: x sends a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message to itself in action TIMEOUT() or x receives a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message and sends a SLOWGREEDYPROBE(src, $destID$, $last'$, $Prev'$, $Next'$)seq in the corresponding action.

(i) x sends a SLOWGREEDYPROBE(x, $destID$, $\emptyset$, $\{x\}$, seq) message m in TIMEOUT() action to itself.
- Invariant 3a holds since $Prev = \emptyset$ and $Next = \{x\}$.
- Invariant 3b holds since $u = x$ and $Next = \{x\}$.
- Invariant 3c holds since $src = x$, $Next = \{x\}$ and $R(\{x\}, destID) = R(x, destID)$.
- Invariant 3d holds since $Prev = \emptyset$.
- Invariant 3e holds and the proof is as follows:
Assume invariant 3e doesn't hold in state s_2, then there exists a node w with $w.id = destID$ and $w \notin R(Next, destID)$ and there is an admissible state s_0 with $x.seqs[destID] < seq$ and $w \in R(x, destID)$.
Because $Next = \{x\}$, $w \notin R(Next, destID) = R(x, destID)$ holds in state s_2. Just like in the proof for the MULTISKIPGRAPH protocol s_0 is a earlier state than s_1 because $x.seqs[destID] < seq$.
Since $w \in R(x, destID)$ is true in state s_0, according to Lemma 14 $w \in R(x, destID)$ is true in all states later than s_0, including state s_2. This is contradictory to the assumption that $w \notin R(x, destID)$ in state s_2. Thus, the invariant 3e must hold in state s_2.

(ii) x sends a SLOWGREEDYPROBE(src, $destID$, $Prev'$,

$Next'$, seq) message m' to a node u in state s_1 by receiving a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message m. The invariants of m must hold since s_1 is admissible.

- Invariant 3a holds and the proof is as follows:

$Prev \cap Next = \emptyset$, $Prev' = Prev \cup \{x\}$ and only right neighbors of node x which are in set $A := \{v \in x.Right | v.id \leq destID \wedge v \notin Prev\}$ are added to $Next$.

Obviously $A \cap Prev' = \emptyset$. Hence, $Next' \cap Prev' = (Next \cup A \setminus \{x\}) \cap Prev' = ((Next \cap Prev') \cup (A \cap Prev')) \setminus \{x\} = (x \cup \emptyset) \setminus \{x\} = \emptyset$.

- Invariant 3b holds since x only sends m' to a node u in $Next'$. $\forall v \in Next'$ it holds that $d(v.id, destID) \geq d(u.id, destID)$, since u is a node in $Next'$ which has minimal distance to the target node according to the action.

- Invariant 3c holds and the proof is as follows:

Since the invariants of m holds, there are $x \in Next$ and $R(Next, destID) \subseteq R(src, destID)$.

$x \in Next$ implies $R(x, destID) \subseteq R(Next, destID) \subseteq R(src, destID)$.

Clearly $R(A, destID) \subseteq R(x, destID) \subseteq R(src, destID)$ holds since $A \subseteq x.Right$ hold.

$R(A, destID) \subseteq R(src, destID)$, $R(Next, destID) \subseteq R(src, destID)$ and $Next' = Next \cup A \setminus \{x\}$ imply $R(Next', destID) \subseteq R(src, destID)$ must hold.

- Invaraint 3d holds and the proof is as follows:

Assume node w exists with $w.id = destID$, then there are two cases:

a. If $w \notin R(Next, destID)$, then $w \notin R(Next', destID)$ since $R(Next', destID) \subseteq R(Next, destID)$. Thus, invariant 3d of message m' holds.
b. If $w \in R(Next, destID)$, because invariant 3d of message m holds, then there exists at least one path from a node in $Next$ to w without traversing the nodes in $Prev$. Consider such a path, then there are two cases:
   - If the path traverses node x, the first edge after node x on that path is from node x to a right neighbor y of x which is closer to $destID$ and not in $Prev' = Prev \cup \{x\}$. Node x changes $Next$ to $Next'$ by adding its right neighbors which are closer to $destID$ than itself and not in $Prev'$, including node y. Thus, there exists a path from node $y \in Next'$ to w without traversing the nodes in $Prev'$.
   - If the path does not traverse node x, then the changes node x performs to the set $Next$ has no effect on the path. Thus, there exists a path from a node in $Next'$ to w without traversing the nodes in $Prev'$.

Both cases show that invariant 3d of message m' holds in state s_2.

- Invariant 3e holds and the proof is as follows:

Assume invariant 3e doesn't hold, then there exists a node w with $w.id = destID$ and $w \notin R(Next', destID)$ in state s_2 and there is a previous admissible state s_0 with $src.seqs[destID] < seq$ and $w \in R(src, destID)$.

Obviously $w.id \neq x.id$, otherwise x sends a PROBESUCCESS() message rather than a SLOWGREEDYPROBE() message.

$w \notin R(Next', destID)$ in state s_2 implies $w \notin R(Next, destID)$ in state s_1.

Assume $w \in R(Next, destID)$ and invariant 3d holds for message m, then there exists one path from a node in $Next$ to w without traversing the nodes in $Prev$. Since node x changes $Next$ to $Next'$ by adding its right neighbors which are closer to $destID$ than itself and not in $Prev' = Prev \cup \{x\}$, then there exists a path from a node in $Next'$ to w without traversing the nodes in $Prev'$. This implies $w \in R(Next', destID)$. This is contradictory to the assumption that $w \notin R(Next', destID)$. Therefore, $w \notin R(Next, destID)$ must hold if $w \notin R(Next', destID)$. Because invariant 3e holds for message m and $w \notin R(Next, destID)$, then for every admissible state with $src.seqs[destID] < seq$ it holds that $w \notin R(src, destID)$. This contradictory to the assumption. So the invariant 3e must hold for message m',

Invariant 4 is violated in state s_2.

Assume a node x sends a PROBESUCCESS($destID$, seq, $dest$) message to a node u in state s_1. This can only happen in action SLOWGREEDYPROBE() when x receives a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message with $x.id = destID$. In this case x sends PROBESUCCESS($destID$, seq, x) to node u. The invariant of this message holds because $x.id = destID$ implies $x \in R(u, destID)$.

Invariant 5 is violated in state s_2.

Assume a node x sends a PROBEFAIL($destID$, seq) message to a node u in state s_1. This can only happen in action SLOWGREEDYPROBE() and it means a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message m has arrived at x with $x.id \neq destID$, $u = src$ and $Next = \{x\}$ and there is no node $y \in x.Right$ with $y.id \leq destID$ which is not in $Prev$. If there exists no node $w = destID$, then the invariant of this message holds for sure. Assume node w exists, then $Next = \{x\}$ and $Next' = Next \setminus \{x\} = \emptyset$ imply $w \notin R(Next', destID)$.

Because invariant 3e of message m holds, then for every admissible state with $u.seqs[destID] < seq$ it holds that $w \notin R(src, destID)$.

$u = src$ implies the invariant of the PROBEFAIL($destID$, seq) holds.

Invariant 6 is violated in state s_2.

Assume a node x send a SEARCH(x, $destID$) message to a node u in state s_1. This can only happen when x gets a PROBESUCCESS($destID$, seq, $dest$) message. The invariant of the PROBESUCCESS($destID$, seq, $dest$) message (invariant 4) implies directly the invariant of the SEARCH(x, $destID$) message. ■

Lemma 16 *In every computation of the* MULTISKIPGRAPH* *protocol, there is an admissible state and after that all states are admissible.*

Proof: According to Lemma 13, there is a state s_1 in which and very subsequent state, every node u has no neighbors stored in *LeftUnknown* and *RightUnknown*. According to the MULTISKIPGRAPH* protocol, any SAFEINTRODUCE(v, src) message with $v \neq src$ is only sent from a node src whose *LeftUnknown* and *RightUnknown* are not empty. Thus, under the fair message receipt assumption there will be a state s_2 after s_1 in which all SAFEINTRODUCE(v, src) messages have been received by nodes. Moreover, any SAFEDELETION(v) message is only sent from a node u if it receives an SAFEINTRODUCE(v, src) message. Thus, under the fair message receipt assumption there will be a state s_3 after s_2 in which all SAFEDELETION(v) messages have been received by nodes. From s_3 on, no SAFEINTRODUCE(v, src) and SAFEDELETION(v) message will be created, thus the first two invaraints hold in state s_3 and in every subsequent state.

The rest of the proof is analog to the proof in Lemma 7 for the MULTISKIPGRAPH protocol. We just analyze the SLOWGREEDYPROBE() message used in the SlowGreedySearch, since other messages used there are the same as in the HybridSearch.

A new SLOWGREEDYPROBE() message can be created in the TIMEOUT() action of a node u or in the SLOWGREEDYPROBE() action when node u receives a SLOWGREEDYPROBE() message. As prove in Lemma 15 every SLOWGREEDYPROBE() message created in the TIMEOUT() action is admissible. For the second case, if the SLOWGREEDYPROBE() message received by node u is not corrupt, then u will also send a SLOWGREEDYPROBE() message which is not corrupt and this is also proven in Lemma 15. If there is an arbitrary corrupt SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message m received by a node u, then it can cause at most one other corrupt SLOWGREEDYPROBE(src, $destID$, $Prev'$, $Next'$, seq)message m' to be sent to a node in $Next'$, and this corrupt message m' can cause another one sent to another node and this goes on and on. However, every node u that receives a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message only changes $Next$ to $Next'$ by adding neighbors which are not in $Prev$ and u will add itself to $Prev$. Since there are finite number of nodes, the set $Prev$ will contain at most all the nodes at some time point. From this time point on, every node that gets a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message will not enlarge the set $Next$ anymore but only remove itself from the $Next$. Either, a SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message comes to the target node and a PROBESUCCESS message will be sent to node src, or the set $Next$ will be empty and a PROBEFAIL message will be sent to node src. Thus, every SLOWGREEDYPROBE() message caused by a corrupt SLOWGREEDYPROBE() message will eventually disappear. In addition, just like the proof in Lemma 7 all corrupt PROBESUCCESS(), PROBEFAIL() and Search() messages will eventually disappear.

Consider that state s_4 after s_3 where all corrupt SLOWGREEDYPROBE(), PROBESUCCESS(), PROBEFAIL() and Search() messages disappeared, then all SlowGreedySearch message invariants hold in s_4 and no corrupt message will be produced by the protocol. s_4 is therefore an admissible state and after s_4 all states stay admissible. ■

Similar to the Lemma 8 the following lemma shows that the reachability to the target node is preserved by the SLOWGREEDYPROBE() action.

Lemma 17 *If there is a* SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) *message in $u.ch$ in an admissible state with $u.id < destID$ and there exists a node w with $w.id = destID$ and $w \in R(Next, destID)$, then u will send a* SLOWGREEDYPROBE(src, $destID$, $Prev'$, $Next'$, seq) *message with $w \in R(Next', destID)$.*

Proof: Consider the following two cases:

(i) $\forall x \in Prev : x.id < u.id$:
In this case, all right neighbors of u which are closer to $destID$ are added to $Next$, since $Prev \cap u.Right = \emptyset$. Thus, u changes $Next$ to $Next'$ such that $R(Next', destID) = R(Next', destID) \setminus \{u\}$ holds. Since $w \in R(Next, destID)$ and $w.id \neq u.id$, $w \in R(Next'.destID)$ holds.

(ii) $\exists x \in Prev_i : x.id > u.id$:
In this case, consider the set $A := \{x \in u.Right | x.id \leq destID\}$. Then there are the following cases:

a. If $Prev \cap A = \emptyset$, then u changes $Next$ to $Next'$ such that $R(Next', destID) = R(Next, destID) \setminus \{u\}$ holds. Thus, $w \in R(Next'.destID)$ holds just like in case (i).
b. If $Prev_i \cap A \neq \emptyset$: Since invariant 3d of message SLOWGREEDYPROBE(u, $destID$, $Prev$, $Next$, seq) holds, then there exists at least one path from a node in the set $Next$ to w without traversing the nodes in $Prev$. As proven in Lemma 15 for invariant 3d, then there exists at least one path in $Next'$ to w without traversing the nodes in $Prev' = Prev_i \cup \{u\}$. This implies $w \in R(Next', destID)$.

■

Lemma 18 *If there is a* SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) *message in $u.ch$ in an admissible state with $u.id < destID$ and there exists a node $w.id = destID$ and $w \in R(u, destID)$, then a* SLOWGREEDYPROBE(src, $destID$, $Prev'$, $Next'$, seq) *message will be in $w.ch$ eventually.*

Proof: Node u receives SLOWGREEDYPROBE(src, $destID$, $Prev$, $Next$, seq) message and invariant 3b of this message holds, which means $u \in Next$ and u is the node in

$Next$ which has maximal id.
$w \in R(u, destID)$ implies $w \in R(Next, destID)$.
According to Lemma 17 node u will send a SLOWGREEDYPROBE(src, $destID$, $Prev_1$, $Next_1$, seq) message with $w \in R(Next_1, destID)$ to a node v_1. By applying Lemma 17 recursively for the subsequent nodes on the forwarding path of a SLOWGREEDYPROBE(src, $destID$, $Prev_i$, $Next_i$, seq) message satisfying $w \in R(Next_i, destID)$, a SLOWGREEDYPROBE(src, $destID$, $Prev'$, $Next'$, seq) message will be in $w.ch$ eventually. ■

Lemma 19 *The* MULTISKIPGRAPH* *protocol guarantees monotonic searchability according to SlowGreedySearch in every computation that starts from an admissible state.*

Proof: **Proof by contradiction:**
Assume two SEARCH(u, $destID$) messages m and m' created in admissible states at time t and t' with $t < t'$ such that m is delivered successfully but m' is not. Let node w be the target node with $w.id = destID$. If m' is created when m is already in $u.WaitingFor(destID)$, then the protocol will handle both messages the same. That is, when m is delivered successfully, m' will also be delivered successfully. Let seq_2 be the sequence number stored locally in node u for $destID$ at time t' when m' is created and seq_1 be the sequence number at time t when m is delivered. The sequence number increases monotonically. If m' is created when m is already sent by node u, then $seq_2 \geq seq_1$. Because m' is not delivered successfully, then there are two possibilities: (1) u receives a PROBEFAIL($destID$, seq) with $seq \geq u.seqs[destID] \geq seq_2$. (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 $u.seqs[destID] < seq$, inclusive the state when m is delivered where the sequence number is $seq_1 \leq seq_2$, that is $w \notin R(u, destID)$. However this is contradictory to the invariant of m which is $w \in R(u, destID)$.

Case (2): According to the protocol u sends SLOWGREEDYPROBE(u, $destID$, $\emptyset$, $\{u\}$, seq) to itself with $seq \geq u.seqs[destID]$ periodically for m'. Since m is delivered successfully and invariant of m holds, that is $w \in R(u, destID)$. According to Lemma 18 a SLOWGREEDYPROBE(src, $destID$, $Prev'$, $Next'$, seq) message will eventually arrive at node w, node w will send a PROBESUCCESS($destID$, seq, w) message to node u. u receives the PROBESUCCESS($destID$, seq, w) message with $seq \geq u.seqs[destID]$, which violates the assumption.

Thus, m' is delivered successfully and monotonic searchability is satisfied. ■