Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures*

Christian Scheideler[1], Alexander Setzer[2], and Thim Strothmann[3]

1 **Paderborn University, Paderborn, Germany**
2 **Paderborn University, Paderborn, Germany**
3 **Paderborn University, Paderborn, Germany**

Abstract

Distributed applications are commonly based on overlay networks interconnecting their sites so that they can exchange information. For these overlay networks to preserve their functionality, they should be able to recover from various problems like membership changes or faults. Various self-stabilizing overlay networks have already been proposed in recent years, which have the advantage of being able to recover from any illegal state, but none of these networks can give any guarantees on its functionality while the recovery process is going on. We initiate research on overlay networks that are not only self-stabilizing but that also ensure that searchability is maintained while the recovery process is going on, as long as there are no corrupted messages in the system. 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 call this property *monotonic searchability*. We show that in general it is impossible to provide monotonic searchability if corrupted messages are present in the system, which justifies the restriction to system states without corrupted messages. Furthermore, we provide a self-stabilizing protocol for the line for which we can also show monotonic searchability. It turns out that even for the line it is non-trivial to achieve this property. Additionally, we extend our protocol to deal with node departures in terms of the Finite Departure Problem of Foreback et al. (SSS 2014). This makes our protocol even capable of handling node dynamics.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Topological Self-Stabilization, Monotonic Searchability, Node Departures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.24

1 Introduction

The Internet has opened up tremendous opportunities for people to interact and exchange information. Particularly popular ways to interact are peer-to-peer systems and social networks. For these systems to stay popular, it is very important that they are highly available. However, once these systems become large enough, changes and faults are not an exception but the rule. Therefore, mechanisms are needed that ensure that whenever there are problems, they are quickly repaired, and all parts of the system that are still functional should not be affected by the repair process. Protocols that are able to recover from arbitrary states are also known as *self-stabilizing* protocols.

Since the seminal paper of Dijkstra in 1974 [4], self-stabilizing protocols have been investigated for many classical problems including leader election, consensus, matching,

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

clock synchronization and token distribution problems. Recently, also various protocols for self-stabilizing overlay networks have been proposed (e.g., [14, 9, 6, 10, 5, 1, 11, 12, 2]). However, for all of these protocols it is only known that they *eventually* converge to the desired solution, but the convergence process is not necessarily *monotonic*. In other words, it is not ensured for two points in time t, t' with $t < t'$ that the functionality of the topology at time t' is better than the functionality at time t.

In this paper, we focus on protocols for self-stabilizing overlay networks that guarantee the *monotonic* preservation of a characteristic that we call *searchability*, i.e., 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. Searchability is a useful and natural characteristic for an overlay network since searching for other participants is one of the most common tasks in real-world networks. Moreover, a protocol that preserves monotonic searchability has the huge advantage that in every state, even if the self-stabilization process has not converged yet, the already built topology can already be used for search requests.

As a starting point for rigorous research on monotonic searchability, we will focus on building a self-stabilizing protocol that preserves monotonic searchability for the line graph. Although the topology itself is fairly simple, to preserve searchability during the self-stabilization process turns out to be quite challenging. Additionally, we study monotonic searchability for the line graph if the node set is dynamic, i.e., nodes are allowed to leave the network.

1.1 Model

We consider a distributed system consisting of a fixed set of nodes in which each node has a unique reference and a unique immutable numerical identifier (or short id). The system is controlled by a protocol that specifies the variables and actions that are available in each node. In addition to the protocol-based variables there is a system-based variable for each node called *channel* whose values are sets of messages. We denote the channel of node u as $u.Ch$ and $u.Ch$ contains all incoming messages to u. Its message capacity is unbounded and messages never get lost. A node can add a message to $u.Ch$ if it has a reference to u. Besides these channels there are no further communication means, so only point-to-point communication is possible.

There are two types of actions. The first type of *action* has the form of a standard procedure $\langle label\rangle(\langle parameters\rangle)$: $\langle command\rangle$, where *label* is the unique name of that action, *parameters* specifies the parameter list of the action, and *command* specifies the statements to be executed when calling that action. Such actions can be called remotely. In fact, we assume that every message must be of the form $\langle label\rangle(\langle parameters\rangle)$ where *label* specifies the action to be called in the receiving node and *parameters* contains the parameters to be passed to that action call. All other messages will be ignored by the nodes. Apart from being triggered by messages, these actions may also be called locally by the nodes, which causes their immediate execution. The second type of action 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. We call an action whose guard is simply **true** a *timeout* action.

The *system state* is an assignment of a value to every variable of each node and messages to each channel. An action in some node p is *enabled* in some system state if its guard evaluates to **true**, or if there is a message in $p.Ch$ requesting to call it. In the latter case the corresponding message is processed (in which case it is removed from $p.Ch$). An action is *disabled* otherwise. Receiving and processing a message is considered as an atomic step.

A *computation* is an infinite fair sequence of system states such that for each state s_i, the next state s_{i+1} is obtained by executing an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. We assume *weakly fair action execution* and *fair message receipt*. Weakly fair action execution means that if an action is enabled in all but finitely many states of the computation, then this action is executed infinitely often. Note that the timeout action of a node is executed infinitely often. Fair message receipt means that if the computation contains a state where there is a message in a channel of a node that enables an action in that node, then that action is eventually executed with the parameters of that message, i.e., the message is eventually processed. Besides these fairness assumptions, we place no bounds on message propagation delay or relative nodes execution speeds, i.e., we allow fully asynchronous computations and non-FIFO message delivery. A *computation suffix* is a sequence of computation states past a particular state of this computation. In other words, the suffix of the computation is obtained by removing the initial state and finitely many subsequent states. Note that a computation suffix is also a computation.

We consider protocols that do not manipulate the internals of node references. Specifically, a protocol is *compare-store-send* if the only operations that it executes on node references is comparing them, storing them in local memory and sending them in a message. That is, operations on references such as addition, radix computation, hashing, etc. are not used. In a compare-store-send protocol, if a node does not store a reference in its local memory, the node may learn this reference only by receiving it in a message. A compare-store-send protocol cannot introduce new references to the system. It can only operate on the references that are already there.

The overlay network of a set of nodes is determined by their knowledge of each other. We say that there is a (directed) *edge* from a to b, denoted by (a, b), if node a stores a reference of b in its local memory or has a message in $a.Ch$ carrying the reference of b. In the former case, the edge is called *explicit* (drawn solid in figures), and in the latter case, the edge is called *implicit* (drawn dashed). With NG we denote the directed *network (multi-)graph* given by the explicit and implicit edges. ENG is the subgraph of NG induced by only the explicit edges. A *weakly connected component* of a directed graph G is a subgraph of G of maximum size so that for any two nodes u and v in that subgraph there is a (not necessarily directed) path from u to v. Two nodes that are not in the same weakly connected component are *disconnected*. We say a node a is to the *left* (*right*, respectively) of a node b if $id(a) < id(b)$ ($id(a) > id(b)$). If there is an edge (a, b) between the two, then a is a *left neighbor* (*right neighbor*). For three nodes a, b, c with $id(a) < id(b), id(a) < id(c)$ (or $id(a) > id(b), id(a) > id(c)$, respectively), we say a node b is *closer* to a than c, if $|id(a) - id(b)| < |id(a) - id(c)|$. If it is clear from the context we sometimes refer to the identifier of a node by dropping the id notation to , e.g., we write $a < b$ instead of $id(a) < id(b)$.

In this paper we are particularly concerned with search requests, i.e., $\textsc{Search}(v, destID)$ messages that are routed along ENG according to a given routing protocol, where v is the sender of the message and $destID$ is the identifier of a node we are looking for. Note that $destID$ does not necessarily belong to an existing node w, since we also want to model search requests to not existing nodes. If a $\textsc{Search}(v, destID)$ message reaches a node w with $id(w) = destID$, the search request *succeeds*; if the message reaches some node u with $id(u) \neq destID$ and cannot be forwarded anymore according to the given routing protocol, the search request *fails*. We assume that nodes themselves initiate $\textsc{Search}()$ requests at will. Therefore, the $\textsc{Search}(destID)$ action is never explicitly called.

We need some additional notation for our results of Section 4, in which we extend the protocol to handle nodes that want to leave the system. A node u has a variable *mode* $\in$ {leaving, staying} that is read-only. If this variable is set to **leaving**, the node is *leaving*; the node is *staying* if the variable is set to **staying**. Note that staying nodes can dynamically decide at any arbitrary state if they want to leave the system by executing a corresponding *leave action*. However, a leaving node cannot switch back to staying. The ultimate goal of a leaving node is to depart from the system. There is one special command that is important for the study of leaving nodes: **exit**. If a node executes **exit** it enters a designated *exit state* and all remaining edges to or from that node are deleted. We call such a node *gone*. A node that is not gone is called *present*. For a gone node all actions are disabled, in particular it will not execute the timeout action regularly.

1.2 Problem Statement

A protocol is *self-stabilizing* if it satisfies the following two properties.

Convergence: starting from an arbitrary system state, the protocol is guaranteed to arrive at a legitimate state.

Closure: starting from a legitimate state the protocol remains in legitimate states thereafter.

A self-stabilizing protocol is thus able to recover from transient faults regardless of their nature. Moreover, a self-stabilizing protocol does not have to be initialized as it eventually starts to behave correctly regardless of its initial state. In *topological self-stabilization* we allow self-stabilizing protocols to perform changes to the overlay network, resp. NG. A legitimate state may then include a particular graph topology or a family of graph topologies.

In this paper we want to build a self-stabilizing protocol for the *linearization problem*, i.e., the nodes are sorted by identifiers and each node stores only two references: its closest successor and its closest predecessor. From a global point of view, the nodes build a *line graph* topology. Of course, searching is easy once a legitimate state has been reached. However, searching reliably during the stabilization phase is much more involved. We say a (self-stabilizing) protocol satisfies *monotonic searchability* according to some routing protocol R if it holds for any pair of nodes v, w that once a $\textsc{Search}(v, id(w))$ request (that is routed according to R) initiated at time t succeeds, any $\textsc{Search}(v, id(w))$ request initiated at a time $t' > t$ will succeed. We do not mention R if it is clear from the context. A protocol is said to satisfy *non-trivial* monotonic searchability if it satisfies monotonic searchability and in every computation of the protocol there is a suffix such that for each pair of nodes v, w for which there is a path from v to w in the target topology $\textsc{Search}(v, id(w))$ requests will succeed.

Furthermore, we give a self-stabilizing protocol that satisfies non-trivial monotonic searchability, solves the linearization problem and solves the *Finite Departure Problem* of [7]. The following problem statement is adapted from [13]:

Finite Departure Problem ($\mathcal{FDP}$): In case the **exit** command is available, eventually reach a system state in which (i) every staying node is awake, (ii) every leaving node is gone and (iii) for each weakly connected component of the initial network graph, the staying nodes in that component still form a weakly connected component.

Consequently, a leaving node u should *safely* execute **exit**, i.e., the removal of u and its incident edges from NG does not disconnect any present nodes and does not violate searchability.

1.3 Related work

The idea of self-stabilization in distributed computing was introduced in a classical paper by E. W. Dijkstra in 1974 [4], in which he looked at 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. [17, 15, 8]. Over the years more and more network topologies were considered, ranging from skip lists and skip graphs [14, 9], to expanders [6], Delaunay graphs [10], hypertrees and double-headed radix trees [5, 1], small-world graphs [11] and a Chord variant [12]. Also a universal algorithm for topological self-stabilization is known [2].

Close to our work is the notion of *monotonic convergence* by Yamauchi and Tixeuil [18]. 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 classic distributed problems (e.g., leader election and vertex coloring) and focus on the amount of non-local information that is needed for them.

Our study of the *Finite Departure Problem* is heavily inspired by [7], in which the authors propose the aforementioned problem to study graceful departures of nodes in a self-stabilizing setting, i.e., nodes that want to leave a distributed system should decide when they can leave without affecting weak connectivity of the topology. They conclude that in general it is not possible to solve the $\mathcal{FDP}$. However, with the use of distributed oracles (which are specialized failure detectors [3]) the authors propose a protocol that solves the problem and arranges the nodes in a line. Additionally, they can show that oracles are not needed if the problem is transformed into a non-decision variant. In [13] the idea is generalized to a protocol framework that solves the $\mathcal{FDP}$ without being reliant on a certain topology and is thereby combinable with most existing overlay protocols.

1.4 Our contribution

To the best our knowledge, this paper presents the first attempt to have stricter requirements towards the self-stabilization process in topological self-stabilization. We define and study *monotonic searchability*, which captures a typical use case for overlay networks, i.e., searching other nodes. More formally, we want to guarantee for a self-stabilizing topology that 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 focus on studying non-trivial monotonic searchability for the list topology. First, we show that in general it is impossible to provide non-trivial monotonic searchability from any initial system state, due to the presence of certain initial messages. This justifies to study searchability only for so-called *admissible system states* in which these messages are not present anymore, as long as the protocol gurantees convergence to these states. We give a self-stabilizing list protocol and an appropriate search protocol that achieve the desired goal and prove their correctness. Moreover, we broaden the elaborateness of the problem statement, by allowing nodes to leave the line topology, i.e., solving the Finite Departure Problem in addition to the aforementioned problems. Also for this combination of problems we present suitable protocols and prove their correctness.

2 Preliminaries

Since gone nodes will never execute any action, we only consider initial states in which all nodes are present. We also restrict the initial state to contain only a finite number of messages that can trigger actions specified by our protocol, since other messages are ignored by the nodes. Finally, we do not allow the presence of references that do not belong to a node in the system. From now on, an initial system state satisfies all of these constraints.

The following propositions are restatements of results in [14] and imply further necessary conditions on initial system states.

1. If a compare-store-send program solves the linearization problem, each computation starts in a weakly connected initial state.
2. If a compare-store-send program solves the linearization problem, each computation starts in a state in which all references belong to present nodes.

A *message invariant* is a predicate of the following form: If there is a message m in the incoming channel of a node, then a predicate P' must hold. A protocol may specify one or more message invariants. An arbitrary message m in a system is called *corrupted* if the existence of m violates one of the message invariants. A state s is called *admissible* if there are no corrupted messages in s. We say a protocol *admissible-message satisfies* a property if the following two conditions hold: (i) in computations in which every state is admissible, it satisfies the property, and (ii) starting from any initial state, there is a computation suffix in which every state is admissible. A protocol *unconditionally satisfies* a property if it satisfies this property starting from any state.

With this notion in mind, we can show that admissible-message satisfaction is necessary for non-trivial monotonic searchability for any routing algorithm R.

▶ **Lemma 1.** *If a compare-store-send self-stabilizing protocol satisfies non-trivial monotonic searchability then this protocol must be admissible-message satisfying.*

The structure of the proof is as follows: we consider an arbitrary unconditionally satisfying protocol and show that it does not satisfy monotonic searchability by creating a bad instance for this protocol. In particular, we exploit that our model does not ensure FIFO delivery of messages. The proof can be found in the full version of this paper [16].

Consequently, to prove non-trivial monotonic searchability for a protocol (according to a given routing protocol R) it is sufficient to show that: (i) the protocol has a computation suffix in which every state is admissible and (ii) the protocol guarantees non-trivial monotonic searchability according to R in admissible states.

For the $\mathcal{FDP}$, it was shown in [7], there is no distributed protocol within our model that can decide when it is safe for a node u to leave the system and thereby solve the $\mathcal{FDP}$. The authors circumvent this impossibility result with the help of oracles. In general, an *oracle* is a predicate that depends on the current system state and the node calling it. In the context of the $\mathcal{FDP}$, an oracle is supposed to advise a leaving node when it is safe to execute **exit**. We use the oracle $\mathcal{NIDEC}$ as introduced in [7] in order to solve the $\mathcal{FDP}$. $\mathcal{NIDEC}$ evaluates to **true** for a node u calling it, if no node $v \neq u$ has a reference to u in its local memory or in a message in $v.Ch$ and if $u.Ch$ is empty. For an in depth discussion of oracles for the $\mathcal{FDP}$, we refer the reader to [7, 13].

3 The Build-List+ and the Search+ protocols

In this section, we present the Build-List+ protocol and the Search+ protocol. Build-List+ solves the linearization problem and is admissible-message satisfying non-trivial monotonic searchability according to Search+. Note that any protocol satisfying non-trivial monotonic searchability must be admissible-message satisfying as shown in Section 2. This section is organized as follows: First, we describe Build-List+ and Search+ in detail (Subsection 3.1). Then, we prove that the Build-List+ protocol solves the linearization problem (Subsection 3.2). Last, we prove that the Build-List+ protocol satisfies non-trivial monotonic searchability according to SEARCH+ (Subsection 3.3). From now on we drop the "according to SEARCH+" clause, since we only consider searchability for SEARCH+.

3.1 Description of Build-List+ and Search+

The BUILD-LIST+ Protocol builds upon the protocol introduced in [15] that solves the linearization problem. For this protocol, every node only keeps a single left and right neighbor. If a node u receives a reference of a node v with $u < v$ ($u > v$, respectively), u either saves v as its new right (left) neighbor if v is closer to u than the current right (left) neighbor w and delegates the reference of w to v or (in case v is not closer), v is not saved and delegated to w. Here, *delegation* means that the reference of a node is sent in a message to another node and not kept in the local memory. A natural (local) search protocol for this topology is to always forward search requests to the neighbor closest to the desired target node, or to abort the search request in case no such neighbor exists. Note that these easy and elegant protocols cannot guarantee monotonic searchability due to three simple facts: (i) due to delegation, it is possible that an explicit edge (u, v) is replaced by an explicit edge (u, w) and an implicit edge (w, v), (ii) consequently, u, v are not in the same weakly connected component in ENG (even though they were before delegation) and (iii) searchability is defined for ENG.

The BUILD-LIST+ protocol introduces the following changes in order to satisfy monotonic searchability: Instead of having a single left and right neighbor, a node u has sets of neighbors *Left* and *Right* (that it sorts implicitly according to id). In the following, whenever we use the notation $Left(u)/Right(u)$, we refer to these sets of a node u. The main principle that we use is that every node w does not delegate any edge to a node v stored in $Left(w)$ or $Right(w)$ directly. Instead it first introduces (using INTRODUCE(v, w)) this node to another node u, waits for an acknowledgement that the edge has been added to $Left(u)$ or $Right(u)$ (which is basically the LINEARIZE(v) message) and then delegates the edge to a node closer to v (using TEMPDELEGATE(v)). More specifically, whenever a node u has multiple neighbors to one side, it does not delegate edges to the closest neighbor directly, but does the following. W.l.o.g. assume that it has multiple neighbors $w_1, \dots, w_\ell$ to the right with $id(w_i) < id(w_{i+1})$. In the TIMEOUT action u introduces w_i to w_{i-1}, with an INTRODUCE(w_i, u) message. Thereby, w_{i-1} knows that it got the reference from u, saves the reference to w_i directly, sends a LINEARIZE(w_i) message back to u and a TEMPDELEGATE(u) to itself (the latter is only to preserve connectivity). Node u can now react to that LINEARIZE(w_i) message, by deleting w_i from its memory and sending the reference to the closest node to the left of w_i in *Right* (which is not necessarily w_{i-1} anymore). Thereby, u preserves a path of explicit edges between u and w_i. Additionally, u sends its own reference to the closest neighbors with a INTRODUCE$(u, \perp)$ message who turn this into a TEMPDELEGATE(u) message. In general, the TEMPDELEGATE(u) action is used to delegate an implicit edge to a node u into one direction (i.e., to the left or to the right) as long as there is a node between the current node and u in *Left* or *Right*. Note that implicit edges are not used for search, thus we do not have to apply the principle of introducing first and delegating afterwards for this kind of edges. However, we have to delegate in order to preserve connectivity and to stabilize to the line eventually. Note that, even though a node has temporarily more references than necessary for the final line topology our protocol still eventually stabilizes to the line, as we will show later. The pseudocode for all BUILD-LIST+ actions is given in Listing 1. Note that a node refers to itself with the expression $self$. Additionally, keep in mind that the timeout action is the only action that is not triggered as a result of another action. Instead, is triggered regularly.

Listing 1 BUILD-LIST+ protocol

```
TIMEOUT
 for all $destID \in Waiting$
   send $forwardProbe(self, destID, \{self\}, self.seq)$ to $self$
 //Let $Left = \{v_1, v_2, \dots, v_k\}$ with $id(v_1) < id(v_2) < \cdots < id(v_k)$
 for all $v_i \in Left$ with $1 \leq i < k$
   send INTRODUCE($v_i, self$) to $v_{i+1}$
 //Let $Right = \{w_1, w_2, \dots, w_l\}$ with $id(w_1) < id(w_2) < \cdots < id(w_l)$
 for all $w_i \in Right$ with $1 < i \leq l$
   send INTRODUCE($w_i, self$) to $w_{i-1}$
 send INTRODUCE($self, \perp$) to $v_1$
 send INTRODUCE($self, \perp$) to $w_1$

INTRODUCE($v, w$)
 if($id(v) < id(self)$)
   if{$w \neq \perp$}
     $Left \leftarrow Left \cup \{v\}$
     send LINEARIZE($v$) to $w$
     send TEMPDELEGATE($w$) to $self$
   else //$w = \perp$
     send TEMPDELEGATE($v$) to $self$
 else if($id(v) > id(self)$)
   //Analogous to the previous case.

LINEARIZE($v$)
 send TEMPDELEGATE($v$) to $self$
 if($id(v) < id(self)$)
   if($Left \neq \emptyset$)
     $x \leftarrow argmax\{id(x')|x' \in Left\}$
     if($v \neq x$)
       $w \leftarrow argmin\{id(w')|w' \in Left\ und\ id(w') > id(v)\}$
       $Left \leftarrow Left \setminus \{v\}$
       send TEMPDELEGATE($v$) to $w$
 else if($id(v) > id(self)$)
   //Analogous to the previous case.

TEMPDELEGATE($u$)
 if($id(u) < id(self)$)
   if($Left = \emptyset$)
     $Left \leftarrow Left \cup \{u\}$
   else //$Left \neq \emptyset$
     $x \leftarrow argmax\{id(x')|x' \in Left\}$
     if($id(x) < id(u)$)
       $Left \leftarrow Left \cup \{u\}$
     else if($id(x) > id(u)$)
       send TEMPDELEGATE($u$) to $x$
 else if{$id(u) > id(self)$}
   //Analogous to the previous case.
```

The SEARCH+ protocol works as follows: Whenever the INITIATENEWSEARCH($destID$) action is called at a node u, u creates a new SEARCH($u, destID$) message and starts to periodically initiate FORWARDPROBE($u, destID, \{u\}, self.seq$) messages that it sends to itself. In the following, assume $id(u) < destID$ (the other case is analogous). Each FORWARDPROBE() message has a set of nodes, called $Next$ attached to it, which contains the nodes the message will visit in its future. It also has a counter seq attached to it whose meaning we will explain later. Whenever a FORWARDPROBE($u, destID, Next, seq$) message is at a node w, w removes itself from $Next$ and adds all its right neighbors x with $id(x) \leq destID$ to $Next$. Then it forwards the FORWARDPROBE($u, destID, Next, seq$) message to the node with minimal id in $Next$. If a FORWARDPROBE($u, destID, Next, seq$) message arrives at a node v with $id(v) = destID$, it directly responds with a PROBESUCCESS($destID, seq, v$) message to u. However, if $Next$ is empty at a node w with $id(w) \neq destID$ after w has added the aforementioned right neighbors, the FORWARDPROBE() message is answered with a PROBEFAIL($destID, seq$) message. In any case, as soon as u receives the response, it acts accordingly: If the answer to a FORWARDPROBE($u, destID, Next, seq$) message is a PROBEFAIL($destID, seq$) message, it drops the corresponding SEARCH($u, destID$) message completely. If the answer is PROBESUCCESS($destID, v$), SEARCH($u, destID$) messages waiting at u are directly sent to v.

Note that if additional SEARCH($u, destID$) messages are created at u while u is still waiting for an answer to an earlier initiated FORWARDPROBE($u, destID$), these requests simply wait together with the previous request (realized by simple $WaitingFor[destID]$ field) and are aborted or sent as soon as the PROBEFAIL($destID$) or PROBESUCCESS($destID, v$) response arrives at u, (i.e., search requests to the same destination are sent out in batches if possible). Furthermore, note that nodes do not memorize whether they have already sent FORWARDPROBE() messages to a certain destination. Due to corrupt initial states, this knowledge could be wrong and nodes relying on this knowledge would wait forever. Therefore, nodes periodically send FORWARDPROBE() messages, instead of only once. Note that because we make no assumptions on the message delivery speed and channels are not FIFO, it is possible that PROBEFAIL() messages arrive at a node u that are answers to FORWARDPROBE() messages initiated long ago. However, in the meantime, there might have been successful responses. To deal with this, each node u stores a sequence number counter seq. Whenever INITIATENEWSEARCH($destID$) is executed by u and there is no SEARCH($u, destID$) that waits for an answer to a FORWARDPROBE($u, destID, Next, seq$) message, u increments $u.seq$, stores the new $u.seq$ value in an entry for v and always attaches the current sequence number ($u.seq$) to each FORWARDPROBE() message u sends. Responses to probes (success and failure) sent by u also contain this sequence number. Whenever a response is sent back to u, u checks whether the sequence number in this message is at least the sequence number stored for $destID$. If not, it simply drops the message, since in that case, the answer belongs to a FORWARDPROBE() message sent for an earlier batch of SEARCH($u, destID$) messages that have already been processed. The complete pseudocode for SEARCH+ is given in Listing 2.

In order to not unnecessarily blow up the pseudocode, we intentionally left out a sanity check for each node, i.e., before executing each action, each node u makes sure that $Left$ only contains nodes v with $v < u$ and that $Right$ only contains nodes v with $u < v$. If this is not the case for some node v, u rearranges the reference to v accordingly. This way, in every computation, the following lemma holds:

▶ **Lemma 2.** *For every node v it holds: For all $x \in Left$, $id(x) < id(v)$, and for all $y \in Right$, $id(v) < id(y)$.*

Listing 2 SEARCH+ protocol

```
InitiateNewSearch($destID$)
 create new message $m = $ Search($self, destID$)
 if ( $WaitingFor[destID] = \emptyset$ )
    $WaitingFor[destID] \leftarrow \{\}$
    $self.seq \leftarrow self.seq + 1$
    $seq[destID] \leftarrow self.seq$
 //Store the messages to $WaitingFor$
 $WaitingFor[destID] \leftarrow WaitingFor[destID] \cup \{m\}$

ForwardProbe($source, destID, Next, seq$)
 if ( $destID = id(self)$ )
    if ( $Next \neq \emptyset$ )
       for all $u \in Next$
          send TempDelegate($u$) to $self$
    send ProbeSuccess($destID, seq, self$) to $source$
    send TempDelegate($source$) to $self$
 else //$destID \neq id(self)$
    if ( $destID > id(self)$ )
       $Next \leftarrow Next \setminus \{self\} \cup \{w \in Right | id(w) \leq destID\}$
       if ( $Next = \emptyset$ )
          send ProbeFail($destID, seq$) to $source$
          send TempDelegate($source$) to $self$
       else //$Next \neq \emptyset$
          $u \leftarrow argmin\{id(u) | u \in Next\}$
          if ( $id(u) < id(self)$ )
             send TempDelegate($u$) to $self$
          else if ( $id(u) < id(argmin\{id(v) | v \in Right\})$ )
             $Right \leftarrow Right \cup \{u\}$
          send ForwardProbe($source, destID, Next, seq$) to $u$
    else if ( $destID < id(self)$ )
       //Analogous to the previous case.

ProbeSuccess($destID, seq, dest$)
 if ( $seq \geq seq[destID]$ )
    /* The message belongs to currently
     * stored search requests to $dest$. */
    send all $m \in WaitingFor[destID]$ to $dest$
    $WaitingFor[destID] \leftarrow \emptyset$
 send TempDelegate($dest$) to $self$

ProbeFail($destID, seq$)
 if ( $seq \geq seq[destID]$ )
    /* The message belongs to currently
     * stored search requests to $dest$. */
    $WaitingFor[destID] \leftarrow \emptyset$
```

3.2 Build-List+ solves the linearization problem

In this section, we prove the following theorem:

▶ **Theorem 3.** Build-List+ *is a self-stabilizing solution to the linearization problem.*

We prove the theorem in three steps: First, we show that starting from any initial state in which NG is weakly connected, NG will always be weakly connected. Second, we show that starting from any initial state, there will be a state in which ENG will be a supergraph of the line graph and that the explicit edges corresponding to the line will never be removed. Third, we prove that all superfluous explicit edges will eventually vanish. Note that all proofs that are omitted in this section can be found in the full version [16].

The first step is represented by the following lemma:

▶ **Lemma 4.** *If a computation of* Build-List+ *starts from a state where NG is weakly connected then in every state, NG remains weakly connected.*

For the second step of the proof of the theorem, we introduce the notation $\mathit{nextLeft}(u) := \mathit{argmax}\{id(v)|v \in \mathit{Left}(u)\}$ and $\mathit{nextRight}(u) := \mathit{argmin}\{id(v)|v \in \mathit{Right}(u)\}$. Furthermore, let $\mathit{length}(u, v)$ for two nodes u and v denote the hop distance in the (ideal) line topology between u and v. We define $rv(v)$ for a node v as $\mathit{length}(v, \mathit{nextRight}(v))$ if $\mathit{Right}(v) \neq \emptyset$ or as n if $\mathit{Right}(v) = \emptyset$; we define $lv(v)$ analogously for $\mathit{nextLeft}(v)$. With this, we define a potential function $\Phi := \sum_{i=1}^{n-1} rv(v_i) + \sum_{i=2}^{n} lv(v_i)$ where $v_1 < v_2 < \cdots < v_n$ are all nodes ordered by their id increasingly. Notice that Φ is bounded from above by $2n(n-1)$ and from below by $2(n-1)$. Also notice that according to the protocol, $\mathit{nextLeft}(v)$ ($\mathit{nextRight}(v)$) can only change if v puts a node closer to v than $\mathit{nextLeft}(v)$ ($\mathit{nextRight}(v)$) into Left (Right). Thus, Φ never increases. We define the *closest neighbor graph* as the graph $G_{NB} = (V, E_{NB})$ where V is the set of all nodes and $(x, y) \in E_{NB}$ iff $y = \mathit{nextRight}(x) \vee y = \mathit{nextLeft}(x)$. Furthermore, we say an edge is *temporary* if it is an implicit edge due to a TempDelegate() message. All other types of implicit edges are called *non-temporary*. One can show the following:

▶ **Lemma 5.** *Assume there is a system state such that Φ does not decrease in any further step of the computation. Then G_{NB} is bidirected and strongly connected.*

We prove this lemma step-by-step, starting with the following lemma:

▶ **Lemma 6.** *Assume a system state such that Φ does not decrease in any further step of the computation. Then G_{NB} is bidirected.*

The definition of a closest neighbor graph and Lemma 2 imply the following:

▶ **Corollary 7.** *If G_{NB} is bidirected and disconnected, every connected component forms a line.*

To show that G_{NB} is also strongly connected, we need two additional lemmata. We start with the following:

▶ **Lemma 8.** *Assume that in a state of the computation of* Build-List+ *G_{NB} is bidirected and disconnected. If there is a non-temporary edge (w, v) with $w \in C_1, v \notin C_1$ for a connected component C_1, then eventually either there will be an explicit or a temporary edge (x, y) with $x \in C_1$ and $y \notin C_1$ or Φ will decrease.*

▶ **Lemma 9.** *Assume that in a state of the computation of* Build-List+ G_{NB} *is bidirected and disconnected. If there is an explicit or a temporary edge* (w, v) *with* $w \in C_1$ *and* $v \notin C_1$ *for a connected component* C_1*, then eventually there will be an explicit or temporary edge* (x, y) *with* $x \in C_1, y \notin C_1$ *and* $length(x, y) < length(w, v)$*, or* Φ *will decrease.*

We are now ready to prove Lemma 5:

Proof. Assume there is an initial state in which Φ does not decrease anymore. Furthermore, assume that the closest neighbor graph G_{NB} is disconnected. Firstly, Lemma 6 guarantees that G_{NB} is bidirected. Furthermore, by Lemma 4, there must be at least one (implicit or explicit) edge (w, v) between a connected component C_1 and another connected component. Together with Lemma 8 this implies that at some point there must be a temporary or explicit edge (x, y) with $x \in C_1$ and $y \notin C_1$. However, then Lemma 9 can be applied. Since there is only a finite number of times that there can be a shorter edge, at some state, Φ must decrease, yielding a contradiction. Thus G_{NB} must be weakly connected. Note that Lemma 6 implies that G_{NB} is also strongly connected, yielding the claim of Lemma 5. ◀

Note that since Φ can never increase and since Φ is bounded from below, Φ can only decrease for a finite number of states. After that, the conditions of Lemma 5 are fulfilled. This lemma and Corollary 7 imply the following corollary:

▶ **Corollary 10.** *For any computation of* Build-List+*, there is a state in which the graph formed by the explicit edges is a supergraph of the line topology.*

For the third step of the proof of the theorem, we have the following lemma:

▶ **Lemma 11.** *If a computation of* Build-List+ *contains a state in which* ENG *is a supergraph of the line topology, then there will be a suffix in which* ENG *is the line topology and no new explicit edges will ever be created again.*

Note that Corollary 10 and Lemma 11 imply that Build-List+ converges to the list. Moreover, Lemma 11 yields the closure property. This finishes the proof of Theorem 3.

3.3 Build-List+ satisfies non-trivial monotonic searchability

In this subsection we prove the following theorem:

▶ **Theorem 12.** Build-List+ *admissible-message satisfies non-trivial monotonic searchability according to* Search+*.*

Note that all proofs that are omitted in this section can be found in the full version [16].

We start with some preliminaries. First we define $R(v)$ as the set of all nodes x with $id(v) < id(x)$ for which there is a directed path from v to x consisting solely of explicit edges (y, z) with $id(y) < id(z)$. Furthermore, we define $R(v, ID) := \{x \in R(v) | id(x) \leq ID\}$. In addition, we define $L(v)$ as the set of all nodes x with $id(x) < id(v)$ for which there is a directed path from v to x consisting solely of explicit edges (y, z) with $id(z) < id(y)$. For a set U, $R(U) := U \cup \bigcup_{u \in U} R(u)$ and $R(U, ID) := \{x \in R(U) | id(x) \leq ID\}$. Accordingly, $L(U) := U \cup \bigcup_{u \in U} L(u)$ and $L(U, ID) := \{x \in L(U) | id(x) \geq ID\}$.

Moreover, we define a state as admissible if the following message invariants hold:

1. If there is an Introduce(v, w) message with $w \neq \perp$ in $u.Ch$, then $v \neq w$, and $u \in R(w)$ (or $u \in L(w)$).
2. If there is a Linearize(v) message in $w.Ch$, then there is a node $u \neq v$ with $u \in Right(w)$ and $v \in R(u)$ if $w < v$ (or $u \in Left(w)$ and $v \in L(u)$ if $v < w$).

3. If there is a ForwardProbe$(source, destID, Next, seq)$ message in $u.Ch$, then
  a. $id(source) < destID$ and $\forall x \in Next : id(x) \geq id(u)$ and $u = argmin_u\{id(u)|u \in Next\}$ (alternatively $destID < id(source)$ and $\forall x \in Next : id(x) \leq id(u)$ and $u = argmax_u\{id(u)|u \in Next\}$).
  b. $id(source) < destID$ and $R(next) \subseteq R(source)$ (or $destID < id(source)$ and $u \in L(source)$).
  c. if v exists such that $id(v) = destID$ and $id(source) < destID$ and $v \notin R(Next, destID)$ (or $id(source) < destID$ and $v \notin L(Next, destID)$) then for every admissible state with $source.seq[destID] < seq$, $v \notin R(source, destID)$ ($v \notin L(source, destID)$).

4. If there is a ProbeSuccess$(destID, seq, dest)$ message in $u.Ch$, then $id(dest) = destID$ and $dest \in R(u)$ if $destID > id(u)$ (or $dest \in L(u)$ if $destID < id(u)$).

5. If there is a ProbeFail$(destID, seq)$ message in $u.Ch$, then either there is no node with id $destID$, or for every admissible state with $u.seq[destID] < seq$, $v \notin R(u)$ (and $v \notin L(u)$), where v such that $id(v) = destID$.

6. If there is a Search$(v, destID)$ message in $u.Ch$, then $id(u) = destID$ and $u \in R(v)$ if $id(v) < destID$ (or $u \in L(v)$ if $destID < id(v)$).

One can show the following Lemma 13 and Lemma 14 which together imply Corollary 15.

▶ **Lemma 13.** *If in a computation of* Build-List+*, there is an admissible state, then all subsequent states are admissible.*

▶ **Lemma 14.** *In every computation of* Build-List+ *there is an admissible state.*

▶ **Corollary 15.** *In every computation of* Build-List+*, there exists a suffix in which every state is admissible.*

For the rest of this subsection, we assume that every computation starts in an admissible state, since we want to show monotonic searchability must hold starting from admissible states only. Furthermore, w.l.o.g., we only consider Search$(u, destID)$ messages with $id(u) < destID$.

Before we can prove Theorem 12, we need an additional result:

▶ **Lemma 16.** *For every message $m =$ ForwardProbe$(v, destID, Next, seq) \in u.Ch$ with $id(u) < destID$, it holds that if there is a node w with $id(w) = destID$ and $w \in R(u)$, then there will be a state with $m' =$ ForwardProbe$(v, destID, Next', seq) \in w.Ch$.*

We are now ready to prove Theorem 12:

Proof. Let m, m' be two Search$(u, destID)$ messages initiated in u in admissible states with m being initiated before m' and assume that m is delivered successfully, but m' is not. Let v be such that $id(v) = destID$. Note that if m' is added to the set $WaitingFor[destID]$ when m is already in the set, then the protocol will handle both messages identical, i.e., if m is successfully delivered to v due to an ProbeSuccess() message, m' is as well. Therefore, m' is added to $WaitingFor[destID]$ when $m \notin WaitingFor[destID]$, which implies $u.seq[destID]$ has increased since the successful delivery of m (according to the protocol). Since we assume that m' is not delivered successfully, either a ProbeFail$(dest, seq)$ message eventually arrives at u with $seq \geq u.s[destID]$, or no ProbeSuccess$(destID, seq, dest)$ with $seq \geq u.s[destID]$, $dest = destID$ will ever arrive at u. We consider both cases individually. In the first case, by the fifth invariant, $v \notin R(u)$ has to hold even though m was already successfully delivered. By the sixth invariant, when m was delivered, $v \in R(u)$, which is why this is a contradiction to Lemma **??**. In the second case, note that ForwardProbe$(u, destID, \{u\}, seq)$ messages are regularly initiated by u with $seq \geq$

$u.s[destID]$ (since $u.seq$ is monotonically increasing). Again, due to the successful delivery of m, by the sixth invariant and Lemma **??**, $v \in R(u)$ when m' was initiated, and therefore, by Lemma 16, a ForwardProbe($u, destID, Next', seq$) message with $seq \geq u.s[destID]$ will eventually be in $v.Ch$, which will be answered with a ProbeSuccess($destID, seq, v$) message, causing m' to be sent to v. By the fair message receipt assumption, this contradicts the assumption that m' is not successfully delivered. ◀

4 The Build-List* and the Search* protocols

For the Build-List+ protocol in Section 3 we implicitly assumed a static node set, i.e., nodes are not allowed to leave or join the network. In this section we want investigate monotonic searchability in terms of the *Finite Departure Problem* ($\mathcal{FDP}$) of [7]. Naturally, a leaving node does not execute InitiateNewSearch(), since it aims at leaving the system. Additionally, a leaving node that is the destination of a ForwardProbe() message, will deliberately answer with ProbeFail(). Consequently, monotonic searchability can only be maintained for pairs of staying nodes.

We note that the $\mathcal{FDP}$ deliberately ignores that new nodes can join the network. However, this abstraction is justified in a self-stabilizing setting, since from an algorithmic point of view for some node u a new node joining the network is the same as getting a message from a node that it has never been in contact with.

In this section, we present the Build-List* and the Search* protocols. In the full version [16], we further show that Build-List* solves the $\mathcal{FDP}$ and also the linearization problem, and extend the proofs of Section 3.3 to show that Build-List* also satisfies non-trivial monotonic searchability according to Search*.

4.1 Description of Build-List* and Search*

For two staying nodes that interact with each other, Build-List* is analogous to Build-List+. Therefore, we only specify the changes in case a node itself is leaving or receives a message from a leaving node. A leaving node distinguishes between two different kinds of neighbors: those that it already had before switching to the leaving mode (which are *Left* and *Right* from Build-List+) and those which it received while being leaving ($Temp_L$ and $Temp_R$). Searchability is only preserved for nodes in the former two sets.

For the ForwardProbe(), Introduce(), Linearize() and TempDelegate() actions, a leaving node u will always save nodes in $Temp_L$ and $Temp_R$ in cases where a staying node saves them in *Left* and *Right*. In its Timeout action, a leaving node u either introduces all its neighbors to each other and executes **exit** if $\mathcal{NIDEC}$ is true or it sends a ReverseAndLinearizeREQ() message to all neighbors. With this ReverseAndLinearizeREQ(dir) message u requests all neighbors to stop holding its reference. As it was shown in [7], leaving nodes should never send their own reference for a successful departure protocol. Therefore, a ReverseAndLinearizeREQ(dir) message only contains a value $dir \in \{left, right\}$ that indicates whether a left or right neighbor should be removed, i.e., u sends a ReverseAndLinearizeREQ(left) message to all its neighbors to the right and and a ReverseAndLinearizeREQ(right) message to all its neighbors to the left. If a node v receives a ReverseAndLinearizeREQ(dir) message, there are two possible scenarios. If v is staying, it sends a ReverseAndLinearizeACK(v,uniqueValue) message to all neighbors in the given direction, which contains its own reference and for each neighbor a uniquely created value (i.e., in our case a local counter or the *id* of a node would be sufficient). This values is also saved as satellite data by v at the corresponding node reference in the neighbor set.

If v is leaving, it behaves like a staying node if the *dir* is right; otherwise it ignores the request. Thereby, leaving nodes with a higher id are given a higher priority for exiting the system. Once a leaving node u receives a ReverseAndLinearizeACK(v,uniqueValue) message, it responds with ReverseAndLinearize(*nodeList*, *uniqueValue*) message that contains the received unique value (for identification purposes) and also all its neighbors that are on the opposite of the node in the message (i.e., if the received node is to the right of u, u sends all left neighbors and vice-versa). A ReverseAndLinearizeACK(v,uniqueValue) message is ignored by a staying node, meaning that it is transformed into a TempDelegate(v) to itself. Finally, the ReverseAndLinearize(*nodeList*, *uniqueValue*) message is received by v and v checks if it has a neighbor with the given unique value. If this is the case, v either finishes the reversal process by deleting the reference to u and saving the newly received neighbors (if v is staying or getting the ReverseAndLinearize(*nodeList*, *uniqueValue*) message from a right neighbor) or v ignores the message by simply saving all nodes in $Temp_L$ (if v is leaving and getting the ReverseAndLinearize(*nodeList*, *uniqueValue*) message from a left neighbor). In case the unique value does not match, the ReverseAndLinearize(*nodeList*, *uniqueValue*) message is not a response to a former ReverseAndLinearizeACK(v,uniqueValue) message and all received nodes are processed by TempDelegate() messages to v itself.

The Search* protocol is very similar to the Search+ protocol. As already mentioned, leaving nodes will neither execute InitiateNewSearch(), nor will they send out a ProbeSuccess() message. In fact the only action that is different in multiple places is the ForwardProbe() action, since we have to make sure that references are not saved in *Left* and *Right* but in $Temp_L$ and $Temp_R$.

Similar to Build-List+, Build-List* performs a sanity check for $Temp_L$, $Temp_R$, *Left* and *Right* before each action. The same is done for the *nodeList* received in a ReverseAndLinearize() message. However, in the last case a failing sanity check (i.e., the nodes in *nodeList* are from two different sides of the current node) directly implies that the message is corrupt and it is safe to process the nodes with TempDelegate(). The pseudocode for Build-List* and Search* can be found in the full version [16].

We have the following results regarding Build-List*:

▶ **Theorem 17.** Build-List* *is a self-stabilizing solution to the $\mathcal{FDP}$.*

▶ **Theorem 18.** Build-List* *is a self-stabilizing solution to the linearization problem.*

▶ **Theorem 19.** Build-List* *admissible-message satisfies non-trivial monotonic searchability according to* Search*.

The proofs of these theorems can be found in the full version [16].

5 Conclusion and Outlook

To the best of our knowledge, we presented the first protocol that self-stabilizes a topology whilst satisfying monotonic searchability. We focused on the line topology as a starting point and extended our protocol such that it additionally solves the Finite Departure Problem. In the design of our protocol, it turned out that the principle of delegating explicit edges only if they have been successfully introduced before is crucial to enable monotonic searchability. A natural open question is whether the application of this principle is sufficient for monotonic searchability. That is, does applying this principle to other protocols that stabilize a topology (e.g., rings, skip-graphs, Delaunay graphs) directly yield monotonic searchability, or do other topologies require more-specialized solutions?

References

1 James Aspnes and Yinghua Wu. O(logn)-time overlay network construction from graphs with out-degree 1. In *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, pages 286–300, 2007.

2 Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. *Theor. Comput. Sci.*, 512:2–14, 2013.

3 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

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

5 Shlomi Dolev and Ronen I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20(5):375–388, 2008.

6 Shlomi Dolev and Nir Tzachar. Spanders: Distributed spanning expanders. *Sci. Comput. Program.*, 78(5):544–555, 2013.

7 Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On stabilizing departures in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems – 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 – October 1, 2014. Proceedings*, pages 48–62, 2014.

8 Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A note on the parallel runtime of self-stabilizing graph linearization. *Theory Comput. Syst.*, 55(1):110–135, 2014. doi:10.1007/s00224-013-9504-x.

9 Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Skip$^+$: A self-stabilizing skip graph. *J. ACM*, 61(6):36:1–36:26, 2014. doi:10.1145/2629695.

10 Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. Towards higher-dimensional topological self-stabilization: A distributed algorithm for delaunay graphs. *Theor. Comput. Sci.*, 457:137–148, 2012.

11 Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. A self-stabilization process for small-world networks. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1261–1271, 2012.

12 Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: A self-stabilizing chord overlay network. *Theory Comput. Syst.*, 55(3):591–612, 2014. doi:10.1007/s00224-012-9431-2.

13 Andreas Koutsopoulos, Christian Scheideler, and Thim Strothmann. Towards a universal approach for the finite departure problem in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems – 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, pages 201–216, 2015.

14 Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. *Theor. Comput. Sci.*, 512:119–129, 2013. doi:10.1016/j.tcs.2012.08.029.

15 Melih Onus, Andréa W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*, 2007.

16 C. Scheideler, A. Setzer, and T. Strothmann. Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures (full version). *ArXiv e-prints*, December 2015. arXiv:1512.06593.

17 Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), 31 August – 2 September 2005, Konstanz, Germany*, pages 39–46, 2005.

18 Yukiko Yamauchi and Sébastien Tixeuil. Monotonic stabilization. In *Principles of Distributed Systems – 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 475–490, 2010.