A

Survey on Algorithms for Self-Stabilizing Overlay Networks[1]

Michael Feldmann[1] Christian Scheideler[1] Stefan Schmid[2]

[1] Paderborn University, Germany [2] Faculty of Computer Science, University of Vienna, Austria

The maintenance of efficient and robust overlay networks is one of the most fundamental and reoccurring themes in networking. This paper presents a survey of state-of-the-art algorithms to design and repair overlay networks in a distributed manner. In particular, we discuss basic algorithmic primitives to preserve connectivity, review algorithms for the fundamental problem of graph linearization, and then survey self-stabilizing algorithms for metric and scalable topologies. We also identify open problems and avenues for future research.

1. INTRODUCTION

Many distributed systems today rely on some kind of *overlay network* connecting the communicating nodes or "peers" of an application using *logical* links: each link corresponds to a path, potentially through many physical links, in the underlying network. The most prominent example are overlay networks over the Internet which allow to route messages according to logical addresses rather than IP addresses, introducing great flexibilities. Well-known overlay networks include Chord [59], Pastry [53], Tapestry [62], CAN [48], Kademlia [42], Viceroy [48], Koorde [28], SkipNet [23], among many others [43; 34; 4]. More recently, overlays are also used by content distribution providers such as Akamai [24], or in crypto-currency infrastructures (e.g., Bitcoin [47]), to improve scalability.

Overlay networks are often fairly transient and dynamic, i.e., nodes join and leave frequently. They hence require mechanisms to support changing memberships. Reasons for such dynamic membership include, e.g., the limited time window during which users are interested in contents shared in a peer-to-peer network, changing popularity of contents, diurnal patterns, failures, etc. Peer-to-peer systems are particularly dynamic as they are designed for open membership and are self-organizing. In general, with an increasing scale, distributed systems are likely to become more dynamic and have to deal with nodes continuously entering and leaving the system.

Besides efficiency, fault-tolerance is arguably one of the most important requirements of large-scale overlay networks. Overlay topologies are usually maintained by the nodes (a.k.a. peers) themselves.

[1]This work has been partially supported by the German Research Foundation (DFG) within the Collaborative Research Center 901 "On-The-Fly Computing" under the project number 160364472-SFB901.

Therefore, *distributed* algorithms are needed to maintain the overlay network and support joining, leaving, and routing between the nodes. These distributed algorithms should also be scalable, given the large size of many overlay networks. Furthermore, such algorithms cannot rely on the assumption that all peers leave the network gracefully, e.g., execute a pre-defined "leave protocol" before departure. Rather, many peers are likely to leave unexpectedly (e.g., crash). Furthermore, topological changes may also happen due to attacks: the larger and hence more popular the overlay network, the more attractive it also becomes for attackers. For example, adversarial nodes may join and leave the network strategically [54], to occupy strategic positions in the overlay or eclipse other nodes. Malicious nodes may further disconnect other nodes by overloading them with requests (denial-of-service attack).

It is hence difficult in practice to rely on certain invariants and assumptions on what can and what cannot happen during the (possibly very long) lifetime of an overlay. Accordingly, it is important that a distributed overlay network be able to *automatically* recover from unexpected or even arbitrary situations. This recovery should also be quick: once in an illegal state, the overlay network may be more vulnerable to further changes or attacks.

This motivates the study of *self-stabilizing* overlay networks. Self-stabilzation is a very powerful concept in fault-tolerance: self-stabilizing algorithms guarantee that in the absence of external influences, they reconverge to a desirable state from *any* initial state (known as the *convergence* property), and then preserves this state (known as the *closure* property). The notion of self-stabilization was first coined by E.W. Dijkstra in 1974 [12]. Leslie Lamport, in his ACM PODC 1983 keynote address, acknowledged self-stabilization as one of the most brillant concepts introduced by Dijkstra [35].

In general, the design of self-stabilizing algorithms is fairly well-understood today. Since Dijkstra's paper, self-stabilization has been studied in many contexts, including graph theory problems, termination detection, clock synchronization, and fault containment [13]. In the context of communication networks, many self-stabilizing algorithms exist, from spanning tree construction [46] to software-defined control [10]. In fact, already in the late 1980s, very powerful results have been obtained on how any synchronous, not fault-tolerant local network algorithm can be transformed into a very robust, self-stabilizing algorithm which performs well both in synchronous and asynchronous environments [5; 6; 36]. However, while these transformations are attractive to strengthen the robustness of local algorithms on a given network topology, e.g., for designing self-stabilizing spanning trees, they are not applicable, or only applicable at high costs, in overlay overlay networks where the topology is subject

to change and optimization itself. Indeed, many decentralized overlay networks (including well-known examples like Chord) are not self-stabilizing, in the sense that the proposed protocols only manage to recover the network from a restricted class of illegal states [1; 3; 59].

Informally, the self-stabilizing overlay network design problem is the following:

(1) An adversary can manipulate the peers' neighborhood (and hence topology) information arbitrarily. In particular, it can remove and add arbitrary nodes and links.
(2) As soon as the adversary stops manipulating the overlay topology, say at some unknown time t_0, the self-stabilization protocols will ensure that eventually, and in the absence of further adversarial changes, a desired topology is reached.

As any self-stabilizing algorithm, a topologically self-stabilizing algorithm must guarantee convergence and closure properties: by local neighborhood changes (i.e., by creating, forwarding, and deleting links with neighboring nodes), the nodes will eventually form an overlay topology with desirable properties (e.g., polylogarithmic degree and diameter) from any initial topology. The system will also stay in a desirable configuration provided that no further external topological changes occur.

We also note the basic fact that in order for a distributed self-stabilizing algorithm to recover any connected topology, the initial topology must at least be *weakly connected*. A directed graph $G = (V, E)$ is *weakly connected*, if the undirected version of G, namely $G' = (V, E')$ is connected, i.e., for two nodes $u, v \in V$ there is a path from u to v in G'.

In this paper, we present a survey on distributed algorithms for maintaining overlay networks. In contrast to the vast existing literature on overlay network designs and algorithms (e.g. [50; 41; 37]), our focus is on self-stabilizing algorithms. In Section 2, we first present a generic model which is useful to design and analyze topologically self-stabilizing algorithms. In Section 3, we discuss the basic primitives of manipulating neighborhoods while preserving connectivity, and discuss their application in the design of self-stabilizing algorithms. Section 4 presents self-stabilizing algorithms for a basic line topology, and we extend our study to metric graphs in Section 5. We discuss scalable topologies and in particular, expander graphs, in Section 6. We survey additional relevant aspects related to monotonic searchability and node departures in Section 7. In Section 8, we conclude and identify open problems.

2. A BASIC MODEL OF TOPOLOGICAL SELF-STABILIZATION

Let us introduce the basic and standard model for self-stabilizing overlay networks. The overlay network is represented as a directed graph $G = (V, E)$ with $n = |V|$. We assume that the set of nodes is static as otherwise a termination of the self-stabilization process may not be reached (this has been shown in [7]). Each peer in the system is represented by a node $v \in V$. Each node $v \in V$ can be identified by its unique reference or its unique identifier $v.id \in \mathbb{N}$ (called *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 knows the reference of some other node v, then u can send a message m to v by putting m into $v.Ch$.

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 information needed for the execution of this action, 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 $true$ or if there is a message in $u.Ch$ that requests to call the action. In both cases, we call the action *enabled*. An action whose guard is simply $true$ is executed periodically. When a node u processes a message m, then m is removed from $u.Ch$.

We define the *system state* to be an assignment of a value 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*. We assume *fair message receipt*, meaning that 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. We place no bounds on message propagation delay or relative node execution speed, i.e., we allow for fully asynchronous computations and non-FIFO message delivery. A self-stabilizing protocol does not manipulate node identifiers and thus only operates on them in *compare-store-send* mode. That is, 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. Note that we compute the hash value of a node's identifier in some protocols, but this does not manipulate the ID itself.

We are interested in the formation and maintenance of a certain graph topology that connects the nodes. As it is standard, we assume that there are no corrupted IDs in the initial state of the system, i.e., node IDs are *read-only*. Note that we are not able to repair initially corrupted node IDs within the scope of this model, as we do not consider the usage of failure detectors. Thus we can assume that node IDs are always correct in all states, as our protocol is compare-store-send. Nevertheless, node channels may contain an arbitrary amount of messages containing false information in initial states: We call these messages *corrupted*. We say the system is in a *legitimate (stable) state*, if the nodes and the edges form the desired graph topology and there are no corrupted messages in the system. We are now ready to define what it means for a protocol to be self-stabilizing:

Definition 2.1 (*Self-stabilization*). 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.

There is a directed edge $(u, v) \in E$, if u stores the 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*. In order for our distributed algorithms to work, we require the directed graph G containing all explicit and implicit edges to stay at least weakly connected at every point in time. Once there are multiple weakly connected components in G, these components cannot be connected to each other anymore as it has been shown in [44] for compare-store-send protocols. For a graph that contains multiple weakly connected components, our protocol converts each of these components to our desired topology.

In general, the following performance metrics are most relevant in topological self-stabilization:

(1) *Convergence Time:* Assuming a synchronous environment (or assuming an upper bound on the message transmission per link), the distributed convergence time measures how many (parallel) communication rounds are required until the final topology is reached.
(2) *Work:* The work measures how many edges are inserted, changed, or removed in total, during the covergence process.

3. FROM CONNECTIVITY PRIMITIVES TO SELF-STABILIZING ALGORITHMS

Before designing distributed algorithms to maintain and repair topologies, we need to answer a most fundamental question: how can nodes manipulate their neighborhoods *locally*, without risking to lose connectivity? And more generally: which primitives exist that allow to iteratively and locally change a graph, such that it eventually reaches its desired final state? Such connectivity primitive operations are a *prerequisite* for the self-stabilizing convergence.

The identification of such primitives however does not yet answer the question how a distributed self-stabilizing algorithm can actually use them. In the following, we hence first discuss the universal primitives for reliable connectivity, and then discuss their use in algorithms.

3.1. Universal Primitives for Reliable Connectivity

Universal connectivity primitives are local graph operations which allow us to transform any topology into any other topology. While we focus on feasibility in the following, we will later use these primitives to design topologically self-stabilizing *algorithms*.

Let us first define the notion of links (u, v). Links can either be explicit or implicit. An explicit link (u, v) (in the following depicted as solid line) means that u knows v, i.e., u stores a reference of v (e.g., v's IP address). An implicit link (u, v) (depicted as dashed line) means that a message including v's reference is currently in transit to u (from some arbitrary sender). We are often interested in the union of the two kinds of links.

As discussed above, a first most fundamental principle in the design of distributed self-stabilizing algorithms is that links can never be deleted:

RULE 1.1. *During the execution of a topologically self-stabilizing algorithm, weak connectivity must always be preserved. In particular, a pointer (i.e., information about a peer) can never be deleted.*

If a link is removed, it may happen that this link is the only link connecting two otherwise disconnected components. Clearly, once disconnected, connectivity can never be established again.

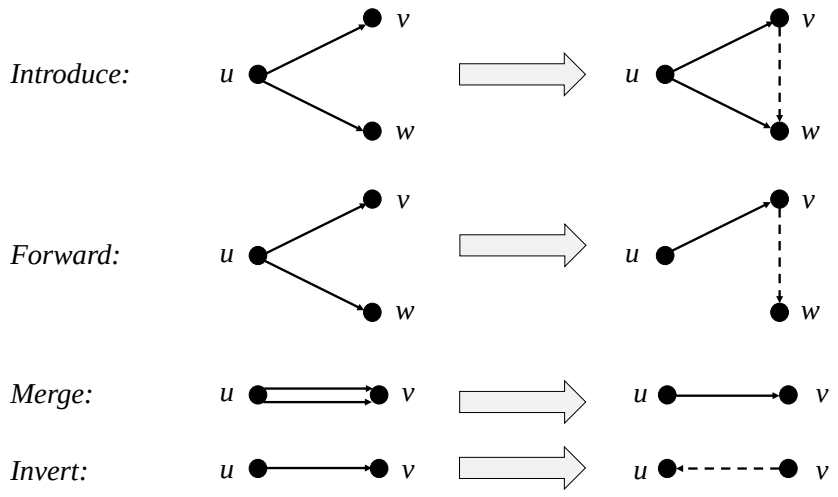

Fig. 1.1. Basic connectivity primitives (*solid*: explicit edges *dashed*: implicit edges).

We next identify four basic primitives which preserve connectivity (cf. Figure 1.1): INTRODUCE, FORWARD, MERGE, INVERT.

(1) INTRODUCE: Assume node u has a pointer to nodes v and w: there are two directed links (u, v) and (u, w). Then, u can introduce w to v by sending the pointer to w to v.
(2) FORWARD: Assume node u has a pointer to nodes v and w, i.e., (u, v) and (u, w). Then, u forwards the reference to w to v and removes the reference of w from its local memory.
(3) MERGE: If u has two pointers to v, i.e., (u, v) and (u, v), then u can merge the two.
(4) INVERT: If u is connected to v, it can invert the link (u, v) to (v, u) by forwarding a pointer to itself to v, and delete the reference to v.

It is easy to see that these primitives indeed preserve weak connectivity. In fact one can show that the INTRODUCE, FORWARD, and MERGE operations even preserve strong connectivity. We also note that we need a compare operation to implement the merge operation: namely, we need to be able to test whether two references point to the same node.

These operations turn out to be very powerful. In fact, three of them are sufficient to transform any weakly connected graph into any strongly connected graph. In other words, they are *weakly universal*:

THEOREM 3.1. *The three primitives* INTRODUCE*,* FORWARD*, and* MERGE *are* weakly universal*: they are sufficient to turn any weakly connected graph* $G = (V, E)$ *into any strongly connected graph* $G' = (V, E')$.

We just provide intuition for this theorem and refer to [50] for more details. Essentially, the proof proceeds in two stages, from $G = (V, E)$ to a complete graph (the clique), and from the clique to $G' = (V, E')$. In the first stage, if in each communication round, each node introduces its neighbors to each other as well as itself to its neighbors, we reach a clique after $O(\log n)$ communication rounds. Due to weak connectivity, it follows that for any two nodes v and w, there is a path from v to w (ignoring link directions). For the second stage, assume $G = (V, E)$ is a clique. Then using FORWARD and MERGE operations, we can transform G into G' as follows (without removing edges in G'). Let (u, w) be an arbitrary edge which needs to be removed, i.e., $(u, w) \notin E'$. Since $G' = (V, E')$ is strongly connected, there is a shortest directed path from u to w in G'. Let v be the next node along this path. Now node u can forward (u, w) to v, i.e., (u, w) becomes (v, w). This will reduce the distance between an unused node pair in G' by 1, and since the maximal distance is $n - 1$, the distance of a superfluous edge can be reduced at most $n - 1$ many times before it merges with an edge in G'. Thus, we eventually obtain G'.

All four primitives together are even *universal*: they are sufficient to transform any weakly connected graph into any weakly connected graph.

THEOREM 3.2. *The four primitives* INTRODUCE*,* FORWARD*,* MERGE*, and* INVERT *are universal: they are sufficient to turn any weakly connected graph* $G = (V, E)$ *into any weakly connected graph* $G' = (V, E')$.

The intuition for this theorem is as follows. Let $G'' = (V, E'')$ be the graph in which for each edge $(u, v) \in E'$, both edges (u, v) and (v, u) are in E''. Note that G'' is strongly connected. Now, according to Theorem 3.1, it is possible to transform any G to G''. So in order to transform G'' to G', we need the INVERT primitive, to remove undesired edges: we invert any undesired edge (u, v) to (v, u) and then merge it with (v, u).

Interestingly, the primitives are not only sufficient but also necessary.

THEOREM 3.3. *The four primitives* INTRODUCE*,* FORWARD*,* MERGE*, and* INVERT *are also necessary.*

The reason is that INTRODUCE is the only primitive which generates an edge, FORWARD is the only primitive which separates a node pair, MERGE is the only primitive which removes an edge, and INVERSION is the only primitive rendering a node unreachable (as one can see in Figure 1.1 we cannot go from u to v anymore once we inverted the explicit edge (u, v) into the implicit edge (v, u)).

3.2. From Connectivity Primitives to Self-Stabilizing Algorithms

Universal primitives allow us to transform any weakly-connected graph G into any weakly-connected graph G'. However, the mere *existence* or *feasibility* of such transformations is often not interesting in practice, if there do not exist efficient distributed algorithms to find a transformation.

Before we show how to derive distributed algorithms, we introduce some fundamental concepts and provide some additional insights into how connectivity can be maintained. A central requirement in topologically self-stabilizing systems is *monotonic reachability*: if v is reachable from u at time t, using explicit or implicit edges, then, if no further failures or errors occur and given a static node set, v is also reachable from u at any time $t' > t$. The following theorem can easily be proved by induction, as long as there are no references to non-existing nodes in the system.

THEOREM 3.4. *The* INTRODUCE*,* FORWARD*, and* MERGE *operations fulfill monotonic reachability.*

Remarks:

(1) There is also the concept of *monotonic searchability*: if u can send a message at time t that is successfully delivered to v according to some routing protocol $\mathcal{R}$, then any further message generated at u at time $t' > t$ with v as its destination is successfully delivered. Monotonic reachability is necessary to implement monotonic searchability, it is not sufficient. We will later (in Section 7) discuss how to realize also monotonic searchability.

(2) One particularly annoying challenge in the design of self-stabilizing algorithms is due to the fact that there may still be corrupted messages in transit in the system. Such messages can threaten the correctness of an algorithm later. In particular, corrupted messages may violate the closure property: although initially in a legal state, the system may move to an illegal state. The set of legal states is hence only a subset of the "correct states".

As it has already been stated by Theorem 3.1, the primitives INTRODUCE, FORWARD and MERGE not only fulfill monotonic reachability but are also weakly universal, so applying these can turn any weakly connected graph into a strongly connected one. This is sufficient for most topologies in practice because usually each node should be allowed to send messages to any other node. It is however not clear on the first glance how to progress from any illegal state to a legitimate one using these primitives. Therefore we need mechanisms to ensure the convergence of the system. A universal approach for this could be to combine a mechanism that allows nodes to find out in finite time if the system is in an illegal state with the transitive closure framework (see Section 6.3). On the positive side, being able to check if the system is in an illegal state is usually possible, allowing the creation of a clique in order to "reset" the system as it is demonstrated in the transitive closure framework. However, in general this is very inefficient because the amount of work for each node can grow linear in the size of the network. Therefore, researchers came up with more efficient protocols tailored to more specific topologies.

4. TOPOLOGICALLY SELF-STABILIZING LINEARIZATION

Many topologically self-stabilizing protocols rely on some basic *Linearization* algorithm [19; 26; 52]. Using linearization, one can establish an ordering of nodes legitimate states. In this section we first present the idea of linearization based on a self-stabilizing protocol for a sorted list [45]. We call that protocol BUILDLIST for the remainder of this survey. Afterwards we explore a protocol for a de Bruijn graph that relies on the linearization technique.

4.1. Linearization Protocol

Before we can describe the actions of BUILDLIST, we need to define the variables for a node u: A pointer $u.left$ storing u's closest left neighbor and a pointer $u.right$ storing u's closest right neighbor. The idea of BUILDLIST is that each node always wants to keep its closest left and right neighbors (based on local information only) and delegate all remaining outgoing connections using the FORWARD primitive.

More formally, BUILDLIST consists of two actions TIMEOUT and LINEARIZE, where TIMEOUT is executed periodically at each node, and LINEARIZE can be called locally or remotely.

In TIMEOUT, a node u first performs a consistency check on its variables $u.left$ and $u.right$: It may happen that in initial states $u.left > u$ (or $u.right < u$). If that is the case then u resets the node

assignment to $u.left$ (or $u.right$) and locally calls LINEARIZE(w) for the removed node w. Furthermore u introduces itself to $u.left$ and $u.right$ via the INTRODUCE primitive in TIMEOUT.

When processing a LINEARIZE(w) message at u with $w < u$, then u first compares the identifier of w with its own identifier and the identifier of $u.left$. Node u distinguishes between the following two cases:

(i) $w < u.left$: This case leads to u forwarding w to $u.left$ by calling LINEARIZE(w) on $u.left$.

(ii) $u.left < w < u$: In this case u replaces $u.left$ by w and forwards the old value of $u.left$ to w via a LINEARIZE call.

We proceed analogously at node u in case $w > u$, this time considering $u.right$ instead of $u.left$. Consider Figure 1.2 for a visualization of the different cases for LINEARIZE.

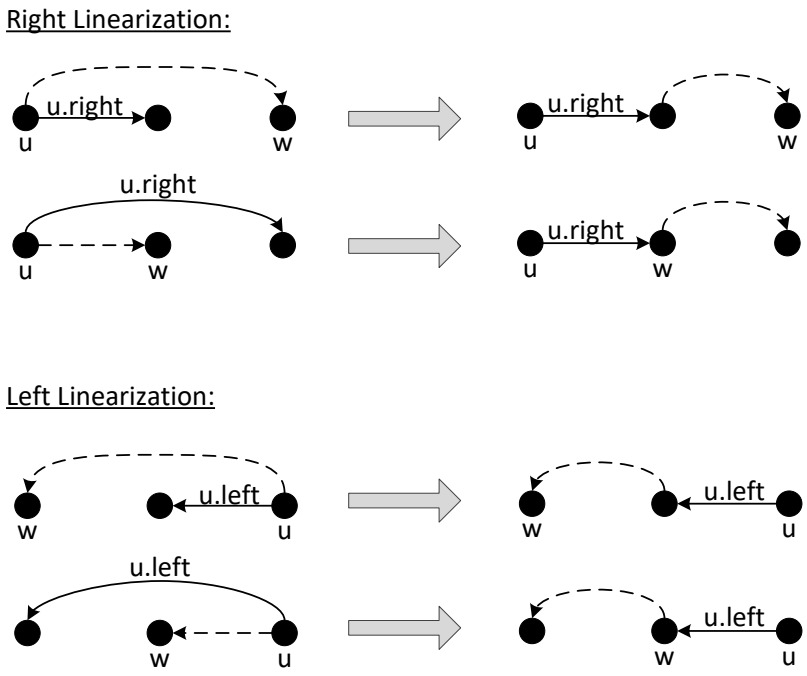

Fig. 1.2. Different cases when calling LINEARIZE(w) at node u.

Using the above actions, one can show that BUILDLIST satisfies convergence and closure, culminating in the following theorem:

THEOREM 4.1. BUILDLIST *is self-stabilizing, i.e.,*

(i) BUILDLIST *transforms any weakly connected graph* $G = (V, E)$ *into a sorted list after* $O(n)$ *rounds (Convergence) and*

(ii) if the explicit edges in G *already form a sorted list, then they are preserved at any point in time (Closure).*

We provide some intuition for the proof: To show convergence, consider a pair of nodes (u, v) with $u < v$ that is adjacent in legitimate states. As the graph G is weakly connected, there exists an undirected path P between u and v. Consider the potential function $\Phi = v_r - v_l$, where v_l is the process with minimum identifier in P and v_r the process with maximum identifier in P. One can show that Φ monotonically decreases over time until $\Phi = v - u$ corresponding to (u, v) being directly connected, which implies convergence. The bound on the convergence time stems from the fact that an implicit edge to some node v has to be delegated along the whole list in a worst-case scenario: In case the list is already fully built with only one edge missing, there are no shortcut edges for an implicit edge in order to reach its target, so it has to traverse the whole list.

In order to show closure, we can argue that an explicit edge (u, v) is only forwarded, if u gets to know a closer neighboring process than v. But this is not possible as processes already form a sorted list, so closure holds.

4.2. The Linearized de Bruijn Graph

Linearization has been used as a basis for several other self-stabilizing topologies. In this section we provide an example for this by examining the *linearized de Bruijn graph* from [51].

Consider the standard de Bruijn graph:

Definition 4.2. Let $d \in \mathbb{N}$. The standard (d-dimensional) *de Bruijn graph* consists of nodes having labels $(x_1, \ldots, x_d) \in \{0, 1\}^d$ and edges $(x_1, \ldots, x_d) \to (j, x_1, \ldots, x_{d-1})$ for all $j \in \{0, 1\}$.

One can route from a source node to a destination node in $O(\log n)$ hops via bitshifting if the network consists of n nodes.

In order to construct a self-stabilizing protocol for a de Bruijn graph, one can use the linearized de Bruijn network (first introduced in [43]) to emulate the classical de Bruijn graph. The idea is to let each real process v emulate 2 additional virtual processes v_0, v_1, resulting in the process v representing 3 nodes v, v_0 and v_1. Using a uniform pseudorandom hash function $h : \mathbb{N} \to [0, 1)$, we project the

identifiers of each real process v onto the $[0,1)$-interval. Afterwards we define the identifiers of the virtual nodes by assigning $\frac{h(v)}{2}$ to v_0 and $\frac{h(v)+1}{2}$ to v_1. A legitimate state of the system is then defined as the sorted list consisting of all real and virtual nodes ordered by their identifiers in $[0,1)$. It has been shown in [43] that such a topology is able to emulate the standard de Bruijn graph, i.e., routing paths in the linearized de Bruijn network are of length $O(\log n)$ w.h.p.

In order to construct a self-stabilizing protocol BUILDDEBRUIJN for the linearized de Bruijn network we let each real and virtual process run the BUILDLIST protocol from Section 4.1. It is easy to see that BUILDLIST converts the graph $G' = (V', E')$ with $V' = \{v, v_0, v_1 \mid v \in V\}$ into a sorted list if G' is weakly connected initially. Unfortunately this is not necessarily the case for initial states, even if $G = (V, E)$ is weakly connected as shown in Figure 1.3.

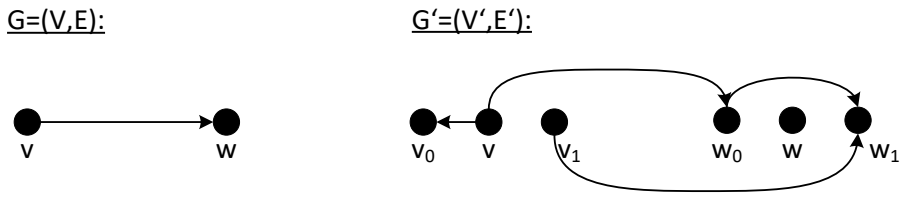

Fig. 1.3. Possible initial state for G' leaving the node w isolated.

In order to eventually reach a state where G' is weakly connected, the authors of [51] introduced the *probing* technique. The idea is to let each node v periodically check if it is in the same connected component as its virtual nodes v_0 and v_1. If this is not the case, then v generates a connection to v_0 and v_1 respectively by locally calling LINEARIZE(v_0) and LINEARIZE(v_1) respectively. To check if a connection to v_0 has to be established, each real process v does the following: It periodically sends out a *probe* along the sorted list to the left. That probe is forwarded along virtual nodes in the sorted list, until it reaches a real node w. w then forwards the probe to its virtual node w_0. From this point on we forward the probe to the right in the sorted list until it reaches v_0 or it gets stuck or overruns v_0. In the first case we know that v and v_0 are in the same connected component. If one of the other two cases arises, v initiates a LINEARIZE(v_0) call. The same approach is used in order to check if v and v_1 are in the same component. Note that the number of hops for each probe is expected to be $O(1)$ in

legitimate states. When combining the BUILDLIST protocol with the probing approach, one can show the following theorem:

THEOREM 4.3. BUILDDEBRUIJN *is self-stabilizing, i.e.,*

(i) BUILDDEBRUIJN *transforms any weakly connected graph* $G = (V, E)$ *into a linearized de Bruijn network after* $O(n)$ *rounds (Convergence) and*

(ii) if the explicit edges in G *already form a linearized de Bruijn network, then they are preserved at any point in time (Closure).*

Following a probing approach similar to the one described above, one can realize a self-stabilizing protocol for the general de Bruijn graph (the q-ary, d-dimensional de Bruijn graph), as it has been shown in [16]. Compared to the standard (d-dimensional) de Bruijn graph (Definition 4.2), the label of a node may now consist of sequences of values $j \in \{0, \ldots, q\}$ instead of bits.

5. SELF-STABILIZING METRICAL GRAPHS

Linearization is not only a building block for more general and scalable self-stabilizing networks, as we will discuss them later in this paper, it is also a a good basis for building another important family of graph topologies: *metric* graphs.

Definition 5.1. Given a set M, a distance function $d : M^2 \to \mathbb{R}$ is a *metric* if for all $x, y, z \in M$

(i) $d(x, y) \geq 0$,

(ii) $d(x, y) = 0$ if and only if $x = y$,

(iii) $d(x, y) = d(y, x)$ and

(iv) $d(x, z) \leq d(x, y) + d(y, z)$.

In the following, we explore graphs that can efficiently be described by a metric, with the focus on networks for geometrical scenarios. Such networks are relevant for example in the area of wireless ad-hoc networks. In particular, in this section we will show that the linearization technique discussed in the previous section can serve as a basis for graphs based on 1-dimensional as well as circular metrics (cf Section 5.1), as well as tree metrics (cf Section 5.4). However, we will also show that when assuming that nodes only know the metric function but not the topology (line, ring, etc.) that should

be formed, it is not possible to design self-stabilizing algorithms solely based on local information at nodes (cf Section 5.2). We therefore introduce another technique in order to build arbitrary graphs for arbitrary metrics (cf Section 5.3).

5.1. Line and Circular Metrics

Consider a line metric as an embedding of the nodes into the one-dimensional space, i.e., we can define d by $d(u,v) = |u.id - v.id|$ for any two nodes $u, v \in V$. It is easy to see that one can use the BUILDLIST protocol from Section 4.1 to converge to a graph that satisfies the line metric.

A simple extension of BUILDLIST would be to let processes form a sorted cycle. For this to work, we have to establish an additional connection between the process with minimal identifier and the process with maximal identifier. We distinguish between list edges and cycle edges. List edges are treated by the BUILDLIST protocol, for cycle edges we introduce the following additional actions:

(i) Upon activation, if a node v does not have a left (or right) list neighbor, it creates a cycle edge to itself and delegates it to $v.left$ (or $v.right$).

(ii) If a node v has a cycle edge to a node w with $v < w$ (or $w < v$) and $v.left \neq \perp$ (or $v.right \neq \perp$), then v delegates the cycle edge to $v.left$ (or $v.right$).

(iii) If node v has a cycle edge to node w that cannot be delegated via action (ii), then upon activation v introduces itself to w, generating the implicit cycle edge (w, v).

(iv) If node v has a (explicit) cycle edge to node w and receives another (implicit) cycle edge to some node $w' \neq w$, then v keeps the edge for which covers the larger distance w.r.t. to the metric function. The other edge, say (w, v'), is removed and replaced by two implicit list edges (v, v') and (w, v').

Consider Figure 1.4 for a visualization of these rules.

One can show that this extension of BUILDLIST (call it BUILDCYCLE) is self-stabilizing:

THEOREM 5.2. BUILDCYCLE *is self-stabilizing, i.e.,*

(i) BUILDCYCLE *transforms any weakly connected graph $G = (V, E)$ into a sorted cycle after $O(n)$ rounds (Convergence) and*

(ii) if the explicit edges in G already form a sorted cycle, then they are preserved at any point in time (Closure).

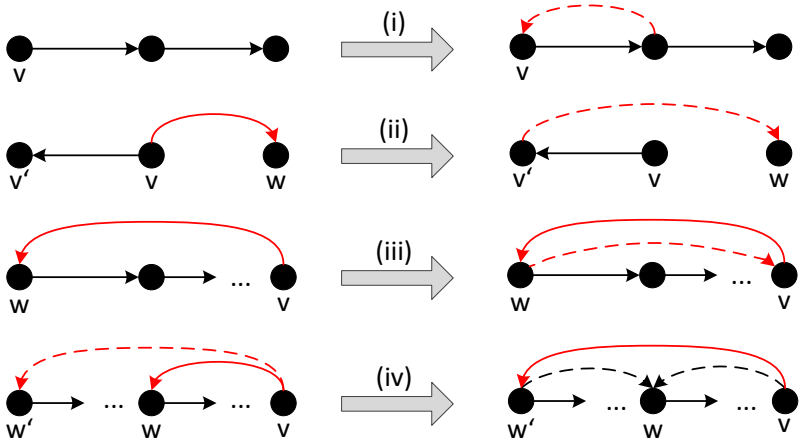

Fig. 1.4. Illustration for the actions of BUILDCYCLE. Red edges denote cycle edges, black edges denote list edges.

Self-stabilizing protocols that make use of a sorted cycle can be found for example in [20; 32; 30].

5.2. Challenges of Local Probing

Although the local probing approach proves useful for many scenarios, it has its limits as we will outline in this section. Consider nodes $v_1, \ldots, v_n$ and imagine we are given a cycle metric $d_C : V^2 \to \mathbb{R}^+$ with $d_C(v_i, v_j) = 1$ with $j = i + 1 \bmod n$ for all $i \in \{1, \ldots, n\}$ (cf Figure 1.5(a)). A cycle graph reflects the metric d_C, which would be a 1-spanner.

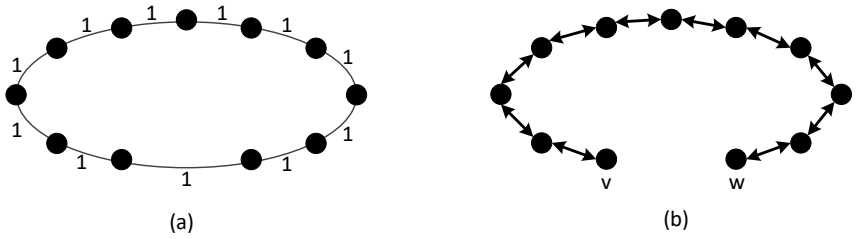

Fig. 1.5. (a) Illustration of the cycle metric. (b) Initial graph for which we cannot locally distinguish between a line and a circular metric.

If nodes only know the metric function d_C and we are given a sorted list initially, then it is impossible using only local probing to reach a sorted cycle. This is because any implicit edge generated by any node

has minimum distance regarding d_C and thus is immediately merged with the corresponding explicit edge. Therefore we cannot generate an implicit edge (v, w) (cf. Figure 1.5(b)) which would be necessary to establish the cycle.

As a consequence it follows that any protocol must be either *non-oblivious* (in the sense that the protocol knows something about the graph topology) or *non-local* (in the sense that requests are processed without local evidence of a violation of the metric) in order to stabilize a k-spanner of the desired topology for any constant k.

An example for a non-oblivious strategy would be the BUILDCYCLE protocol from Section 5.1. An example for a non-local strategy is the probing approach from Section 4.2 that is used to stabilize the linearized de Bruijn graph. We generalize this approach in the next section.

5.3. General Metrics with Global Probing

We next discuss non-local strategies that can be used in order to stabilize any arbitrary given metric given via a distance function $d_M : V^2 \to \mathbb{R}$. The first strategy makes use of probing: At each node u we periodically generate a probing request that is handled in two phases:

(1) Follow a random sequence of nodes $v_1, v_2, \ldots$ that lie on a shortest path to u (i.e., $d_M(v_i, i) + d_M(v_i, v_{i+1}) = d(v_{i+1}, u)$) up to a node v_k, for which we cannot delegate the request any further.
(2) From v_k we delegate the request to a random neighbor w_k and follow a random shortest path back to u. If a node w is encountered that does not have a neighbor lying on the shortest path to u, create the edge (w, u).

This strategy suffices to detect edges that are contained in the desired topology, but are still missing in the overlay.

A different approach has been discussed in [20]: The idea is to let nodes form a sorted cycle via the previously presented BUILDCYCLE protocol. Furthermore each node v maintains a pointer $v.test$ to some node in V. Via delegation these pointers constantly traverse the cycle in a common direction. The actual construction of the metric is then done by each node v periodically: Using d_M, v checks if the edge $(v, v.test)$ should be added to the graph. Also v checks, for each of its outgoing edges, whether it can be removed from the graph. One can show that this algorithm is able to stabilize any arbitrary metric.

5.4. Special Metrics with Local Probing

There exist special metrics for which we can construct self-stabilizing protocols based on local probing only. Consider a *tree metric* $d_T : V^2 \to \mathbb{R}^+$ that assigns a weight to each possible edge in the overlay. The following protocol BUILDMST from [22] is able to maintain a minimum spanning tree, i.e., a connected graph $MST = (V, E)$, for which the sum of the edge weights $\sum_{e \in E} d_T(e)$ is minimum. For nodes u, v, w define $u \prec (v, w)$ as a shorthand for $(d_T(i, v) < d_T(v, w)) \wedge (d_T(u, w) < d_T(v, w))$. This means that u is closer to both v and w with respect to d_T, and hence that the MST should not contain the edge (v, w) in order to connect u, v and w. BUILDMST works as follows: Let each node v maintain a set $N_v \subseteq V$ that stores the neighboring nodes of v in the MST. Upon activation a node $v \in V$ performs the following actions for each of its current neighbors $w \in N_v$: It checks whether there is a node $u \in N_v$ for which $u \prec (v, w)$ holds. If such a node exists then v delegates w to u and removes w from N_v. Otherwise v introduces itself to w by sending its reference to w. Consider Figure 1.6 for a visualization of these rules.

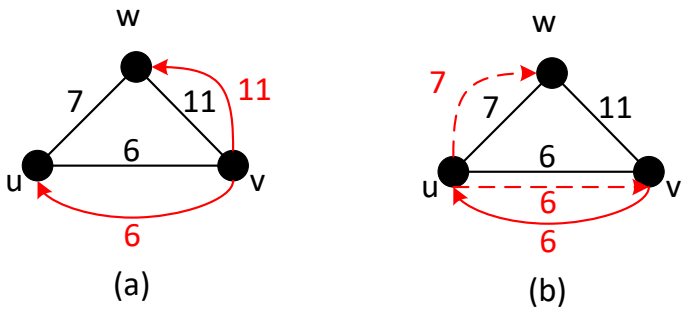

Fig. 1.6. An exmaple of the protocol's execution. The black edges indicate the distances between the nodes with respect to d_T. Red edges denote the overlay's edges. In this example, v first delegates w to u and then introduces itself to u

Using a potential function, one can show that BUILDMST satisfies convergence and closure, i.e., BUILDMST is self-stabilizing:

THEOREM 5.3. *Given a weakly connected graph $G = (V, E)$ and a tree metric $d_T : V^2 \to \mathbb{R}^+$.* BUILDMST *is self-stabilizing, i.e.:*

(i) BUILDMST *eventually transforms G into a minimum spanning tree after $O(n^2)$ rounds (Convergence) and*

(ii) if the explicit edges in G already form a minimum spanning tree, then they are preserved at any point in time (Closure).

The proof works similar to the one for the line metric: Let G_s be the directed graph containing all explicit and implicit edges when the system is in state s. Then we define the potential of G_s to be the weight of the minimum spanning tree that can be constructed from all edges in G_s when ignoring their direction. It can be shown that this potential decreases monotonically throughout time. As the weight of the (globally) optimal minimum spanning tree, i.e., the minimum spanning tree that considers all possible edges between nodes, is a lower bound for the potential function, it is clear that the potential cannot decrease indefinitely.

5.5. Euclidean Metrics

Instead of identifying processes via their identifier, we identify processes v via their geographical position in this section, i.e., $v = (v_x, v_y) \in \mathbb{R}^2$. We first define the Delaunay graph:

Definition 5.4. The *Delaunay graph* $G = (V, E)$ consists of nodes $V \in \mathbb{R}^2$ and edges E, where $(v, w) \in E$ if there exists a circle C through v and w such that no other node is within C.

Note that if $(v, w) \in E$, then other nodes are allowed to lie on the border of the circle C going through v and w (see Figure 1.7).

Denote the *Euclidean distance* between v and w by $||v, w|| = \sqrt{(v_x - w_x)^2 + (v_y - w_y)^2}$. One can easily verify that the Euclidean distance is a metric. The underlying metric for the Delaunay graph is the *Delaunay triangulation*, which is an approximation of the Euclidean metric with the advantage of being locally checkable. In more details, the length of any shortest path between any two nodes $u, v \in V$ in the Delaunay graph is at most $1.998 \cdot ||u, v||$ as it has been shown in [61].

Each node u has a variable $u.N \subseteq V$ that consists of u's current neighbors. Similar to BUILDLIST, BUILDDELAUNAY consists of two actions TIMEOUT and INTRODUCE, where TIMEOUT is executed periodically at each node and INTRODUCE can be called locally or remotely. INTRODUCE is used in the same manner as LINEARIZE in BUILDLIST, i.e., we use INTRODUCE to forward nodes in the Delaunay graph until they reach their correct neighbor.

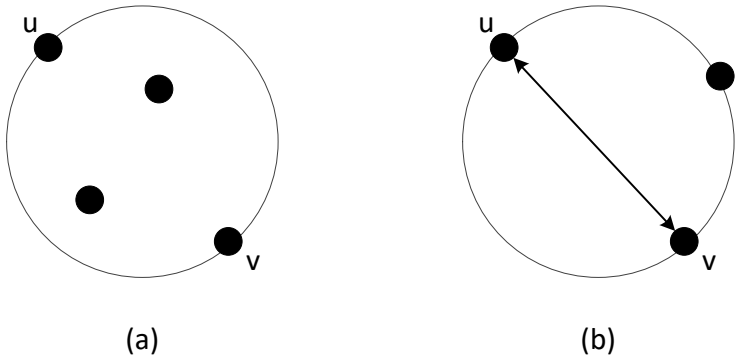

Fig. 1.7. (a) Example scenario, where $(u, v) \notin E$ due to the fact that any cycle going through u and v contains at least one of the two inner nodes inside of it. (b) Scenario where $(u, v) \in E$.

In TIMEOUT node u performs the following three actions:

(i) u checks for each of its neighbors $v \in u.N$, if (u, v) belongs to the Delaunay graph from u's local point of view, i.e., if there exists a circle C going through u and v that does not contain any other node out of $u.N$. If not, then u removes v from $u.N$ and locally calls INTRODUCE(v).

(ii) u introduces itself to its neighbors in $u.N$ by calling INTRODUCE(u) on them.

(iii) Let $v_1, \ldots, v_k \in N$ be u's neighbors ordered in clockwise direction. Node u introduces v_i to v_{i+1} and vice versa, if the angle $\sphericalangle v_i u v_{i+1}$ is smaller $180\,^\circ$.

Consider Figure 1.8(a) for a visualization of the actions (ii) and (iii) of TIMEOUT.

Upon receipt of an INTRODUCE(v) message, node u does the following: It first checks whether the edge (u, v) is contained in the Delaunay graph with node set $u.N \cup \{u, v\}$ and adds v to $u.N$ if that is the case. If not, then u determines a node out of $u.N$ to forward v to in the following way: u first determines the nodes $w, w' \in u.N$ with minimal angle to v in clockwise direction and counterclockwise direction respectively (see Figure 1.8(b)). Then u forwards v to the node $w'' \in \{w, w'\}$ that minimizes $||w'', v||$ by calling INTRODUCE(v) on w''.

One can show that BUILDDELAUNAY satisfies convergence and closure, i.e., BUILDDELAUNAY is self-stabilizing:

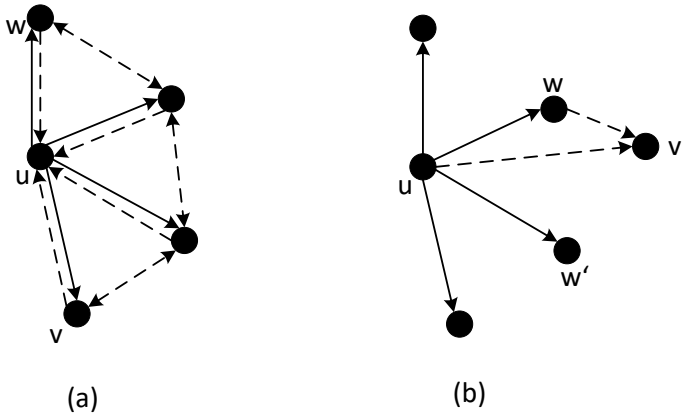

Fig. 1.8. (a) Visualization of the actions (ii) and (iii) of TIMEOUT. Observe that u does not introduce v and w to each other, as the angle $\sphericalangle vuw > 180\,^\circ$. (b) Example for node u determining the node $w \in u.N$ to forward v to. The implicit edge (w, v) is generated as u calls INTRODUCE(v) on w.

THEOREM 5.5. *Given a weakly connected graph $G = (V, E)$ with a set of processes $V \in \mathbb{R}^2$.* BUILDDELAUNAY *is self-stabilizing, i.e.,*

(i) BUILDDELAUNAY *eventually transforms G into a Delaunay graph after $O(n^3)$ rounds (Convergence) and*

(ii) if the explicit edges in G already form a Delaunay graph, then they are preserved at any point in time (Closure).

To show convergence, one can use a potential function. Observe that an implicit edge (u, v) is either transformed into an explicit edge at u, or is forwarded to a node $w \in u.N$ such that the edge gets shorted w.r.t. the Euclidean distance. Now consider the potential Φ that counts the number of nodes $w \notin u.N$ that should be contained in $u.N$ in legitimate states. One can show that Φ monotonically decreases until $\Phi = 0$, which yields convergence.

Closure can be shown by arguing that each implicit edge is being merged with an explicit one in legitimate states, because an implicit edge cannot be transformed into an explicit edge that does not belong to the Delaunay graph.

BUILDDELAUNAY can be applied in higher-dimensional scenarios as well, i.e., in a d-dimensional scenario we consider balls of dimension d instead of two-dimensional circles.

A protocol for self-stabilizing quadtrees has been proposed in [15]. The protocol makes use of a space-filling curve to obtain an ordering $\prec$ of all processes. Given this ordering $\prec$, we can apply BUILDLIST to let all processes form a sorted list. Using the established list edges, we use a probing approach similar to the one presented in Section 4.2 to generate additional edges that guarantee that the diameter of the quadtree is $O(\log n)$, given that Euclidean distance between any two processes is large enough. Same as for the Delaunay graph, the protocol can be generalized to higher dimensions, such that self-stabilizing octtrees can be realized.

6. SELF-STABILIZING EXPANDERS

A most attractive family of networks are networks based on expander graph topologies. Expander graphs are sparse graphs with strong connectivity properties that protect the network from disconnecting when nodes shut down. More formally, if k nodes got shut down (for example by an adversarial attack), then at most $O(k)$ nodes get disconnected from the network. Expander graphs are known to enable efficient communication at low cost, and can be highly scalable. They hence come with many applications, from overlays [2] to datacenter interconnects [60]. In the following, we review the design of a self-stabilizing network based on skip graphs. Subsequently, we discuss the design of self-stabilizing random graphs. We conclude by discussing a powerful and general framework to design self-stabilizing networks.

6.1. Self-Stabilizing Skip Graphs

The first self-stabilizing and scalable overlay network was SKIP+ [25], a self-stabilizing variant of the skip graph family [2; 23]. Similarly to the original skip graphs, SKIP+ features a polylogarithmic degree and diameter. However, in constrast to the original skip graph versions, SKIP+ contains additional edges which enable *local detectability*: only with these edges it can be ensured that at least one node will always notice, locally, if the overall network is not in the desired state yet.

SKIP+ distinguishes between stable edges and temporary edges. Similarly to the linearization example above, temporary edges will travel through the topology (i.e., they are forwarded), and eventually merge or stabilize. Node v considers an edge (v, w) to be temporary if from v's point of view (v, w) does not belong to SKIP+ and so v will try to forward it to some of its neighbors for which the edge would be more relevant. If on the other hand (v, w) belongs to SKIP+ from v's point of view, then v

considers (v, w) to be a stable edge and will make sure that the connection is bidirected, i.e., it will propose (w, v) to w.

As many self-stabilizing algorithms, the self-stabilizing protocol for SKIP+ is very simple: the nodes in SKIP+ continuously must execute three rules.

(1) ***Rule 1: Create Reverse Edges and Introduce Stable Edges.*** This rule makes sure that a directed edge becomes a bidirected edge, introducing the nodes to each other. Also, stable edges are created where needed.
(2) ***Rule 2: Forward Temporary Edges.*** This rule is used for forwarding temporary edges to neighboring nodes. Eventually, the edges will stabilize or merge.
(3) ***Rule 3: Introduce All and Linearize.*** The rule has two parts. It performs some kind of local transitive closure, where nodes introduce all their neighbors to each other. Moreover, the rule is responsible for sorting neighboring nodes according to their identifiers. (In a skip graph, nodes are ordered on each level, facilitating search operations.)

The three rules are continuously checked and executed in parallel by all nodes. However, while the algorithm itself is simple, its analysis is non-trivial. In a nutshell, the stabilization proof is based on the observation that the execution of the algorithm can be divided into phases in which certain properties (milestones) are achieved. In particular, the execution can be thought of being divided into a bottom-up and a top-down phase. The bottom-up phase (i.e., from skip graph level 0 upwards), connected components for increasingly larger prefixes are formed in the identifier space. This will be accomplished by Rule 1 (where new nodes in the range of a node are discovered and where ranges may be refined) and Rule 3 (where an efficient variation of a local transitive closure is performed). Once the connected components are formed, in the second phase of the algorithm (the division into phases is a purely analytical one) will form a sorted list out of each prefix component. This is accomplished in a top-down fashion by merging the two already sorted subcomponents into a sorted larger component until all nodes in the bottom level form a sorted list.

Jacob et al. [25] show the following result.

THEOREM 6.1. *SKIP+ is self-stabilizing, i.e.,*

(i) SKIP+ transforms any weakly connected graph $G = (V, E)$ into a skip graph after $O(\log^2 n)$ rounds (Convergence) and

(ii) if the explicit edges in G already form a skip graph, then they are preserved at any point in time (Closure).

A single join event (i.e., a new node connects to an arbitrary node in the system) or leave event (i.e., a node just leaves without prior notice) can be handled with polylogarithmic work.

An improved version of the self-stabilizing SKIP+ graph is HSKIP+ [17] which reduces the stabilization time in practice and needs less work for single join or leave events.

6.2. Self-Stabilizing Random Graphs

Another attractive family of overlay topologies are random graphs. Random graphs are known to enable efficient communication, and are also used in other contexts, such as datacenter networks [58] and also serve as models for social networks and in particular small-world networks [29]. A self-stabilizing small-world network has been proposed in [30]. The topology consist of a sorted ring with additional shortcuts, i.e., each node has a *long-range link* to a random node in the system.[2] Similar to the probing approach for the linearized de Bruijn graph, in this algorithm, each node periodically sends out probes to ensure that the network is connected through non-long-range links. Long-range links are maintained through a technique called *Move and Forget* adapted from [11]: As time proceeds, a node v may forget its long-range link with a certain probability that is increasing over time. Node v periodically asks its long-range link $v.lrl$ for its direct ring neighbors l and r. Once v forgets the long-range link, it sets $v.lrl$ to either l or r, each with probability $1/2$.

A special technique to let a topology converge to some random graph that is an expander has been proposed in [39; 40]. Expander graphs can be constructed by a series of edge flips in regular undirected graphs. The authors introduce the k-Flipper transformation rule, which considers a given path of length $k + 2$ and interchanges the end vertices of the path. It can be shown that the continuous use of the 1-Flipper transformation on a d-regular graph G_0 constructs all connected d-regular graphs with the same probability. For $d \in \omega(1)$ a random connected d-regular graph is a $\Theta(d)$-expander graph asymptotically almost surely, i.e., with probability $p \geq 1 - o(1)$.

[2]Note that depending on the probability distribution of these long-range links, the random graph may or may not be an expander.

A generalization of the 1-Flipper rule achieves the following result, which informally states that within a polynomial number of edge flips one is able to construct a random d-regular connected graph that is a $\Theta(d)$-expander.

THEOREM 6.2 ([39]). *If we choose $d \in \Omega(\log n)$ applying $\mathcal{O}(dn)$ random $\Theta(d^2 n^2 \log 1/\varepsilon)$-Flipper operations transforms any given d-regular connected graph into a connected d-regular graph with expansion $\Theta(d)$ with high probability.*

A different rule called "Flip" for edge flipping has been introduced in [27]: one such operation replaces the edges (i, j) and (k, l) by edges (i, k) and (j, l), if and only if i and l are adjacent to each other. Starting from a connected graph, it has been shown that this Flip operation defines a Markov chain on all connected graphs, i.e., we can construct any connected graph G' when starting with the connected graph G by a series of Flip operations. Finally, it has been shown in [14] that the Flip Markov chain is rapidly mixing for regular graphs.

6.3. Transitive Closure Framework

A framework to derive self-stabilizing algorithms for any desired topology is the Transitive Closure Framework (TCF) [9]. The idea of the framework is as follows: Each node periodically receives its 2-neighborhood and locally checks if it is in a legitimate state. If there exists a node in the network, for which this is not the case, then it becomes a *detector* and spreads the event that the system is in an illegitimate state through the network. This leads to every node eventually becoming a detector. Detectors expand their neighborhood to eventually contain all nodes in V. Thus the system eventually reaches a clique. Once the clique has been generated, each node is able to compute its correct set of neighbors within one round, using the FORWARD primitive to delegate edges not needed by a node until the edge is merged at some node (recall that we are not allowed to simply delete an edge as connectivity may be lost this way).

The following bound on the runtime of TCF has been shown in [9]:

THEOREM 6.3 ([9]). *The Transitive Closure Framework is self-stabilizing, i.e.,*

(i) it transforms any weakly connected graph $G = (V, E)$ into any locally-checkable family of overlay networks in at most $D(n) + \log n + 1$ rounds, where $D(n)$ is the maximum distance between a non-detector node and its closest detector (Convergence) and

(ii) if the explicit edges in G already form the desired topology, then they are preserved at any point in time (Closure).

One may confer that $D(n)$ may become quite large when considering any initial state. However, the authors of [9] showed for a wide variety of overlay networks that $D(n)$ and the diameter of the network in a legitimate state are asymptotically identical. For example, this leads to a $O(\log n)$-time self-stabilizing algorithm for the SKIP+ graph, which is optimal, as any self-stabilizing algorithm needs time of at least the diameter of the network in legitimate states.

Another framework for generating different families of graphs is AVATAR [8], which also has the advantage that the degree of nodes increases only by a polylogarithmic factor in expectation during stabilization.

7. OTHER ASPECTS

There are several specific aspects in topological self-stabilization which have received special attention. For example, besides supporting joins and leaves, overlay networks also need to provide fast search. Another important aspect is how to support special leave operations, beyond simple crashes. In the following, we discuss these two aspects in more detail.

7.1. Monotonic Searchability

Consider the sorted list from Section 4.1 and assume a node v wants to send a packet P to the node with identifier $id \in \mathbb{N}$. Before it can do that, v first has to search for the reference of the node with id t in the sorted list. This is done by v generating a SEARCH(v, id) request that is forwarded along the list until it *terminates*, i.e., it either arrives at the desired node, or the underlying routing algorithm concludes that the request cannot be forwarded anymore. Upon termination the result of the request is directly sent back to v (this is why the request has to contain v's reference). A trivial routing algorithm would be to just forward SEARCH(v, id) at each node u via either $u.left$ or $u.right$, depending on which of these node's ids are closer to id (Algorithm 1).

Algorithm 1 Routing Algorithm for the Sorted List

```
1: procedure SEARCH(v, sid)                                   ▷ Executed by node u
2:     if u.id = sid then
3:         "Success", terminate
4:     if u.left < sid < u.id or u.id < sid < u.right then
5:         "Failure", terminate                                ▷ Guarantees liveness
6:     if sid < u.id then
7:         u.left ← SEARCH(v, sid)
8:     else
9:         u.right ← SEARCH(v, sid)
```

A desired property for SEARCH requests is to guarantee *liveness*, i.e., to guarantee that each SEARCH request eventually terminates. One can easily see that the above algorithm for the sorted list trivially satisfies liveness. More involved strategies have to be considered if we also want to guarantee *safety* properties. Regarding SEARCH requests, *monotonic searchability* is considered as an important property:

Definition 7.1 *([57])*. A self-stabilizing protocol satisfies *monotonic searchability* according to some routing protocol $\mathcal{R}$, if it holds for any pair of nodes v, w that once a SEARCH$(v, w.id)$ request (that is routed according to $\mathcal{R}$) initiated at time t succeeds, any SEARCH$(v, w.id)$ request initiated at time $t' > t$ will succeed.

Protocols that satisfy monotonic searchability have the advantage to be able to provide guarantees on SEARCH requests even while the recovery process is going on. Due to the modification of edges by the self-stabilizing protocol, realizing monotonic searchability is a non-trivial problem: One can see that the above routing algorithm for the sorted list does not satisfy monotonic searchability. Consider Figure 1.9 for an example.

Research on monotonic searchability was initated in [56] where the authors came up with a protocol for the sorted list that satisfies monotonic searchability. The idea is that nodes are allowed to have multiple left and right neighbors while the system recovers. An edge (u, w) that should be forwarded to some node v is not directly forwarded but only removed from u's local storage after v has acknowledged the implicit edge to w. The authors showed that this extension still leads to the system converging to

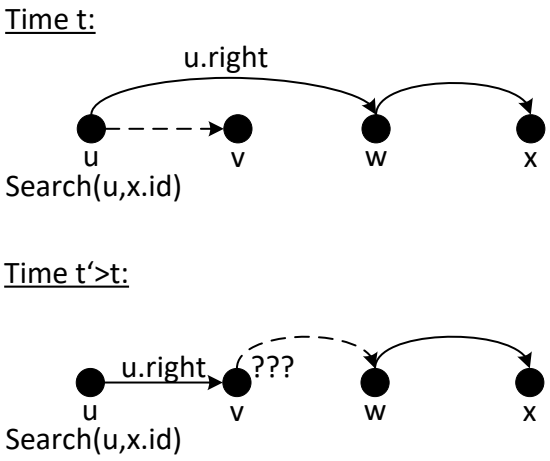

Fig. 1.9. Example illustrating why the trivial routing algorithm for the sorted list fails to guarantee monotonic searchability. At time t the search request is able to reach the target node x, but this cannot be guaranteed anymore after u performed LINEARIZE at time $t' > t$ and the search request arrives at v before the reference of w indicated by the implicit edge (v, w).

a sorted list. Furthermore, when using a appropriate routing protocol, monotonic searchability can be shown as well.

Research on monotonic searchability culminated in a generic approach that can be applied to a wide range of self-stabilizing protocols such that monotonic searchability is guaranteed [57]. Along with a generic search protocol, the approach introduces a new set of primitives for manipulating edges that allow the safer delegation of edges than the primitives described in Section 3.1, while still being provably universal. The cost that one pays when applying that protocol is that SEARCH requests may traverse up to $\Omega(n)$ hops until termination when searching for non-existent nodes in the system.

In order to avoid the above costs, one can still ask if there are protocols for specific topologies that satisfy monotonic searchability. Recently the authors of [38] presented a specialized protocol in order to guarantee monotonic searchability in the perfect skip graph. The perfect skip graph is the deterministic version of the skip graph and has the advantage over the SKIP+ graph that it can be built in a self-stabilizing manner using a probing approach, without having to rely on additional edges to enable local checkability.

Last but not least we note that the protocol for the self-stabilizing quadtree from [15] has been constructed in a way such that monotonic searchability can be guaranteed trivially.

Whether there are even more efficient topology-specific protocols remains an open problem.

7.2. Node Departures

Another challenging task arises when studying node departures, i.e., when nodes are allowed to leave the system. In an ideal scenario nodes may just leave the system, as we then rely on the self-stabilizing protocol to stabilize the system again. This approach however is flawed for two reasons: First, if the expansion of the topology is not high enough (consider for example the sorted list), then a leaving node may disconnect the overlay, implying that stabilization to a sorted list is not possible. The second reason is that even if the expansion of the topology is high, it is still not guaranteed to be high in illegitimate states, but only in legitimate states. Thus, leaving nodes may still be able to disconnect the overlay.

In topological self-stabilization, node departures have been first studied in [18], where nodes are either *staying* (the process wants to remain in the system) or *leaving* (the process wants to be excluded from the system). Leaving nodes are allowed to enter the state **exit** or **sleep**. A process that is in state **exit** is gone, i.e., it does not execute any actions anymore. If a process is in state **sleep**, it does not execute any action until another process invokes an action on it via a message. Processes that are neither in states **exit** or **sleep** are called *awake*. The challenge is to stabilize the system to a legitimate state with the following properties:

(i) Every staying process is awake.
(ii) Every leaving process is either in state **exit** or permanently in state **sleep**.
(iii) All staying processes form a weakly connected component.

The authors of [18] introduced the following problems:

— *Finite Departure Problem ($\mathcal{FDP}$)*: Eventually reach a legitimate state in case that only the state **exit** is available.
— *Finite Sleep Problem ($\mathcal{FSP}$)*: Eventually reach a legitimate state in case that only the state **sleep** is available.

An important (negative) result is that the $\mathcal{FDP}$ cannot be solved by local control protocols. Instead one has to make use of oracles:

THEOREM 7.2. *Any self-stabilizing solution for the $\mathcal{FDP}$ has to rely on an oracle.*

On the positive side however, the $\mathcal{FSP}$ can actually be solved by local control protocols.

Research on node departures culminated in an universal protocol for the $\mathcal{FDP}$ [33] that can be applied to a wide range of self-stabilizing protocols. As already mentioned above, this is of great importance even if the expansion of the topology in legitimate states is high.

Recently it has been shown that when considering a new interconnection model based on relays, the $\mathcal{FDP}$ can be solved even without relying on an oracle [55].

8. CONCLUSION AND OPEN QUESTIONS

This paper presented an overview of the state-of-the-art techniques to design self-stabilizing overlay networks: overlays which are highly fault-tolerant in the sense that they reconfigure to reach a desirable state, in a distributed manner, from arbitrary initial states. In particular, we proceeded in a bottom-up manner, starting with basic connectivity primitives, then studying algorithms for the fundamental linearization problem, moving to more general geometric graphs, and finally reaching scalable networks based on skip graphs and random graphs.

While today we have a fairly good understanding of the design of algorithms for self-stabilizing networks, a number of important problems remain open. In the following, we discuss some of them.

— ***Upper and Lower Bounds on Work:*** The main focus in existing literature on the design of self-stabilizing topologies is on the *feasibility and correctness* of distributed algorithms. While we have presented some results on the *runtime* of self-stabilizing algorithms in this paper, particularly little is known today (with some exceptions [25],[31]) about the *work* required for topological self-stabilization. Results on upper and lower bounds on the work of self-stabilizing algorithms for different overlay topologies, will be very interesting.

— ***Transient Behavior:*** We also lack a good understanding of the achievable *transient* properties, during convergence. For example, while self-stabilizing networks such as SKIP+ are scalable in the sense that they rely on graphs providing a polylogarithmic degree and diameter, scalability may not be ensured *during convergence*: during convergence, the degree of a node may temporarily raise significantly, far beyond the maximum initial or final node degree. A major open question hence regards the design of algorithms which also provide scalability *during convergence*. To give

another example, consider the diameter: it may be desirable that the diameter never significantly increases during convergence. At the same time, we also note that for some specific problems, such as linearization, achieving, e.g., convergence without increasing degrees, is simple, and an interesting avenue for future research is to investigate to which extent such results can be generalized.

— ***Locality:*** While a self-stabilizing algorithm by definition will re-establish a desired property from *any* initial configuration, it is desirable that the parallel convergence time as well as the overall work is proportional to "how far" the initial topology is from the desired one. We currently lack a general theoretical understanding of what can and cannot be achieved in terms of such local properties. We also lack a deep understanding of how the "distance" between initial and desired topology can be defined generally; it may also depend on the application.

— ***Churn Tolerance:*** We currently lack insights into the rate of joins and leaves an overlay network can tolerate. Indeed, while there exists work on overlays which are "churn-tolerant" [34; 49; 21], this line of research has been largely independent of research on self-stabilizing algorithms. Bringing these two fields together may lead to very interesting and relevant research insights.

REFERENCES

[1] Dana Angluin, James Aspnes, Jiang Chen, Yinghua Wu, and Yitong Yin. Fast construction of overlay networks. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–154. ACM, 2005.

[2] James Aspnes and Gauri Shah. Skip graphs. *Acm transactions on algorithms (talg)*, 3(4):37, 2007.

[3] James Aspnes and Yinghua Wu. O (logn)-time overlay network construction from graphs with out-degree 1. In *International Conference On Principles Of Distributed Systems*, pages 286–300. Springer, 2007.

[4] Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust dht. *Theory of Computing Systems*, 45(2):234–260, 2009.

[5] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 206–219. IEEE, 1988.

[6] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science (FOCS)*, pages 258–267. IEEE, 1991.

[7] Markus Benter, Mohammad Divband, Sebastian Kniesburges, Andreas Koutsopoulos, and Kalman Graffi. Ca-re-chord: A churn resistant self-stabilizing chord overlay network. In *2013 Conference on Networked Systems, NetSys 2013, Stuttgart, Germany, March 11-15, 2013*, pages 27–34, 2013.

[8] Andrew Berns. Avatar: A time- and space-efficient self-stabilizing overlay network. In *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, pages 233–247, 2015.

[9] Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 62–76, 2011.

[10] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. Renaissance: A self-stabilizing distributed SDN control plane. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 233–243, 2018.

[11] Augustin Chaintreau, Pierre Fraigniaud, and Emmanuelle Lebhar. Networks become navigable as nodes move and forget. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 133–144, 2008.

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

[13] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

[14] Tomás Feder, Adam Guetz, Milena Mihail, and Amin Saberi. A local switch markov chain on given degree graphs with application in connectivity of peer-to-peer networks. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 69–76, 2006.

[15] Michael Feldmann, Christina Kolb, and Christian Scheideler. Self-stabilizing overlays for high-dimensional monotonic searchability. In *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, pages 16–31, 2018.

[16] Michael Feldmann and Christian Scheideler. A self-stabilizing general de bruijn graph. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, pages 250–264, 2017.

[17] Matthias Feldotto, Christian Scheideler, and Kalman Graffi. Hskip+: A self-stabilizing overlay network for nodes with heterogeneous bandwidths. In *14th IEEE International Conference on Peer-to-Peer Computing, P2P 2014, London, United Kingdom, September 9-11, 2014, Proceedings*, pages 1–10, 2014.

[18] 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.

[19] 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.

[20] Robert Gmyr, Jonas Lefèvre, and Christian Scheideler. Self-stabilizing metric graphs. *Theory Comput. Syst.*, 63(2):177–199, 2019.

[21] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. *SIGCOMM Comput. Commun. Rev.*, 36(4):147–158, August 2006.

[22] Thorsten Götte, Christian Scheideler, and Alexander Setzer. On underlay-aware self-stabilizing overlay networks. In *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, pages 50–64, 2018.

[23] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[24] Cheng Huang, Angela Wang, Jin Li, and Keith W Ross. Understanding hybrid cdn-p2p: why limelight needs its own red swoosh. In *Proc. 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 75–80. ACM, 2008.

[25] 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.

[26] 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.

[27] Thomas Janson, Peter Mahlmann, and Christian Schindelhauer. A self-stabilizing locality-aware peer-to-peer network combining random networks, search trees, and dhts. In *16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010*, pages 123–130, 2010.

[28] M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *International Workshop on Peer-to-Peer Systems*, pages 98–107. Springer, 2003.

[29] Jon M Kleinberg. Navigation in a small world. *Nature*, 406(6798):845, 2000.

[30] 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.

[31] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. A deterministic worst-case message complexity optimal solution for resource discovery. In *Structural Information and Communication Complexity - 20th International Colloquium, SIROCCO 2013, Ischia, Italy, July 1-3, 2013, Revised Selected Papers*, pages 165–176, 2013.

[32] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: A self-stabilizing chord overlay network. *Theory Comput. Syst.*, 55(3):591–612, 2014.

[33] 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.

[34] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *International Workshop on Peer-to-Peer Systems*, pages 13–23. Springer, 2005.

[35] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *ACM SIGOPS Operating Systems Review*, 19(4):34–44, 1985.

[36] Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: Self-stabilization on speed. In *Symposium on Self-Stabilizing Systems*, pages 17–34. Springer, 2009.

[37] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, Steven Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and tutorials*, 7(1-4):72–93, 2005.

[38] Linghui Luo, Christian Scheideler, and Thim Strothmann. MULTISKIPGRAPH: A self-stabilizing overlay network that maintains monotonic searchability. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 845–854, 2019.

[39] Peter Mahlmann and Christian Schindelhauer. Peer-to-peer networks based on random transformations of connected regular undirected graphs. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 155–164, 2005.

[40] Peter Mahlmann and Christian Schindelhauer. Distributed random digraph transformations for peer-to-peer networks. In *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*, pages 308–317, 2006.

[41] Peter Mahlmann and Christian Schindelhauer. *Peer-to-peer-netzwerke: Algorithmen und Methoden*. Springer-Verlag, 2007.

[42] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[43] Moni Naor and Udi Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 7-9, 2003, San Diego, California, USA (part of FCRC 2003)*, pages 50–59, 2003.

[44] Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. *Theor. Comput. Sci.*, 512:119–129, 2013.

[45] Melih Onus, Andrea Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 99–108, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[46] Radia J. Perlman. An algorithm for distributed computation of a spanningtree in an extended LAN. In *SIGCOMM '85, Proceedings of the Ninth Symposium on Data Communications, British Columbia, Canada, September 10-12, 1985*, pages 44–53, 1985.

[47] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *See https://lightning.network/lightning-network-paper.pdf*, 2016.

[48] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, pages 161–172, 2001.

[49] Sean Rhea, Dennis Geels, Timothy Roscoe, John Kubiatowicz, et al. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, volume 6, pages 127–140. Boston, MA, USA, 2004.

[50] Andréa W. Richa and Christian Scheideler. Overlay networks for peer-to-peer networks. In *Handbook of Approximation Algorithms and Metaheuristics, Second Edition, Volume 2: Contemporary and Emerging Applications*. 2018.

[51] Andréa W. Richa, Christian Scheideler, and Phillip Stevens. Self-stabilizing de bruijn networks. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 416–430, 2011.

[52] Christina Rickmann, Christoph Wagner, Uwe Nestmann, and Stefan Schmid. Topological self-stabilization with name-passing process calculi. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 19:1–19:15, 2016.

[53] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[54] Christian Scheideler. How to spread adversarial nodes?: rotate! In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 704–713. ACM, 2005.

[55] Christian Scheideler and Alexander Setzer. Relays: A new approach for the finite departure problem in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, pages 239–253, 2018.

[56] Christian Scheideler, Alexander Setzer, and Thim Strothmann. Towards establishing monotonic searchability in self-stabilizing data structures. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 24:1–24:17, 2015.

[57] Christian Scheideler, Alexander Setzer, and Thim Strothmann. Towards a universal approach for monotonic searchability in self-stabilizing overlay networks. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 71–84, 2016.

[58] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 225–238, 2012.

[59] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[60] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 205–219. ACM, 2016.

[61] Ge Xia. The stretch factor of the delaunay triangulation is less than 1.998. *SIAM J. Comput.*, 42(4):1620–1659, 2013.

[62] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.