Abstract—In this paper we present two major results: First, we introduce the first self-stabilizing version of a supervised overlay network (as introduced in [1]) by presenting a self-stabilizing supervised skip ring. Secondly, we show how to use the self-stabilizing supervised skip ring to construct an efficient self-stabilizing publish-subscribe system. That is, in addition to stabilizing the overlay network, every subscriber of a topic will eventually know all of the publications that have been issued so far for that topic. The communication work needed to process a subscribe or unsubscribe operation is just a constant in a legitimate state, and the communication work of checking whether the system is still in a legitimate state is just a constant on expectation for the supervisor as well as any process in the system.

Keywords—Topological Self-stabilization; Supervised Overlay; Publish-Subscribe System

I. INTRODUCTION

The publish subscribe paradigm ([4], [5]) is a very popular paradigm for the targeted dissemination of information. It allows clients to subscribe to certain topics or contents so that they will only receive information that matches their interests. In the traditional client-server approach the dissemination of information is handled by a server (also called broker), which has the benefit that the publishers are decoupled from the subscribers: the publisher does not have to know the relevant subscribers, and the publisher and subscribers do not have to be online at the same time. However, in this case the availability of the publish subscribe system critically depends on the availability of the server, and the server has to be powerful enough to handle the dissemination of the publish requests. An alternative approach is to use a peer-to-peer system. However, if no commonly known gateway is available, the peer-to-peer system cannot recover from overlay network partitions. In practice, peer-to-peer systems usually have a commonly known gateway since otherwise new peers may not be able to get in contact with a peer that is currently in the system (and can therefore process the join request). In our supervised overlay network approach we assume that there is a commonly known gateway, called supervisor, that just handles subscribe and unsubscribe requests but does not handle the dissemination of publish requests, which will be handled by the subscribers in a peer-to-peer manner. We are interested in realizing a topic-based supervised publish subscribe system, which means that peers can subscribe to certain topics (that are usually relatively broad and predefined by the supervisor).

Topic-based publish subscribe systems have many important applications. Apart from providing a targeted news service, they can be used, for example, to realize a group communication service [6], which is considered an important building block for many other applications ranging from chat groups and collaborative working groups to online market places (where clients publish service requests), distributed file systems or transaction systems. To ensure the reliable dissemination of publish requests in a topic-based publish subscribe system, we present a self-stabilizing supervised publish subscribe system, which ensures that for any initial state (including overlay network partitions) eventually a legitimate state will be reached in which all subscribers of a topic know about all publish requests that have been issued for that topic. We also show that the overhead for the supervisor in our system is very low. In fact, the message overhead of the supervisor is just a constant for subscribe and unsubscribe operations, and the supervisor has a low maintenance overhead in a legitimate state.

A. Model

We model the overlay network of a distributed system as a directed graph $G = (V, E)$, where $n = |V|$. Each peer is represented by a node $v \in V$. Each node $v \in V$ is identified by its unique reference or 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. We assume a channel to be able to store any finite number of messages, and messages are never duplicated or get lost in the channel. If a node $u$ has the reference of some other node $v$, $u$ can send a message $m$ to $v$ by putting $m$ into $v.Ch$. There is a directed edge $(u, v) \in E$ whenever $u$ stores a reference of $v$ in its local memory or 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. Note that every node is assumed to know the supervisor, and this information is read-only, so $G$ always contains a directed star graph from all peers to the supervisor.

*This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901)
Nodes may execute actions: An action is just a standard procedure and has the form \(\text{label}(\text{parameters})\) : \(\text{command}\), where \text{label} is the name of that action, \text{parameters} defines the set of parameters and \text{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 \(\text{label}(\text{parameters})\). When a node \(u\) processes a message \(m\), then \(m\) is removed from \(u.Ch\). Additionally, there is an action that is not triggered by messages but is executed periodically by each node. We call this action \text{TIMEOUT}.

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 \text{initial state}. We assume fair message receipt, meaning that every message of the form \(\text{label}(\text{parameters})\) 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. Consider the \text{TIMEOUT} action as an example for this. We place no bounds on message propagation delay or relative node execution speed, i.e., we allow fully asynchronous computations and non-FIFO message delivery. Our protocol does not manipulate node identifiers and thus only operates on them in compare-store-send mode, i.e., the nodes are only allowed to compare node IDs, store them in a node's local memory or send them in a message.

In this paper we assume for simplicity that there are no corrupted IDs (i.e., IDs of unavailable nodes) in the initial state of the system. However, dealing with them is easy when having a failure detector that is eventually correct since, due to the supervisor, the correctness of our protocol cannot be endangered by sending messages to non-available nodes. Since our protocol just deals with IDs in a compare-store-send manner, this implies that node IDs will always be non-corrupted for all computations. Nevertheless, the node channels may initially contain an arbitrary finite number of messages containing false information. We call these messages corrupted, and we will argue that eventually there will not be any corrupted messages in the system. We will show that our protocol realizes a self-stabilizing supervised publish-subscribe system.

**Definition 1 (Self-stabilization).** A protocol is self-stabilizing w.r.t. a set of legitimate states 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.

### B. Related Work

The concept of self-stabilizing algorithms for distributed systems goes back to the year 1974, when E. W. Dijkstra introduced the idea of self-stabilization in a token-based ring [7]. Many self-stabilizing protocols for various types of overlays have been proposed, like sorted lists [8], de Bruijn graphs [9], Chord graphs [2] and many more. There is even a universal approach, which is able to derive self-stabilizing protocols for several types of topologies [10].

The cycle topology is particularly important for our work. Our cycle protocol is based on [3], in which the authors construct a self-stabilizing cycle that acts as a base for additional long-range links, both together forming a small-world network.

The paper closest to our work is by Kothapalli and Scheideler [1]. The authors provide a general framework for constructing a supervised peer-to-peer system in which the supervisor only has to store a constant amount of information about the system at any time and only has to send out a constant number of messages to integrate or remove a node. However, their system is not self-stabilizing.

In the literature there are publish-subscribe systems that are self-stabilizing: e.g. in [11] the authors present different content-based routing algorithms in a self-stabilizing (acyclic) broker overlay network that clients can publish messages to. Their main idea is a leasing mechanism for routing tables such that it is guaranteed that once a client subscribes to a topic there is a point in time such that every publication which is issued thereafter is delivered to the newly subscribed client (i.e., there are no guarantees for older publications). While the authors focus on the routing tables and take the overlay network as a given ingredient, our work focuses on constructing a self-stabilizing supervised overlay network and then using it to obtain a self-stabilizing publish-subscribe system.

A self-stabilizing publish-subscribe system for wireless ad-hoc networks is proposed in [12], which builds upon the work of [13]: Similar to our work, the authors arrange nodes in a cycle with shortcuts and present a routing algorithm that makes use of these shortcuts to deliver new publications for topics to subscribers only after \(O(n)\) steps. Subscribe and unsubscribe requests are processed by updating the routing table at nodes. Both systems described above differ from our approach, as they solely focus on the routing scheme and updates of the routing tables, while we focus on updating the topology upon subscribe/unsubscribe requests. Additionally, our system is able to deliver publications in \(O(\log n)\) steps, if we use flooding, since we use a network with logarithmic diameter. Furthermore, we are also able to deliver all publications of a domain to a new subscriber after only a constant number of rounds.
There is a close relationship between group communication services (e.g., [6], [14]) and publish-subscribe systems. Processes are ordered in groups in both paradigms and group-messages are only distributed among all members of some group. Self-stabilizing group communication services are proposed in [15] for ad-hoc networks and in [16] for directed networks. However, there are some key differences: In group communication services, participants have to agree on group membership views. This results in a high memory overhead for each member of a group, as nodes in a group technically form a clique. On the other hand subscribers of topics in publish-subscribe systems are in general not interested in any other members of the topic. For our approach, this results in logarithmic worst-case and constant average case degree for subscribers.

C. Our Contributions

To the best of our knowledge, we present the first self-stabilizing protocol for a supervised overlay network. We focus on a topology that is a ring with shortcuts which we call skip ring. The corresponding protocol BUILDRING is split up into two subprotocols: One protocol is executed at the supervisor (see Section III-A), the other one is executed by each subscriber (Section III-B). Our basic protocol assumes that all references actually belong to existing nodes. However, we also present an extension (see Section III-C) to handle references to non-existing nodes and unannounced failures of nodes. In contrast to the supervised overlay network proposed in [1], our new protocol lets the supervisor handle multiple insertions/deletions in parallel without having to rely on confirmations from other nodes, however, at the cost of storing much more references than the solution in [1].

The skip ring shares some similarities with other shortcut-based peer-to-peer systems like Chord networks [2] or skip graphs [17]. However, our network has a better congestion than these networks, as the supervised approach allows a much more balanced distribution of these nodes.

We show how to use the supervised skip ring to obtain a self-stabilizing publish-subscribe system (see Section IV) in which each skip ring corresponds to a topic. Every subscriber of a topic eventually gets all publications that have been issued so far for that topic. The shortcuts in the skip ring are helpful when using flooding to distribute new publications among all subscribers, since a skip ring of \( n \) nodes has diameter \( \log n \).

In our self-stabilizing publish-subscribe system the message overhead of the supervisor is linear in the number of topics (but not in the number of subscribers), if we use a simple generalization strategy in which each topic corresponds to one skip ring. This, of course, decreases the applicability of our system in large-scale scenarios. However, better scalability can be achieved by organizing topics in a hierarchical manner, or by having different supervisors for each topic. For the latter scenario, one could make use of a self-stabilizing distributed hash table (with consistent hashing) for all supervisors, in which a sub-interval of \([0,1)\) is assigned to each supervisor. By hashing IDs of topics in the same manner, each supervisor is then only responsible for the topics in its sub-interval. Since solutions for self-stabilizing distributed hash tables already exist in the literature (see e.g. [18]), we do not elaborate on them further in this paper.

Due to space restrictions, pseudocode and full versions of selected proofs are deferred to the full version of this paper [19].

II. Preliminaries

In this section we formally introduce the topology for a skip ring (Section II-A). As a base for our self-stabilizing protocol we introduce the BUILDRING protocol from [3] to arrange nodes in a sorted ring (Section II-B).

A. Skip Ring

Let \( l : \mathbb{N}_0 \rightarrow \{0,1\}^* \) be a mapping with the property that \( l(x) := (x_{d-1}, \ldots, x_0)_d \) for every \( x \in \mathbb{N}_0 \) with binary representation \( (x_d, \ldots, x_0) \) (where \( d \) is minimum possible). Intuitively, \( l \) takes the leading bit of the binary string representing the input value and moves it to the units place. In our setting the supervisor will use \( l \) to assign a (unique) label to each subscriber. Labels are generated in the order: 0, 1, 01, 11, 001, 011, 101, 111, 0001,... Note that \( l \) is invertible. We call the value \( l(x) \in \{0,1\}^* \) of \( l \) a label. Denote by \( |\text{label}| \) the minimum number of bits used to encode label. A label \( y = (y_1 \ldots y_d) \in \{0,1\}^d \) may either be represented as a bit string or as a real-valued number within \([0,1)\) by evaluating the function \( r : \{0,1\}^* \rightarrow [0,1) \) with \( r(y) := \sum_{i=1}^{d} y_i / 2^i \). The function \( r \) induces an ordering of all nodes in a ring, which will be used in the following to define the skip ring:

Definition 2 (Skip Ring). A skip ring \( SR(n) \) is a graph \( G = (V, E_R \cup E_S) \) with \( n \) nodes. \( G \) is defined as follows:

- Each node \( v \in V \) has a unique label denoted by \( \text{label}_v \in \{0,1\}^* \) with \( l^{-1}(\text{label}_v) < n \).
- \( (u,v) \in E_R \leftrightarrow (u,v) \) are consecutive in the ordering induced by \( r \). Denote the edges in \( E_R \) as ring edges.
- \( (u,v) \in E_S \leftrightarrow (u,v) \) is part of the sorted ring w.r.t. node labels over all nodes in \( K_i, i \in \{1, \ldots, \lceil \log n \rceil \} \), where \( K_i := \{ w \in V \mid |\text{label}_w| \leq i \} \). Denote \( (u,v) \in E_S \) as a shortcut on level \( i \), if \( i = \max\{|\text{label}_u|, |\text{label}_v|\} \).

The label of a node \( v \in V \) is independent from its unique ID \( v.id \) and will be determined by the supervisor.

The intuition behind \( E_R \) and \( E_S \) is that we want all nodes with label of length at most \( k \) to form a (bidirected) sorted ring for all \( k \in \{1, \ldots, \lceil \log n \rceil \} \). For \( k = \lceil \log n \rceil \) these edges are stored in \( E_R \), for \( k < \lceil \log n \rceil \) they are stored in \( E_S \). Due to the way we defined the function \( l \) it holds that for all \( x \in \{2^d, \ldots, 2^{d+1} - 1\} \) the values \( r(l(x)) \) are uniformly distributed among all nodes in a sorted ring for all \( k \in \{1, \ldots, \lceil \log n \rceil \} \).
spread in between old values \( r(l(y)) \) with \( y \in \{0, \ldots, 2^d - 1\} \). This implies that the longer a node is a participant of the system, the more shortcuts it has. This makes sense from a practical point of view, since older and thus more reliable nodes hold more connectivity responsibility in form of more shortcuts.

The decision whether two nodes are connected or not only depends on the labels of the nodes, which means that an arrival/departure of a node only affects its neighbors (see Section IV-A for details). Figure 1 illustrates \( SR(16) \).

The following Lemma follows from the definition of \( SR(n) \):

**Lemma 3 (Node Degree).** In a legitimate state, the degree of nodes in a skip ring is logarithmic in the worst case and constant in the average case.

**B. Self-Stabilizing Ring**

The base of our self-stabilizing protocol is the BUILDRING protocol from [3] that organizes all nodes in a sorted ring according to their labels, using linearization [8]: Each node \( v \in V \) stores edges to its closest left and right neighbors (denoted by \( v.left, v.right \in V \)) according to \( v \)'s label (denoted by \( v.label \)). Any other nodes \( u \) are delegated by \( v \) to either \( v.left \) or \( v.right \) (depending on which node is closer to \( u \)). Additionally the node with minimum label stores an edge to the node with maximum label and vice versa, such that the sorted ring is closed. Nodes \( v \) periodically introduce themselves to their neighbors \( v.left \) and \( v.right \) in the sorted ring: This means that \( v \) sends a message to \( v.left/v.right \) containing a reference to itself. This way nodes can check, if the sorted ring is in a legitimate state from their point of view or not.

In our setting nodes may assume corrupted labels for their neighboring nodes in any nonlegal state: If node \( v \in V \) has an edge to \( w \in V \), then \( v \) locally stores the tuple \((label_w, w)\). While the reference to \( w \) is assumed to be correct by definition at any time, \( w \)'s variable \( w.label \) may change to a different value at some point in time. Unfortunately, \( v \) still has the old label value associated with \( w \), implying that \( label_w \neq w.label \). As a consequence, we extend the BUILDRING protocol as follows: Whenever a node \( v \in V \) introduces itself to another node \( w \in V \), then \( v \) informs \( w \) about the label \( label_w \) that \( v \) thinks is assigned to \( w \). Node \( w \) then checks the label for correctness by comparing \( w.label \) with \( label_w \), and if \( label_w \neq w.label \), \( w \) sends \( v \) its correct value of \( w.label \).

Including the modifications mentioned above, the extended BUILDRING protocol is still self-stabilizing:

**Lemma 4.** The BUILDRING protocol with its extension is self-stabilizing.

**III. SELF-STABILIZING SUPERVISED SKIP RING**

In this section we first extend the skip ring topology by introducing a supervisor. The description of our BUILDSR is then split into two sub-protocols: One sub-protocol is executed by the supervisor, the other one is executed by every other node. Adapting publish-subscribe terminology, we denote a node \( v \in V \) as subscriber for the rest of the paper.

Recall that every subscriber \( v \) is assumed to know the supervisor \( s \), and this information is read-only, so the graph \( G \) always contains edges \((v, s)\). The assumption of having such a supervisor is not far-fetched, because even pure peer-to-peer systems need a common gateway that acts as an entrance point for peers.

Our goal in this section is to construct a self-stabilizing protocol in which subscribers form a skip ring with the help of the supervisor, starting from any initial state. The extension to a self-stabilizing publish-subscribe system is then described in Section IV.

**A. Supervisor Protocol**

The first part of the BUILDSR protocol is executed by the supervisor. The supervisor maintains the following variables:

- A database \( \subseteq \{0,1\}^* \times V \) containing subscribers and their corresponding labels. Denote \( n := |\text{database}| \).

- A variable next \( \in \mathbb{N} \) that is used to notify subscribers in a round-robin fashion.

In the supervisor’s **TIMEOUT** method, the supervisor chooses a subscriber \( v \) in a round-robin fashion (using the variable **next**) from its database. Then the supervisor sends a message to \( v \) containing \( v \)'s label \( label_v \), as well as the correct values for \( v \)'s predecessor \( pred_v \) and successor \( succ_v \) according to the database. We call such a triple \((pred_v, label_v, succ_v)\) the configuration for \( v \).
In addition to the above action, the supervisor has to check the integrity of its database: We say that the database of $s$ is corrupted, if at least one of the following conditions hold:

(i) There exists a tuple $(\text{label}, v) \in \text{database}$ with $v = \perp$. (There exists a tuple without any subscriber.)
(ii) There exist entries $(\text{label}_1, v_1), (\text{label}_2, v_2) \in \text{database}$ with $\text{label}_1 \neq \text{label}_2$ and $v_1 = v_2$. (There exist multiple tuples storing the same subscriber)
(iii) There exists $i \in \{0, \ldots, n-1\}$, s.t. for all $(\text{label}_v, v) \in \text{database}$ it holds $\text{label}_w \neq l(i)$. (There are labels missing)
(iv) There exists $i \geq n$, s.t. there is a tuple $(\text{label}, v) \in \text{database}$ with $\text{label} = l(i)$. (There exists a tuple with an incorrect label)

All of these cases may occur in initial states. Note that when using a hashmap for the database, we do not need to check explicitly whether there are multiple tuples with the same label or whether there are tuples with the label set to $\perp$. We perform the following actions to tackle the above 4 cases:

(i) Upon detecting a tuple $(\text{label}, \perp) \in \text{database}$, we simply remove it from the database.
(ii) Whenever a subscriber $v$ wants to unsubscribe or request its configuration, the supervisor first searches the database for all tuples $(l, w)$ with $w = v$. It then removes all duplicates except the tuple with lowest label, guaranteeing that $v$ is associated with no more than one label.
(iii) In TIMEOUT the supervisor checks for all $i \in \{0, \ldots, n-1\}$ if there is a tuple $(l(i), v)$ stored in the database. If not, then the supervisor takes the tuple $(l(j), w) \in \text{database}$ with maximum $j \in \mathbb{N}$, $j > i$ and replaces its label with $l(i)$.
(iv) It is easy to see that the action for (iii) also solves case (iv).

Observe that all of these actions are performed locally by the supervisor, i.e., they generate no messages. Therefore we assume that the database of the supervisor is always in a non-corrupted state from this point on.

B. Subscriber Protocol

In this section we discuss the part of the BUILDSR protocol that is executed by each subscriber. First, we present the variables needed for a subscriber. Note that we intentionally omit the reference to the supervisor $s$ here, since $s$ is assumed to be hard-coded. A subscriber $v \in V$ stores the following variables:

- $v.\text{label} \in \{0, 1\}^* \cup \{\perp\}$: The unique label of $v$ or $\perp$ if $v$ has not received a label yet.
- $v.\text{left}, v.\text{right}, v.\text{ring} \in \{0, 1\}^* \times (V \cup \{\perp\})$: Left and right neighbor in the ring as well as the cyclic connection in case $r(v.\text{label})$ is minimal/maximal.

- $v.\text{shortcuts} \subset \{0, 1\}^* \times (V \cup \{\perp\})$: All of $v$’s shortcuts.

For the rest of the protocol description, we use $v.\text{left}$ and $v.\text{right}$ to indicate $v$’s left (resp. right) neighbor in the ring even if the left (resp. right) neighbor is stored in $v.\text{ring}$ instead of $v.\text{left}$ (resp. $v.\text{right}$). We also may refer to the variables $v.\text{left}, v.\text{right}, v.\text{ring}$ as $v$’s direct ring neighbors. Recall that each subscriber executes the extended BUILDRING protocol from Section II-B.

1) Receiving correct Labels: For now we focus on the ring edges only. Our first goal is to guarantee that every subscriber $v$ eventually stores its correct label in $v.\text{label}$.

Recall that we have periodic communication from the supervisor to the subscribers, i.e., the supervisor periodically sends out the configurations to all subscribers stored in its database. This action alone does not suffice in order to make sure that every subscriber eventually stores its correct label, since in initial states the database may be empty and subscriber labels may store arbitrary values. Thus, we also need periodic communication from subscribers to the supervisor. The challenge here is to not overload the supervisor with requests in legitimate states of the system. Each subscriber $v$ periodically executes the following actions:

(i) If $v.\text{label} = \perp$, then $v$ asks the supervisor to integrate $v$ into the database and send $v$ its correct configuration.
(ii) If $v.\text{label} \neq \perp$, then, with probability $1/(2^{k^2} \cdot k^2)$, $v$ asks the supervisor for its correct configuration, where $k = |v.\text{label}|$.

Action (ii) is dedicated to handle subscribers that have incorrect labels or already store a label, but are not known to the supervisor. Upon receiving a configuration request from a subscriber $v$, the supervisor integrates $v$ into the database (if it is not already contained in the database) and sends $v$ its configuration and thus its correct label.

We still need some further actions to tackle special initial states: Imagine a subscriber $v$ having a label such that the probability mentioned in (ii) becomes negligible. In case $v$ is not contained in the supervisor’s database yet, $v$ will send a configuration request to the supervisor with only very low probability. The following action is able to solve this problem under the assumption that there exists a subscriber $v$ that is already contained in the supervisor’s database and has $v$ stored as one of its direct ring neighbors.

(iii) W.l.o.g. let $w.\text{left} = (\text{label}_w, v)$. If $w$ receives a configuration from the supervisor and $\text{pred} \neq v$, it checks whether $w.\text{left}$ is closer than the left ring neighbor $(\text{label}_p, \text{pred}) \in \{0, 1\} \times V$ proposed by the configuration, i.e., $|r(\text{label}_w) - r(\text{w.\text{label}})| \leq |r(\text{label}_p) - r(\text{w.\text{label}})|$. In case this holds, $w$ requests the supervisor to send the correct configuration to $w.\text{left}$.

The assumption for action (iii) may not hold in all initial states, i.e., there is a connected component in which all
subscribers have stored labels such that the probability mentioned in (ii) becomes negligible. Note that actions (i)-(iii) suffice to show convergence in theory. In order to improve the time it takes the network to converge, we introduce one last periodic action:

(iv) Subscriber $v$ periodically requests its configuration with probability $1/2$ from the supervisor if it determines, based only on its local information, that its label is minimal.

We now sketch why eventually all subscribers in a connected component $C$ get their correct label. This is obviously the case when all subscribers in $C$ are stored in the supervisor’s database as the supervisor will then periodically hand out the correct labels in a round-robin fashion. Denote a subscriber that is already stored in the supervisor’s database as recorded. Action (iv) guarantees that we quickly have at least one recorded subscriber in a connected component $C$. Assume that $C$ still contains non-recorded subscribers. As long as the supervisor is able to introduce new recorded subscribers to recorded subscribers in $C$, $C$’s size grows, but since the number of subscribers is finite, $C$ will eventually become static. For such a static connected component $C$ we know that due to BUILDRING, subscribers in $C$ eventually form a sorted ring. Then there exists at least one ring edge from a recorded subscriber $v \in C$ to a non-recorded subscriber $w \in C$. Furthermore, $v$’s correct ring neighbor $w'$ indicated by its configuration has to be further away from $v$ than $w$ (so for instance $r(v.label) < r(w.label) < r(w'.label)$ if we consider the right neighbor of $v$). This holds because no new subscriber can be introduced to a recorded subscriber in $C$. Once $v$ receives its configuration from the supervisor, it triggers action (iii) and requests the configuration for $w$, leading to $w$ being inserted into the supervisor’s database and thus reducing the number of non-recorded subscribers in $C$ by one. This inductively implies that eventually all subscribers in $C$ are recorded.

We now want to bound the expected number of requests that are periodically sent out to the supervisor when the system is in a legitimate state. For the next lemma, denote a timeout interval as the time in which every subscriber has called its TIMEOUT method exactly once.

**Theorem 5.** Consider a supervised skip ring with $n$ subscribers in a legitimate state. The expected number of configuration requests sent out by all subscribers is less than $1$ in each timeout interval.

**Proof (Sketch):** In a legitimate state, the maximum length of a subscriber’s label is equal to $\log(n)$ and only action (ii) is executed by subscribers. Requests are only sent from a subscriber $v$ to the supervisor with probability based on $v$’s label length $|v.label|$. The expected number of configuration requests sent out by subscribers with label of length $k$ is equal to $\sum_{i=1}^{2^{k-1}} 1/(2^{k} \cdot k^{2}) = 1/(2k^{2})$. In summary, the expected number of configuration requests that are sent out by all subscribers is equal to $\sum_{k=1}^{\log(n)} 1/(2k^{2}) < 1$. ■

2) Maintaining Shortcuts: Now we describe how subscribers establish and maintain shortcut edges. Recall that we have shortcuts on levels $k = 1, \ldots, \lfloor \log n \rfloor$, where $k = \lfloor \log n \rfloor$ represents the ring edges that are already established. A subscriber $v$ with label of length $k = |v.label|$ has exactly 2 shortcuts on each level in $\{k, \ldots, \lfloor \log n \rfloor\}$ in a legitimate state.

We first describe how a subscriber is able to compute all its shortcut labels locally, based only on the information of its left and right direct ring neighbors. The following approach only computes the respective labels in $\{0, 1\}$ that a node should have shortcuts to, but not the subscribers that are associated with these labels. The idea is the following: In general, a subscriber $v \in V$ has only shortcuts to other subscribers that lie on the same semicircle as $v$, i.e., either the semicircle of subscribers within the interval $[0, 1/2]$ or the semicircle of subscribers within the interval $[1/2, 1]$ (where the 1 is represented by the subscriber with label 0). Consider a subscriber $v$ with $r(v.label) \in (0, 1)$ and its two ring neighbors $w, u$ such that $v.left = (\text{label}_u, w)$ and $v.right = (\text{label}_w, u)$. If $v$ recognizes that $|v.label| < |\text{label}_w|$, then $v$ knows that it has to have a shortcut with label $s$ and $r(s) = 2 \cdot r(\text{label}_w) - r(v.label)$, because node $w$ was previously inserted between subscribers with labels $s$ and $v.label$. After this, $v$ can apply this method recursively, i.e., it checks if the computed label $s$ if $|v.label| < |s|$ until it reaches a label of less or equal length. This same procedure is applied analogously for $v.right$.

As an example, recall the (stable) ring from Figure 1. Suppose we want to compute all shortcut labels for the subscriber with (real-valued) label $1/4$, based only on the labels of its direct ring neighbors, which are $3/16$ and $5/16$. We know that the label $3/16$ has length 4, which is greater than the length of label $1/4$, which is 2. Thus, we get a shortcut $s_1$ for $1/4$ with label $2 \cdot (3/16) - 1/4 = 1/8$. The label $1/8$ has length 3, which is still greater than 2. Hence we compute a shortcut $s_2$ with label $2 \cdot (1/8) - 1/4 = 0$. Finally we know that the length of label 0 is 1, which is smaller than 2, which terminates the algorithm. The computation of shortcut labels to $3/8$ and $1/2$ works analogously.

We are now ready to define the self-stabilizing protocol that establishes and maintains shortcuts for all subscribers. Consider a subscriber $v$ with label length $|v.label| = k$. On TIMEOUT, $v$ checks if $v.shortcuts$ contains subscribers $(\text{label}_u, u), (\text{label}_w, w)$ on level $k$. If that is the case, then $v$ introduces $u$ to $w$, by sending a message to $w$ containing the reference of $u$ as well as $u$’s label $\text{label}_u$. Also, $v$ introduces $w$ to $u$ in the same manner. Note that for $|v.label| = \lfloor \log n \rfloor$, $v$ has to consider its two ring neighbors instead of $v.shortcuts$. On receipt of such an introduction message consisting of the pair $(\text{label}_w, w)$, $u$ checks if it has
a shortcut \((\text{label}_{w'}, u')\) with \(\text{label}_{w'} = \text{label}_w\). If that is the case, then \(u\) replaces the existing node reference \(w'\) by \(w\) and, if \(w' \neq w\), forwards the reference of \(w'\) on the sorted ring via the BuildRING protocol. This way it is guaranteed that shortcuts are established in a bottom-up fashion.

C. Handling Subscriber Failures

We now consider the case that subscribers \(v \in V\) are allowed to crash without warning. In this case the address \(v.id\) ceases to exist. Consequently, even though nodes may still send messages to \(v\), these messages do not invoke any action on \(v\). Note that we do not consider subscriber failure, since it is assumed to be hard-coded. The challenge here is to restore the system to a correct supervised skip ring that does not contain \(v\), i.e., we need to exclude \(v\) from the system. In pure peer-to-peer systems this scenario is a problem, since we have to maintain failure detectors [20] at each node in order to be able to determine if some neighboring node has crashed. This leads to an increased overhead in the complete system. However, in our setting it suffices to establish only one single failure detector at the supervisor, because we only need to make sure that the database will eventually contain the correct data. Consequently, if the supervisor notices that subscriber \(v\) has crashed, it just has to remove \(v\) from its database. By periodically executing the actions for restoring a corrupted database we know that the database will eventually contain the correct data.

IV. SELF-STABILIZING PUBLISH-SUBSCRIBE SYSTEM

In this section we show how to use our BuildSR protocol as a self-stabilizing publish-subscribe system. We start by discussing some general modifications and then describe the operations subscribe, unsubscribe and publish.

Let \(T \subset \mathbb{N}\) be the set of available topics that one may subscribe to. To construct a publish-subscribe system out of our self-stabilizing supervised overlay network, we basically run a BuildSR protocol for each topic \(t \in T\) at the supervisor. Thus, the supervisor has to extend its database to be in \((\{0,1\}^*)^{\#T} \times V\). From here on, we assume that each message contains the topic it refers to, such that the receiver of such a message can match it to the respective BuildSR protocol. Once a subscriber wants to subscribe to some topic \(t \in T\), it starts running a new BuildSR protocol for topic \(t\). Upon unsubscribing, the subscriber may remove the respective BuildSR protocol, once it gets the permission from the supervisor, implying that the supervisor has removed the subscriber from its database. By assigning the topic number to each message that is sent out, we can identify the appropriate protocol at the receiver. For convenience, we still consider only one supervised skip ring for the rest of the paper.

A. Subscribe/Unsubscribe

When processing a subscribe \((v)\) operation, the supervisor executes the following actions (denote by \(n\) the number of nodes in the database before the subscribe/unsubscribe request):

1) Insert \((l(n), v)\) into the database.
2) Send \(v\) its correct configuration \((\text{pred}_v, l(n), \text{succ}_v)\).

The correctness of subscribe follows immediately, since our protocol is self-stabilizing. Note that the supervisor can easily extract the tuples \(\text{pred}_v\) and \(\text{succ}_v\) from the database, since all tuples are sorted based on the value of their labels. Our approach has the advantage that it spreads multiple sequential subscribe operations through the skip ring, meaning that a pre-existing subscriber is involved (i.e., it has to change its configuration) only for two consecutive subscribe operations. Afterwards its configuration remains untouched until the number of subscribers has doubled. This is due to the definition of the label function \(l\). As an example consider the skip ring \(SR(16)\) from Figure 1 and assume that there are 16 new subscribers that want to join. Then these new subscribers are inserted in between consecutive pairs of old subscribers on the ring, as they receive (real-valued) labels \(1/32, 3/32, 5/32, \ldots\).

When processing an unsubscribe \((v)\) operation, the supervisor executes the following actions:

1) Remove \((\text{label}_v, v)\) from the database.
2) Get the tuple \((\text{label}_w, w)\) with \(\text{label}_w = l(n-1)\) from the database and replace \(\text{label}_w\) with \(v\)'s label \(\text{label}_v\) in the database.
3) Send \(w\) its new configuration \((\text{pred}_v, \text{label}_v, \text{succ}_v)\).
4) Inform \(v\) that it is granted permission to delete all its connections to other subscribers.

After both subscribers have received their correct label from the supervisor, the ring will stabilize itself. Note that the supervisor's database is already in a legitimate state after the initial subscribe (resp. unsubscribe) message has been processed by the supervisor. Therefore, the supervisor does not rely on additional information from subscribers to stabilize its database. The following lemma states the correctness of unsubscribe.

Lemma 6. After a subscriber \(v\) has sent an unsubscribe \((v)\) request to the supervisor, \(v\) eventually gets disconnected from the graph induced by \(E_R \cup E_S\).

It follows from the above descriptions that the supervisor only has to send out a constant number of messages per subscribe/unsubscribe request:

Theorem 7. In a legitimate state, the message overhead of the supervisor and subscribers is constant for subscribe/unsubscribe operations.

B. Publish

In the following paragraphs we extend our protocol to be able to provide publish operations in a self-stabilizing manner. Note that the presented approach is used only to realize a self-stabilizing publication-dissemination-approach. There
exist dedicated protocols (e.g., flooding, see Section IV-C) that realize a more efficient distribution of publications among the subscribers. A self-stabilizing protocol for publications is able to correct eventual mistakes that occurred in the flooding approach. For storing publications at each subscriber, we use an extended version of a Patricia trie [21] to effectively determine missing publications at subscribers. We first define the Patricia trie and later on present a protocol that is able to merge all publications in all Patricia tries. This results in each subscriber storing all publications.

A trie is a search tree with node set $T$ over the alphabet $\Sigma = \{0, 1\}$. Every edge is associated with a label $c \in \Sigma$. Additionally, every key $x \in \Sigma^k$ that has been inserted into the trie can be reached from the root of the trie by following the unique path of length $k$ whose concatenated edge labels result in $x$.

A Patricia trie is a compressed trie in which all chains (i.e., maximal sequences of nodes of degree 1) are merged into a single edge whose label is equal to the concatenation of the labels of the merged trie edges. We store a Patricia trie at each subscriber $v \in V$, denoted by $v.T$. Each leaf node in a Patricia trie stores a publication $p \in \mathcal{P}^*$, where $\mathcal{P} = \{0, 1\}$ is the alphabet for publications. Note that each inner node $t \in T$ of a Patricia trie has exactly 2 child nodes denoted by $c_1(t), c_2(t) \in T$. Furthermore, we want to assign a label to each node: The label $t.label \in \Sigma^k$ of an inner node $t \in T$ is defined as the longest common prefix of the labels of $t$’s child nodes (with $\perp$ being the empty word). If $t$ is a leaf node storing a publication $p \in \mathcal{P}^*$, we define $t$’s label to be the unique key generated by the collision-resistant hash function $h_m : \mathbb{N} \times \mathcal{P}^* \rightarrow \{0, 1\}^m$, where a pair $(v.id,p) \in \mathbb{N} \times \mathcal{P}^*$ contains the unique ID of the subscriber $v \in V$ that generated the publication $p$. Note that the constant $m \in \mathbb{N}$ and the hash function $h_m$ are known to all subscribers, ensuring that every label for a publication has the same length.

In addition to node labels, we let nodes store (unique) hash values: We use another collision-resistant hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^*$ and define the hash value $t.hash$ of a leaf node $t$ as $h(t.label)$. If $t$ is an inner node, then $t.hash$ is defined as the hash of the concatenation of the hashes of $t$’s child nodes, i.e., $t.hash := h(h(c_1(t).label) \circ h(c_2(t).label))$. This approach is similar to a Merkle-Hash Tree (MHT) [22], which also hashes data using a collision-resistant hash function and building a tree on these hashes. However, our approach does not require one-way hash functions, which is a standard assumption in MHTs, because we do not require our scheme to be cryptographically secure.

If a subscriber $v \in V$ wants to publish a message $p \in \mathcal{P}^*$ over the ring, $v$ just inserts $p$ into its own Patricia trie. The publication $p$ is then spread among all subscribers of the ring by the following protocol, executed at each subscriber $v \in V$: Subscriber $v$ periodically sends a request CHECKTrie$(v, r_v)$ to one of its ring neighbors (chosen randomly) containing $v$ itself and the root node $r_v \in v.T$ of $v$’s Patricia trie. Note that sending an arbitrary node $t \in v.T$ along a message CHECKTrie means that we only store $t.label$ and $t.hash$ in the request while ignoring $t$’s outgoing edges. Upon receiving a request CHECKTrie$(v, t_u)$ with $t_u \in v.T$, a subscriber $u \in V$ does the following: It searches for the node $t_u \in u.T$ with label $t_u.label = t_v.label$ and checks if $t_u.hash = t_v.hash$. The following three cases may happen:

(i) $t_u.hash = t_v.hash$: Then we know that the set of publications stored in the subtrie of $u.T$ with root node $t_u$ are the same as the set of publications stored in the subtrie of $v.T$ with root node $t_v$. Subscriber $u$ does not send any response to $v$ in this case.
(ii) $t_u.hash \neq t_v.hash$: Then the contents of the subtries with roots $t_u, t_v$ differ in at least one publication. In order to detect the exact location, where both Patricia tries differ, $u$ responds to $v$ by sending a request CHECKTrie$(u, c_1(t_u), c_2(t_u))$ to $v$, which is handled by $v$ as two separate CHECKTrie requests CHECKTrie$(u, c_1(t_u))$ and CHECKTrie$(u, c_2(t_u))$.

(iii) $t_u$ does not exist in $u.T$: Then $v.T$ contains publications that do not exist in $u.T$. Subscriber $u$ is able to compute the label prefix of those missing publications: First, $u$ searches for the node $c \in u.T$ with label prefix $t_u.label$ and $|c.label|$ minimal, i.e., $c.label = t_u.label \circ b_1 \circ \ldots \circ b_k$ with $b_1, \ldots, b_k \in \{0, 1\}$ and $|c.label| = |t_v.label| + k$ minimal. If such a node $c$ exists, then $u.T$ may contain at least all publications with label prefix $c.label$. Furthermore, $u$ knows that all publications with label prefix $t_u.label \circ (1 - b_1)$ are missing in its Patricia trie. As a consequence, $u$ requests $v$ to continue checking the subtrie with root node of label $c.label$ and to deliver all publications with label prefix $p = t_u.label \circ (1 - b_1)$ to $u$. It does so by sending a CHECKANDPUBLISH$(u, c, p)$ request to $v$, where $v$ internally calls CHECKTrie$(u, c)$ and, in addition, delivers all publications with label prefix $p$ to $u$. In case that a node $c$ as described above cannot be found in $u.T$, $u$ just requests $v$ to deliver all publications with prefix $t_u.label$ to $u$, since that subtrie is missing in $u.T$.

With this approach, only those publications are sent out that are assumed to be missing at the receiver.

As an example consider two subscribers $u,v \in V$ with Patricia tries as shown in Figure 2. Note that $P_3$ is missing in $v.T$. We describe how $v$ will eventually receive $P_3$.

First assume that $u$ sends out a CHECKTrie$(u, r_u)$ message to $v$ in its TIMEOUT method with $r_u$ being the root node of $u$’s Patricia trie. Subscriber $v$ then compares the hash $r_v.hash$ with the hash of its root node, which is not equal. Thus, $v$ sends a message CHECKTrie$(v, (0, h(h(P_1) \circ h(P_2))), (100, h(P_3)))$ to $u$, which forces $u$ to compare the hashes of the nodes with labels 0 resp. 100 to
the hashes $h(h(P_1) \circ h(P_2))$ resp. $h(P_3)$. Both comparisons result in the hashes being equal, which ends the chain of messages at subscriber $u$.

Now assume, that $v$ sends out a $\text{CHECKTRIE}(v, r_v)$ message to $u$ in its $\text{TIMEOUT}$ method with $r_v$ being the root node of $v$’s Patricia trie. Subscriber $u$ compares the $r_v,\text{hash}$ with $v,\text{hash}$ and spots a difference. Thus, it sends a message $\text{CHECKTRIE}(u, (0, h(h(P_1) \circ h(P_2))), (10, h(h(P_3) \circ h(P_4))))$ to $v$. For the node with label 0 this results in both hashes being equal, but then $v$ cannot find a node with label 10 in its Patricia trie, which is why $v$ sends a message $\text{CHECKANDPUBLISH}(v, (100, h(P_3)), p = 101)$ to $u$. Note that the node with label 100 is the node with label of minimum length for which 10 is a prefix. Thus, $p = (10 \circ (1 - 0)) = 101$. The $\text{CHECKANDPUBLISH}$ request forces $u$ to compare the hashes of its node with label 100 to the given hash $h(P_3)$, which results in both hashes being equal. Furthermore, $u$ sends all publications with labels of prefix 101 to $v$, which is only the publication $P_4$. After $v$ has inserted $P_4$ in its Patricia trie, both tries are equal, resulting in two equal root hashes.

The example shows that it is important at which subscriber the initial $\text{CHECKTRIE}$ request is started.

C. Flooding

As an extension to the above approach, we can make use of shortcuts to spread new publications over the ring: Whenever a subscriber $u \in V$ generates a new publication $p$, $u$ inserts $p$ into $u.T$ and broadcasts $p$ over the ring, by sending a $\text{PUBLISHNEW}(p)$ message to all of its neighbors $v$ with $(u, v) \in E_R \cup E_S$. Upon receiving such a $\text{PUBLISHNEW}(p)$ message, a subscriber $v \in V$ checks if $p$ is already stored in $v.T$. If not, then $v$ inserts $p$ into $v.T$ and continues to broadcast $p$ by forwarding the $\text{PUBLISHNEW}$ message to its neighbors. In case that $p$ is already stored in $v.T$, $v$ just drops the message. By applying this flooding approach on top of the self-stabilizing publish protocol, we can achieve faster delivery of new publications in practice (recall that the skip ring has a diameter of $\log n$). Still, if a new subscriber joins a topic, it has to rely on the core $\text{BUILDSR}$ protocol to receive all publications. Furthermore, note that we do not rely on flooding to show convergence of publications.

V. Analysis

In this section we show that $\text{BUILDSR}$ is self-stabilizing according to Definition 1. We also show that eventually all subscribers are storing all publications in their respective Patricia tries. The combination of the first two theorems yields that $\text{BUILDSR}$ is self-stabilizing:

**Theorem 8** (Network Convergence). Given any initially weakly connected graph $G = (V, E_R \cup E_S)$ with $n$ nodes, $\text{BUILDSR}$ transforms $G$ into a skip ring $SR(n)$.

**Proof (Sketch):** We can argue that the supervisor’s database eventually contains non-corrupted information. Afterwards we show the convergence for the supervisor, i.e., eventually all subscribers $v \in V$ will be stored in the supervisor’s database. The convergence of the subscriber network follows in two steps: First we argue with the help of Lemma 4 that the sorted ring of subscribers eventually converges. Finally we prove via induction that all shortcuts are eventually established for each subscriber.

**Theorem 9** (Network Closure). If the explicit edges in $G = (V, E_R \cup E_S)$ already form a supervised skip ring $SR(n)$, then they are preserved at any point in time if no subscribers join or leave the system.

Furthermore, we can show that the delivery of publications is done in a self-stabilizing manner.

**Theorem 10** (Publication Convergence). Consider an initially weakly connected graph $G = (V, E_R \cup E_S)$ and assume that there are publications $P \subseteq P^*$ in the system, stored at arbitrary subscribers $v \in V$. Then eventually all subscribers store a Patricia trie consisting of all publications in $P$.

**Proof:** (Sketch) For a subscriber $u \in V$ let $P_u \subseteq P$ be the set of publication stored in the leaf nodes of $u.T$.  

---

![Diagram](image_url)
We define the potential of a pair \((u, v)\) of subscribers by \(\phi(u, v) = |(P_u \cup P_v) \setminus P_u|\) and the potential of all subscribers as \(\Phi = \sum_{(u, v) \in E_R} \phi(u, v)\). One can show that \(\Phi\) is monotonically decreasing over time and, thus, \(\Phi = 0\) eventually. This shows the theorem, since it is easy to see that \(\Phi = 0 \iff P_u = P\) for all \(u \in V\).

Finally we also show convergence for publications.

**Theorem 11** (Publication Closure). Consider a stable supervised skip ring \(SR(n)\) and assume that all subscribers store the exact same Patricia trie containing publications \(P \subseteq \mathcal{P}\). Then no Patricia trie is modified by a subscriber as long as no subscriber issues a publish request and no further subscriber joins the system.

**VI. CONCLUSION**

In this paper we proposed a self-stabilizing protocol for the supervised skip ring, which can be extended to a self-stabilizing publish-subscribe system. The system is able to effectively handle node insertions/deletions and is furthermore able to efficiently recover from node failures.

As parts of our protocol are randomized (subscriber periodically call the supervisor with a certain probability), one may investigate if there are deterministic self-stabilizing protocols for supervised overlay networks. These can probably be established by using a token-passing scheme. Depending on the rate of join/leave requests, the supervisor may adjust the speed of the token. Then the space overhead for the supervisor could be reduced as it only needs to know the number of subscribers \(n\). However, it may be harder to prove the self-stabilization property, as the token-passing scheme has to be able to deal with multiple connected components, so we leave this to future works.

**REFERENCES**

[1] K. Kothapalli and C. Scheideler, “Supervised peer-to-peer systems,” in ISPA 2005. IEEE Computer Society, 2005, pp. 188–193.

[2] S. Kniesburges, A. Koutsofopoulos, and C. Scheideler, “Re-chord: A self-stabilizing chord overlay network,” Theory Comput. Syst., vol. 55, no. 3, pp. 591–612, 2014.

[3] ——, “A self-stabilization process for small-world networks,” in IPDPS 2012. IEEE Computer Society, 2012, pp. 1261–1271.

[4] P. T. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, “The many faces of publish/subscribe,” ACM Comput. Surv., vol. 35, no. 2, pp. 114–131, 2003.

[5] F. Fabret, H. Jacobsen, F. Llirbat, J. L. M. Pereira, K. A. Ross, and D. E. Shasha, “Filtering algorithms and implementation for very fast publish/subscribe,” in SIGMOD 2001, 2001, pp. 115–126.

[6] A. Fekete, N. A. Lynch, and A. A. Shvartsman, “Specifying and using a partitionable group communication service,” ACM Trans. Comput. Syst., vol. 19, no. 2, pp. 171–216, 2001.

[7] E. W. Dijkstra, “Self-stabilizing systems in spite of distributed control,” Commun. ACM, vol. 17, no. 11, pp. 643–644, 1974.

[8] M. Onus, A. W. Richa, and C. Scheideler, “Linearization: Locally self-stabilizing sorting in graphs,” in ALENEX 2007. SIAM, 2007.

[9] A. W. Richa, C. Scheideler, and P. Stevens, “Self-stabilizing de bruijn networks,” in SSS 2011, ser. Lecture Notes in Computer Science, vol. 6976. Springer, 2011, pp. 416–430.

[10] A. Berns, S. Ghosh, and S. V. Pemmaraju, “Building self-stabilizing overlay networks with the transitive closure framework,” Theor. Comput. Sci., vol. 512, pp. 2–14, 2013.

[11] G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, A. Ulbrich, and L. Fiege, “Self-stabilizing publish/subscribe systems: Algorithms and evaluation,” in Euro-Par 2005, ser. Lecture Notes in Computer Science, vol. 3648. Springer, 2005, pp. 664–674.

[12] G. Siegemund and V. Turau, “PSVR - self-stabilizing publish/subscribe communication for ad-hoc networks (short paper),” in SSS 2016, ser. Lecture Notes in Computer Science, vol. 10083, 2016, pp. 346–351.

[13] G. Siegemund, V. Turau, and K. Maamra, “A self-stabilizing publish/subscribe middleware for wireless sensor networks,” in NetSys 2015. IEEE Computer Society, 2015, pp. 1–8.

[14] Y. Amir, C. Nita-Rotaru, J. R. Stanton, and G. Tsudik, “Secure spread: An integrated architecture for secure group communication,” IEEE Trans. Dependable Sec. Comput., vol. 2, no. 3, pp. 248–261, 2005.

[15] S. Dolev, E. Schiller, and J. L. Welch, “Random walk for self-stabilizing group communication in ad hoc networks,” IEEE Trans. Mob. Comput., vol. 5, no. 7, pp. 893–905, 2006.

[16] S. Dolev and E. Schiller, “Self-stabilizing group communication in directed networks,” Acta Inf., vol. 40, no. 9, pp. 609–636, 2004.

[17] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig, “A distributed polylogarithmic time algorithm for self-stabilizing skip graphs,” in PODC 2009. ACM, 2009, pp. 131–140.

[18] ——, “Skip\(^*:\) A self-stabilizing skip graph,” J. ACM, vol. 61, no. 6, pp. 36:1–36:26, 2014.

[19] M. Feldmann, C. Kolb, C. Scheideler, and T. Strothmann, “Self-stabilizing supervised publish-subscribe systems,” CoRR, vol. abs/1710.08128, 2017. [Online]. Available: http://arxiv.org/abs/1710.08128

[20] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” J. ACM, vol. 43, no. 2, pp. 225–267, 1996.

[21] D. R. Morrison, “PATRICIA – practical algorithm to retrieve information coded in alphanumeric,” J. ACM, vol. 15, no. 4, pp. 514–534, 1968.

[22] R. C. Merkle, “A digital signature based on a conventional encryption function,” in CRYPTO 1987, ser. Lecture Notes in Computer Science, vol. 293. Springer, 1987, pp. 369–378.