

# Self-Stabilizing Supervised Publish-Subscribe Systems\*

Michael Feldmann<sup>1</sup>, Christina Kolb<sup>2</sup>, Christian Scheideler<sup>3</sup>, and Thim Strothmann<sup>4</sup>

- 1 Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany  
mfeldma2@mail.upb.de
- 2 Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany  
ckolb@mail.upb.de
- 3 Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany  
scheideler@upb.de
- 4 Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany  
thim@mail.upb.de

---

## 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 [14]) 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.

**1998 ACM Subject Classification** C.2.4 Distributed Systems

**Keywords and phrases** Topological Self-stabilization, Publish-Subscribe, Supervised Overlay

## 1 Introduction

The publish subscribe paradigm ([7, 8]) 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

---

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



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 [9], 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.

## 1.1 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.

Nodes may execute *actions*: An action is just a standard procedure and has the form  $\langle label \rangle(\langle parameters \rangle) : \langle command \rangle$ , where *label* is the name of that action, *parameters* defines the set of parameters and *command* defines the statements that are executed when calling that action. It may be called locally or remotely, i.e., every message that is sent to a node has the form  $\langle label \rangle(\langle parameters \rangle)$ . 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 *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 *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. Consider the *TIMEOUT* action as an example for this. We place no bounds on message propagation delay or relative node execution speed, i.e., we

allow fully asynchronous computations and non-FIFO message delivery. Our protocol does not manipulate node identifiers and thus only operates on them in *compare-store-send* mode, i.e., 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.

## 1.2 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 [4]. Many self-stabilizing protocols for various types of overlays have been proposed, like sorted lists [18], de Bruijn graphs [19], Chord graphs [13] and many more. There is even a universal approach, which is able to derive self-stabilizing protocols for several types of topologies [2].

The cycle topology is particularly important for our work. Our cycle protocol is based on [12], 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 [14]. 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 [17] 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 [20], which builds upon the work of [21]: 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  $\mathcal{O}(n)$  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  $\mathcal{O}(\log n)$  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., [9, 1]) 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 [6] for ad-hoc networks and in [5] 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.

### 1.3 Our Contribution

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 BUILDSR is split up into two subprotocols: One protocol is executed at the supervisor (see Section 3.1), the other one is executed by each subscriber (Section 3.2). Our basic protocol assumes that all references actually belong to existing nodes. However, we also present an extension (see Section 3.3) to handle references to non-existing nodes and unannounced failures of nodes. In contrast to the supervised overlay network proposed in [14], 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 [14].

The skip ring shares some similarities with other shortcut-based peer-to-peer systems like Chord networks [13] or skip graphs [10]. 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 4) 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. [11]), we do not

elaborate on them further in this paper.

## 2 Preliminaries

In this section we formally introduce the topology for a skip ring (Section 2.1). As a base for our self-stabilizing protocol we introduce the BUILDRING protocol from [12] to arrange nodes in a sorted ring (Section 2.2).

### 2.1 Skip Ring

Let  $l : \mathbb{N}_0 \rightarrow \{0, 1\}^*$  be a mapping with the property that  $l(x) := (x_{d-1} \dots x_0 x_d)$  for every  $x \in \mathbb{N}_0$  with binary representation  $(x_d \dots x_0)_2$  (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  $|label|$  the minimum number of bits used to encode *label*. A label  $y = (y_1 \dots y_d) \in \{0, 1\}^d$  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  $label_v \in \{0, 1\}^*$  with  $l^{-1}(label_v) < n$ .
- $(u, v) \in E_R \Leftrightarrow (u, v)$  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  $K_i$ ,  $i \in \{1, \dots, \lceil \log n \rceil - 1\}$ , where  $K_i := \{w \in V \mid |label_w| \leq i\}$ . Denote  $(u, v) \in E_S$  as a *shortcut on level  $i$* , if  $i = \max\{|label_u|, |label_v|\}$ .

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

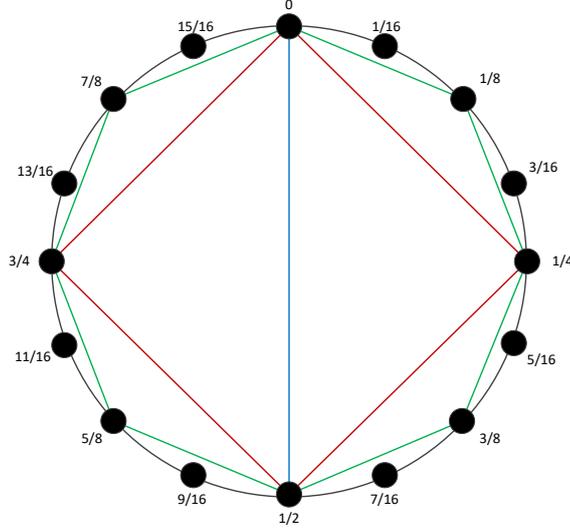
The intuition behind  $E_R$  and  $E_S$  is that we want all nodes with label of length at most  $k$  to form a (bidirected) sorted ring for all  $k \in \{1, \dots, \lceil \log n \rceil\}$ . For  $k = \lceil \log n \rceil$  these edges are stored in  $E_R$ , for  $k < \lceil \log n \rceil$  they are stored in  $E_S$ . Due to the way we defined the function  $l$  it holds that for all  $x \in \{2^d, \dots, 2^{d+1} - 1\}$  the values  $r(l(x))$  are uniformly spread in between old values  $r(l(y))$  with  $y \in \{0, \dots, 2^d - 1\}$ . 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 a arrival/departure of a node only affects its neighbors (see Section 4.1 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.*

**Proof.** For convenience, we define  $k := |label_v|$ . Node  $v$  has 2 shortcuts to nodes with label of length  $k'$  for each  $k' \geq k$ . Having  $n$  nodes in the system, we know that  $k'$  is upper bounded by  $\log(n)$ , which sums up the degree of  $v$  to be  $2 \cdot (\log n - k + 1) = \mathcal{O}(\log n)$ .



■ **Figure 1** A skip ring consisting of 16 nodes. The triples are of the form  $(x, l(x), r(l(x)))$  where  $x \in \{0, \dots, 15\}$ ,  $l(x)$  is the corresponding label and  $r(l(x))$  is the real valued version of the label. Black edges are ring edges ( $k = 4$ ), green edges are shortcuts for  $k = 3$ , red edges for level  $k = 2$  and the blue edge is the shortcut for  $k = 1$ .

Next, we want to compute the average degree of a node. We count the overall number of edges in a stable system containing  $n$  subscribers. Let  $f(k)$  denote the number of subscribers with label of length  $k$ . We have

$$f(k) = \begin{cases} 2 & k = 1 \\ 2^{k-1} & k > 1 \end{cases}$$

Recall that the maximum length of a label is equal to  $\log(n)$  in a stable state. Combining this fact with the above formula for the node degree, we get the following result for the number of edges in  $|E_R \cup E_S|$ :

$$\begin{aligned} |E_R \cup E_S| &= \sum_{k=1}^{\log(n)} f(k)(2(\log(n) - k + 1)) \\ &= 4\log(n) + \sum_{k=2}^{\log(n)} 2^{k-1}(2(\log(n) - k + 1)) \\ &= 2\log(n) + \sum_{k=1}^{\log(n)} 2^k(\log(n) - k + 1) \\ &= 2n\log(n) + 2n - 2 - \sum_{k=1}^{\log(n)} 2^k k \\ &= 4n - 4 \end{aligned}$$

Dividing this value by  $n$  yields an upper bound of  $4 = \Theta(1)$  for the average node

degree. ◀

## 2.2 Self-Stabilizing Ring

The base of our self-stabilizing protocol is the BUILDING protocol from [12] that organizes all nodes in a sorted ring according to their labels, using linearization [18]: 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 BUILDING 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 BUILDING protocol is still self-stabilizing:

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

**Proof.** In case that there are no corrupted labels, the extended BUILDING protocol behaves the same as the standard BUILDING protocol, so we refer the reader to [12] to verify that the protocol is indeed self-stabilizing.

We are going to show that in case there exist corrupted node labels, these will eventually vanish. W.l.o.g. consider the variable  $u.right$  for a node  $u$ . We define the *trace* of  $u.right$  as the chain of values  $((label_{v_1}, v_1), (label_{v_2}, v_2), \dots)$  that are allocated to  $u.right$  while the system stabilizes. By definition of the standard BUILDING protocol, labels stored in the trace for  $u.right$  are monotonically decreasing. Furthermore, the trace is finite, since the number of labels (corrupted or correct) is finite. Let  $(label_{v_k}, v_k)$  be the last node of this trace, i.e., eventually it holds  $u.right = (label_{v_k}, v_k)$ . Then  $u$  will introduce itself in its TIMEOUT method to  $v_k$  by sending a message  $M$  storing  $u$  itself and  $label_{v_k}$  to  $v_k$ . Upon receiving  $M$ ,  $v_k$  is able to check if  $label_{v_k} = v_k.label$  and send a reply storing the (correct) label  $v_k.label$  to  $u$  in case  $label_{v_k} \neq v_k.label$ , s.t.  $u$  corrects its label  $label_{v_k}$ . This implies that the number of corrupted labels is reduced, but there may be a new trace generated for  $u.right$ . But since the number of corrupted node labels is finite and is not duplicating, eventually, the overall number of corrupted node labels will reduce to 0. ◀

## 3 Self-stabilizing Supervised Skip Ring

In this section we first extend the skip ring topology by introducing a supervisor. The description of our BUILD SR 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 4.

### 3.1 Supervisor Protocol

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

- A *database*  $\subset \{0, 1\}^* \times V$  containing subscribers and their corresponding labels. Denote  $n := |\text{database}|$ .
- A variable  $\text{next} \in \mathbb{N}$  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  $\text{next}$ ) from its *database*. Then the supervisor sends a message to  $v$  containing  $v$ 's label  $\text{label}_v$ , as well as the correct values for  $v$ 's predecessor  $\text{pred}_v$  and successor  $\text{succ}_v$  according to the *database*. We call such a triple  $(\text{pred}_v, \text{label}_v, \text{succ}_v)$  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  $i \in \{0, \dots, n-1\}$ , s.t. for all  $(\text{label}_v, v) \in \text{database}$  it holds  $\text{label}_v \neq l(i)$ . (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  $i \in \{0, \dots, n-1\}$  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.

## 3.2 Subscriber Protocol

In this section we discuss the part of the BUILD SR 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:

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

For the rest of the protocol description, we use  $v.left$  and  $v.right$  to indicate  $v$ 's left (resp. right) neighbor in the ring even if the left (resp. right) neighbor is stored in  $v.ring$  instead of  $v.left$  (resp.  $v.right$ ). We also may refer to the variables  $v.left, v.right, v.ring$  as  $v$ 's *direct ring neighbors*. Recall that each subscriber executes the extended BUILD RING protocol from Section 2.2.

### 3.2.1 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  $v.label$ .

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

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

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.** Since the system has  $n$  subscribers, the maximum length of a subscriber's label is equal to  $\log(n)$  in a legitimate state. In a legitimate state, only the second action (ii) is executed by subscribers, as all subscribers have stored their correct configuration. Thus, requests are only sent from a subscriber  $v$  to the supervisor with probability based on  $v$ 's label length  $|v.label|$ . The number of subscribers with label of length  $k$  is equal to  $2^{k-1}$  and the probability that a subscriber with label of length  $k$  contacts the supervisor in its TIMEOUT procedure is equal to  $1/(2^k \cdot k^2)$ . It follows that 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  $\sum_{k=1}^{\log(n)} 1/(2k^2) < 1$ . ◀

### 3.2.2 Maintaining Shortcuts

In this section we describe how subscribers establish and maintain shortcut edges. Recall that we have shortcuts on levels  $k = \{1, \dots, \lceil \log n \rceil\}$ , where  $k = \lceil \log n \rceil$  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  $\{k, \dots, \lceil \log n \rceil\}$  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  $v.left = (label_w, w)$  and  $v.right = (label_u, u)$ . If  $v$  recognizes that  $|v.label| < |label_w|$ , then  $v$  knows that it has to have a shortcut with label  $s$  and  $r(s) = 2 \cdot r(label_w) - r(v.label)$ , 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 for 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  $(label_u, u), (label_w, w)$  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  $label_u$ . Also,  $v$  introduces  $w$  to  $u$  in the same manner. Note that for  $|v.label| = \lceil \log n \rceil$ ,  $v$  has to consider its two ring neighbors instead of  $v.shortcuts$ . On receipt of such an introduction message consisting of the pair  $(label_w, w)$ ,  $u$  checks if it has a shortcut  $(label_{w'}, w')$  with  $label_{w'} = label_w$ . 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 BUILDING protocol. This way it is guaranteed that shortcuts are established in a bottom-up fashion.

### 3.3 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 supervisor 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 [3] 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.

## 4 Self-Stabilizing Publish-Subscribe System

In this section we show how to use our BUILD SR 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  $\mathcal{T} \subset \mathbb{N}$  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 BUILD SR protocol for each topic  $t \in \mathcal{T}$  at the supervisor. Thus, the supervisor has to extend its *database* to be in  $(\{0, 1\}^*)^{|\mathcal{T}|} \times V$ . 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 BUILD SR protocol. Once a subscriber wants to subscribe to some topic  $t \in \mathcal{T}$ , it starts running a new BUILD SR protocol for topic  $t$ . Upon unsubscribing, the subscriber may remove the respective BUILD SR 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.

### 4.1 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  $(pred_v, l(n), succ_v)$ .

The correctness of subscribe follows immediately, since our protocol is self-stabilizing. Note that the supervisor can easily extract the tuples  $pred_v$  and  $succ_v$  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  $1/32, 3/32, 5/32, \dots$

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

1. Remove  $(label_v, v)$  from the *database*.
2. Get the tuple  $(label_w, w)$  with  $label_w = l(n - 1)$  from the *database* and replace  $label_w$  with  $v$ 's label  $label_v$  in the *database*.
3. Send  $w$  its new configuration  $(pred_v, label_v, succ_v)$ .

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  $E_R \cup E_S$ .*

**Proof.** Assume that subscriber  $v \in V$  sends an  $unsubscribe(v)$  request to the supervisor. By definition of the unsubscribe protocol, the supervisor removes  $v$  from its *database* and sends  $v$  its configuration that is  $(\perp, \perp, \perp)$ . Hence,  $v$  sets  $v.label = \perp$  and answers all incoming introduction messages from other subscribers with the request to delete the connection to  $v$ . By definition of BUILD SR, every subscriber  $u$  that has an edge  $(u, v) \in E_R \cup E_S$  will eventually introduce itself to  $v$ , leading to  $v$  eventually getting disconnected from the graph induced by  $E_R \cup E_S$ . This proves the lemma. ◀

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.*

## 4.2 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 4.3) 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 [16] 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  $T$  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  $\bar{h}_m : \mathbb{N} \times \mathcal{P}^* \rightarrow \{0, 1\}^m$ , 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  $\bar{h}_m$  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) [15], 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.

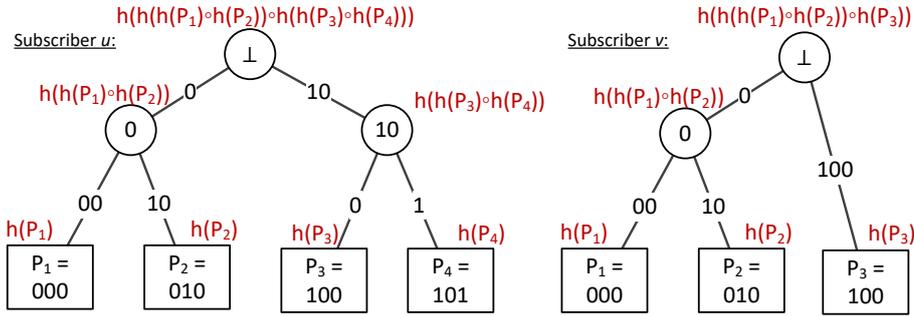
If a subscriber  $v \in V$  wants to publish a message  $p \in \mathcal{P}^*$  over the ring,  $v$  just inserts  $p$  into its own Patricia trie. The publication  $p$  is then spread among all subscribers of the ring by the following protocol, executed at each subscriber  $v \in V$ : Subscriber  $v$  periodically sends a request  $\text{CHECKTRIE}(v, r_v)$  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  $\text{CHECKTRIE}$  means that we only store  $t.label$  and  $t.hash$  in the request while ignoring  $t$ 's outgoing edges. Upon receiving a request  $\text{CHECKTRIE}(v, t_v)$  with  $t_v \in v.T$ , 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 subtree of  $u.T$  with root node  $t_u$  are the same as the set of publications stored in the subtree 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 subtrees 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  $\text{CHECKTRIE}(u, c_1(t_u), c_2(t_u))$  to  $v$ , which is handled by  $v$  as two separate  $\text{CHECKTRIE}$  requests  $\rightarrow \text{CHECKTRIE}(u, c_1(t_u))$  and  $\text{CHECKTRIE}(u, c_2(t_u))$ .
- (iii)  $t_v$  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  $t_v.label$  and  $|c.label|$  minimal, i.e.,  $c.label = t_v.label \circ b_1 \circ \dots \circ b_k$  with  $b_1, \dots, b_k \in \{0, 1\}$  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  $t_v.label \circ (1 - b_1)$  are missing in its Patricia trie. As a consequence,  $u$  requests  $v$  to continue checking the subtree with root node of label  $c.label$  and to deliver all publications with label prefix  $p = t_v.label \circ (1 - b_1)$  to  $u$ . It does so by sending a  $\text{CHECKANDPUBLISH}(u, c, p)$  request to  $v$ , where  $v$  internally calls  $\text{CHECKTRIE}(u, c)$  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  $t_v.label$  to  $u$ , since that subtree 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  $P_4$  is missing in  $v.T$ . We describe how  $v$  will eventually receive  $P_4$ .

First assume that  $u$  sends out a  $\text{CHECKTRIE}(u, r_u)$  message to  $v$  in its  $\text{TIMEOUT}$  method with  $r_u$  being the root node of  $u$ 's Patricia trie. Subscriber  $v$  then compares the hash  $r_u.hash$



■ **Figure 2** Example Patricia tries  $u.T$  and  $v.T$  for two subscribers  $u, v \in V$ .

with the hash of its root node, which is not equal. Thus,  $v$  sends a message  $\text{CHECKTRIE}(v, (0, h(h(P_1) \circ h(P_2))), (100, h(P_3)))$  to  $u$ , which forces  $u$  to compare the hashes 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  $r_v.hash$  with  $r_u.hash$  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.

### 4.3 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.

## 5 Analysis

In this section we show that 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 BUILDSR is self-stabilizing:

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

**Proof.** First of all, note that eventually all corrupted messages are received. Furthermore, a corrupted message cannot trigger an infinite chain of corrupted messages, i.e., eventually the false information is either corrected or received but not spread anymore. We assume this fact for the rest of the proof.

We start with the supervisor and prove the following lemma:

► **Lemma 9** (Supervisor Validity). *Eventually the supervisor's database will not be corrupted anymore.*

**Proof.** We show for every condition that may occur in a corrupted *database* that the condition does not occur anymore at some point in time.

- (i) Assume that there exists a tuple  $(label, v) \in database$  with  $v = \perp$ . Then this tuple is simply deleted from the database (see Algorithm 3, line 41).
- (ii) Assume that there are entries  $(label_1, v_1), (label_2, v_2) \in database$ ,  $label_1 \neq label_2$  and  $v_1 = v_2$ . At some point in time the supervisor wants to send the configuration to  $v_1$  (Algorithm 3, line 5), which leads to the supervisor calling CHECKMULTIPLECOPIES( $v_1$ ). Therefore, the supervisor is able to detect the entries  $(label_1, v_1), (label_2, v_2)$ . W.l.o.g. assume that  $(label_1, v_1)$  is detected before  $(label_2, v_2)$  in the for-loop of CHECKMULTIPLECOPIES. Then the supervisor just removes the redundant tuple  $(label_2, v_2)$  from the *database*, which resolves this condition.
- (iii) Assume there exists  $i \in \{0, \dots, n-1\}$ , such that for all  $(label_v, v) \in database$  it holds  $label_v \neq l(i)$ . In this case, the supervisor is able to detect this corruptness: It asks for the entry with label  $l(i)$  in the *database* (Algorithm 3, line 43). Then the supervisor proceeds as follows: It replaces the label of the entry  $(l(j), w)$  with maximum  $j > i$  with the label  $l(i)$  (see Algorithm 3, line 45). Note that  $(l(j), w)$  has to exist in this case, since otherwise  $i \geq |database| = n$ , which is a contradiction.
- (iv) Finally, assume that there exists  $i \geq n$ , s.t. there is a tuple  $(label, v) \in database$  with  $label = l(i)$ . Then there exists a number  $j \in \{0, \dots, n-1\}$ , for which there is no tuple  $(l(j), w) \in database$ , so eventually  $(label, v)$  gets its label changed to  $(l(j), v)$ .

In the end we showed that all 4 conditions that may occur in a corrupted *database* do not occur in the *database* from some point in time on, which concludes the proof. ◀

Using this lemma, we are ready to prove the convergence of the supervisor:

► **Lemma 10** (Supervisor Convergence). *After the supervisor's database has reached a non-corrupted state, all subscribers  $v \in V$  will eventually become recorded.*

**Proof.** Recall that we call a subscriber *recorded*, if it is stored in the supervisors's *database*. Note that the supervisor does not remove subscribers from its (non-corrupted) *database* unless it gets the request to do so, which is not the case as we do not consider UNSUBSCRIBE operations here.

Let  $v \in V$  be a non-recorded subscriber. Then either  $v.label = \perp$  or  $v.label \neq \perp$ . Recall the actions described in Section 3.2.1. If  $v.label = \perp$ , then  $v$  requests its configuration from the supervisor via action (i) and becomes recorded.

Now let  $v.label \neq \perp$ . If  $v$  does not have connections to other subscribers (i.e.,  $v.left = v.right = \perp$ ), then  $v$  requests its configuration from the supervisor via action (iv), as  $v$  thinks that it is the subscriber with minimal label, so  $v$  becomes recorded.

It remains to consider the general case, where we are given a connected component  $C$  of subscribers. Action (iv) guarantees that  $C$  quickly contains at least one subscriber that is recorded. 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. We show that for such a static connected component  $C$ , eventually all subscribers will become recorded: Consider the potential function

$$\Phi(C) = |\{v \in C \mid v \text{ is non-recorded}\}|.$$

We show that eventually,  $\Phi(C) = 0$ . Following the same argumentation from above, it is easy to see that  $\Phi$  is never increasing. Let  $\Phi(C) = c$  for an arbitrary constant  $c > 0$ . Then all subscribers in  $C$  eventually form a sorted ring due to BUILDRING. This implies that there exists a ring edge  $(v, w)$  from a subscriber  $v$  that is already contained in the supervisor's *database* to a subscriber  $w$  that is not yet contained in the supervisor's *database*. W.l.o.g. let  $v.right = w$ . Since  $w$  is not known to the supervisor, the *database* has to contain a different right neighbor for  $v$ . Let  $w' \in C, w' \neq w$  be this neighbor, i.e., as the supervisor sends  $v$  its correct configuration, it tells  $v$  that  $w'$  should be its right ring neighbor. But then it has to hold  $r(v) < r(w) < r(w')$ . Note that  $r(v) < r(w') < r(w)$  cannot hold, since this contradicts the fact that subscribers in  $C$  have already arranged themselves in a sorted ring and no new subscriber can be introduced to a recorded subscriber in  $C$ , as  $C$  is already static. This implies that upon receiving its configuration,  $v$  triggers action (iii) and requests the configuration for  $w$  at the supervisor, reducing  $\Phi$  by one and finishing the proof. ◀

Having the supervisor's *database* converged, we know that the ring of subscribers eventually converges:

► **Lemma 11** (Ring Convergence). *After each subscriber  $v \in V$  has stored its correct configuration from the supervisor the ring induced by edges  $E_R$  has converged.*

**Proof.** The supervisor periodically sends the correct configuration to each subscriber in a round-robin fashion (Algorithm 3, line 5). This implies that after  $n$  calls of the supervisor's TIMEOUT procedure, each subscriber has stored its correct label. Note that this does not necessarily include the correct ring neighbors: A subscriber  $v \in V$  may have received its configuration  $(pred_v, label_v, succ_v)$  from the supervisor, but the subscriber  $u$  stored in  $pred_v$  (resp.  $succ_v$ ) may not yet. This may result in  $v$  modifying  $v.left = pred_v$  via BUILDRING, because  $u$  may not have received its correct label. Since at least all labels are correct now, each subscriber receives its configuration from the supervisor the second time and does not change its list neighbors anymore. ◀

Finally, we need to prove the convergence of the shortcuts for all subscribers:

► **Lemma 12** (Shortcut Convergence). *Assume that each subscriber  $v \in V$  has its correct configuration stored. Then all correct shortcut links will eventually be established at some point in time for all subscribers  $v \in V$ .*

**Proof.** Using Lemma 11, we assume that the sorted ring induced by edges  $E_R$  is already correctly built. We perform an induction over the levels  $i \in \{1, \dots, \lceil \log n \rceil\}$  of shortcuts and show that all shortcuts on each level are eventually established. The induction base ( $i = \lceil \log n \rceil$ ) trivially holds, as shortcuts on level  $\lceil \log n \rceil$  are ring edges in  $E_R$ . For the induction hypothesis, assume that all shortcuts on level  $i$  have already been established, i.e., all nodes in  $K_i := \{w \in V \mid |\text{label}_w| \leq i\}$  already form a sorted ring (recall Definition 2). In the induction step we show that all shortcuts on level  $i - 1$  are eventually established. It is easy to see that  $K_{i-1} \subset K_i$  holds. Denote the sorted ring over nodes in  $K_i$  as  $R_i$ . Observe that each node  $v \in K_i \setminus K_{i-1}$  has two neighbors  $u, w$  in  $R_i$  with  $u, w \in K_{i-1}$ . Thus, by definition of our protocol,  $v$  eventually introduces  $u$  to  $w$  and vice versa when calling its TIMEOUT method. This implies that the shortcuts  $(u, w)$  and  $(w, u)$  are established. The above argumentation implies that the ring  $R_{i-1}$  is established eventually, which concludes the induction. ◀

Having shown the convergence of the supervisor (Lemma 10), the sorted ring for all subscribers (Lemma 11) and the convergence of the shortcuts for all subscribers (Lemma 12), we have proved the convergence of the overall system (Theorem 8). ◀

► **Theorem 13 (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.*

**Proof.** We need to show closure for the supervisor's *database* as well as for the skip ring. Again, we start at the supervisor:

► **Lemma 14 (Supervisor Closure).** *If the explicit edges in  $SR(n)$  already form a supervised skip ring, then the supervisor's database does not get modified anymore, if no subscriber joins or leaves the system.*

**Proof.** The supervisor's *database* is only modified, if SUBSCRIBE requests for new subscribers arrive at  $s$  or a subscriber  $v \in V$  unsubscribes by sending an UNSUBSCRIBE request to  $s$ . Both scenarios are forbidden by assumption of the lemma. ◀

► **Lemma 15 (Ring Closure).** *If the explicit edges in  $SR(n)$  already form a supervised skip ring, then the set  $E_R$  does not get modified anymore, if no subscriber joins or leaves the system.*

**Proof.** Messages that are generated by the extended BUILDING protocol do not modify the edge set  $E_R$ , since closure of the extended BUILDING protocol (Lemma 4) holds. Observe that introduction messages for shortcuts do not modify the variables  $v.\text{left}, v.\text{right}$  and  $v.\text{ring}$  for a subscriber  $v \in V$ . Implicit edges generated by configurations sent out by the supervisor  $s$  are just merged with the existing explicit edges at the receiving subscriber  $v$ , since  $v$  already stores the correct configuration. ◀

► **Lemma 16 (Shortcut Closure).** *If the explicit edges in  $SR(n)$  already form a supervised skip ring, then the set  $E_S$  does not get modified anymore, if no subscriber joins or leaves the system.*

**Proof.** Note that shortcuts are only modified in INTRODUCESHORTCUT or (Algorithm 4, line 22). INTRODUCESHORTCUT is only called to introduce a node  $u$  to some shortcut  $w$ , which already exists, since no node generates an introduction message for two nodes that are not allowed to be connected by a shortcut, as one can easily see via induction. ◀

By combining Lemmas 14, 15 and 16, we obtain Theorem 13. ◀

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

► **Theorem 17 (Publication Convergence).** *Consider an initially weakly connected graph  $G = (V, E_R \cup E_S)$  and assume that there are publications  $P \subset \mathcal{P}^*$  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.** First note that in our protocol, no publish messages are deleted from the Patricia tries, i.e., once a subscriber  $v \in V$  has a publication  $p \in P$  stored in its Patricia trie  $v.T$ , it will never remove  $p$  from  $v.T$ . Therefore, we apply Theorem 8 and assume that the explicit edges in  $G$  already form a supervised skip ring. For a subscriber  $u \in V$  let  $P_u \subseteq P$  be the set of publication stored in the leaf nodes of  $u.T$ . We define the potential of a pair  $(u, v)$  of subscribers by

$$\phi(u, v) = |P_{u,v} \setminus P_v|,$$

where  $P_{u,v}$  is a shorthand expression for  $P_u \cup P_v$ . Note that  $\phi(u, v) = \phi(v, u)$  does not hold in general, since intuitively speaking,  $\phi(u, v)$  returns the number of publications stored in  $u.T$  that are missing in  $v.T$ . The potential over all subscribers is then defined as

$$\Phi = \sum_{(u,v) \in E_R} \phi(u, v).$$

It is easy to see that the following Corollary holds:

► **Corollary 18.**  $\Phi \geq 0$  at any point in time and  $\Phi = 0 \Leftrightarrow P_u = P$  for all subscribers  $u \in V$ .

We obtain convergence, if we can show that  $\Phi$  is monotonically decreasing and eventually  $\Phi = 0$ . We show that  $\Phi$  is monotonically decreasing with the following claim:

► **Lemma 19.**  $\Phi$  never increases.

**Proof.** By definition,  $\Phi$  only increases, if there is  $(u, v) \in E_R$  for which  $\phi(u, v)$  increases. This implies that there is a subscriber  $w \in V$ ,  $u \neq w \neq v$  that has sent  $u$  a set of publications  $P_w \subset P$  via a PUBLISH request that are not contained in  $v.T$  yet. Then  $\phi'(u, v) = |(P_{u,v} \cup P_w) \setminus P_v| = \phi(u, v) + |P_w|$ . But this also implies that  $\phi(w, u)$  decreases by  $|P_w|$ , because  $\phi'(w, u) = |P_{w,u} \setminus (P_u \cup P_w)| = \phi(w, u) - |P_w|$ , leaving  $\Phi$  at the same value as before. ◀

To complete the proof, we still need to show that eventually  $\Phi = 0$ .

► **Lemma 20.** *As long as there exists  $\phi(u, v) > 0$  for some edge  $(u, v) \in E_R$ , there is a computation after which  $\phi(u, v)$  has decreased.*

**Proof.** Let  $\phi(u, v) > 0$  for two subscribers  $u, v \in V$  that are connected via a ring edge  $(u, v) \in E_R$ . Let  $p \in u.T$  be the node with minimal label length  $|p.label|$ , for which it holds that all publications stored in the leafs of the subtree of  $p$  are missing in  $T_v$ . Note that  $p.label$  is a prefix for all those publications. Obviously, such a node always exists, when there is one or more publication missing in  $v.T$ . Assume to the contrary that we are in state  $s$  with  $\phi(u, v) > 0$  and for all possible computations,  $\phi(u, v)$  does not decrease. We state a computation in which  $u$  delivers all publications with prefix  $p.label$  to  $v$ , resulting in a decrease of  $\phi(u, v)$ .

W.l.o.g. consider the node  $t \in u.T$  with label length  $|t.label|$  minimal, for which  $t.label$  is a prefix of  $p.label$  and for which there does not exist a node in  $v.T$  with label  $t.label$ . Such a node exists, because in case there is no inner node in  $u.T$  with these properties, we can choose  $t = p$ . Note that we consider  $p.label$  to be a prefix of itself. Now we look at the path  $(r_u = t_1, \dots, t_k = t)$  from the root node  $r_u$  of  $u.T$  to  $t$ . It holds that for all nodes  $t_i$  with  $i \neq k$  on this path, there exists a node  $t'_i \in v.T$  with the same label as  $t_i$ , i.e.,  $t'_i.label = t_i.label$ . Otherwise our choice for  $t$  would be wrong, since  $|t.label|$  is not minimal. Since  $h$  is a collision-resistant hash function for the hash values of the nodes, we have for  $i \in \{1, \dots, k-1\}$  that  $t_i.hash \neq t'_i.hash$ . We prove the following claim:

► **Claim 21.** *Eventually, subscriber  $u$  sends a  $\text{CHECKTRIE}(u, t)$  request to  $v$ .*

**Proof.** Consider the path  $(r_u = t_1, \dots, t_k = t)$  from above from the root node  $r_u$  of  $u.T$  to  $t$ . Assume that  $k$  is odd. By definition of our protocol,  $u$  will eventually send a  $\text{CHECKTRIE}(u, r_u)$  request to  $v$ . Since the root hashes are not equal,  $v$  sends a  $\text{CHECKTRIE}(v, t_2)$  request to  $u$ . Since we assumed that  $t_i.hash \neq t'_i.hash$  for all  $i \in \{1, \dots, k-1\}$ , this chain continues with  $u$  sending  $\text{CHECKTRIE}(u, t_j)$  requests to  $v$ ,  $j$  odd, until  $u$  sends a  $\text{CHECKTRIE}(u, t_k)$  request to  $v$ , which proves the lemma. The case where  $k$  is even works analogously when starting at subscriber  $v$ . ◀

Applying the claim above, we now assume that  $v$  has received a  $\text{CHECKTRIE}(v, t)$  request. By our initial assumptions for  $t$  it holds that there is no node with label  $t.label$  contained in  $v.T$ . Thus,  $v$  searches for a node  $c \in v.T$  with label  $c.label = (t.label \circ b_1 \circ \dots \circ b_k)$  of minimum length and responds to  $u$  with a  $\text{CHECKANDPUBLISH}(v, c, p')$  request. Here,  $p' = t.label \circ (1 - b_1)$  if  $c$  exists, otherwise  $p' = t.label$ .

► **Claim 22.**  $p' = p.label$ .

**Proof.** For  $p' = t.label$ , we know that there does not exist a node with a label that has  $t.label$  as a prefix. Hence, all publications with prefix  $t.label$  are missing at  $v.T$ , implying  $t.label = p.label$ , because we chose  $p.label$  to be of minimal length. For  $p' = t.label \circ (1 - b_1)$ , we know because of the existence of  $c \in v.T$  and the non-existence of a node with label  $t.label$  in  $v.T$  that there is no node with label  $t.label \circ (1 - b_1)$  stored in  $v.T$ . Thus, all publications with prefix  $t.label \circ (1 - b_1)$  are missing in  $v.T$ . Since we chose  $p.label$  to be of minimal length we get  $t.label \circ (1 - b_1) = p.label$ . ◀

Subscriber  $u$  responds to the  $\text{CHECKANDPUBLISH}$  request by sending all publications  $P \subset \mathcal{P}^*$  to  $v$  that have prefix  $p'$ . Since  $p' = l_p$ ,  $\phi(u, v)$  decreases and the lemma follows. ◀

By combining Corollary 18, Lemma 19 and Lemma 20 we proved the theorem. ◀

Finally we also show *convergence* for publications.

► **Theorem 23 (Publication Closure).** *Consider a stable supervised skip ring  $SR(n)$  and assume that all subscribers store the exact same Patricia trie containing publications  $P \subset \mathcal{P}^*$ . 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.*

**Proof.** In this stable network the only type of request regarding publications is the periodic  $\text{CHECKTRIE}$  request. Thus, at the receiver, the hash values of the Patricia tries' root nodes are compared. But since all nodes store the same Patricia trie, both hashes are equal, resulting in no further message being sent out as an answer to the  $\text{CHECKTRIE}$  request. ◀

## 6 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 Yair Amir, Cristina Nita-Rotaru, Jonathan Robert Stanton, and Gene Tsudik. Secure spread: An integrated architecture for secure group communication. *IEEE Trans. Dependable Sec. Comput.*, 2(3):248–261, 2005.
- 2 Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. *Theor. Comput. Sci.*, 512:2–14, 2013.
- 3 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
- 4 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
- 5 Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. *Acta Inf.*, 40(9):609–636, 2004.
- 6 Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Trans. Mob. Comput.*, 5(7):893–905, 2006.
- 7 Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
- 8 Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João L. M. Pereira, Kenneth A. Ross, and Dennis E. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 115–126, 2001.
- 9 Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.
- 10 Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 131–140. ACM, 2009.
- 11 Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Skip<sup>+</sup>: A self-stabilizing skip graph. *J. ACM*, 61(6):36:1–36:26, 2014.
- 12 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. IEEE Computer Society, 2012.

- 13 Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: A self-stabilizing chord overlay network. *Theory Comput. Syst.*, 55(3):591–612, 2014.
- 14 Kishore Kothapalli and Christian Scheideler. Supervised peer-to-peer systems. In *8th International Symposium on Parallel Architectures, Algorithms, and Networks, ISPAN 2005, December 7-9, 2005, Las Vegas, Nevada, USA*, pages 188–193. IEEE Computer Society, 2005.
- 15 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.
- 16 Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
- 17 Gero Mühl, Michael A. Jaeger, Klaus Herrmann, Torben Weis, Andreas Ulbrich, and Ludger Fiege. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 664–674. Springer, 2005.
- 18 Melih Onus, Andréa W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007.
- 19 Andréa W. Richa, Christian Scheideler, and Phillip Stevens. Self-stabilizing de bruijn networks. In *SSS 2011, Grenoble, France, October 10-12, 2011.*, volume 6976 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2011.
- 20 Gerry Siegemund and Volker Turau. PSVR - self-stabilizing publish/subscribe communication for ad-hoc networks (short paper). In *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, volume 10083 of *Lecture Notes in Computer Science*, pages 346–351, 2016.
- 21 Gerry Siegemund, Volker Turau, and Khaled Maamra. A self-stabilizing publish/subscribe middleware for wireless sensor networks. In *2015 International Conference and Workshops on Networked Systems, NetSys 2015, Cottbus, Germany, March 9-12, 2015*, pages 1–8. IEEE Computer Society, 2015.

## **A** Pseudocode

In order to not blow up the pseudocode, we may use a tuple  $t_1 = (\text{label}_{t_1}, v_{t_1})$  for comparisons regarding the label or for remote calls, i.e., we may write  $t_1 < t_2$  instead of  $\text{label}_{t_1} < \text{label}_{t_2}$  when comparing labels, or in case we want to send a message to  $v_{t_1}$ , we may just write  $t_1 \leftarrow \text{REMOTECALL}$  instead of  $v_{t_1} \leftarrow \text{REMOTECALL}$ . This prevents us from always having to formally define the contents of each tuple  $t \in \{0, 1\}^* \times V$  in the code.

---

**Algorithm 1** BUILDLIST protocol w.r.t. corrupted IDs executed by  $u \in V$ 


---

```

1: procedure BUILDLISTTIMEOUT
2:   if  $u.left \leq u.label$  then                                     ▷ Analogously for  $u.right \geq u.label$ 
3:      $u.left \leftarrow \text{CHECK}(u, u.left, LIN)$ 
4:   else
5:      $u \leftarrow \text{LINEARIZE}(u.left)$ 
6:      $u.left \leftarrow \perp$ 
7:
8: procedure CHECK( $sender, label, flag \in \{CYC, LIN\}$ )
9:   if  $u.label \neq label$  then
10:     $sender \leftarrow \text{INTRODUCE}(u, flag)$ 
11:   else
12:     $\text{INTRODUCE}(sender, flag)$ 
13:
14: procedure LINEARIZE( $v$ )
15:   if  $u.label \neq \perp$  then
16:     if  $v = u.left \vee v = u.right$  then
17:       if  $v = u.left \wedge label_v \neq u.left$  then                                     ▷ Analogously for  $u.right$ 
18:         if  $label_v \leq u.label$  then
19:           Replace label in  $u.left$  with  $label_v$ 
20:         else
21:            $u \leftarrow \text{LINEARIZE}(v)$ 
22:            $u.left \leftarrow \perp$ 
23:       else
24:         if  $label_v \leq u.left$  then                                     ▷ Analogously for  $u.right \leq label_v$ 
25:            $u.left \leftarrow \text{LINEARIZE}(v)$ 
26:         if  $u.left < label_v \leq u.label$  then                                     ▷ Analogously for  $u.right$ 
27:            $v \leftarrow \text{LINEARIZE}(u.left)$ 
28:            $u.left \leftarrow v$ 
29:       else
30:          $v \leftarrow \text{REMOVECONNECTIONS}(u)$                                      ▷ Lets  $v$  remove  $u$  from its local storage

```

---

---

**Algorithm 2** Extended BUILDRING protocol executed by  $u \in V$ 


---

```

1: procedure BUILDRINGTIMEOUT
2:   if  $u.ring = \perp$  then
3:     if  $u.left = \perp \wedge u.right \neq \perp$  then  $\triangleright$  Analogously for  $u.right = \perp \wedge u.left \neq \perp$ 
4:        $u.right \leftarrow \text{INTRODUCE}(u, CYC)$ 
5:   else if  $u.label = \perp$  then
6:      $u.ring \leftarrow \text{REMOVECONNECTIONS}(u)$   $\triangleright$  Lets  $u.ring$  delete  $u$ 
7:      $u.ring \leftarrow \perp$ 
8:   else
9:     if  $u.left \neq \perp \wedge u.ring > u.label$  then  $\triangleright$  Analogously for  $u.right$ 
10:       $u.left \leftarrow \text{INTRODUCE}(u.ring, CYC)$ 
11:       $u.ring \leftarrow \perp$ 
12:     if  $(u.left = \perp \wedge u.ring > u.label) \vee (u.right = \perp \wedge u.ring < u.label)$  then
13:        $u.ring \leftarrow \text{CHECK}(u, u.ring, CYC)$ 
14:   BUILDLISTTIMEOUT
15:
16: procedure INTRODUCE( $c = (label_v, v)$ ,  $flag \in \{CYC, LIN\}$ )
17:   if  $u.label \neq \perp$  then
18:     if  $u.ring = v \wedge u.ring \neq label_v$  then
19:       if  $(label_v < u.label \wedge u.ring < u.label) \vee (label_v > u.label \wedge u.ring > u.label)$ 
then
20:          $u.ring \leftarrow (label_v, v)$ 
21:       else
22:          $u \leftarrow \text{INTRODUCE}((label_v, v))$ 
23:          $u.ring \leftarrow \perp$ 
24:       if  $flag = CYC$  then
25:         if  $u.ring = \perp$  then
26:           if  $(label_v < u.label \wedge u.right = \perp)$  then  $\triangleright$  Analogously for  $u.left$ 
27:              $u.ring \leftarrow v$ 
28:           if  $label_v < u.label \wedge u.right \neq \perp$  then  $\triangleright$  Analogously for  $u.left$ 
29:              $u.right \leftarrow \text{INTRODUCE}(v, CYC)$ 
30:         else if  $(u.ring < u.label \wedge label_v < u.label) \vee (u.ring > u.label \wedge label_v >$ 
 $u.label)$  then
31:            $u.ring = \text{argmax}_{w \in \{c, u.ring\}} \{|w - u.label|\}$ 
32:            $w' \leftarrow \{c, u.ring\} \setminus w$ 
33:            $u \leftarrow \text{INTRODUCE}(w', LIN)$ 
34:            $w \leftarrow \text{INTRODUCE}(u.ring, LIN)$ 
35:         else
36:            $u \leftarrow \text{INTRODUCE}(v, LIN)$ 
37:            $u \leftarrow \text{INTRODUCE}(u.ring, LIN)$ 
38:            $u.ring \leftarrow \perp$ 
39:       else
40:          $\text{LINEARIZE}(v)$   $\triangleright$  Method from BUILDLIST with corrupted node ids
41:     else
42:        $v \leftarrow \text{REMOVECONNECTIONS}(u)$   $\triangleright$  Lets  $v$  remove  $u$  from its local storage

```

---

**Algorithm 3** BUILD<sub>SR</sub> protocol executed by the supervisor

---

```

1: procedure TIMEOUT(true) ▷ Periodically executed!
2:   CHECKLABELS
3:   next  $\leftarrow$  next + 1 mod n
4:   Let  $(label, v) \in database$  with  $label = l(next)$ 
5:   GETCONFIGURATION(v)
6: procedure SUBSCRIBE(v)
7:   if  $\forall (l, u) \in database : u \neq v$  then
8:      $database \leftarrow database \cup \{(l(n), v)\}$  ▷ Update the database
9:     Let  $pred_v, succ_v \in database$  be the predecessor/successor of v in the ring
10:     $v \leftarrow SETDATA(pred_v, l(n), succ_v)$ 
11:   else
12:     GETCONFIGURATION(v) ▷ Just send all of v's connections to v
13: procedure UNSUBSCRIBE(v)
14:   CHECKMULTIPLECOPIES(v)
15:   if  $\exists (label_u, u) \in database : u = v$  then
16:     if  $n > 1 \wedge l^{-1}(label_u) \neq n - 1$  then
17:       Let  $pred_v, succ_v \in database$  be the predecessor/successor of v in the ring
18:       Let  $(label_w, w) \in database$  with  $label_w = l(n - 1)$ 
19:        $database \leftarrow database \cup \{(label_u, w)\} \setminus \{(label_u, v)\} \setminus \{(label_w, w)\}$ 
20:        $w \leftarrow SETDATA(pred_v, label_u, succ_v)$  ▷ Replace w's id
21:     else
22:        $database \leftarrow database \setminus \{(label_u, v)\}$  ▷ Just update database
23:      $v \leftarrow SETDATA(\perp, \perp, \perp)$  ▷ Let v delete its label and connections
24: procedure GETCONFIGURATION(u)
25:   CHECKMULTIPLECOPIES(u)
26:   if  $\exists (label_v, v) \in database : v = u$  then
27:     Let  $pred_v, succ_v \in database$  be the predecessor/successor of v in the ring
28:      $v \leftarrow SETDATA(pred_v, (label_v, v), succ_v)$ 
29:   else
30:      $u \leftarrow SETDATA(\perp, \perp, \perp)$  ▷ u not part of the database
31: procedure CHECKMULTIPLECOPIES(v)
32:   rem  $\leftarrow$  false
33:   for all  $(l, w) \in database$  do
34:     if  $rem = false \wedge w = v$  then
35:       rem  $\leftarrow$  true
36:     else if  $w = v$  then
37:        $database \leftarrow database \setminus \{(l, w)\}$ 
38: procedure CHECKLABELS ▷ Only called in TIMEOUT
39:   for all  $(l, v) \in database$  do
40:     if  $v = \perp$  then
41:        $database \leftarrow database \setminus \{(l, v)\}$  ▷ Remove tuple with empty value
42:     for all  $i \in \{0, \dots, n - 1\}$  do
43:       if  $\forall (l, v) \in database : l \neq l(i)$  then ▷ Tuple could not be found
44:         Get the tuple  $(l(j), w)$  with maximum  $j > i$ 
45:          $database \leftarrow database \setminus \{(l, \perp), (l(j), w)\} \cup \{(l(i), w)\}$  ▷ Set label for w

```

---

---

**Algorithm 4** BUILD SR protocol executed by subscribers  $u \in V$ 


---

```

1: procedure TIMEOUT(true) ▷ Periodically executed
2:   BUILD RING TIMEOUT
3:   Check and update  $u.shortcuts$ , s.t. it contains the correct labels afterwards.
4:   if  $u.label = \perp$  then ▷ As long as  $u$  has no label: Subscribe to supervisor
5:      $supervisor \leftarrow$  SUBSCRIBE( $u$ )
6:   else ▷ Just ask the supervisor for the correct configuration
7:     if  $u$  thinks that  $u.label$  is minimal then
8:       Let  $k := |u.label|$ 
9:       With probability  $1/2$ :  $supervisor \leftarrow$  GET CONFIGURATION( $u$ )
10:    else
11:      With prob.  $1/(2^k \cdot k^2)$ :  $supervisor \leftarrow$  GET CONFIGURATION( $u$ )
12:    if  $\exists(l_u, u), (l_w, w) \in v.shortcuts$  on level  $k = |v.label|$  then
13:       $u \leftarrow$  INTRODUCE SHORTCUT( $l_w, w$ )
14:       $w \leftarrow$  INTRODUCE SHORTCUT( $l_u, u$ )
15:
16: procedure SET DATA(pred, label, succ)
17:    $u.label \leftarrow label$ 
18:   if  $u.left \neq \perp \wedge (pred = \perp \vee |u.left - u.label| < |pred - u.label|)$  then
19:      $supervisor \leftarrow$  GET CONFIGURATION( $u.left$ ) ▷ Analog. for  $u.right, u.ring$ 
20:   Update  $u.left, u.right, u.ring$  w.r.t.  $pred, succ$  and  $label$ 
21:
22: procedure INTRODUCE SHORTCUT( $l, v$ )
23:   if  $\exists(l', v') \in u.shortcuts : l' = l$  then
24:     if  $v' \neq v$  then
25:        $u.shortcuts \leftarrow u.shortcuts \setminus \{(l', v')\}$  ▷ Remove old shortcut for label  $l$ 
26:     if  $v' \neq \perp$  then
27:       LINEARIZE( $(l', v')$ )
28:      $u.shortcuts \leftarrow u.shortcuts \cup \{(l, v)\}$ 
29:   else
30:     LINEARIZE( $v$ )

```

---

---

**Algorithm 5** Self-Stabilizing publication protocol for a subscriber  $u \in V$ 

---

```

1: procedure PUBLISHTIMEOUT
2:   Choose a random node  $v$  from  $u.left, u.right, u.ring$ 
3:   Let  $r$  be the root node of  $T_u$ , with label  $l_r$  and hash  $h_r$ 
4:    $v \leftarrow \text{CHECKTRIE}(u, (l_r, h_r))$ 
5:
6: procedure PUBLISH( $P$ )
7:   for all  $p \in P$  do
8:     if  $p$  not stored in the Patricia trie then
9:       INSERT( $p, T_u$ ) ▷ Insert  $p$  into  $T_u$ 
10:
11: procedure CHECKTRIE( $sender \in V, tuples \subset (\{0, 1\}^*)^2$ )
12:   for all  $(l, h) \in tuples$  do
13:      $v \leftarrow \text{SEARCHNODE}(l, T_u)$  ▷ Search for the node with label  $l$  in  $T_u$ 
14:     if  $v \neq \perp$  then
15:       if  $h_v \neq h \wedge v$  is an inner node of  $T_u$  then ▷ If  $h_v = h$ , subtrees are equal
16:         Let  $v_1, v_2$  be  $v$ 's child nodes
17:          $sender \leftarrow \text{CHECKTRIE}(u, \{(l_{v_1}, h_{v_1}), (l_{v_2}, h_{v_2})\})$ 
18:       else
19:         Let  $c \in T_u$  with minimal label  $l_c = (l \circ c_1 \circ \dots \circ c_k)$  for which  $l$  is a prefix
20:         if  $c \neq \perp$  then
21:            $sender \leftarrow \text{CHECKANDPUBLISH}(u, (l_c, h_c), p = (l \circ (1 - c_1)))$ 
22:         else
23:            $sender \leftarrow \text{CHECKANDPUBLISH}(u, \emptyset, l)$ 
24:
25: procedure CHECKANDPUBLISH( $sender \in V, tuples \subset (\{0, 1\}^*)^2, pf \in \{0, 1\}^*$ )
26:   CHECKTRIE( $sender, tuples$ )
27:    $P \leftarrow$  All publications with prefix  $pf$  from  $T_u$ 
28:    $sender \leftarrow \text{PUBLISH}(P)$ 
29:
30: procedure PUBLISHNEW( $p \in \mathcal{P}^k$ )
31:   if  $p$  not stored in the Patricia trie then
32:     INSERT( $p, T_u$ ) ▷ Insert  $p$  into  $T_u$ 
33:     for all  $v \in \{u.left, u.right, u.ring\} \cup u.shortcuts$  do
34:        $v \leftarrow \text{PUBLISHNEW}(p)$ 

```

---