Towards a Universal Approach for the Finite Departure Problem in Overlay Networks*

Andreas Koutsopoulos, Christian Scheideler, Thim Strothmann

Department of Computer Science,
University of Paderborn, Germany,
{koutsopo,scheidel,thim}@mail.upb.de

Abstract. A fundamental problem for overlay networks is to *safely* exclude leaving nodes, i.e., the nodes requesting to leave the overlay network are excluded from it without affecting its connectivity. There are a number of studies for safe node exclusion if the overlay is in a well-defined state, but almost no formal results are known for the case in which the overlay network is in an arbitrary initial state, i.e., when looking for a *self-stabilizing* solution for excluding leaving nodes. We study this problem in two variants: the *Finite Departure Problem ($\mathcal{FDP}$)* and the *Finite Sleep Problem ($\mathcal{FSP}$)*. In the $\mathcal{FDP}$ the leaving nodes have to irrevocably decide when it is safe to leave the network, whereas in the $\mathcal{FSP}$, this leaving decision does not have to be final: the nodes may resume computation when woken up by an incoming message. We are the first to present a self-stabilizing protocol for the $\mathcal{FDP}$ and the $\mathcal{FSP}$ that can be combined with a large class of overlay maintenance protocols so that these are then guaranteed to safely exclude leaving nodes from the system from any initial state while operating as specified for the staying nodes. In order to formally define the properties these overlay maintenance protocols have to satisfy, we identify four basic primitives for manipulating edges in an overlay network that might be of independent interest.

1 Introduction

Any distributed system must be based on some overlay network that specifies which nodes can directly send messages to which other nodes in the system. For distributed systems across the Internet, this is achieved by the nodes storing IP addresses of other nodes in that system, and in this case a node is said to be able to directly send a message to another node whenever it knows its IP address. A basic prerequisite for an overlay network which allows all pairs of nodes to exchange information is that it is connected, and a fundamental problem for overlay networks is to *preserve* connectivity while nodes are leaving, i.e., the nodes requesting to leave the overlay network are eventually excluded from it without disconnecting any staying nodes. Since due to permanent or transient failures a distributed system may rarely be in an ideal state, it would be desirable to find *self-stabilizing* protocols for the exclusion of

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

leaving nodes, i.e., from *any* initial state connectivity is preserved. While this seems to be a fundamental problem, only recently first solutions were found.

Foreback et al. [15] proposed to study this problem in two variants: the *Finite Departure Problem ($\mathcal{FDP}$)* and the *Finite Sleep Problem ($\mathcal{FSP}$)*. In the $\mathcal{FDP}$ the leaving nodes have to irrevocably decide when it is safe to leave the network, whereas in the $\mathcal{FSP}$, this leaving decision does not have to be final: the nodes may resume computation when woken up by an incoming message. On the negative side, Foreback et al. showed that there is no self-stabilizing local-control protocol for the $\mathcal{FDP}$. But if an oracle is available, then an appropriate local-control protocol can be constructed. Moreover, a variant of that protocol can solve the $\mathcal{FSP}$ without using an oracle. However, these protocols require that there is a fixed total order on the nodes (e.g., their names or IP addresses do not change), and they only work for a specific overlay maintenance protocol that aims at organizing the nodes in a sorted list.

In this paper, we present a self-stabilizing protocol for the $\mathcal{FDP}$ that can extend a large class of overlay maintenance protocols so that they are then guaranteed to eventually exclude the leaving nodes without risking disconnection and while the overlay maintenance protocol is operating as specified for the staying nodes. As a by-product, we present a set of four basic primitives for the manipulation of edges in overlay networks that are safe and universal in a sense that connectivity is preserved and that, in principle, one can get from any weakly connected graph to any other weakly connected graph. This might be of independent interest as we expect our insights to simplify the design and analysis of overlay maintenance protocols in the future.

1.1 Model

We consider a distributed system consisting of a fixed set of processes in which each process has a unique reference (like its IP address). We refer to processes and their references interchangeably. The system is controlled by a protocol that specifies the variables and actions that are available in each process. In addition to the protocol-based variables there is a system-based variable for each process called *channel* whose values are sets of messages. We denote the channel of process u as $u.Ch$ and $u.Ch$ contains all incoming messages to u. Its message capacity is unbounded and messages never get lost. A process can add a message to $u.Ch$ if it knows u (resp. its reference). Besides these channels there are no further communication means, so only point-to-point communication is possible.

A process u has a variable *mode(u)*$\in$ {leaving,staying} that is read-only. If this variable is set to **leaving**, the process is *leaving*; the process is *staying* if the variable is set to **staying**.

There are two types of actions. The first type of action has the form of a standard procedure $\langle label \rangle(\langle parameters \rangle) : \langle command \rangle$, where *label* is the unique name of that action, *parameters* specifies the parameter list of the action, and *command* specifies the statements to be executed when calling that action. Such actions can be called remotely. In fact, we assume that every message must be of the form $\langle label \rangle(\langle parameters \rangle)$ where *label* specifies the action to be called in the receiving process and *parameters* contains the parameters to be passed to that action call. All other messages will be ignored by the processes. Apart from being triggered by

messages, these actions may also be called locally by the process, which causes their immediate execution. The second type of action has the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$, where *label* and *command* are defined as above and *guard* is a predicate over local variables. We call an action whose guard is simply **true** a *timeout* action.

There are three special commands that are important for the study of our finite departure problem. Whenever a process u wants to send a message to a process whose reference is stored in variable v, it executes $v \leftarrow label(parameters)$, which asks the process referenced by v to execute action *label* with parameter list $parameters$. In addition, there are **exit** and **sleep**. If a process executes **exit** it enters a designated *exit state*. We call such a process *gone*. If a process executes **sleep**, it enters a *sleep state*. Such a process is *asleep*. If a process never wakes up again, it is called *permanently asleep*. A process that is neither gone nor asleep is called *awake*. See Figure 1 for the corresponding state graph for a process.

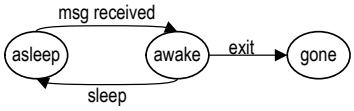


Fig. 1. The state graph for a process in our model.

The *system state* is an assignment of a value to every variable of each process and messages to each channel. An action in some process p is *enabled* in some system state if its guard evaluates to **true** and p is awake, or there is a message in $p.Ch$ requesting to call it and p is awake or asleep. In the latter case, p becomes awake again as soon as the corresponding message is processed (in which case it is removed from $p.Ch$). The action is *disabled* otherwise. Hence, while a gone process never wakes up again, an asleep process may wake up again when processing an appropriate message.

A *computation* is an infinite fair sequence of system states such that for each state s_i, the next state s_{i+1} is obtained by executing an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. We assume *weakly fair action execution* and *fair message receipt*. Weakly fair action execution means that if an action is enabled in all but finitely many states of the computation when the corresponding process is awake, and the process is awake for infinitely many states, then this action is executed infinitely often. Note that unless a process is gone or permanently asleep at some point (i.e., it never wakes up again), its timeout action is executed infinitely often. Fair message receipt means that if the computation contains a state where there is a message in a channel of a process that is not gone and that enables an action in that process, then that action is eventually executed with the parameters of that message, i.e., the message is eventually processed. Besides these fairness assumptions, we place no bounds on message propagation delay or relative process execution speeds, i.e., we allow fully asynchronous computations and non-FIFO message delivery.

We consider protocols that do not manipulate the internals of process references. Specifically, a protocol is *copy-store-send* if the only operations that it executes on

process references is copying them, storing them in local memory and sending them in a message. That is, operations on references such as addition, radix computation, hashing, etc. are not used. In a copy-store-send protocol, if a process does not store a reference in its local memory, the process may learn this reference only by receiving it in a message. A copy-store-send protocol cannot introduce new references to the system. It can only operate on the references that are already there.

The overlay network of a set of processes is determined by their knowledge of each other. We say that there is a (directed) *edge* from a to b, denoted by (a,b), if process a stores a reference of b in its local memory or has a message in $a.Ch$ carrying the reference of b. In the former case, the edge is called *explicit* (drawn solid in figures), and in the latter case, the edge is called *implicit* (drawn dashed). The edges form a directed *process (multi-)graph* PG. A *weakly connected component* of a directed graph G is a subgraph of G of maximum size so that for any two processes u and v in that subgraph there is a (not necessarily directed) path from u to v. Two processes that are not in the same weakly connected component are *disconnected*. We call a process p *hibernating* if p is asleep, $p.Ch$ is empty and all processes q that have a directed path to p in PG are also asleep and have an empty $q.Ch$. The following claim was shown in [15].

Claim. For any copy-store-send protocol and any system state of that protocol in which process p is hibernating, p is permanently asleep.

1.2 Problem Statement

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

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

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

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

In the following, a process is called *relevant* if it is neither gone nor hibernating. Otherwise we call it *irrelevant*. Since hibernating and gone processes will never execute any action, for the self-stabilization we only consider initial states in which all processes are relevant. We also restrict the initial state to contain only a finite number of messages that can trigger actions, since other messages are ignored by the processes. Finally, we do not allow the presence of references that do not belong to a process in the system. Their handling would require failure/presence detectors which is beyond the scope of this paper. From now on, an initial system state satisfies all of these constraints.

A system state is *legitimate* if (i) every staying process is awake, (ii) every leaving process is either hibernating or gone, and (iii) for each weakly connected component of the initial process graph, the staying processes in that component still form a weakly connected component. Now we are ready to formally state the following two problems.

Finite Departure Problem **($\mathcal{FDP}$)** : eventually reach a legitimate state for the case that the **sleep** command (and therefore the sleep state) is *not* available (but only **exit**).

Finite Sleep Problem **($\mathcal{FSP}$)** : eventually reach a legitimate state for the case that the **exit** command (and therefore the gone state) is *not* available (but only **sleep**).

A self-stabilizing solution for these problems must be able to solve these from any initial state and to satisfy the closure property afterwards. Notice that (i) and (ii) can trivially be maintained in a legitimate state, so for the closure property one just needs to ensure that (iii) is also maintained.

A process p can *safely* leave a system if the removal of p and its incident edges from PG does not disconnect any relevant processes. As shown in [15], there is no distributed algorithm within our model that can decide when it is safe for a process p to leave the system. Hence, we need oracles.

1.3 Oracles

An *oracle* $\mathcal{O}$ is a predicate that depends on the current system state and the process calling it. In the context of the $\mathcal{FDP}$, an oracle is supposed to advise a leaving process when it is safe to execute **exit**, thus we restrict our attention to protocols that *only* allow a leaving process to do so if the given oracle is **true** for it. Such a protocol is also said to *rely* on the oracle. Moreover, we restrict our attention to oracles that *only* depend on the current process graph of relevant processes and the calling process, i.e., oracles are of the form $\mathcal{O}{:}\mathcal{PG}\times P\to\{\textbf{true},\textbf{false}\}$ where $\mathcal{PG}$ is the set of process graphs and P is the set of processes.

We define the following oracle that we will use throughout the paper. Oracle $\mathcal{SINGLE}$ evaluates to **true** for a process u if u has edges with at most one other relevant process.

1.4 Related Work

To the best of our knowledge, the results by Foreback et al. [15] were the first attempt to rigorously analyze self-stabilizing process departures for overlay networks. The phenomenon they unearthed about the impossibility to locally decide when it is safe to leave the network is similar to the results of Fisher et al. [14] on the *consensus problem*, which is not solvable in an asynchronous system even if only a single process may crash. However, solutions to the *stabilizing consensus problem*, in which it is not required that each process irrevocably commits to a final value but that eventually they arrive at a common, stable value without being aware of that, are known [3, 11]. The impossibility can also be circumvented by the use of specialized oracles known as failure detectors [9].

Due to the popularity of peer-to-peer networks, the research literature on this subject is extensive [2, 4, 5, 8, 17, 24, 28]. While departure algorithms have been proposed in these papers, none of these protocols are self-stabilizing. Cases in which the rate of churn is limited have already been considered [1, 18, 23]. Kuhn et al. [23] present a solution that organizes nodes into cliques of $\Theta(\log n)$ size that they call super-nodes. Hayes et al. [18] handle limited churn with a topological repair strategy

called Forgiving Graph. For the case that the nodes have a sufficient amount of time to react, Saia et al. [26] propose an algorithm that repairs the network after an arbitrary number of deletions. Limited churn has also been studied in the context of adversarial nodes [6, 27]. While there is almost no work on self-stabilizing node departures, several self-stabilizing peer-to-peer protocols have already been proposed [10, 12, 19, 20, 25, 22]. The studied topologies range from simple line and ring structures [16] to skip lists and skip graphs [25, 20], expanders [13], Delaunay graphs [21], and a Chord variant [22]. Also a universal algorithm for topological self-stabilization is known [7]. However, none of these provide any means to exclude nodes that want to leave the network in a self-stabilizing manner.

1.5 Our results

Our main result is a self-stabilizing local-control protocol presented in Section 3 that can solve the $\mathcal{FDP}$ when relying on the $\mathcal{SINGLE}$ oracle. The $\mathcal{SINGLE}$ oracle was chosen for its simplicity, since we expect it to be easily implementable via timeouts in practice. The only interfaces our protocol needs to an underlying communication layer is that it can send a message to a process identified by some reference (by executing $v \leftarrow label(parameters)$ for some variable v holding a reference) and that it can check (via $v = w$) whether two references v and w point to the same or different processes. This has the advantage that the underlying layer is given full flexibility concerning the management of referencing information and that it does not have to pass any of that information (apart from whether two references point to the same process) to the process layer, which might be useful for anonymous networks. Instead, the protocols in [15] require that there is a fixed order on the processes. Also, the protocols in [15] were designed with a fixed topology in mind while this is not the case for our new protocol, which allows it to be easily integrated into existing overlay maintenance protocols, as we will demonstrate in this paper in Section 4. In order to simplify the analysis and formally specify the class of overlay maintenance protocols that can be used in conjunction with our departure protocol, we introduce four basic primitives for manipulating edges in the process graph in Section 2 and prove some fundamental results about them which might be of independent interest. We point out that the solutions in Sections 3 and 4 require the additional constraint that initially there exists at least one staying process per connected component of the overlay network.

2 Preliminaries

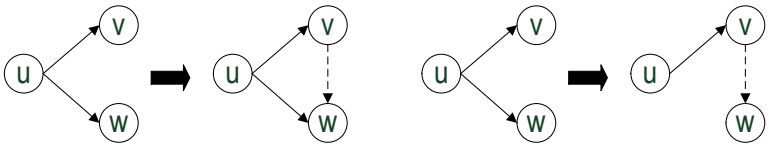
An important property for any overlay management protocol is the fact that weak connectivity is never lost by its own actions. Therefore, it is highly desirable that every process only executes primitives that preserve weak connectivity. Here we introduce four primitives for manipulating edges in an overlay network that are safe in a sense that they preserve weak connectivity (as long as there is no fault). This implies that *any* distributed protocol whose actions can be decomposed into these four primitives is guaranteed to preserve weak connectivity. The four primitives are:

Introduction If a process u has a reference to two processes v and w, u *introduces* w to v if it sends a message to v containing a reference to w while keeping the reference to w.

Delegation If a process u has a reference to two processes v and w, then u *delegates* w's reference to v if it sends a message to v containing a reference to w and deletes the reference to w.

Fusion If a process u has two references v and w with $v=w$, then it *fuses* them if it only keeps one of these references.

Reversal If a process u has a reference to some other process v, then it *reverses* the connection if it sends a reference of itself to v and deletes the reference to v.



(a) Introduction primitive (b) Delegation primitive

(c) Fusion primitive (d) Reversal primitive

Fig. 2. The four primitives in pictures.

Note that we assume that u,v,w are pairwise distinct. The only exception is *self-introduction*, a special case of introduction , where u sends a reference of itself to v, but does not delete its reference to v. The four primitives have the advantage that they can be executed locally by every process in a wait-free fashion (as none of the primitives requires any feedback). Also, they just need the ability to check whether two references point to the same process (see Fusion) to be implementable. Other than that, access to the contents of the references is not needed, which is useful. Moreover, it holds:

Lemma 1. *Introduction, Delegation, Fusion, and Reversal preserve weak connectivity.*

Proof. The statement obviously holds for Introduction since only additional edges are introduced. In Delegation an edge (u,w) is deleted, but there still exists a path from u to w via v, so u and w are still in the same weakly connected component. Fusion deletes an edge only if it is superfluous for weak connectivity. The Reversal rule deletes an edge (u,v) but replaces it with an edge (v,u), thereby also preserving weak connectivity. □

Let $\mathcal{P}$ denote the set of all distributed protocols where all interactions between processes can be decomposed into the four primitives. Not surprisingly, all of the self-stabilizing topology maintenance protocols proposed so far (e.g., [10, 12, 19, 20, 25, 22])

satisfy this property (as otherwise they would risk disconnection). Lemma 1 implies that any protocol in $\mathcal{P}$ preserves weak connectivity, which was previously shown individually for each cited protocol. Note that the first three primitives even preserve strong connectivity in a sense that for any pair of nodes u,v with a directed path in PG there will always be a directed path from u to v in PG when only allowing these three primitives. We say that a set of primitives is *universal* if the primitives allow one to get from any weakly connected graph $G=(V,E)$ to any other weakly connected graph $G'=(V,E')$ for PG. The set is *weakly universal* if G' is strongly connected.

Theorem 1. *Introduction, Delegation, Fusion, and Reversal are universal.*

Proof. We give a general strategy how to transform an arbitrary weakly connected graph $G=(V,E)$ into any other weakly connected graph $G'=(V,E')$. At first, note that if every process continuously introduces all neighbors to each other, including self-introduction, then the topology of PG is eventually transformed from G into a clique (in fact, $O(\log n)$ rounds of communication are sufficient for that as the distances between the nodes are essentially cut in half in each round of introduction).

Next we show that by using Delegation and Fusion, one can transform PG from the clique to the bidirected extension $G''=(V,E'')$ of G', i.e., the graph where for any edge $(u,v)\in E'$ there are edges $(u,v),(v,u)\in E''$. To do so, we make use of the fact that G'' is strongly connected. Consider an arbitrary edge (u,w) in PG that is not in E''. Since G'' is strongly connected, there exists a shortest path from u to w in G'' and therefore also in PG (as we first want to keep all edges in G''). Let v_1 be the first neighbor of u along that shortest path. Then u delegates the reference of w to v_1. Now the process v_1 (and all other processes on the shortest path) proceed similar to u by forwarding the reference to w along the shortest path up to the last process v_k, who is a neighbor of w. Process v_k uses Fusion to merge the edge with $(v_k,w)\in E''$. By applying this procedure to all edges not in E'', all that remains is G''.

At last we can use Reversal and Fusion to get from G'' to G'. To do so, every edge (u,v) that is in E'', but not in E' is reversed by u. Then the newly created edge (v,u) is fused with the already existing edge $(v,u)\in E'$. □

Note that Theorem 1 only shows that *in principle* it is possible to get from any weakly connected graph to any other weakly connected graph. From the proof we can conclude the following corollary.

Corollary 1. *Introduction, Delegation and Fusion are weakly universal.*

Furthermore, Introduction, Delegation, Fusion and Reversal are not only sufficient for universality but also necessary, i.e., by removing one primitive, universality is lost.

Theorem 2. *Introduction, Delegation, Fusion and Reversal are necessary for universality.*

Proof. To prove the lemma, we show that each primitive has a unique function that cannot be replaced by the other primitives. Introduction is the only primitive that can create new edges, so without it, any Graph G' with $|E'|>|E|$ cannot be reached from G. Fusion is the only primitive that reduces the overall number of edges. Delegation

is necessary, since by using only Introduction, Fusion and Reversal, a protocol can never locally disconnect two specific processes. Finally, consider an example graph G consisting of two processes u and v and an edge (u,v). Reversal is necessary to reach the goal topology G' that consists solely of the edge (v,u). □

3 Process Departures

In this section we present a self-stabilizing protocol for the $\mathcal{FDP}$ that only needs to compare references for equality as needed for the four primitives.

Our protocol consists of various actions. In the $present(v)$ action, a reference v is introduced to some process (i.e., the sending of a $present(v)$ message to u, corresponds to Introduction primitive). Moreover, in the $forward(v)$ action, the reference v is delegated to the executing process.

We assume that whenever a process a sends a request to call *present* or $forward$ containing a reference of a process b to another process c, it automatically sends some relevant information it knows about b along with it. In this section the only relevant information is the *mode* of b, which we denote as $a.mode(b)$ (i.e., a's knowledge of b's mode), which can be **staying** or **leaving**. Note that since we aim at a self-stabilizing protocol, $a.mode(b)$ might be incorrect (i.e., $a.mode(b) \neq mode(b)$) since b might have a different mode than a thinks it has. In this case we say that process a contains invalid (mode) information about process b. A system is in an *invalid* state if there exists at least one relevant process u with invalid information stored in u itself or in some incoming message in $u.Ch$. In both cases we say that u *has invalid information*. If no node has invalid information, the system state is said to be *valid*.

We denote the set of all references a process u stores in its local memory as the neighborhood set $u.N$ of u. Note that $u.N$ is not a variable of u but just a notation we use, which simplifies our protocol description and the proofs. Along with each $v \in u.N$, process u also stores its knowledge of the mode of v, denoted by $u.mode(v)$. Our solution makes use of a special variable called *anchor* whose reference is the only one *not* being in $u.N$ (i.e., it will be treated differently than all other references of v throughout our protocol). The *anchor* will only be used by the leaving nodes, so in a legitimate state, the *anchor* of a staying process is empty, denoted by $\perp$. The anchor of a leaving process v is a reference to some process which, according to the local information of v is a staying process. Therefore, each time v gets a message from a third process w, v forwards w to its anchor by a $forward$ message in the hope of eliminating all references to itself. Each process has a periodically executed *timeout* action. In case a process u is leaving, it sends a $present(u)$ message to its anchor in the *timeout* action (in order to verify it has a staying anchor). If it is staying, it sends a $present(u)$ message to all neighbors (to make other processes aware of it). This is an implementation of our earlier presented *self-introduction* primitive. Periodically executed self-introduction can ensure that invalid information vanishes from the system, as we will show later. Additionally, leaving processes consult $\mathcal{SINGLE}$ in *timeout*, and if it evaluates to true, the process is safe to perform **exit**. The actions of our protocol are presented in Algorithms 1- 3.

Algorithm 1 *u.timeout*

```
 1: if u.anchor≠⊥ and u.mode(anchor)=leaving then
 2:     u←present(u.anchor)                                   ▷ ♢
 3:     u.anchor:=⊥
 4: if mode(u)=leaving then
 5:     if u.N=∅ then
 6:         if SINGLE(u) then
 7:             exit
 8:         else
 9:             if u.anchor≠⊥ then
10:                 u.anchor←present(u)                       ▷ ♢
11:     else
12:         for all v∈u.N do
13:             u←forward(v)                                  ▷ ♢
14:         u.N:=∅
15: else
16:     if u.anchor≠⊥ then
17:         u←present(u.anchor)                               ▷ ♢
18:         u.anchor:=⊥
19:     for all v∈u.N do
20:         if u.mode(v)=leaving then
21:             u.N:=u.N\{v}
22:         v←present(u)                                      ▷ ♢ or ♣
```

Algorithm 2 u.present(v)

```
 1: if v=u.anchor and u.mode(v)=leaving then
 2:     u.anchor:=⊥                                           ▷ ♠
 3: if u.mode(v)=leaving then
 4:     if mode(u)=leaving then
 5:         v←forward(u)                                      ▷ ♣
 6:     else
 7:         if v∈u.N then
 8:             u.N:=u.N\{v}
 9:         v←forward(u)                                      ▷ ♣
10: else
11:     if mode(u)=leaving then
12:         if u.anchor≠⊥ then
13:             v←forward(u)                                  ▷ ♣
14:         else
15:             u.anchor:=v
16:     else
17:         u.N:=u.N∪{v}                                      ▷ ♠
```

Algorithm 3 u.forward(v)

```
 1: if v=u.anchor and u.mode(v)=leaving then
 2:     anchor:=⊥                                          ▷ ♠
 3: if u.mode(v)=leaving then
 4:     if mode(u)=leaving then
 5:         if u.anchor=⊥ then
 6:             v←forward(u)                               ▷ ♣
 7:         else
 8:             u.anchor←forward(v)                        ▷ ♡
 9:     else
10:         if v∈u.N then
11:             u.N:=u.N\{v}
12:         v←forward(u)                                   ▷ ♣
13: else
14:     if mode(u)=leaving then
15:         if u.anchor≠⊥ then
16:             u.anchor←forward(v)                        ▷ ♡
17:         else
18:             u.anchor:=v
19:     else
20:         u.N:=u.N∪{v}                                   ▷ ♠
```

3.1 Correctness proof

To show that our proposed protocol is a self-stabilizing solution to the $\mathcal{FDP}$, it remains to show two properties.

***Safety*:** The protocol never disconnects any relevant processes.
***Liveness*:** All leaving processes are eventually gone.

Lemma 2. *If a computation of our protocol starts in a state where the subgraph PG of relevant processes is weakly connected, it remains weakly connected in every state of the computation.*

To prove safety we make use of the results from Section 2.

Proof. First of all, note that each relevant process is also awake, since obviously gone processes cannot be relevant. The proof of the lemma relies on the fact that our protocol that the (awake) processes run is a composition of the four primitives presented in Section 2. To illustrate this, the protocol is annotated with the symbols ♢,♡,♠,♣. Each symbol represents a primitive: ♢ is (Self-)Introduction, ♡ is Delegation, ♠ is Fusion and ♣ is Reversal. Therefore, we can use the result of Lemma 1 and the fact that $\mathcal{SINGLE}$ preserves weak connectivity in the only case in which we do not use a primitive, (i.e., a process executes **exit**). This proves the lemma. □

It remains to show that our protocol makes progress such that all leaving processes eventually leave the system. Due to space constraints, we only sketch the proof of Lemma 3.

Lemma 3. *Leaving processes eventually execute the* ***exit*** *command, thereby preserving liveness.*

Sketch of Proof: Let Φ_t be a potential function that denotes the amount of invalid information present in the system at some time t, i.e., Φ_t is equal to the number of edges (x,y), either explicit or implicit, such that $mode(y) \neq x.mode(y)$.

The only way Φ_t could increase is if invalid information is copied. In order to do so, a process u has to forward invalid information about process v to another process w , since the information sent about oneself is always valid. The only spots in the pseudocode where this can potentially happen are lines 8 and 16 of the $forward$ action, where u sends $forward(v)$ to $u.anchor$. However, in that case v is not saved by u. So , even if u sends invalid information about v to $u.anchor$, the invalid information is not duplicated in the system. Therefore, $\Phi_t \geq \Phi_{t'}$ for any $t' > t$. To show that the system state is eventually valid, it suffices to show that as long as $\Phi_t > 0$ it holds that for any t there is a $t' > t$ such that $\Phi_{t'} < \Phi_t$.

Let (u,v) be an edge that contains invalid information at time t. We have to show that for every combination of the values of $mode(u)$ and $mode(v)$ the potential drops. Due to page limitations we skip this part of the proof.

The statement of Lemma 3 follows, since we can show that eventually a leaving process which has an edge to or from some staying process u executes **exit**. By using this argument inductively we have that eventually all leaving processes execute **exit**. □

From Lemma 2 and 3 we can conclude the following Theorem.

Theorem 3. *The protocol depicted in Algorithms 1- 3 together with the oracle* $\mathcal{SINGLE}$ *is a self-stabilizing solution to the* $\mathcal{FDP}$*.*

4 Embedding in existing overlay protocols

In this section we show how the protocol that was developed in Section 3 can be combined with a large class of distributed overlay protocols. Note that the original protocol does not necessarily have to be self-stabilizing. However, it must satisfy our safety requirement, i.e., no action should disconnect processes. The framework given below solves the $\mathcal{FDP}$ (Section 4.1).

4.1 FDP for arbitrary protocols

Consider any protocol $P \in \mathcal{P}$ (i.e., P is based on the four primitives and hence satisfies the safety condition). In order to combine our protocol with P, P has to fulfill two algorithmic requirements. First, P conducts periodic *self-introduction*, i.e., it has a periodically executed (*timeout*) action, in which the executing process introduces itself to all processes in its neighborhood (among other activities). The timeout action of P is called *P-timeout*. Second, P has a *postprocess* action, which is able to handle messages that cannot be delivered, i.e., if a message $v \leftarrow label(parameters)$ cannot be delivered, *postprocess* is able to act accordingly in order to reintegrate the information into the process. We need *postprocess* to handle messages that cannot be delivered because references of leaving processes are in *parameters*. Therefore, we require that *postprocess*

uses the $forward$ messages of our original protocol to get rid of these references. The exact inner workings of *postprocess* are closely tied to P itself, therefore we do not specify how *postprocess* deals with references of staying processes and other variables that are in *parameters*. Apart from being useful for us to solve the $\mathcal{FDP}$, such a *postprocess* action is also helpful in cases of messages that need to be reintegrated into the process, for example because their delivery failed before. Note that all self-stabilizing overlay protocols proposed so far can easily be adapted to satisfy these requirements.

Let P' be the protocol framework that combines P and the already presented protocol to solve the $\mathcal{FDP}$. The idea of P' is to introduce an action *preprocess* that is used every time a process u sends a message $v \leftarrow label(parameters)$ in P. Instead of sending this message directly u calls its $preprocess(label, parameters)$ action. This action saves the message in a message list and verifies the mode of v and each process x in *parameters* by sending a $verify(u)$ message to that process. These $verify$ messages are resent in timeout, if an answer has not been received yet. Once all processes have answered by a $process(v)$ message, u either sends the message $v \leftarrow label(parameters)$ if all processes in *parameters* are staying, otherwise it calls the local $postprocess(parameters)$ action. The *postprocess* action makes sure that all *leaving* processes in *parameters* are excluded and that staying processes are reintegrated into P. Note that *preprocess* and *postprocess* can only be called locally. To enhance readability we write $(x_1,...,x_k)$ instead of *parameters* in all algorithms, thereby focusing on the process references of messages in P and leaving out the part of *parameters* which does not contain process references, but just additional information. However, this additional information in *parameters* is not lost by *preprocess* and *postprocess*, but we do not interfere with it.

Another addition is that every process u executing P' is required to maintain an additional variable $u.anchor$, which (in a valid configuration) has the value $\perp$ if u is staying. Moreover, each process maintains a list variable $u.mlist$ which stores all the messages u wants to send. These are sent out once the valid information from the x_i processes arrive by *process* messages. In addition to our earlier protocol the mode information of a reference saved in *mlist* can have the additional value *unknown* to indicate that a *process* message has not been received yet, i.e., the *mode* is not verified. However, the mode information of the node itself ($mode(v)$) of course still only can have the value *leaving* or *staying*. It remains to specify how leaving nodes react, if a $label(parameters)$ message of P is received. A leaving node, will not execute the corresponding action of P but sends *present* messages to all processes in *parameters*, in order to remove possible references to it.

Due to space constraints the framework for constructing the modified protocol P' cannot be presented in this paper. Note that even though we do not present them in a specific algorithm, all actions of P have to adhere to the changes presented in the last paragraph (not only *timeout*). Furthermore, the *present* and $forward$ actions from the last section are changed in case a staying process gets a reference from another staying process.

Correctness proof We need to show that all leaving processes are eventually gone and, in case P is self-stabilizing, that protocol P' eventually works like P (e.g., reaches

the same target topology). The proof of Theorem 4 proceeds analogously to the proofs of Section 3.

Theorem 4. *Let $P \in \mathcal{P}$ be a distributed overlay protocol which solves some distributed problem $\mathcal{DP}$ with the already mentioned requirements. Then there is another protocol P' constructed as described above, such that P' eventually solves $\mathcal{FDP}$. In addition, if P is self-stabilizing, then P' also solves $\mathcal{DP}$.*

Analogous to the results in [15], we can overcome the use of oracles by relaxing the $\mathcal{FDP}$ to the $\mathcal{FSP}$. In this problem a process u can either be asleep or awake. If u is asleep, it does not preform any actions besides waiting for incoming messages. If it is awake, it conducts the desired protocol as usual. Once an asleep process receives a message, it automatically becomes awake again and executes the corresponding actions.

5 Conclusion

We presented a self-stabilizing protocol for the $\mathcal{FDP}$ and the $\mathcal{FSP}$ that can be combined with a large class of overlay maintenance protocols. Additionally, we identified four basic primitives for manipulating edges in an overlay network that preserve weak connectivity and are universal.

In the future we want to investigate stronger safety conditions for overlay networks than just connectivity.

References

1. Keno Albrecht, Fabian Kuhn, and Roger Wattenhofer. Dependable peer-to-peer systems withstanding dynamic adversarial churn. In *Dependable Systems*, pages 275–294, 2006.
2. David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP*, pages 131–145, 2001.
3. Dana Angluin, Michael J. Fischer, and Hong Jiang. Stabilizing consensus in mobile networks. In *DCOSS*, pages 37–50, 2006.
4. James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007.
5. Baruch Awerbuch and Christian Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *SODA*, pages 318–327, 2004.
6. Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust dht. *Theory Comput. Syst.*, 45(2):234–260, 2009.
7. 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.
8. Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, and Mark Thober. Pagoda: a dynamic overlay network for routing, data management, and multicasting. In *SPAA*, pages 170–179, 2004.
9. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
10. Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list and skip graph. *Theor. Comput. Sci.*, 428:18–35, 2012.

11. Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing consensus with the power of two choices. In *SPAA*, pages 149–158, 2011.
12. Danny Dolev, Ezra Hoch, and Robbert van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *PODC*, volume 4878, 2007.
13. Shlomi Dolev and Nir Tzachar. Spanders: Distributed spanning expanders. *Sci. Comput. Program.*, 78(5):544–555, 2013.
14. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
15. Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On stabilizing departures in overlay networks. In *SSS*, pages 48–62, 2014.
16. Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *LATIN*, pages 294–305, 2010.
17. Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems*, 2003.
18. Thomas P. Hayes, Jared Saia, and Amitabh Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, 25(4):261–278, 2012.
19. Thomas Hérault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, and Joffroy Beauquier. Brief announcement: Self-stabilizing spanning tree algorithm for large scale systems. In *SSS*, pages 574–575, 2006.
20. Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC*, pages 131–140, 2009.
21. Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. Towards higher-dimensional topological self-stabilization: A distributed algorithm for delaunay graphs. *Theor. Comput. Sci.*, 457:137–148, 2012.
22. Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: a self-stabilizing chord overlay network. In *SPAA*, pages 235–244, 2011.
23. Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Distributed Computing*, 22(4):249–267, 2010.
24. Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC*, pages 183–192, 2002.
25. Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *SSS*, pages 356–370, October 2011.
26. Jared Saia and Amitabh Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *IPDPS*, pages 1–12, 2008.
27. Christian Scheideler. How to spread adversarial nodes?: rotate! In *STOC*, pages 704–713, 2005.
28. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.