On Stabilizing Departures in Overlay Networks

Dianne Foreback[1], Andreas Koutsopoulos[2], Mikhail Nesterenko[1], Christian Scheideler[2], and Thim Strothmann[2]

[1] Kent State University
[2] University of Paderborn

Abstract. A fundamental problem for peer-to-peer systems is to maintain connectivity while nodes are leaving, i.e., the nodes requesting to leave the peer-to-peer system are excluded from the overlay network without affecting its connectivity. There are a number of studies for safe node exclusion if the overlay is in a well-defined state initially. Surprisingly, the problem has not been formally studied yet for the case in which the overlay network is in an arbitrary initial state, i.e., we are 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 if necessary. We show that there is no self-stabilizing distributed algorithm for the $\mathcal{FDP}$, even in a synchronous message passing model. To allow a solution, we introduce an oracle called $\mathcal{NIDEC}$ and show that it is sufficient even for the asynchronous message passing model by proposing an algorithm that can solve the $\mathcal{FDP}$ using $\mathcal{NIDEC}$. We also show that a solution to the $\mathcal{FSP}$ does not require an oracle.

1 Introduction

Peer-to-peer systems allow computers to interact and share resources without the need for a central server or centralized authority. This ability to self-organize has made peer-to-peer systems very popular. Since participation in such systems is usually voluntary, the peers may arrive and depart at any time. A peer may even leave the network without notice. Therefore, maintaining a connected overlay network is a challenging task. Many strategies help to alleviate this problem. They include using an overlay network with a high expansion or separating the peers into more reliable super-peers forming an overlay network on behalf of the other peers that just connect to one or more super-peers. While these strategies may work well in practice, rigorous research on when it is safe to leave the network is still in its infancy. The goal of this paper is to lay the foundation for a rigorous treatment of node departures in the context of self-stabilization. In fact, we are the first to provide answers to the question:

Is it possible to design a distributed algorithm that allows any collection of nodes to eventually leave a network from any initial state without losing connectivity?

Self-stabilization makes the above question non-trivial. A self-stabilizing algorithm recovers from an arbitrary initial state. Hence, a self-stabilizing node departure algorithm has to handle the states where the departing node is about to leave and may disconnect the network.

1.1 System Model

We consider a distributed system consisting of a fixed set of processes with fixed identifiers, IDs for short, that are globally ordered. We refer to processes and their identifiers interchangeably. The system is controlled by an algorithm that specifies the variables and actions that are available in each process. In addition to the algorithm-based variables there is a system-based variable called *channel* whose values are sets of messages. The channel message capacity is unbounded, and messages will never get lost. We assume non-FIFO message delivery, fair-message receipt and point-to-point communications (multi-cast and broadcast primitives are not considered). We treat all messages sent to a process p as belonging to a single incoming channel C_p. Each process has a read-only boolean variable called *leaving*. If this variable is **true**, the process is *leaving*; the process is *staying* otherwise.

An *action* has the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. $label$ is a name to differentiate actions. $guard$ either detects the presence of a particular message type in the incoming channel, or it is a predicate over local variables. We call an action whose guard is simply **true** a *timeout* action. $command$ is a sequence of statements that assign new values to process variables or send messages to other processes. Two other possible statements are **exit** and **sleep**. If a process executes **exit** it enters a designated *exit state*. Such a process is *gone*. If a process executes **sleep**, it enters the *sleep state*. Such a process is *asleep*. If a process is neither gone nor asleep, it is called *awake*.

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** in this state and p is awake, or its guard detects the presence of a particular message type in C_p and p is asleep. In the latter case, p becomes awake again, i.e., it leaves its sleep state. The action is *disabled* otherwise. Hence, while a gone process will never wake up again, an asleep process may wake up again when receiving an appropriate message.

A *computation* is an infinite fair sequence of states such that for each state s_i, the next sate 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 two kinds of fairness of computation: weak fairness of action execution and fair message receipt. *Weak fairness* of action execution means that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. Note that unless a process is gone or permanently asleep (i.e., it never wakes up again) at some point, 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, this computation also contains a later state where this message is not present in the channel, i.e., the message is received. Besides these fairness assumptions, we place no bounds on message propagation delay or relative process execution speeds, i.e. we consider fully asynchronous computations.

A *computation suffix* is a sequence of computation states past a particular state of this computation. In other words, the suffix of the computation is obtained by removing the initial state and finitely many subsequent states. Note that a computation suffix is also a computation.

We consider algorithms that do not manipulate the internals of process identifiers. Specifically, an algorithm is *copy-store-send* if the only operations that it executes on process IDs is copying them, storing them in local process memory and sending them in a message. That is, operations on IDs such as addition, radix computation, hashing, etc. are not used. In a copy-store-send algorithm, if a process does not store an ID in its local memory, the process may learn this ID only by receiving it in a message. A copy-store-send algorithm cannot introduce new IDs to the system. It can only operate on the IDs that are already there.

1.2 Problem Statement

An algorithm is *self-stabilizing* if it satisfies the following two properties. *Convergence:* starting from an arbitrary system state, the algorithm is guaranteed to arrive at a legitimate state. *Closure:* starting from a legitimate state the algorithm remains in legitimate states thereafter. A self-stabilizing algorithm is thus able to recover from transient faults regardless of their nature. Moreover, a self-stabilizing algorithm does not have to be initialized as it eventually starts to behave correctly regardless of its initial state.

Before we define a legitimate state for the problems considered in this paper, we restrict the set of initial states to exclude trivially useless parts of that state. For that we first need some notation.

A (directed) *link* is a pair of identifiers (a, b) that is defined as follows: either a message carrying identifier b is in the incoming channel of process a, or process a stores identifier b in its local memory. We say that process a *points* to b or *has a link* to b. When we describe a link, we always state the pointing process first. The links form a directed *process (multi-)graph* PG. A *(weakly) connected component* in some directed graph G is a subgraph of G of maximum size so that for any two nodes u and v in that subgraph there is a (not necessarily) directed path from u to v. Two nodes that are not in the same weakly connected component are *disconnected*. A process p is *hibernating* if p is asleep and C_p is empty and all processes q that have a directed path to p in PG are also asleep and have an empty C_q.

Proposition 1. *For any copy-store-send algorithm and any system state of that algorithm in which process p is hibernating, p is permanently asleep.*

Proof. Let $PG(p)$ be the subgraph containing all processes q with a directed path to p. A process q in $PG(p)$ can only be woken up by a message, but such a message would have to come from a process q' outside of $PG(p)$, which would require a link (q', q) in PG. Since such a link does not exist, the proposition follows. □

Also initially gone processes are useless as they will never perform any computation. Hence, we assume that the initial state only consists of non-gone and non-hibernating processes. We also restrict the initial state to contain only messages that can trigger an action since the others will be ignored. Finally, we do not allow the presence of identifiers 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 will always satisfy 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 our two problems.

Finite Departure Problem ($\mathcal{FDP}$): eventually reach a legitimate state for the case that only the **exit** command is available.

Finite Sleep Problem ($\mathcal{FSP}$): eventually reach a legitimate state for the case that only the **sleep** command is available.

A self-stabilizing solution for these problems has to 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.

1.3 Oracles

Since we will need the use of (specific types of) oracles in order to show the further results, we introduce the notation and the relevant definitions in this section. In the following, a process is called *relevant* if it is neither gone nor hibernating. Otherwise we call it *irrelevant*. 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 we will see later, there is no algorithm within our model that can decide when it is safe for a process p to leave the system. Hence, we need oracles. An *oracle* $\mathcal{O}$ is a predicate that depends on the 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 leave the network, so we restrict our attention to algorithms that *only* allow a leaving process to call **exit** if the oracle is **true** for it. Such an algorithm is also said to *rely* on the oracle. Moreover, we restrict our attention to oracles that *only* depend on the current process graph of the relevant processes and the calling process, i.e., the oracles are of the form $\mathcal{O}$: $\mathcal{PG} \times \mathcal{P} \to \{$**true**,**false**$\}$ where $\mathcal{PG}$ is the set of process graphs and $\mathcal{P}$ is the set of processes. For example, we may define oracle $\mathcal{EXIT}$ to be **true** for some process u if u can safely leave the system. Certainly, this oracle needs global information and is therefore expensive to implement. So we are focusing on more local oracles. To define these oracles we need to introduce additional notation.

A link (v, w) in PG with $v \neq w$ is *relevant* for some process u if $u = w$ and v is not gone, or it is implied by a message in C_u carrying the ID of w (i.e., $u = v$). Otherwise, the link is *irrelevant* for u. Note that the links implied by process IDs stored in u are also irrelevant (meaning that u does not have to learn about them since it already knows them).

An oracle $\mathcal{O}$ is *id-sensitive* for some process u if its output depends on links relevant for u. An oracle $\mathcal{O}$ is *strictly id-sensitive* if for every process u the oracle's output *only* depends on the links relevant for u. Hence, the oracle ignores irrelevant links. Note that an action that changes the system state without affecting relevant links also does not affect the output of a strictly id-sensitive oracle. Naturally, a strictly id-sensitive oracle is also (regularly) id-sensitive. An oracle is *id-insensitive* if it is not id-sensitive. That

is, the output of an id-insensitive oracle does not depend on the links relevant for the process.

We define the following strictly id-sensitive oracles. Oracle $\mathcal{NID}$ (no identifiers) evaluates to **true** if the system does not contain an identifier of u in v or C_v for some relevant process $v \neq u$. Oracle $\mathcal{EC}$ (empty channel) evaluates to **true** for a particular process u if the incoming channel of u is empty. Oracle $\mathcal{NIDEC}$ is a conjunction of $\mathcal{NID}$ and $\mathcal{EC}$. That is, $\mathcal{NIDEC}$ evaluates to **true** if both $\mathcal{NID}$ and $\mathcal{EC}$ evaluate to **true**. Note that $\mathcal{NIDEC}$ is less powerful than $\mathcal{NID}$ and $\mathcal{EC}$ used jointly since the algorithm using $\mathcal{NIDEC}$ is not able to differentiate between the conditions separately reported by $\mathcal{NID}$ and $\mathcal{EC}$. Oracle $\mathcal{ONESID}$ evaluates to **true** for a process u if u shares links with at most one relevant process.

Within a class of oracles $\mathcal{C}$, an oracle $\mathcal{O}$ is *necessary* for the $\mathcal{FDP}$ if for every algorithm $\mathcal{A}$ relying on an oracle $\mathcal{O}' \in \mathcal{C}$ with $\mathcal{O}'(s,u) =$**true** while $\mathcal{O}(s,u) =$**false** for some system state s and process u, $\mathcal{A}$ cannot be a self-stabilizing solution to the $\mathcal{FDP}$.

An Oracle $\mathcal{O}$ is *semi-persistent* if the actions of other processes cannot invalidate it. That is, once a semi-persistent oracle is **true** for process u, it remains **true** regardless of actions of processes other than u. Out of the oracles we defined, $\mathcal{NID}$ and $\mathcal{NIDEC}$ are semi-persistent while $\mathcal{EC}$, $\mathcal{ONESID}$ and $\mathcal{EXIT}$ are not.

1.4 Our Contribution

First, we show that without an id-sensitive oracle there is no self-stabilizing solution for the $\mathcal{FDP}$ within our model. Afterwards we show that among all id-sensitive oracles $\mathcal{ONESID}$ is necessary to solve the $\mathcal{FDP}$. On the other hand, we prove that $\mathcal{NIDEC}$ is sufficient to solve the $\mathcal{FDP}$ by providing a self-stabilizing algorithm for the $\mathcal{FDP}$ relying on $\mathcal{NIDEC}$.

Problem $\mathcal{FSP}$, in contrast to the $\mathcal{FDP}$, does not require the processes to irrevocably exit the system. This will allow us to design a self-stabilizing algorithm for the $\mathcal{FSP}$ that does not need any oracle.

1.5 Related Work

The difficulty of the Finite Departure Problem resembles that of fault-tolerant agreement in distributed systems. Fault-tolerant agreement has been studied in the context of the famous Consensus Problem. It is shown [**?**] that the problem is not solvable in an asynchronous system even if only a single process may crash. This implies that no self-stabilizing solution for the Consensus Problem either. This impossibility is circumvented through the use of specialized oracles known as failure detectors [**?**].

Due to the popularity of peer-to-peer networks, the research literature on this subject is extensive [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. While departure algorithms have been proposed in these papers, none are self-stabilizing. In fact, a rigorous treatment of when it is safe to leave the system has not been yet attempted. Cases in which the rate of churn is limited have already been considered [**?**,**?**,**?**]. Kuhn et al [**?**,**?**,**?**] handle this limitation by organizing the nodes into cliques of $\Theta(\log n)$ size that they call super-nodes. Hayes et al. [**?**] 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. [**?**] propose a network maintenance algorithm called DASH to repair the network resulting from an arbitrary number of deletions. Limited churn has also been studied in the context of adversarial nodes [**?**,**?**,**?**]. While there is no work on self-stabilizing node departures, several self-stabilizing peer-to-peer algorithms are proposed [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. The studied topologies range from a simple line and ring [**?**,**?**], to skip lists and skip graphs [**?**,**?**], expanders [**?**], the Delaunay graph [**?**], hypertree [**?**], and Chord [**?**]. Also a universal algorithm for topological self-stabilization is known [**?**]. However, none of these provide any means to exclude nodes that want to leave the network.

2 Basic Properties of the $\mathcal{FDP}$

In this section we show that the $\mathcal{FDP}$ requires an id-sensitive oracle. Moreover, if only strictly id-sensitive oracles are considered, then $\mathcal{ONESID}$ is necessary. The below proposition is a restatement of the results obtained in [**?**,**?**]. Intuitively it says that once disconnected, the system may not be able to reconnect again.

Proposition 2. *[?,?] If a computation of a copy-store-send algorithm starts in a state where two processes u and v are disconnected in PG, u and v remain disconnected in PG in every state of this computation.*

Theorem 1. *Any self-stabilizing solution to the $\mathcal{FDP}$ has to rely on an id-sensitive oracle.*

Proof. Assume that algorithm $\mathcal{A}$ is a self-stabilizing solution to the $\mathcal{FDP}$ that relies on an id-insensitive oracle $\mathcal{O}$. We consider following counter-example. Consider a system of at least three processes. The computation of $\mathcal{A}$ starts in a state where all processes but one, process v, are weakly connected. Hence, by Proposition 2, v remains disconnected from the system for the rest of the computation. Among the connected processes, u is leaving. Since $\mathcal{A}$ is a solution to the $\mathcal{FDP}$, the computation will eventually reach a state s_1 in which u calls **exit** in some action A enabled in s_1. See Figure 1 for an illustration.

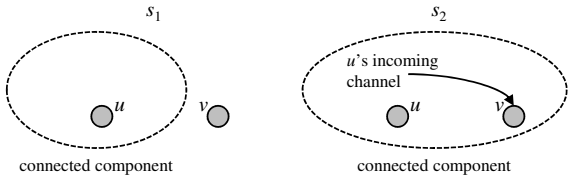

Fig. 1. Illustration for the proof of Theorem 1.

We take s_1 and construct another state s_2 where there is a message carrying the ID of v in the incoming channel of process u. In s_2, all processes of the system are weakly connected. Observe that the process graphs PG_1 for state s_1 and PG_2 for state s_2 differ only by the new, relevant link (u, v). Since $\mathcal{O}$ is id-insensitive, both the state of u and

the output of $\mathcal{O}$ for u are the same for s_1 and s_2. Hence, action A is also enabled in u, and it may execute in the same way in s_2 as in s_1, which implies that u may call **exit**. This would disconnect v from the rest of the system. By Proposition 2, v remains disconnected from the system for the rest of the computation.

Hence, contrary to our initial assumption, $\mathcal{A}$ is not a self-stabilizing solution to the $\mathcal{FDP}$. A similar argument applies to the case in which process v or C_v holds an identifier of u. □

Theorem 1 immediately implies the following corollary.

Corollary 1. *A self-stabilizing solution to the $\mathcal{FDP}$ is impossible without an oracle.*

Interestingly, the impossibility even holds in a synchronous communication model. Consider the model in which each round consists of two stages: in stage 1, every process receives all messages from the previous round, and in stage 2, every process executes any number of its enabled actions. Let us transform the state s_1 in the proof of Theorem 1 into a state s_2 in which v has a link to u. If this is the state of the initial round, u cannot receive a message from v in that round, since there was no prior round, so u still executes the **exit** statement. Hence, the system gets disconnected. We now address the strict id-sensitivity property of oracles.

Lemma 1. *If a self-stabilizing solution to the $\mathcal{FDP}$ relies on a strictly id-sensitive oracle, then this oracle evaluates to **true** only if a process has relevant links with at most one relevant process.*

Proof. Assume there exists an algorithm $\mathcal{A}$ that is a self-stabilizing solution to the $\mathcal{FDP}$ which uses a strictly id-sensitive oracle $\mathcal{O}$ such that there exists a state s_1 where the oracle evaluates to **true** for some leaving process u while it shares relevant links with at least two staying processes v and w. That is, either u has an identifier of v or w in its incoming channel or u's identifier is in the memory of v or w or their respective incoming channels. We construct state s_2 by removing all links from w except for the links to u. Since $\mathcal{O}$ is strictly id-sensitive, this does not change the output of $\mathcal{O}$. Notice that in s_2, process w is disconnected from the system except for the links to u.

Let us now consider a computation σ of $\mathcal{A}$ where u is leaving. Since $\mathcal{A}$ is a solution to the $\mathcal{FDP}$, u should eventually reach a state s_3 in σ in which it executes the **exit** statement in some enabled action A. Since A relies on $\mathcal{O}$, $\mathcal{O}$ must be true in this case.

We construct a system state s_4 where the state of u is the same as in s_3 while the state of the rest of the system is the same as in s_2. Since this does not change the links relevant for u compared to s_2, this does not change the output of $\mathcal{O}$ compared to s_2. On the other hand, the local state of u and the output of $\mathcal{O}$ for u is the same in s_4 as in s_3. Hence, action A must be enabled in s_4, and it may execute in the same way in s_4 as in s_3, which implies that u may call **exit**. This, however, would disconnect process w from the rest of the staying processes. According to Proposition 2, w remains disconnected from the system for the rest of the computation. Thus, contrary to the initial assumption, $\mathcal{A}$ is not a self-stabilizing solution to the $\mathcal{FDP}$. □

Lemma 1 leads to the following theorem.

Theorem 2. *Among all strictly id-sensitive oracles, the oracle $\mathcal{ONESID}$ is necessary to obtain a self-stabilizing solution to the $\mathcal{FDP}$.*

We conjecture that $\mathcal{ONESID}$ is also sufficient to solve the $\mathcal{FDP}$ but it is not semi-persistent. Semi-persistent oracles have the nice property that the action atomicity may be relaxed without affecting the result of the computation. It is known (cf., for example, [**?**]) that for oracle-free message-passing algorithms, for every low-atomicity computation (i.e., atomicity of action execution is only required within a process) there is an equivalent high-atomicity computation (i.e., only one action may be executed in the entire system at a time). In general, for a program with oracles this may no longer be true as the oracle may change its value in a low-atomicity computation due to the actions of other processes. This can be particularly critical in an algorithm for the $\mathcal{FDP}$, where a process may think that it is safe to leave the network because its oracle evaluated to **true** but at the time when it calls **exit** the oracle may be **false** again. However, a semi-persistent oracle cannot be affected in that way. Actions of other processes can only change its value from **false** to **true**. Hence the following proposition.

Proposition 3. *If an algorithm in an asynchronous message-passing system, whose successful performance relies only on the* **true***-value of oracles, uses semi-persistent oracles only, then for every low-atomicity computation, there is an equivalent high-atomicity computation.*

Since low-atomicity computation is what happens in practice, it is therefore preferable to find a solution relying on a semi-persistent oracle like $\mathcal{NIDEC}$.

3 Solution for the $\mathcal{FDP}$

In this section we present a self-stabilizing algorithm called $\mathcal{SDA}$ that solves the Finite Departure Problem with the help of $\mathcal{NIDEC}$. We focus on the case that PG consists of a single connected component. However, the results transfer to PG being split up into multiple connected components. The algorithm is shown in Figure 2.

For ease of exposition, we say that identifier q is to the *right* of identifier p if $q > p$ and to the *left* of p if $q < p$. In algorithm $\mathcal{SDA}$, to maintain connectivity, each process p contains variables $left$ and $right$ that store process IDs that are respectively less than and greater p. If $left$ or $right$ does not contain an identifier, it contains $-\infty$ or $+\infty$ respectively. To ensure a safe process departure, $\mathcal{SDA}$ uses $\mathcal{NIDEC}$ oracle.

Algorithm $\mathcal{SDA}$ uses two message types: *intro* and *reverse*. Message *intro* carries a single process ID and serves as a way to introduce processes to one another. Message *reverse* does not carry an ID. Instead, this message carries a boolean value denoted as **revright** or **revleft**. This message is a request for the recipient process to remove the respective left or right ID from its memory and send its own ID back.

We now describe the actions of the algorithm. Some of the actions contain message sending statements involving IDs stored in the $left$ and $right$ variables. If the variable contains $\pm\infty$, the sending action is skipped. To simplify the presentation of the algorithm, this is omitted in Figure 2.

The algorithm has three actions. The first action is *timeout*, which is a periodically triggered action, that has as a general goal to introduce the node to its neighbors. If the process is staying, it sends its ID to its right and left neighbor. If the process is leaving, it sends messages to its neighbors requesting them to remove its ID from their memory. If the process is leaving and the $\mathcal{NIDEC}$ oracle signals that it is safe to leave, the process introduces its neighbors to each other to preserve system connectivity and then exits by executing the **exit** statement. The second action is *introduce*. It receives and handles *intro* messages received by a node. The operation of this action depends on the relation between the ID carried by the message and the IDs stored in $left$ and $right$. The process either forwards *intro*(id) to its left or right neighbor to handle it; or, if id happens to be closer to p than $left$ or $right$, then p replaces the respective neighbor and instead introduces the old neighbor identifier to id. The third action, *reverse*, handles the neighbors' requests to leave, i.e. the *rev* messages received by a node. If p receives this message, it sets the respective variable to $+\infty$ or to $-\infty$ and, to preserve system connectivity, sends its own ID to this process. To break symmetry, if p itself is leaving, it ignores the request from its left neighbor.

3.1 Correctness proof

For $\mathcal{SDA}$ to be a self-stabilizing solution to the $\mathcal{FDP}$ it remains to show two properties. *Safety*: $\mathcal{SDA}$ never disconnects any relevant processes. *Liveliness*: All leaving processes eventually exit the system.

Lemma 2. *If a computation of $\mathcal{SDA}$ starts in a state where the graph PG of the non-gone processes is weakly connected, the graph PG of the non-gone processes remains weakly connected in every state of this computation.*

Proof. We demonstrate the correctness of the lemma by showing that none of the actions of $\mathcal{SDA}$ disconnects PG. Action *timeout* only adds links to PG if $\mathcal{NIDEC}$ is **false** and cannot disconnect it in this case. If $\mathcal{NIDEC}$ is **true**, PG does not contain links pointing to p and the only outgoing links are $(p, left)$ and $(p, right)$. If p is connected to the rest of PG by at most one link (i.e., $left$ or $right$ does not store an ID), the departure does not disconnect PG. If both $left$ and $right$ store an ID, the leaving of p does not disconnect PG because p sends $intro(left)$ to $right$ and $intro(right)$ to $left$ and thereby preserves weak connectivity between the remaining processes.

Let us consider *introduce*. If the received id is the same as p or as $left$ or $right$, the message is ignored. However, this does not disconnect PG. Let us consider the case of $id < p$. The case of $id > p$ is similar. There are two subcases to address. In case $id < left$, p sends $intro(id)$ to $left$. That is, in PG, the link (p, id) is replaced with $(left, id)$. Since p stores the recipient identifier in $left$, i.e. PG has a link $(id, left)$, the graph connectivity is preserved. The other case is $left < id < p$. In this case, p replaces $left$ with id and forwards the old value to id. That is, the links (p, id) and $(p, left)$ are replaced by (p, id) and $(id, left)$. This replacement preserves PG connectivity.

The rev message received by a *reverse* action may force p to set either $right$ or $left$ to infinity thus removing a link from PG. Let us consider the case of $right$ being set to $+\infty$, the other case is similar. This operation removes $(p, right)$ from PG. However,

constant p : process identifier
variables $leaving$: $boolean$, read only, **true** when p wants to leave
$left$: process ID less than p, $-\infty$ if undefined
$right$: process ID greater than p, $+\infty$ if undefined
$p.C$: channel of incoming messages of process p
messages $intro(id)$, introduces process identifier
$rev(direction)$, requests recipient to reverse edge
$direction$ is **revleft** or **revright**

actions

timeout: **true** $\longrightarrow$
  if not $leaving$ **then**
    send $intro(p)$ **to** $left$,
    send $intro(p)$ **to** $right$
  else // leaving
    send rev(**revleft**) **to** $right$
    send rev(**revright**) **to** $left$
* **if** $\mathcal{NIDEC}$ **then**
    if $left \neq -\infty$ **and** $right \neq +\infty$ **then**
      send $intro(left)$ **to** $right$
      send $intro(right)$ **to** $left$
** **exit**

introduce: $intro \in p.C \longrightarrow$
  receive $intro(id)$
  if $id < left$ **then**
    send $intro(id)$ **to** $left$
  if $left < id < p$ **then**
    send $intro(left)$ **to** id
    $left := id$
  if $p < id < right$ **then**
    send $intro(right)$ **to** id
    $right := id$
  if $right < id$ **then**
    send $intro(id)$ **to** $right$

reverse: $rev \in p.C \longrightarrow$
  receive $rev(direction)$
  if $direction =$ **revleft then**
    if not $leaving$ **then**
      send $intro(p)$ **to** $left$
      $left := -\infty$
  else // $direction$ is **revright**
    send $intro(p)$ **to** $right$
    $right := +\infty$

Fig. 2. Algorithm $\mathcal{SDA}$ for process p. $\mathcal{SSA}$ is obtained by omitting the line marked with * (i.e. the use of $\mathcal{NIDEC}$) and replacing the line indicated with ** (i.e. the **exit** command) by the **sleep** command.

reverse sends a message $intro(p)$ to $right$. That is, it replaces the link $(p, right)$ with $(right, p)$, so weak connectivity of PG is preserved. □

The liveliness part of the correctness proof is more involved. Due to the way IDs are handled by $\mathcal{SDA}$, the development of a link can be traced over the course of the computation. Recall that a link (p, q) is associated with an ID of q stored in p or a message in C_p. The actions of $\mathcal{SDA}$ may transform (p, q) into a different link (p', q'). Only the following cases can occur:

1. The *introduce* action stores q in $left$ or $right$ or drops the q since it is equal to p, $left$ or $right$. In both cases, we stay with the link (p, q).
2. The *introduce* action may delegate the ID of q to some process p': then (p, q) changes to (p', q). Note that whenever this happens, $p' \in [p, q]$.
3. The *reverse* action reverses the link (p, q) to (q, p). Note that whenever this happens, p is staying or p is leaving and $p < q$.

The changes (i.e., cases 2 and 3) to a link (p, q) over time form a sequence of links $(p, q) = (p_0, q_0), (p_1, q_1), (p_2, q_2), \ldots$ that we call the *trace* of (p, q). The cases listed above imply the following lemma.

Lemma 3. *(Monotonicity) For every (p', q') in the trace of (p, q), $p', q' \in [p, q]$.*

This and the fact that we have a finite number of processes may seem to imply that every trace is finite, but for now we cannot exclude the case that a link is reversed infinitely often between two processes. It will only be implied later when we know that eventually all leaving processes will exit the system.

Consider an arbitrary fixed computation of $\mathcal{SDA}$. A link that does not change any more in it is called *stable*. A *steady chain* of processes $x_k, \ldots, x_0$ is a sequence of leaving and not yet gone processes of increasing order with stable links (x_i, x_{i-1}). A steady chain is *maximal* if it cannot be extended to the left or right. See Figure 3 for an illustration. Note that at every state of the computation, every leaving process is part of at least one maximal steady chain (which might just be a chain consisting of itself). Also, the following holds:

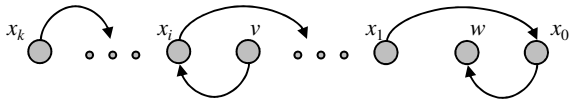

Fig. 3. Illustration of a steady chain.

Lemma 4. *A maximal steady chain can only change in two ways: either (1) process x_k exits the system, or (2) the chain is extended to the left or right due to new stable edges.*

Since the number of processes is finite, this means that eventually a maximal steady chain is stable, i.e., it does not change any more for the rest of the computation. We call this a *stable chain*. Now, we can prove the following lemma.

Lemma 5. *In every computation of $\mathcal{SDA}$, the only stable chain is the empty chain.*

Proof. Consider on the contrary that we have a non-empty stable chain $x_k, \ldots, x_0$. Our goal will be to prove that eventually there is no incoming link from non-gone processes in PG to x_k.

First, suppose there is an incoming link (p, x_k) with $p < x_k$. If there is a reversal in the trace of that link, then we end up with a link (x_k, p') with $p \leq p' < x_k$. If this causes x_k to delegate p' away, then due to the Monotonicity Lemma that link will never include x_k again. Otherwise, x_k stores p' in *left*, and since a leaving process never reverses its link to *left*, x_k either eventually delegates p' away, which will mean that the link never includes x_k again, or x_k holds on to that link, which means that (x_k, p') can never become an incoming link to x_k again. So suppose that there is no reversal in the trace of (p, x_k). Then its trace is finite, which means that eventually it becomes a stable link (p', x_k). We will argue via two cases that this cannot happen.

(1a) If p' is staying, then p' will eventually introduce itself to x_k. This will create a new edge (x_k, p') in PG. If this link is not delegated by x_k, x_k will eventually ask p' to reverse its link to x_k, which it will do, but that would contradict the assumption that (p', x_k) is stable. If x_k delegates (x_k, p'), then we keep track of that link until we get to a link (x, p') that gets reversed or is stable. In the former case, p' would delegate x_k to x, and in the latter case, p' would also either delegate x_k to x or reverse (p', x_k), depending on whether x is staying or leaving. Hence, in any case, (p', x_k) would not be stable, a contradiction.

(1b) If p' is leaving, then we distinguish between two cases. If x_k is not aware of p', then the chain can be extended to p' because (p', x_k) is stable, which contradicts our assumption to have a stable chain. If x_k is aware of p', then x_k will eventually ask p' to reverse its right edge, which will cause the link (p', x_k) to be reversed which again contradicts our assumption that (p', x_k) is stable.

Next, consider the case that there is an incoming link (p, x_k) with $p > x_k$. If there is a reversal in the trace of that link, we end up with a link (x_k, p') with $x_k < p' \leq p$. If this causes x_k to delegate p' away, then due to the Monotonicity Lemma the trace that link will never include x_k again. Otherwise, it must hold that $x_k < p' \leq x_{k-1}$. If $p' = x_{k-1}$, the edge would become stable, and otherwise, x_k would delegate x_{k-1} to p', which would contradict the assumption that (x_k, x_{k-1}) is stable. So in any case this link will eventually not be an incoming link to x_k any more. Thus, suppose that there is no reversal in the trace of (p, x_k). Then its trace is finite, which means that eventually it becomes a stable link (p', x_k). We will again argue via two cases that this cannot happen.

(2a) If p' is staying, then p' will eventually introduce itself of x_k. If $x_k < p' < x_{k-1}$, then x_k would delegate x_{k-1} away, contradicting our assumption that (x_k, x_{k-1}) is stable. If $p' > x_{k-1}$, then similar arguments as for case (1a) above will show that (p', x_k) is not stable, also contradicting our assumption.

(2b) If p' is leaving, p' will eventually ask x_k to reverse its right edge, which it will do, contradicting our assumption that (x_k, x_{k-1}) is stable.

Moreover, x_k will never create an incoming edge to itself since it would only do that when asked to reverse (x_k, x_{k-1}), but since (x_k, x_{k-1}) is stable, this will not happen. Hence, eventually x_k has no incoming edge. This implies that eventually x_k has no

more messages to process, so $\mathcal{NIDEC}$ will eventually be **true**. Therefore, x_k can exit the system, which contradicts our assumption that the chain is stable. □

Lemmas 2 and 5 lead to the following theorem.

Theorem 3. *Algorithm $\mathcal{SDA}$ and the $\mathcal{NIDEC}$ oracle provide a self-stabilizing solution to the $\mathcal{FDP}$.*

4 Finite Sleep Problem

We can overcome the use of oracles by changing to the Finite Sleep Problem. Algorithm $\mathcal{SSA}$, which solves this problem, is almost identical to $\mathcal{SDA}$ shown in Figure 2. The only differences are that no oracle is checked and that the **sleep** command is used instead of **exit**.

For the correctness proof of $\mathcal{SSA}$, we show that the safety and liveliness properties hold. To satisfy the page constraints, we moved the proofs in this section to the appendix. We first define and prove the conditions that must prevail for a process to remain permanently asleep.

Lemma 6. *In the $\mathcal{SSA}$ algorithm, a process p is permanently asleep if and only if p is hibernating.*

The lemma implies that given that our initial state satisfies the conditions in Section 1.2, no process will initially be permanently asleep. Additionally, the following lemma holds.

Lemma 7. *If a computation of $\mathcal{SDA}$ starts in a state where the graph PG of the non-hibernating processes is weakly connected, the graph PG of the non-hibernating processes remains weakly connected in every state of this computation.*

Lemmas 6 and 7 imply safety. So it remains to prove liveliness. Notice that if $\mathcal{NIDEC}$ is true (as a condition, not oracle) for a process p, then p would hibernate in $\mathcal{SSA}$ after calling *timeout*. Hence, it follows together with the proof of Lemma 7 that a process becomes hibernating in $\mathcal{SSA}$ if and only if $\mathcal{NIDEC}$ is true for it. On the other hand, a non-hibernating process is not permanently asleep. Therefore, the liveliness proof is identical to the liveliness proof of $\mathcal{SDA}$, which implies the following theorem.

Theorem 4. *$\mathcal{SSA}$ provides a self-stabilizing solution to the $\mathcal{FSP}$.*

5 Conclusion

In this paper, we showed that an id-sensitive oracle is required for a self-stabilizing solution to the Finite Departure Problem. We proved that among strictly id-sensitive oracles, $\mathcal{ONESID}$ is necessary for a solution to the problem. We showed that a more restrictive oracle $\mathcal{NIDEC}$ is sufficient by presenting an algorithm that solves the Finite

Departure Problem using $\mathcal{NIDEC}$. We suspect that $\mathcal{ONESID}$ is also sufficient but it remains to be proven.

Observe that the $\mathcal{SSA}$ algorithm, besides solving the $\mathcal{FDP}$, also organizes the staying processes in a sorted list. It would be interesting to consider building more complex and robust topologies such as the skip-list or skip-graph [**?**,**?**,**?**].

It would also be interesting to study the power of individual components of $\mathcal{NIDEC}$: $\mathcal{NID}$ and $\mathcal{EC}$. Specifically, we would like to determine the extent of the states from which the algorithm using only one of the components may recover.

A Proofs in Section 4

A.1 Proof of Lemma 6

The backwards direction (if p is hibernating then p is permanently asleep) directly follows from Proposition 1. So it remains to prove the other direction.

Suppose that there is a process q that has a directed path along the processes $q_0 = q, q_1, \ldots, q_\ell = p$ to p and q is either not asleep, or C_q is non-empty. Without loss of generality, we may assume that for all other processes q_i with $i \geq 1$, C_q is empty (otherwise set q as the process with largest i with non-empty C_{q_i}). Hence, for all $i \geq 1$, q_{i+1} is initially stored in q_i. Since q is either awake and knows q_1, or C_q contains a message with q_1, and q can only fall asleep in *timeout*, q is guaranteed to eventually process the link (q, q_1) by either calling the *timeout* (which may contact q_1), *introduce* (which may contact or delegate q_1), or *reverse* action (which may contact q_1). If q_1 gets delegated, the receiving process is also guaranteed to process q_1. We continue the trace of (q, q_1) in this case (which causes the involved starting points to be woken up) until we reach a process q' where q_1 is not delegated any more. This must eventually happen since the number of processes is finite. Hence, q_1 is eventually contacted, which will wake up q_1. Since q_1 initially stores q_2, q_1 is therefore also guaranteed to eventually process the link (q_1, q_2). The same arguments as for q_1 then guarantee that also q_2 eventually processes the link (q_2, q_3). Hence, by induction, eventually p is woken up, which completes the proof. □

A.2 Proof of Lemma 7

We know from Lemma 2 that none of the actions of $\mathcal{SSA}$ disconnects the graph PG of the non-exited processes. Thus, as long as no process falls asleep after an action (which can only happen if a leaving process calls *timeout*), the lemma holds. Suppose now that a leaving process p calls *timeout*. Our first goal is to show that no other process can become hibernating in this case. Consider any process $q \neq p$ that is non-hibernating and that has a directed path from p. We distinguish between two cases.

(1) If the directed path from p to q leads through a process q' stored in a message in C_p, then p cannot become hibernating and therefore q cannot become hibernating as well.

(2) If the directed path from p to q leads though $left$ or $right$ of p, then q cannot become hibernating because p will contact $left$ and $right$ in *timeout*.

Hence, if p does not become hibernating after *timeout*, weak connectivity is preserved. It remains to consider the case that p becomes hibernating. In this case, C_p is empty. Also, there cannot be a path from a non-hibernating node q to p. Hence, $\mathcal{NIDEC}$ would be **true** for p (if evaluated by it). Since we know from Lemma 2 that in this case p may even exit the system without causing disconnectivity, we can also allow p to hibernate without risking disconnectivity for the non-hibernating processes. □