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

Andreas Koutsopoulos
University of Paderborn, Germany
koutsopo@mail.upb.de

Christian Scheideler
University of Paderborn, Germany
scheideler@.upb.de

Thim Strothmann
University of Paderborn, Germany
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.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.2.2 [**Computer-Communication Networks**]: Network Protocols

Keywords

Distributed Systems; Self-Stabilization; Overlay Networks; Process Departures

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

SPAA '15, June 13-15, 2015, Portland, OR, USA
ACM 978-1-4503-3588-1/15/06.
http://dx.doi.org/10.1145/2755573.2755614.

1. INTRODUCTION

Any distributed system must be based on some overlay network which 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. If the overlay is in a well-defined state, scenarios in which the rate of node departures and arrivals is limited have already been studied [1, 5, 7]. However, due to permanent or transient failures a distributed system may rarely be in an ideal state, so it would be desirable to find *self-stabilizing* protocols for the exclusion of 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. [4] 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 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 disconnectivity 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.

2. MODEL & PROBLEM STATEMENT

We consider a distributed system consisting of a fixed set of processes in which each process has a unique reference (like its IP address). 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 it contains all incoming messages to u. Its message capacity is unbounded and messages never get lost. Messages are delivered according to the standard asynchronous message passing model, so every message is eventually delivered, but the delivery might happen in any order. A process u has a variable $mode(u) \in \{\text{leaving}, \text{staying}\}$ 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 special commands that are important for the study of our finite departure problem: **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*. 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 are also asleep and have an empty $q.Ch$.

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, we only consider initial states in which all processes are relevant for the self-stabilization. We also restrict the initial state to contain only 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.

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 also 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 does not disconnect any relevant processes. As shown in [4], 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.

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 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.

3. OUR RESULTS

Our main result is a self-stabilizing local-control protocol that can solve the $\mathcal{FDP}$ when relying on the $\mathcal{SINGLE}$ oracle. The only interfaces that it needs to an underlying communication layer is that it can send a message to a process identified by some reference and that it can check 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. This flexibility is a major enhancement, since the previously known protocols to solve the $\mathcal{FDP}$ [4] requires that there is a fixed order on the processes. Also, the protocols in [4] were designed with a fixed topology in mind while this is not the case for our new protocol, which allows for an easy integration into existing overlay maintenance protocols, as we can also demonstrate. 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. 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.

We can show some fundamental results about these primitives, i.e. they are safe (they always preserve weak connectivity) and universal (*in principle* it is possible to get from any weakly connected graph to any other weakly connected

graph). Furthermore, these four primitives are not only sufficient for universality but also necessary, i.e., by removing one primitive, universality is lost. With $\mathcal{P}$ we 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., [2, 3, 6, 8]) are in $\mathcal{P}$(as otherwise they would risk disconnection of their topology).

3.1 Process Departures

Our new self-stabilizing protocol that solves the $\mathcal{FDP}$ only needs to compare references for equality as needed for the four primitives. Since the protocol is self-stabilizing it is possible that information is *invalid* in an initial state, i.e., a process u assumes that another process v is leaving, even though v is staying.

Our solution makes use of a special variable called *anchor* which will only be used by the leaving nodes, so in a legitimate state, the *anchor* of a staying process is empty. The anchor is a reference of a process which a leaving process v assumes to be staying. Therefore, each time v gets a message from a third process w, v forwards w to its anchor by using the delegation primitive in the hope of eliminating all references to itself. In a nutshell, our protocol works as follows. Each process has a periodically executed *timeout* action. In case a process u is leaving, it introduces itself to its anchor in the *timeout* action (in order to verify it has a staying anchor). If it is staying, it introduces itself to all neighbors (to make other processes aware of it). This so-called *self-introduction*, which is is a special case of the introduction primitive, ensures that invalid information vanishes. Additionally, a leaving processes consults $\mathcal{SINGLE}$ in *timeout*, and if it evaluates to true, the process is safe to perform **exit**. Moreover, processes have to react to incoming introduction and delegation messages. A staying process saves references of other staying processes and reverses references to leaving processes. A leaving process will never save any references, except if it needs an anchor. In case the leaving process has an anchor it forwards all incoming messages to that anchor. Otherwise it makes use of the reversal primitive, i.e., it sends its own reference to the process whose reference was contained in the received message. 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.

It turns out that safety is easy to prove, since we only use the primitives to realize our protocol. In order to show liveness we first prove that all invalid information that is present in an initial state of the system eventually vanishes by a potential argument. Once all information is valid, we can prove Theorem 3.1 by induction, i.e., as long as there is a leaving process that is not gone yet, one leaving process executes **exit** eventually.

THEOREM 3.1. *Every leaving process eventually executes the **exit** command, thereby preserving liveness.*

Our developed protocol can be combined with a large class of distributed overlay protocols, i.e., with most protocols from $\mathcal{P}$ (see Theorem 3.2). In fact a protocol has to fulfill only two additional 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). Second, P has a *postprocess* action, which is able to handle messages that should not be delivered, i.e., if we do not want to deliver a message, *postprocess* integrate the information contained in that message back into the sending process. This is important, since we want to avoid that P spreads information about leaving processes. In order to do so our enhanced protocol makes sure that a message is only delivered once the mode of all processes referenced in that message has been checked. If one single process is leaving, the message is not delivered and *postprocess* reintegrates the information. Note that all self-stabilizing overlay protocols proposed so far can easily be adapted to satisfy these two requirements. Moreover, we can adapt our protocol in order to solve the $\mathcal{FSP}$, so that an oracle is not required anymore. Therefore, our protocol achieves the same feasibility result as the protocol of [4] with lesser assumptions and an enhanced adaptability and flexibility.

THEOREM 3.2. *Let $P \in \mathcal{P}$ be a distributed overlay protocol which solves some distributed problem $\mathcal{DP}$ with the already mentioned requirements. Then there we can construct another protocol P', such that P' eventually solves the $\mathcal{FDP}$. In addition, if P is self-stabilizing, then P' also solves $\mathcal{DP}$.*

We note that our algorithms assume that there is at least one staying process. For the degenerate case in which all processes want to leave the network, we need a more intricate protocol that declares some leaving process as *pseudo-anchors* in order to solve the departure problem.

4. 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] Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list and skip graph. *Theor. Comput. Sci.*, 428:18–35, 2012.

[3] Danny Dolev, Ezra Hoch, and Robbert van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *PODC*, volume 4878, 2007.

[4] Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On stabilizing departures in overlay networks. In *SSS*, pages 48–62, 2014.

[5] 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.

[6] 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.

[7] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Distributed Computing*, 22(4):249–267, 2010.

[8] Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *SSS*, pages 356–370, October 2011.