Stabilizing Finite Churn in Peer-to-Peer 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

July 18, 2013

Techical report: TR-KSU-CS-2013-02
Department of Computer Science
Kent State University

Abstract. We define and study the Finite Leave Problem which abstracts churn in peer-to-peer networks. In this problem, a process that leaves the system should not disconnect it. We address this problem in the asynchronous message passing system model. In this model, problem does not have a self-stabilizing solution. To enable the solution we use oracles. We define $\mathcal{NIDEC}$ oracle and prove it to be necessary to solve the finite leave problem. We then study a particular case of the Finite Leave Problem: Finite Leave with Linearization (topological sort). In this problem, the remaining processes have to sort themselves in the increasing order of their identifiers. We present a self-stabilizing algorithm that solves this problem using $\mathcal{NIDEC}$. With the help of other oracles we extend the solution to handle system disconnections and identifiers that are not present in the system.

1 Introduction

Motivation. A peer-to-peer overlay network is formed by nodes, called peers, storing identifiers of other peers in their memory. The routing is carried out by the underlying network, such as the Internet. So long as such an overlay network is connected, it can provide effective means of decentralized data storage, exchange and communication. Due to their popularity, the research literature on peer-to-peer networks is extensive [1–3, 5, 12, 17, 22–24].

The number of participants in a peer-to-peer network on the Internet may grow to hundreds of thousands and even millions. Moreover, the peer participation in the network may be brief and transitory [25]. Hence handling continuous node departures and arrivals, called churn, is fundamental for peer-to-peer system design. In such large

* The paper is eligible for best student paper award.

scale systems, churn is compounded by faults and misconfigurations of various and often unexpected kinds.

Despite the importance of the subject, few studies systematically address robust churn handling. This is in part due to the complexity of the problem. If a peer leaves, the messages to this peer are lost and the references to it become invalid. If processing power of the peers and speed of message propagation between them varies, it is difficult to design a peer-to-peer algorithm which allows a peer to leave the system without disconnecting it.

In this paper, we study churn in the context of self-stabilization and consider churn-tolerant linearization [20] which is a fundamental task for peer-to-peer system construction. We address it in the asynchronous message passing system model.

Tools. Self-stabilization is a holistic approach to fault tolerance: a self-stabilizing algorithm is designed to recover from an arbitrary initial state. Thus, regardless of the nature and extent of the fault or misconfiguration, the system is guaranteed to return to correct operation once the influence of the fault stops. A number of self-stabilizing peer-to-peer algorithms are proposed [6, 7, 9, 10, 13, 14, 18, 21].

The asynchronous message-passing system is a classic model for exploring the fundamental properties of algorithms. In such a model, there is no bound on message propagation delay or on relative process execution speed. This model is well suited to represent massive peer-to-peer systems on the Internet.

It turns out, it is impossible to design a self-stabilizing program in the asynchronous message-passing system model that solves the Finite Leave Problem. The reason is that in an arbitrary initial state, the leaving process may not be aware of other processes either holding its identifier or sending messages to it. The departure of such a process may disconnect the system. Since the peer-to-peer system is held together by the peers' knowledge of each other, once it is disconnected, it is impossible for the peers to find each other again without outside help.

We circumvent this impossibility through the use of oracles. An oracle is a construct that is itself impossible to implement in an asynchronous system, yet it enables the solution for a particular problem [8]. In effect, an oracle encapsulates the impossible and shows the bounds of the achievable for the algorithm design.

Related work. Kuhn et al [15] address churn with the idea of stable supernodes to be maintained by churning peers. In effect, the redundant peers maintain the stability of the supernodes. Their solution may require a significant number of peers to maintain system stability. Benter et al [4] consider self-stabilizing solution to churn in synchronous systems. There are several linearization studies [20, 18, 11]. In particular, Mohd Nor et al [19] consider peer-to-peer linearization with oracles.

Our contribution. We state the Finite Leave Problem where each process in the system either has to leave the system or has to stay and form a specified topology. In particular, the Finite Leave Linearization Problem requires the processes to sort themselves. We define oracle $\mathcal{NIDEC}$ which returns **true** for a particular process u

if its identifier is not present anywhere in the system and the incoming channel of u is empty. We prove that this oracle is necessary to solve the Finite Leave Problem. We then present an algorithm $\mathcal{SL}$ which uses $\mathcal{NIDEC}$ to solve the Finite Leave Linearization Problem. We observe that $\mathcal{NIDEC}$ is persistent in the sense that once it evaluates to **true** for a particular process, it remains in this state regardless of the actions of the other processes. This enables the programs using $\mathcal{NIDEC}$ to remain correct with low atomicity program actions. We describe how algorithm $\mathcal{SL}$, with the help from other oracles, can be further extended to handle system disconnections and identifiers that are not present in the system. We conclude with oracle implementation details and future research directions.

2 Model and Problem Statement

Peer-to-peer networks. A peer-to-peer overlay network consists of a set of N processes with unique identifiers. We refer to processes and their identifiers interchangeably. Processes may be ordered on the basis of these identifiers. Processes a and b are *consequent*, denoted $\mathbf{cnsq}(a, b)$, if $(\forall c : c \in N : (c < a) \vee (b < c))$. That is, two consequent processes do not have an identifier between them. For the sake of completeness, we assume that $-\infty$ is consequent with the smallest id process in the system. Similarly, the largest id process is consequent with $+\infty$.

Processes communicate by passing messages through channels. A *link* is a pair of identifiers (a, b) 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 *a points* to *b* or *has a link* to *b*. Note that the link is directed. When we define a link we always state the pointing node first. We state node that is pointed to second. The *process connectivity* graph CP is the graph formed by the links of the identifiers stored by the processes. A *channel connectivity* multigraph CC includes both locally stored and message-based links. Self-loop links are not considered. By this definition, CP is a subgraph of CC.

A peer-to-peer network is *linearized* if and only if each process points to its consequent process, i.e. CP forms a bi-directed sorted list. When discussing a linearized network, processes with identifiers greater than p are to the *right* of p, while processes with identifiers smaller than p are to the *left* of p. That is, we consider processes arranged in the increased order of identifiers from left to right.

Communication model. Each program contains a set of variables and actions. A *channel* C is a particular variable type whose values are sets of messages. Channel message capacity is unbounded. Message loss is not considered. The order of message receipts does not have to match transmission order. That is, we assume non-FIFO channels. We treat all messages sent to a particular process as belonging to a single incoming channel.

An *action* has the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. *label* is a name to differentiate actions. *guard* can be of several forms. It can detect the presence of a message in the incoming channel, it can be a predicate over local variables, or it

be just **true**. In the last case, the corresponding action is *timeout*. This action is to be executed periodically by the given process. *command* is a sequence of statements assigning new values to the variables of the process or sending messages to other processes.

Program state is an assignment of a value to every variable of each process and messages to each channel. A program state may be arbitrary. We assume that all process identifiers, either in the channels or in process variables, are present in the system. An action is *enabled* in some state if its guard is **true** in this state. It is *disabled* in this state otherwise. A timeout action is always enabled. We consider programs with timeout actions, hence, in every state there is at least one enabled action.

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 the command of 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. *Fair message receipt* means that if the computation contains a state where there is a message in a channel, 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 programs that do not manipulate the internals of process identifiers. Specifically, a program is *copy-store-send* if the only operations that it does with process identifiers is copying them, storing them in local process memory and sending them in a message. That is, operations on identifiers such as addition, radix computation, hashing, etc. are not used. In a copy-store-send program, if a process does not store an identifier in its local memory, the process may learn this identifier only by receiving it in a message. A copy-store-send program can not introduce new identifiers to the system, it can only operate on the ids that are already there.

Oracles. An *oracle* is a predicate on the global system state to be used in a guard of an action. Some oracles may not be implementable in an asynchronous system. Such oracles enable otherwise impossible solutions.

Since the process that uses the oracle is not supposed to implicitly derive information about the state of the system from its own state, the oracle predicate may not contain the local variables of the process either.

However, potentially, the oracle predicate may mention arbitrary variables of the global system state. The implementation of such oracles may be problematic. An

oracle is *minimalistic* if for every process u that uses it, it only mentions the incoming channel of u and the identifiers of u elsewhere in the system.

We define the following minimalistic oracles. Oracle $\mathcal{NID}$ evaluates to **true** for a particular process u if CC does not contain a link pointing to u. In other words, no other process stores u in its local variables, neither is u present in the messages of the incoming channels of other processes. Oracle $\mathcal{EC}$ 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** when 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 program using $\mathcal{NIDEC}$ is not able to differentiate between the conditions separately reported by $\mathcal{NID}$ and $\mathcal{EC}$.

Finite leave problem statement. Each process has a read-only boolean variable *leaving* whose value is the same throughout the computation. If this variable is **true**, the process is *leaving*; the process is *staying* otherwise. A leaving process may be in a designated *exit* state where it may execute no actions. Once a leaving process moves to the exit state, all the links pointing to this process are removed. That is, the incoming messages to this process are lost and the identifiers are deleted.

Every computation of a solution to the *Finite Leave Problem* ($\mathcal{FL}$) should contain a suffix with the following properties. In every state of the suffix: (i) the process connectivity graph CP of the staying processes is the same and forms a prescribed topology while (ii) each leaving process is in the exit state. For example, the *Finite Leave Linearization Problem* ($\mathcal{FLL}$) requires the staying processes to linearize.

We consider problems that require single-component topologies. That is, the target topology of staying processes should remain weakly connected.

Proposition 1. *[18, 19] If a computation of a copy-store-send program starts in a state where the channel connectivity graph CC is disconnected, the graph is disconnected in every state of this computation.*

Hence, once the departure of a node causes the disconnection of CC, it is impossible to regain connectivity. Thus, a solution to the single-component Finite Leave Problem has to maintain connectivity throughout its every computation. Therefore, we assume that computations of self-stabilizing solutions to the single-component $\mathcal{FL}$ start from states where CC is weakly connected.

For oracle-based solutions to $\mathcal{FL}$, we assume that the oracle evaluates to **true** when it is safe for this process to leave the system. We also define the solution as *normal* if every process may execute the *exit* action leaving the system while it points to at least one other process.

3 Necessary Condition

In this section we show that a self-stabilizing program needs the $\mathcal{NIDEC}$ oracle to solve the Finite Leave Problem. Intuitively, without this oracle, a process may not be able to determine whether its departure disconnects the system.

Lemma 1. *If a normal self-stabilizing solution to the single-component Finite Leave problem is using a minimalistic oracle, then this oracle has to evaluate to* ***true*** *only if the incoming channel of the process using the oracle does not contain process identifiers.*

Proof: Assume there exists a normal algorithm $\mathcal{A}$ that is a self-stabilizing solution to the Finite Leave Problem that uses a minimalistic oracle $\mathcal{O}$ such that it evaluates to **true** for some process u in some system state s_1 even though its channel contains a message with process identifier v.

Let us consider the system where v is not present at all. Since $\mathcal{A}$ is a normal solution to $\mathcal{FL}$, there is a computation of $\mathcal{A}$ where u is leaving while holding at least one identifier w. That is, in this computation, u is executing the *exit* action in some state s_2.

Let us add v back to the system. We construct a system state s_3 as follows. The local state of process u is the same in s_3 and in s_2. The incoming channel contents of process u as well as the links pointing to u are the same in s_3 and in s_1. We link the rest of the system such that, except for the links to and from u, processes v and w belong to two disconnected components.

Let us examine the constructed state s_3. Since the links pointing to u as well as the contents of the channel are the same as in s_1, oracle $\mathcal{O}$ evaluates to **true**. Since the state of process u is the same as in s_2, the *exit* action of algorithm $\mathcal{A}$ taking the process out of the system is enabled. We execute this action and then execute the actions of $\mathcal{A}$ in arbitrary fair manner.

Since, by assumption, $\mathcal{A}$ is a self-stabilizing solution to the single-component $\mathcal{FL}$, this computation has to contain a suffix with a single-component system topology. However, the first action of this computation disconnects the system. By Proposition 1, the system remains disconnected for the rest of the computation. That is, this computation may not contain a suffix with a single-component system topology. This means that, contrary to our initial assumption, $\mathcal{A}$ is not a solution to the single-component $\mathcal{FL}$. The lemma follows. □

Lemma 2. *If a normal self-stabilizing solution to the single-component Finite Leave Problem is using a minimalistic oracle, then this oracle has to evaluate to* ***true*** *only if there are no processes pointing to the process using the oracle.*

Proof: (Outline) The proof proceeds similarly to the proof of Lemma 1. We assume that there is a solution to $\mathcal{FL}$ that uses an oracle which evaluates to **true** for some process u even if there is another process v pointing to u. We then construct a state with the *exit* action is enabled and whose execution disconnects the system which, int turn, invalidates our assumption of the existence of the solution to $\mathcal{FL}$ using this oracle. □

Lemmas 1 and 2 lead to the following theorem.

Theorem 1. *Oracle $\mathcal{NIDEC}$ is necessary to enable a normal self-stabilizing solution to the single-component Finite Leave Problem.*

4 Solution

Description. In this section we present a self-stabilizing algorithm called $\mathcal{SL}$ that solves the Finite Leave Linearization Problem $\mathcal{FLL}$ problem with the help of oracle $\mathcal{NIDEC}$. The algorithm is shown in Figure 1. The operation of the algorithm is as follows. Each process p has a read-only variable *leaving* that is initially set to **true** to indicate whether the process needs to leave the system. Each process maintains variables *right* and *left* that store other process identifiers. These variables store identifiers that are less than and more than p respectively. If such a variable does not hold an identifier, we assume it stores ∞. To ensure correctness of process leaving, the algorithm uses $\mathcal{NIDEC}$ oracle.

Algorithm $\mathcal{SL}$ uses two message types: *intro* and *req*. Message *intro* carries a single process identifier and serves as a way to introduce processes to one another. Message *req* does not carry an identifier. Instead, this message carries a boolean value which we denote as **remright** or **remleft**. This message is a request for the recipient process to remove the respective left or right identifier from its memory. This lack of identifier in *req* allows the leaving process to retrieve its own identifier from the other processes without re-introducing it with each message.

We now describe the actions of the algorithm. Some of the actions contain message sending statements involving identifiers stored in *left* and *right* variables. If the variable contains ∞, the sending action is skipped. To simplify the presentation of the algorithm, this detail is omitted in Figure 1.

The algorithm has four actions. The first action is *timeout*. It is executed periodically. If the process is staying, it sends its identifier to its right and left neighbor. If the process is leaving, it sends messages to the neighbors requesting them to remove its identifier from their local memory. The second action is *introduce*. It receives and handles *intro*. The operation of this action depends on the relation between the identifier *id* carried by the message and identifiers stored in *left* and *right*. The process either forwards *id* to its left or right neighbor to handle; or, if *id* happens to be closer to p than *left* or *right*, p replaces the respective identifier and instead sends the old identifier to *id* to handle.

The third action, *request*, handles the neighbors' requests to leave. If p receives such a request, it sets the respective variable to ∞ and, to preserve system connectivity, sends its own identifier to the leaving process. To break symmetry, if p is leaving itself, it ignores leaving request from its left neighbor.

The last action is *exit*. If the process is leaving and $\mathcal{NIDEC}$ oracle signals that it is safe to leave, then the process mutually introduces its neighbors to preserve system connectivity and then exits.

Correctness proof. Once every leaving process exits, $\mathcal{SL}$ operates exactly as the linearization component of Corona [18] and ensures the system linearization. We summarize this claim in the following proposition. See the Appendix for the proof of this proposition.

constant p // process identifier
variables

$leaving$: $boolean$, read only // application level, **true** when process needs to leave
$left$: process ids less than p, $-\infty$ if undefined
$right$: process ids greater than p, $+\infty$ if undefined
$\mathcal{NIDEC}$: no p in the system and empty incoming channel oracle

messages

$intro(id)$, carries process identifier, confirms connectivity
$req(direction)$, requests recipient to remove neighbor
$direction$ may be either **remleft** or **remright**

actions

timeout: **true** $\longrightarrow$
if not $leaving$ **then**
send $intro(p)$ **to** $left$,
send $intro(p)$ **to** $right$
else // leaving
send req(**remleft**) **to** $right$
send req(**remright**) **to** $left$

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** $right$

request: $req \in p.C \longrightarrow$
receive $req(direction)$
if $direction$ = **remleft then**
if not $leaving$ **then**
send $intro(p)$ **to** $left$
$left := -\infty$
else // $direction$ is **remright**
send $intro(p)$ **to** $right$
$right := +\infty$

exit: $\mathcal{NIDEC}$ **and** $leaving \longrightarrow$
if $left \neq -\infty$ **and** $right \neq +\infty$ **then**
send $intro(left)$ **to** $right$
send $intro(right)$ **to** $left$
exit

Fig. 1. Algorithm $\mathcal{SL}$ for process p.

Proposition 2. *[18] If every processes in a computation of $\mathcal{SL}$ is staying, then the algorithm linearizes the system.*

The proof of correctness of $\mathcal{SL}$ contains two parts: safety and liveness. The safety part demonstrates that the operation of $\mathcal{SL}$ does not disconnect the system and the liveness part shows that all leaving processes exit the system.

Lemma 3. *If a computation of $\mathcal{SL}$ starts in a state where the communication graph CC is connected, the graph remains connected in every state of this computation.*

Proof: We demonstrate correctness of the lemma by showing that none of the actions of $\mathcal{SL}$ disconnect CC. Action *timeout* may only send messages. This action only adds links to CC and cannot disconnect it.

Let us consider action *introduce*. This action receives $intro(id)$ message from the incoming channel of process p and thus removes a link (p, id) from CC. This may potentially disconnect the graph. The operation of *introduce* depends on the value of id. If $id = p$, i.e. the message carries the same identifier as the receiving process, this message forms a self-loop link (p, p). This link is not included in CC and the message receipt does not affect CC. We now consider $p < id$. The case of $id < p$ is similar. If $p < id < right$, *introduce* sets $right = id$. Let variable $right$ hold identifier q before the action execution. This action then removes link (p, q) from CC. However, this action also sends message $intro(q)$ to id. That is, *introduce* replaces link (p, q) with two links (p, id) and (id, q). Thus, q is still reachable from p and the connectivity of CC is preserved. If $id = right$, action *introduce* removes the message and does no further operations. This removes link (p, id) from CC which may potentially disconnect CC. However, since $id = right$, link $(p, right)$ is already present in CC and the graph remains connected after one of the two identical links are removed. If $id > right$, action *introduce* forwards id to $right$ thus replacing link (p, id) with a path $(p, right)$ and $(right, id)$ and preserving connectivity of CC.

Let us consider action *request*. This action receives *req* message. This message does not carry an identifier. Hence, its receipt does not affect CC. However, *request* may force p to set either $right$ or *left* to infinity thus removing a link from CC. Let us consider the case of $right$ being set to $+\infty$, the other case is similar. This operation removes $(p, right)$ from CC. However, *request* sends message $intro(p)$ to $right$. That is, it replaces a link $(p, right)$ with $(right, p)$. In effect, this action changes the direction of the link in CC, which preserves the weak connectivity of the graph.

The last action is *exit*. This action makes the process exit the system thus removing links from CC that contain this process. This action is enabled if $\mathcal{NIDEC}$ is **true**. This means that identifier p is not present elsewhere in the system and the incoming channel of process p is empty. That is, CC does not contain links pointing to p and the only outgoing links are $(p, left)$ and $(p, right)$. Note that if either *left* or $right$ are undefined, then p is connected to the rest of the graph through a single link. Hence, the process' departure does not disconnect it. Now, if both *left* and $right$ are defined, the leaving of p may potentially disconnect them. However, before leaving, p sends $intro(left)$ to $right$ and $intro(right)$ to *left*. This replaces links $(p, left)$ and $(p, right)$

with two links $(left, right)$ and $(right, left)$ preserving the connection between these two processes.

To summarize, none of the actions of $\mathcal{SL}$ disconnect CC. Hence the lemma. □

The liveness part of the correctness proof is somewhat involved. To break symmetry, each leaving process ignores disconnection requests from its left neighbor. Hence, it would appear that the rightmost leaving process should leave first. Yet, this may not be the case. Indeed, process u may have difficulty disconnecting from a left leaving processes v that is pointing to u. Process u may be pointing to some other process w to the left of u. Thus, u is requesting disconnection from w instead of v. Since a leaving process only sends request messages that do not carry identifiers, u may not be aware of v at all.

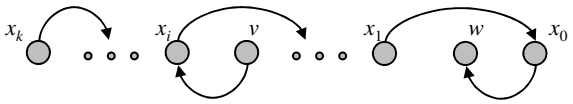

Fig. 2. Illustration of a steady chain for the proofs of Lemmas 4 and 5.

To proceed with the proof, we need to introduce additional notation. A *steady chain* $x_k, \ldots, x_0$ of leaving processes for a particular computation is defined as follows. The first process x_0 is the rightmost leaving process. Each subsequent process x_i points to process x_{i-1} and does not remove this link until x_i leaves. See Figure 2 for illustration. A steady chain is *maximal* if it cannot be further extended to the left. That is, either no leaving process to the left of x_k points to it, or such process removes this link before leaving. Multiple steady chains may be present in a computation. However, a steady chain of at least one process x_0 is present in every computation of the algorithm.

Lemma 4. *In every computation of $\mathcal{SL}$, if processes x_0 and x_k are respectively the first and last in a maximal steady chain, then eventually staying processes to the right of x_0 stop pointing to x_0 and staying processes to the left of x_k stop pointing to x_k.*

Proof: (Outline) We prove the lemma for x_0. The argument for x_k is similar. By definition, x_0 is the rightmost leaving process. Thus, processes to its right are staying. No leaving process sends messages with its own identifier. Thus, once the initial messages with its identifier are received, the message with x_0 may be sent only by a process forwarding this identifier towards x_0. Let v be the staying process with the largest identifier among right processes pointing to x_0. If v removes x_0's identifier, it never points to x_0 again. Thus, we need to show that v eventually does so.

Since x_0 is the rightmost leaving processes, all processes to the right of x_0 are staying. Therefore, none of them sends $req($**remright**$)$. Hence, after initial such messages are received, once defined, $right$ variable of process x_0 never becomes undefined

again. Moreover, if it changes, it can only hold processes progressively closer to x_0. Thus, if x_0 ever points to v and then points to some other process, this other process is closer to x_0 than v. Process v is staying. Therefore, it periodically sends $intro(v)$ to x_0. When x_0 receives this message, its actions depend on the contents of its *right* variable. If *right* is undefined or greater than v, then *right* is set to v. If *right* is less than v then v is forwarded to *right*.

Now, if *right* is defined, x_0 periodically sends request messages to *right*. If *right* holds v, then such message is sent to v. Once v receives such a request, v stops pointing to x_0.

If *right* holds an identifier less than v, once x_0 receives $intro(v)$, it forwards it to *right*. The process *right* may forward v's identifier further. Eventually, this forwarding stops at some process u. Since u is to the right of x_0, u is staying. Process u holds v in its *right* variable and periodically sends $intro(u)$ to v. Since u is to the right of x_0, once v receives this message, it starts pointing to u and stops pointing to x_0.

To summarize, in every computation, eventually v stops pointing to x_0. Hence, the lemma. □

Lemma 5. *In every computation of $\mathcal{SL}$, if a process x_i such that $i > 0$ belongs to a steady chain, then eventually no process, staying or leaving, to the right of x_i points to x_i.*

Proof: (Outline) Let us consider an arbitrary computation of $\mathcal{SL}$ with any process x_i in a steady chain of leaving processes. Let x_i be such that there are processes to the right of x_i that point to x_i. Let v be the rightmost such process. Since x_i is leaving, it does not send messages with its own identifier. Hence, once initial messages are received, if v stops pointing to x_i it will not point to x_i again.

If v is leaving, it periodically sends $req($**remright**$)$ to x_i. If x_i receives such a message, it removes its right link. However, by the definition of steady chain, x_i does not remove its right link before exiting the system. This means that v removes the identifier of x_i before sending a request.

If v is staying, it periodically sends $intro(v)$ to x_i. The processing of this message depends on the value of *right* at x_i. The identifier of v cannot be less than the identifier held in *right*. Otherwise, the value of *right* changes once x_i receives $intro(v)$. However, by the definition of steady chain, x_i does not change this until x_i leaves. Hence, v must be greater than *right*. In this case, once $intro(v)$ is received, v is forwarded to *right*. By using an argument similar to the proof of Lemma 4, we can show that v eventually stops pointing to x_i. □

Lemma 6. *In every computation of $\mathcal{SL}$, each leaving process exits the system.*

Proof: We prove the lemma by showing that in every computation, at least one leaving process eventually exits. Let us consider the leftmost process x_k in a maximal steady chain of the leaving processes. Since this chain is maximal, every leaving process to the left of x_k eventually stops pointing to it. By Lemma 4 every staying processes to the left of x_k eventually does so as well.

Lemmas 5 and 4 indicate that every process to the right of x_k eventually stops pointing to it as well. In other words, eventually, no process in the system points to x_k. In this case, no process sends messages to x_k. Once x_k receives all incoming messages, the guard of the *exit* action is enabled which allows x_k to leave the system. □

Proposition 2 as well as Lemmas 3 and 6 lead to the following theorem.

Theorem 2. *Algorithm $\mathcal{SL}$ and oracle $\mathcal{NIDEC}$ provide a self-stabilizing solution to the Finite Leave Linearization Problem $\mathcal{FLL}$.*

5 Extensions

Persistency. If a *persistent* oracle for a process u is **true** in a system state, then the actions of processes other than u cannot change the output of the oracle. An oracle that detects that it is safe for a process to leave does not have to be persistent. For example, an oracle may detect that the removal of all the incoming links to the departing process u preserves the weak connectivity of the remaining graphs due to alternative paths connecting the other processes. However, in general, such an oracle is not persistent because the departures of the processes in the alternative paths may make it unsafe for u to leave.

Oracle persistency is a useful property for a practical distributed system where it may take time to gather oracle information and then execute the dependent command. We capture this with the following discussion.

The *low atomicity message passing system* is a message passing system where the algorithm actions are restricted as follows. The action can either use oracles and receive messages, or send messages. This low atomicity model reflects the delay of possible command execution since message receipt and message sending actions may be interleaved by actions of other processes.

Two action executions are *causally related* [16] if (i) they happen in the same process, (ii) one is sending a message and the other is receiving the same message, (iii) one action uses an oracle that mentions the process of the other action. Two action executions are *concurrent* if they are not causally related. Two computations are *equivalent* if they only differ by the order of their concurrent actions.

Proposition 3. *If a message-passing asynchronous system algorithm uses persistent oracles only, then for every low-atomicity computation of this algorithm, there is an equivalent high-atomicity computation.*

Observe that $\mathcal{NIDEC}$ is persistent. Hence, the following theorem.

Theorem 3. *Algorithm $\mathcal{SL}$ and $\mathcal{NIDEC}$ oracle provide a self-stabilizing solution to the Finite Leave Linearization Problem $\mathcal{FLL}$ in the low atomicity asynchronous message passing system.*

Connectivity oracle and non-existent id detector. Algorithm $\mathcal{SL}$ fails to operate correctly if the system starts in a disconnected state or if the system contains links to the identifiers that are not present in the system. To make a complete system, we may introduce the following oracles. Oracle $\mathcal{CONNECT}$ detects that the system is disconnected and injects an identifier from one disconnected component to the other thus reconnecting it. Oracles $\mathcal{DETECTRIGHT}$ and $\mathcal{DETECTLEFT}$ return true if the respective right and left identifiers are non-existent. We summarize the addition of these oracles in the following proposition.

Proposition 4. *Algorithm $\mathcal{SL}$ with $\mathcal{NIDEC}$ as well as $\mathcal{CONNECT}$, $\mathcal{DETECTRIGHT}$ and $\mathcal{DETECTLEFT}$ oracles provide a self-stabilizing solution to the finite leave linearization problem $\mathcal{FLL}$ if the initial state is disconnected and it contains non-existent identifiers.*

Oracle implementation. Let us discuss the implementation of the oracles introduced in this paper. Oracles $\mathcal{NIDEC}$, $\mathcal{DETECTRIGHT}$ and $\mathcal{DETECTLEFT}$ may be implemented in a synchronous system using timeouts. For example, each process periodically sends a heartbeat message containing its identifier to its right and left neighbor. If process u does not receive such messages for a specified period of time, it assumes that no process points to u. If no process points to u, after some time, the incoming channel of u is empty and oracle $\mathcal{NIDEC}$ at u can be set to **true**.

Oracle $\mathcal{CONNECT}$ requires a bootstrap service to which every process is connected. This service would keep track of some unique property of a weakly connected component, e.g. the largest identifier. Once the bootstrap service observes that there are two sets of nodes that report different identifiers as their largest, the service detects system disconnect and injects the largest identifier of one component into the other.

Observe that all four of the above oracles are persistent. Hence, the correctness of their implementation is not affected by the delay in reporting of the detected condition. This simplifies their implementation. On the other hand, these oracles are to be used by self-stabilizing algorithms. Therefore, their own implementation has to be self-stabilizing. Such implementation is left for future research.

6 Conclusion

In conclusion we would like to address future research directions. Linearization, addressed in this paper is an elementary task of peer-to-peer network construction. It would be interesting to study whether our algorithm can be extended to more efficient structures such as skip-list or skip-graph as some other self-stabilizing algorithms were [9, 18, 21].

In this paper, we showed that $\mathcal{NIDEC}$ is necessary to enable a self-stabilizing solution to the Finite Leave Problem. This means that only $\mathcal{NIDEC}$ allows the system to stabilize from an arbitrary state. However, it would also be interesting to

study the power of individual components of $\mathcal{NIDEC}$: no-identifier oracle $\mathcal{NID}$ and empty channel oracle $\mathcal{EC}$. Specifically, it would be interesting to consider from what topologies and initial states these oracles allow recovery.

Finally, in this paper, we formally addressed finite churn. The difficulty of this problem was to ensure that all leaving processes safely depart before staying processes attempt to construct the required topology such as a linear sorted list. A more challenging task, to be addressed in the future, is to deal with infinite churn where arbitrary many processes may join and leave the system. In this case, the system cannot wait until all leaving processes depart. Instead, the system has to stabilize despite ongoing churn.

References

1. David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 131–145, New York, NY, USA, 2001. ACM.
2. James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37:1–37:25, 2007.
3. Baruch Awerbuch and Christian Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 318–327, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.
4. Markus Benter, Mohammad Divband, Sebastian Kniesburges, Andreas Koutsopoulos, and Kalman Graffi. Ca-re-chord: A churn resistant self-stabilizing chord overlay network. In *NetSys*, pages 27–34, 2013.
5. Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, and Mark Thober. Pagoda: a dynamic overlay network for routing, data management, and multicasting. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 170–179, New York, NY, USA, 2004. ACM.
6. Silvia Bianchi, Ajoy Datta, Pascal Felber, and Maria Gradinariu. Stabilizing peer-to-peer spatial filters. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 27, Washington, DC, USA, 2007. IEEE Computer Society.
7. Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.
8. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
9. Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list and skip graph. *Theoretical Computer Science*, 428:18–35, 2012.
10. Danny Dolev, Ezra Hoch, and Robbert van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *OPODIS 2007: Proceedings of the 11th International Conference on the Principles of Distributed Systems*, volume 4878 of *Lecture Notes in Computer Science*, pages 343–357. Springer, December 2007.
11. 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 Alejandro López-Ortiz, editor, *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, volume 6034 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2010.
12. Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: a scalable overlay network with practical locality properties. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.
13. 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.
14. R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proc. of 28th ACM Symp. on Principles of Distributed Computing*, pages 131–140, 2009.
15. Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Distributed Computing*, 22(4):249–267, 2010.
16. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
17. Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of Distributed Computing*, pages 183–192, New York, NY, USA, 2002. ACM.
18. Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 356–370, October 2011.
19. Rizal Mohd Nor, Mikhail Nesterenko, and Sebastien Tixeuil. Linearizing peer-to-peer systems with oracles. Technical Report TR-KSU-CS-2012-02, Dept. of Computer Science, Kent State University, July 2012.
20. Melih Onus, Andréa W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *ALENEX 2007: Proceedings of the Workshop on Algorithm Engineering and Experiments*. SIAM, January 2007.
21. Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors. *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*. ACM, 2011.
22. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.
23. Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.
24. 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 Transactions on Networking*, 11(1):17–32, February 2003.
25. Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 189–202, October 2006.

Appendix

The proofs of Proposition 2 and supporting lemmas are adopted from [18].

Lemma 7. If a computation of $\mathcal{SL}$ starts in a state where for some process a there are two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such chat $a < c < b$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \leq c$.

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $b < c < a$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \geq c$.

Intuitively, Lemma 7 states that if there is a link in the incoming channel of a process that is shorter than what the process already stores, then, the process' links will eventually be shortened. The proof is by simple examination of the algorithm.

Lemma 8. If a computation of $\mathcal{SL}$ starts in a state where for some process a there is an edge $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such that $a < b < c$, then the computation contains a state where there is a link $(d, c) \in CP$, where $d \leq b$.

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $c < b < a$, then this computation contains a state where there is a link $(d, c) \in CP$, where $d \geq b$.

Intuitively, the lemma states that if there is a longer link in the channel, it will be shortened by forwarding the id to its closer successor.

Lemma 9. If a computation of $\mathcal{SL}$ starts in a state where for some processes a, b, and c such that $a < c < b$ (or $a > c > b$), there are edges $(a, b) \in CP$ and $(c, a) \in CC$, then the computation contains a state where either some edge in CP is shorter than in the initial state or $(a, c) \in CP$.

Proof: The timeout action in process c is always enabled. When executed, it adds $message(c)$ to the incoming channel of process a. Then, the lemma follows from Lemma 7. □

Lemma 10. If a computation starts in a state where there is a link $(a, b) \in CP$, then the computation contains a state where some link in CP is shorter than in the initial state or there is a link $(b, a) \in CP$.

Proof: Assume without loss of generality that $a < b$. Once a executes its always enabled timeout action, link (b, a) is added to CC. We need to prove that either some link in CP is shortened or this link is added to CP.

Let us consider a link $(b, c) \in CP$ such that $c < b$. There can be three cases with respect to the relationship between a and c. In case $c < a$, the lemma follows from Lemma 7. In case $c = a$, the claim of the lemma is already satisfied. The case of $c > a$ is the most involved.

According to Lemma 8, if $c > a$, the computation contains a state where a shorter link to a belongs to CC. That is, there is a process d such that $a < d \leq c$ and $(a, d) \in CC$. Let us consider link $(e, d) \in CP$ such that $e < d$.

If $e < a$, then, according to Lemma 7, some link in CP shortens. If $e = a$, then some link in CP shortens according to Lemma 9. In both cases the claim of this lemma is satisfied.

Let us now consider the case where $e > a$. According to Lemma 8, the link to process a in CC shortens. The same argument applies to the new shorter link to a in CC. That is, either some link in CP shortens or a link to a shortens. Since the length of the link to a is finite, some link in CP eventually shortens. Hence the lemma. □

Lemma 11. If the computation is such that if $(a, b) \in CP$ then $(b, a) \in CP$ in every state of the computation, then this computation contains a suffix where $((a, b) \in CP) \Leftrightarrow ((a, b) \in CC)$

Lemma 11 states that if CP does noes not change in a computation then eventually, the links in CP contain all the links of CC.

Proof: (of the lemma)

That is, there is a pair of consequent processes u and v that are not neighbors. By condition of the lemma, CP is strongly connected. This means that there is a path from u to v.

Let us consider the shortest such path. Since u and v are not neighbors, the path has to include processes to the left or to the right of both u and v. Assume without loss of generality $u < v$ and the path includes processes to the right of u and v. Let us consider the rightmost process in this path w. Let x and y be the processes that respectively precede and follow w in this path. Since w is the rightmost, both x and w are to the left of w.

Note that each process in CP can have at most one outgoing left and one outgoing right neighbor. By the condition of the lemma the outgoing neighbor of a process is also its incoming neighbor. Since x precedes w in the path from u to v and y follows w, x is the incoming and y is the outgoing neighbors of w. Yet, x and y are both to the left of w. This means that $x = y$. However, this also means that w can be eliminated from the path from u to v and can be this way shortened. However, we considered the shortest path from u and v. It cannot be further shortened. We arrived at a contradiction that proves the if part of the lemma.

The only if part follows form the observation that each process can only have a single right and single left neighbor. That is, a process is already a neighbor with the consequent process it cannot be a neighbor with any other process. □