Linearizing Peer-to-Peer Systems with Oracles

Rizal Mohd Nor*†, Mikhail Nesterenko* and Sébastien Tixeuil‡

July 15, 2012

Technical report: TR-KSU-CS-2012-02
Department of Computer Science
Kent State University

Abstract

We study distributed linearization or topological sorting in peer-to-peer networks. We define strict and eventual versions of the problem and consider these problems with and without input identifiers that do not exist in the system. None of these problems are solvable in the asynchronous message-passing systems. We define a collection of oracles and prove which oracle combination is necessary to enable a solution in the message-passing systems for each variant of the linearization problem. We then present a linearization algorithm. We prove that this algorithm and a specific combination of oracles solves each stated variant of the linearization problem.

Corresponding author: Mikhail Nesterenko, Department of Computer Science, Kent State University, Kent, OH, 44242, USA, fax: +1 (330) 672-0737, email: mikhail@cs.kent.edu

Keywords: linearization, topological sort, peer-to-peer systems, oracles, failure detectors, necessary and sufficient conditions, asynchronous message passing systems, distributed algorithms

1 Introduction

Problem definition. Construction of structured peer-to-peer systems in asynchronous systems appears to have fundamental limitations such as inability to connect a disconnected network or discard peer identifiers that are not present in the system [17]. In this paper we endeavor to systematically study these limits and delineate the impossible from the achievable. We pattern our work after the classic impossibility of crash-robust consensus [10] and resolving it with failure detector oracles [4, 5].

We state the linearization (topological sort) problem requiring each process p to determine its two peers whose identifiers are consequent to p. Similar to the relationship between consensus and agreement problems, linearization is a basic task required for most popular peer-to-peer systems construction [1, 2, 14, 15, 16, 19, 20]. On the other hand, linearization is simple enough problem to see how the results established for linearization are applicable of peer-to-peer systems in general.

Similar to consensus, we define two variants of the problem: strict linearization, where each process has to output its consequent identifiers exactly once; and eventual linearization where a process may make a finite number of mistakes in its output. We introduce a restriction that is specific to peer-to-peer systems: the initial input may contain only process identifiers that exist in the system. We study the linearization problems with and without this restriction, i.e. we consider four different linearization problem variants.

*Kent State University, USA.
†International Islamic University, Malaysia.
‡UPMC Sorbonne Universités & IUF, France. This work was supported in part by ANR project SHAMAN.

Our contribution. We show that none of the four variants of the linearization problem are solvable in the asynchronous message-passing systems. Analogous to the consensus problem, we use the concept of oracles that encapsulate the impossible. We define the weak connectivity oracle that detects that the system is disconnected and restores its connectivity. We show that this oracle is necessary to solve all four variants of the problem. We define the participant detector oracle that removes non-existent identifiers from the system. We then show that this oracle is necessary to solve the linearization problem that allows non-existent identifier input. We define the oracle property of subset splittability. Intuitively, a subset splittability oracle does not provide information about the state of the outside system to a particular subset of processes. We then prove that non-subset splittable oracle is necessary to solve strict linearization.

On the constructive side, we present a linearization algorithm and show that it solves each variant of the linearization problem with a particular combination of oracles. Specifically, our algorithm solves eventual linearization problem with existent identifiers using only weak connectivity oracle; the addition of participant detector oracle enables solution to the problem with non-existent identifiers. Taken together with the necessary results, this demonstrates that the particular combinations of oracles are necessary and sufficient to solve the variants of the linearization problem with existing identifiers. We define the consequent detector oracle, a specific non-subset splittable oracle that can output consequent identifier once the process stores it in its memory. We then show that using the consequent detector oracle our algorithm solves the strict linearization problem. These results are summarized in Figure 2.

Related literature. Onus et al [18] recognize the importance of linearization as a fundamental problem in peer-to-peer system construction and study it in the context of self-stabilization [7, 21]. Gall et al [11] consider linearization performance bounds. In the impossibility part of their work, Mohd Nor et al [17] outline the limits of solvability of peer-to-peer problems in the message-passing systems. Emek et al [9] study various definitions of connectivity for overlay networks. There are several studies on participant detectors [3, 13] for consensus.

2 Notation and Execution Model

Peer-to-peer systems. A peer-to-peer overlay system consists of a set N of processes with unique identifiers. When it is clear from the context we refer to a process and its identifier interchangeably. A process stores other process identifiers in its local memory. Once the peer identifier is stored, the process is able to communicate with its peer by sending messages to it. Message routing is handled by the underlying network. We thus assume that the peers are connected by a communication channel. Processes may store identifiers of peers that do not exist in the system. If a message is sent to such non-existent identifier it is discarded. A process a *forwards* identifier b to process c, if a sends message containing identifier b to process c and erases b from its memory.

The peer identifiers are assumed to be totally ordered, i.e. for any two distinct identifiers a and b, either $a < b$ or $a > b$. Two processes a and b of set N are *consequent*, denoted $\mathbf{cnsq}(a, b)$ if any other process that belongs to N is either less than a or greater than b. Negative infinity is consequent with the smallest process of N and positive infinity is consequent with the largest process. Note that the total order of identifiers implies that if two non-identical sets are merged, the consequent process changes for at least one process in each set.

Graph terminology helps in reasoning about peer-to-peer systems. A *link*, denoted (a, b), between a pair of identifiers a and b is defined as follows: either message $message(b)$ carrying identifier b is in the incoming channel of process a, or process a stores identifier b in its local memory. Thus defined link is

directed. When referring to link (a, b), we always state the predecessor process first and the successor process second.

A *channel connectivity* multigraph CC includes both locally stored and message-based links. Self-loop links are not considered. Links to non-existent identifiers are not considered. Note that besides the processes CC may contain two nodes $+\infty$ and $-\infty$ and the corresponding links to them. Graph CC reflects the connectivity data that is stored in the process memory and, implicitly, in communication channels messages.

Computation model. Each process contains a set of variables and actions. A *channel* is a special kind of variable whose values are sets of messages. That is, we consider non-FIFO channels. The channels may contain an arbitrary number of messages, i.e. the channels are unbounded. We assume that the only information any message can carry is process identifiers. We further assume that each message carries only one identifier. Message loss is not considered. Since message order is unimportant we consider all messages sent to a particular process as belonging to the single incoming channel of this process.

An action has the form $\langle guard \rangle \longrightarrow \langle command \rangle$. *guard* is either a predicate over the process variables or the incoming channel or **true**. In the latter case the predicate and its action are *timeout*. *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. An action is *enabled* in some program state if its guard is **true** in this state. The action is *disabled* otherwise. A timeout action is always enabled.

A computation on a set N of processes is an fair sequence of states such that for each state s_i, the next state s_{i+1} is obtained by executing the command of an action the the processes of N that is enabled in s_i. This disallows the overlap in action execution. That is, the action execution is atomic. The computation is either infinite or it ends in a state where no actions are enabled. This execution semantics is called interleaving semantics or central demon [8]. We assume two kinds of fairness: weak fairness of action execution of fairness of 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 the channel, this computation contains a later state where this message is no longer in the channel. Besides the fairness, our computation model places no bounds on message propagation delay or relative process execution speed, i.e. we consider fully asynchronous computations.

Computation suffix is the 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-forward* if the only operations that it does with process identifiers is comparing 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-forward algorithm, 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-forward algorithm can not introduce new identifiers to the system, it can only operate on the ids that are already there. If a computation of a copy-store-forward algorithm starts from a state where every identifier is existing, each state of this computation contains only existing identifiers.

Oracles. An *oracle* is a specialized set of actions used to abstract a problem in distributed computing. The actions of a single oracle may be defined in multiple processes. An oracle action of a process may mention the state of variables of other processes and even the global state of the whole system.

An oracle is *subset splittable* for a linearization algorithm $\mathcal{A}$ if there exists two non-intersecting set of processes S_1 and S_2 as well as a computation σ_1 on S_1 of $\mathcal{A}$ and state s_2 of processes in S_2 with the following property. If an oracle is enabled in some state s_1 of processes of S_1 of σ_1, this oracle is also enabled in $s_1 \cup s_2$. That is, if the processes of S_2 in state s_2 are added to s_1, the oracle still remains enabled. An oracle is just subset splittable, if it is subset splittable for any linearization algorithm. Intuitively, subset splittability prevents a subset of processes from learning about the state of the rest of the system on the basis of an oracle. Subset splittable and not-subset splittable oracles are respectively denoted as $\mathcal{SS}$ and $\mathcal{NSS}$.

A linearization algorithm is *proper* if it satisfies the following requirements.

- If a process a has identifiers b and c, such that $a < b < c$ then process a forwards c to b. The requirement is similar in the opposite direction. That is, process forwards identifier closer to its destination.
- A process that does not contain identifiers to its right or left is *orphan*. A process does not orphan itself. That is, the process does not discard its only single left, or single right, identifier. Note that oracle actions may still orphan the process.

3 The Linearization Problem and Solution Oracles

Linearization problem statement. The linearization problem is stated as follows. Each process p of a given set N of processes, is input a left l and a right r neighbor such that $l < p$ and $r > p$. These values may be $-\infty$ and $+\infty$ respectively. The communication channels are empty. In the solution, each process should output two identifiers: cl and cr such that each is consequent with p. The smallest process should output negative infinity as its left neighbor while the largest process should output positive infinity as it right neighbor.

Depending on the certainty of the output, the problem has two variants. The *strict linearization problem* $\mathcal{SL}$ requires each process to output its neighbors exactly once and allows only correct output. The *eventual linearization problem* $\mathcal{EL}$ states that each computation contains a suffix where the output of each process is correct. That is, each process is allowed to make a finite number of mistakes. The problem statement also depends on whether non-existent identifiers may be present in the initial state. *Non-existing identifier variant* $\mathcal{NID}$ allows such identifiers while *existing-only identifiers variant* $\mathcal{EID}$ prohibits them.

The combination of these conditions defines four different linearization problem statements. When we refer to the specific linearization problem, we state the particular conditions. For example, strict linearization problem with non-existing identifiers is referred to as $\mathcal{SL}$+$\mathcal{NID}$.

Oracles. The oracle actions are shown in Figure 1. An oracle may have one or two actions. The two actions operate on the right and left variable of the process and have respective subscripts r and l.

We define the following oracles to be used in solving the linearization problem. Weak connectivity oracle $\mathcal{WC}$ has a single action that selects a pair of processes p and q such that they are disconnected in the channel connectivity graph CC and adds q to the incoming channel of p connecting them. Participant detector $\mathcal{PD}$ oracle removes a non-existent identifier stored in p. The actions of neighbor output oracle $\mathcal{NO}$ just output identifiers stored in left and right variables of p. In fact, $\mathcal{NO}$ is not a true oracle. It is trivially built from scratch as it uses only local variables of p. However, for ease of exposition $\mathcal{NO}$ actions are described among oracles. The actions of consequent process detector $\mathcal{CD}$ are similar to the actions of $\mathcal{NO}$ in effect. However, each action of $\mathcal{CD}$ outputs the stored identifier only if it is consequent with p. That is, unlike $\mathcal{NO}$, the guard of $\mathcal{CD}$ mentions all the identifiers of the system.

process p

constants and global variables

N, // set of processes in the system
CC // system channel connectivity graph

shortcuts

$\mathbf{cnsq}(a, b) \equiv (\forall c : c \in N : (c < a) \vee (b < c))$

local variables

r, l, // input, right ($> p$) and left ($< p$) neighbors
cl, cr // output, right and left consequent process, initially $\perp$

oracle actions

$\mathcal{WC}$: CC contains disconnected components $C1$ and $C2$ such that
$(p \in C1) \wedge (q \in C2) \longrightarrow$
send $message(q)$ **to** p

$\mathcal{PD}$l: $l \notin N \longrightarrow l := -\infty$
$\mathcal{PD}$r: $r \notin N \longrightarrow r := +\infty$

$\mathcal{NO}$l: $cl \neq l \longrightarrow cl := l$
$\mathcal{NO}$r: $cr \neq r \longrightarrow cr := r$

$\mathcal{CD}$l: $(cl \neq l) \wedge \mathbf{cnsq}(l, p) \longrightarrow cl := l$
$\mathcal{CD}$r: $(cr \neq r) \wedge \mathbf{cnsq}(p, r) \longrightarrow cr := r$

Figure 1: Linearization algorithm oracles.

Lemma 1 *Oracles $\mathcal{NO}$, $\mathcal{PD}$ and $\mathcal{WC}$ are subset splittable while $\mathcal{CD}$ is not.*

Proof: Indeed, $\mathcal{NO}$ is trivially subset splittable since its guards only mention local variables. To see why $\mathcal{PD}$ is subset splittable, consider a set of processes S_1 and a computation σ_1 of some algorithm $\mathcal{A}$ on this set. We form another set of processes S_2 such that it does not intersect with S_1 and does not contain any of the non-existing identifiers appearing in σ_1. Let s_2 be an arbitrary state of processes of S_2. If some identifier nid is non-existent in a state s_1 of σ_1 it remains non-existent if processes in state $s_1 \cap s_2$. Hence, if an action of $\mathcal{PD}$ is enabled in s_1, is enabled in $s_1 \cup s_2$ as well.

Let us now consider $\mathcal{WC}$. Again, let S_1 be a set of processes and σ_1 be a computation of some algorithm $\mathcal{A}$ on it. Let S_2 be a set of processes that does not intersect with S_1. Let state s_2 of processes S_2 be such that none of these processes stores identifiers from S_1. Let us consider a state that is formed by merging some state s_1 of σ_1 and s_2. If channel connectivity graph CC is disconnected in s_1, it remains disconnected in $s_1 \cup s_2$. Hence, if an action of $\mathcal{WC}$ is enabled in s_1, it is also enabled in $s_1 \cup s_2$. That is, $\mathcal{WC}$ is subset splittable.

Let us discuss $\mathcal{CD}$. Consider an arbitrary set of processes S_1 and a computation σ_1 of some linearization algorithm $\mathcal{A}$ on it. Each process of a linearization algorithm has to output process identifiers consequent with itself. If a process stores consequent identifiers, its $\mathcal{CD}$ actions are enabled. However, since the

identifier space is totally ordered, regardless of the composition of S_2, if S_2 is added to S_1, at least one process in S_1 changes its consequent process. This disables an action of $\mathcal{CD}$. Hence, $\mathcal{CD}$ is not subset splittable. □

4 Necessary Conditions

Lemma 2 *If a computation of a copy-store-forward algorithm starts in an arbitrary state where the channel connectivity graph CC is disconnected. Either the graph remains disconnected for the rest of the computation or the computation contains an execution of a weak connectivity oracle action.*

Proof: Let us consider the computation σ that contains a state where the channel connectivity graph CC is at least weakly connected. Let s_2 be the first such state. Assume, without loss of generality, that in s_2 process a has a link to process b in CC while in all previous states, including the state s_1 that directly precedes s_2, the two processes are disconnected. The link may be due to the action of the algorithm or an oracle. Let us consider the possibility of algorithm action first.

Since processes in the message passing system do not share local memory, an algorithm action may create link (a, b) in CC only by adding process b to the incoming channel of a. That is, some process c sends a message carrying b to a. This message transmission moves the system from s_1 to s_2. To send a message to a, process c needs to hold the identifier of a in state s_1. That is, c has to be connected to a in s_1. Also, c sends identifier b to a. That is, c is connected to b in s_1. This means that for this message transmission, a and b need to be weakly connected in s_1. However, we assumed that s_2 is the first state where a and b are connected.

Hence, the action that moves the system from s_1 to s_2 has to be an oracle action. This action connects two disconnected processes. That is, it has to be the action of the weak connectivity oracle. □

Theorem 1 *Every solution to the linearization problem requires a weak connectivity oracle.*

Proof: Let $\mathcal{A}$ be a linearization algorithm. Let us consider the set of processes to be linearized. Let us further consider a computation of $\mathcal{A}$ that starts in a state where this set is separated into two arbitrary subsets S_1 and S_2. Since process identifiers are totally ordered, there has to be at least two consequent processes $p_1 \in S_1$ and $p_2 \in S_2$. Since $\mathcal{A}$ is a linearization algorithm, p_1 has to eventually output p_2. According to Lemma 2, this may only happen if the computation contains the actions of the weak connectivity oracle. □

Theorem 2 *A solution to the strict linearization problem requires a non-subset splittable oracle.*

Proof: Assume the opposite. Let there be an algorithm $\mathcal{A}$ that solves the strict linearization problem with only subset splittable oracle $\mathcal{O}$. Since $\mathcal{O}$ is subset splittable, there are two non-intersecting sets of processes S_1 and S_2 as well as a computation σ_1 of $\mathcal{A}$ on S_1 and a state s_2 of S_2 such that the addition of s_2 to every state of σ_1 keeps the actions of $\mathcal{O}$ in processes of S_1 enabled.

We construct a computation σ_3 of $\mathcal{A}$ on $S_1 \cup S_2$ as follows. The computation starts with the initial state of σ_1 merged with s_2. We then consider the first action of σ_1, if the action is non-oracle, since processes of S_1 in σ_3 have the same initial state as in σ_1, the action is enabled and can be executed. If the first action is an oracle $\mathcal{O}$ action, since the oracle is subset splittable, this action is enabled and can be executed. We proceed building σ_3 by sequentially executing the actions of σ_1. Computation σ_1 is produced by $\mathcal{A}$ that, by assumption, is a solution to the strict linearization problem. By the statement of the problem, during σ_1, every process has to output the identifier of its consequent process exactly once. We stop adding the actions of σ_1 to σ_3 once every process of S_1 does so. We conclude the construction of σ_3 by executing the actions of $\mathcal{A}$ and $\mathcal{O}$ in an arbitrary fair manner. Thus constructed, σ_3 is a computation of $\mathcal{A}$.

Let us examine σ_3. By construction, every process p_1 in S_1 outputs an identifier that p_1 is consequent with in S_1. Since the identifier state space is totally ordered, the consequent identifiers of at least one process of S_1 differ if S_2 is added to S_1. This means that this process outputs incorrect identifier in σ_3 that is executed on $S_1 \cup S_2$. However, this violates the requirements of the strict linearization problem. This means that, contrary to our initial assumption, $\mathcal{A}$ is not a solution to $\mathcal{SL}$ and the strict linearization problem indeed requires a non-subset splittable oracle. □

Theorem 3 *A proper solution to the linearization problem that allows non-existing identifiers requires a participant detector oracle.*

Proof: Assume the opposite. Let $\mathcal{A}$ be a proper algorithm that solves a linearization problem with non-existing identifiers and does not use $\mathcal{PD}$. That is, oracles used by the the algorithm do not remove non-existing identifiers.

Let us construct a computation σ on some set of processes. We select the initial state of σ as follows. Processes do not have links to existing identifiers. That is, each process is disconnected from all other processes. Each process stores exactly two non-existing identifiers. For any two neighbor processes p_1 and p_2 such that $p_1 < p_2$, the non-existing identifier np_1 stored at p_1 is such that $p_1 < np_1 < p_2$, the non-existing identifier np_2 stored at p_2 is $p_1 < np_2 < p_2$. That is, the non-existing ids are between neighbors. If the process has the largest, or smallest identifier in the set, this process contains respectively lower and higher non-existing identifier. Note that since $\mathcal{A}$ is proper, processes do not orphan themselves. Hence, they cannot remove the non-existing identifiers. Since $\mathcal{A}$ does not use participant detector oracle, the oracles that it does use cannot remove the non-existing identifiers either. Thus, the only identifier action that the oracles can do is adding identifiers.

Since $\mathcal{A}$ is proper, a process cannot orphan itself. Hence, the actions of the algorithm cannot remove the non-existent identifiers from this initial state. Since $\mathcal{A}$ is copy-store-forward, the actions cannot add new identifiers to the system. That is, the only possible actions that affect the topology of the system are oracle actions. We construct σ as follows. We execute an enabled oracle action. The oracle may add an identifier id_1 to some process p_1. Assume, without loss of generality, $id > p_1$. Process p_1 already holds $np_1 > p_1$. There may be two cases. In the first case, id_1 is greater than np_1. Since $\mathcal{A}$ is proper, id_1 is forwarded to np_1. Since np_1 is non-existing, id_1 is lost and the system remains disconnected. Let us consider the case where id_1 is less than np_1. In this case id_1 is non-existing. Since $\mathcal{A}$ is proper, p_1 keeps id_1 and forwards np_1 to id_1. That is np_1 is discarded. The system, however, remains disconnected.

The resultant state resembles the initial state of σ in the sense that the only actions that may be enabled are the actions of an id-adding oracle. In similar manner, we continue constructing σ by executing an enabled oracle action and then letting the algorithm handle the added identifier. We proceed with this construction until there are no more enabled oracle actions or indefinitely.

In this computation, no process has the identifier of its consequent process. Hence, the process may not output the neighbor's identifiers in σ. That is, contrary to our assumptions, $\mathcal{A}$ is not a solution to the linearization problem with non-existing identifiers. □

The theorems of this section specify the oracles that are necessary to solve each variant of the linearization problem. These requirements are summarized in Figure 2(a).

5 Linearization Solutions

Algorithm description. The linearization algorithm $\mathcal{L}$ contains to actions: $\mathcal{REC}$ and $\mathcal{TO}$. The actions are shown in Figure 3. The first is a message receipt action $\mathcal{REC}$. This action is enabled if the incoming channel of process p contains a message bearing some identifier id. If the received id is greater than the

	$\mathcal{EL}$	$\mathcal{SL}$
$\mathcal{EID}$	$\mathcal{WC}$	$\mathcal{WC}$+$\mathcal{NSS}$
$\mathcal{NID}$	$\mathcal{WC}$+$\mathcal{PD}$	$\mathcal{WC}$+$\mathcal{PD}$+$\mathcal{NSS}$

(a) Necessary oracles.

	$\mathcal{EL}$	$\mathcal{SL}$
$\mathcal{EID}$	$\mathcal{L}$+$\mathcal{WC}$+$\mathcal{NO}$	$\mathcal{L}$+$\mathcal{WC}$+$\mathcal{CD}$
$\mathcal{NID}$	$\mathcal{L}$+$\mathcal{WC}$+$\mathcal{NO}$+$\mathcal{PD}$	$\mathcal{L}$++$\mathcal{WC}$+$\mathcal{CD}$+$\mathcal{PD}$

(b) Solution algorithm and oracles sufficient for solution.

Figure 2: Necessary and sufficient conditions for a linearization problem solution.

left neighbor r of p, p forwards this identifier to p to process. If id is between p and r, then p, selects id to be its new right neighbor and forwards the old neighbor for r to handle. Process p handles received id smaller than its own in a similar manner. If p receives its own identifier, p discards it. The second action is a timeout action $\mathcal{TO}$. It is always enabled. This means that the correction of the algorithm does not depend on the timing of the action execution, which is left up to the implementer. The action sends identifier p to its right and left neighbor provided they exist. Note that the linearization algorithm is proper.

Lemma 3 *If channel connectivity graph contains only existing identifiers, the operation of the linearization algorithm $\mathcal{L}$ in combination with any of the oracles does not disconnect any pair of processes in the channel connectivity graph CC.*

Proof: Let us consider the actions of the oracles first. The actions of $\mathcal{WC}$ may only add identifiers to CC. Hence it does not disconnect the processes in CC. Since there are no non-existent identifiers, the actions of $\mathcal{PD}$ are disabled. Oracles $\mathcal{NO}$ and $\mathcal{CD}$ only copy the identifiers in the same process. Hence, they do not affect CC either.

Let us now consider the action of $\mathcal{L}$. The operation of receive action $\mathcal{REC}$ action depends on the value of the received identifier id. If id is the same as p, it is discarded. However, since self-loops are not considered in CC, this discarding of the identifier does not change CC. Let us consider the case $p > id$. If $id > r$, then p forwards id to r to deal with. That is, the link (p, id) in CC is replaced by the path (p, r), (r, id). If $p > id \geq r$, process p replaced its right neighbor with p and forwards its old right neighbor to id. That is, the link (p, id) is preserved in CC while (p, r) is replaced by (p, id), (id, r). In either case no path in CC is disconnected. The case of $p < id$ is similar. The timeout action $\mathcal{TO}$ only adds links to CC so it does not disconnect it. □

Lemma 4 *Starting from an arbitrary state that contains only existing identifiers, the linearization algorithm $\mathcal{L}$ in combination with any of the oracles, arrives at a state where the channel connectivity graph CC is connected.*

Proof: Indeed, if CC is disconnected, actions of $\mathcal{WC}$ are enabled in the processes of the disconnected components. Once such action is executed, the two components are connected. According to Lemma 3, the components are not disconnected again regardless of used oracles. Hence, CC is eventually connected in every computation of the linearization algorithm where $\mathcal{WC}$ is used. □

Lemma 5 *Any computation of the linearization algorithm $\mathcal{L}$ in combination with participant detector oracle $\mathcal{PD}$ and any other oracles has a suffix with only existing identifiers.*

Proof: Observe that none of the oracles introduce new non-existing identifiers. Since, linearization algorithm is copy-store-send, it does not create new identifiers either. Hence, to prove the lemma we need to show that all non-existent identifiers present in the initial state are removed.

Note that each process of the linearization algorithm either keeps an identifier or forwards it to its neighbors. That is, processes of $\mathcal{L}$ do not duplicate non-existent identifiers. Moreover, the identifier is

process p

local variables
r, l // input, right ($> p$) and left ($< p$) neighbors

algorithm action
$\mathcal{REC}$: $message(id)$ is in the coming channel of $p \longrightarrow$
receive $message(id)$
if $id > p$ **then**
if $id < r$ **then**
if $r < +\infty$ **then**
send $message(r)$ **to** id
$r := id$
else
send $message(id)$ **to** r
if $id < p$ **then**
if $id > l$ **then**
if $l > -\infty$ **then**
send $message(l)$ **to** id
$l := id$
else
send $message(id)$ *to* l

$\mathcal{TO}$: **true** $\longrightarrow$
if $l > -\infty$ **then**
send $message(p)$ **to** l
if $r < +\infty$ **then**
send $message(p)$ **to** r

Figure 3: Linearization algorithm actions.

forwarded only in one direction: either to the left or to the right. This means that during the computation each identifier will be forwarded a finite number of times. Let us consider process p that holds non-existent identifier nid and does not forward it. Since nid is non-existent, an action of participant detector $\mathcal{PD}$ is enabled at p. Since nid is not forwarded, the action remains enabled until executed. Once executed, the action removes the non-existent identifier. That is, every non-existent identifier is eventually removed. □

Lemma 6 *Starting from an arbitrary state where CC is connected and only existing identifiers are present, the linearization algorithm combined with the timeout oracle and regardless of the operation of other oracles contains a suffix where the variables r and l of each process p hold identifiers consequent with p.*

The proof of Lemma 6 is in the appendix.

Theorem 4 *The linearization algorithm combined with neighbor output, and weak connectivity oracles solves eventual linearization with existing identifiers problem. The linearization algorithm combined with*

consequent process detector and weak connectivity oracles solves strict linearization with existing identifiers problem.

The addition of participant detector enables the solution to the non-existent identifier variants of these problems.

The specific oracles sufficient for each problem solution as stated in Theorem 4 are summarized in Figure 2(b).

Proof: Let us first address the case of existing identifiers only. According to Lemma 6, if a computation starts in an arbitrary state where CC is connected, this computation contains a suffix where each process p stores its consequent identifiers in r and l. The argument differs depending on whether $\mathcal{NO}$ or $\mathcal{CD}$ is being used.

In case $\mathcal{NO}$ is used, if p stores different identifiers in r and cr, then $\mathcal{NO}$r is enabled. Once executed, the identifier stored in r is output. That is, if there is a suffix of a computation containing consequent right identifier in r of p, there is a suffix that contains this identifier cr. Similar argument applies to the left identifier of p. That is, every computation of $\mathcal{L}$+$\mathcal{NO}$+$\mathcal{WC}$ contains a suffix where consequent left and right neighbors are output. In other words, this combination of the linearization algorithm and oracles solves $\mathcal{EL}$+$\mathcal{EID}$.

Let us consider the case of $\mathcal{CD}$. Note that consequent process detector oracle outputs the identifier if and only if it is consequent. However, every computation of the algorithm contains a suffix where each process stores its consequent identifiers. If the process holds its consequent identifier, $\mathcal{CD}$ is enabled. Once $\mathcal{CD}$ is executed, the correct identifier is output. That is, every computation of $\mathcal{L}$+$\mathcal{CD}$+$\mathcal{WC}$ every process outputs its consequent identifiers exactly once. In other words, this combination of the linearization algorithm and oracles solves $\mathcal{SL}$+$\mathcal{EID}$.

Let us address the case of non-existing identifiers. According to Lemma 5, participant process detector oracle $\mathcal{PD}$ eventually removes non-existent identifiers from the system. That is, very computation contains a suffix with only existing identifiers. In this case $\mathcal{NO}$ eventually outputs correct identifiers that satisfies the conditions of eventual linearization problem.

By its specification, consequent process detector oracle $\mathcal{CD}$ never outputs non-existent identifiers. That is, the presence of non-existent identifiers does not compromise the solution to the strict linearization problem if $\mathcal{CD}$ is used.

Hence, the addition of $\mathcal{PD}$ enables the solution of the non-existing identifier variants of the linearization problems. □

6 Oracle Implementation and Optimality

Oracle nature and implementation. The three oracles required to solve the linearization problem variants described in this paper are weak connectivity, participant process detector and consequent process detector. None of them are implementable in the computation model we consider. Nonetheless, let us discuss possible approaches to their construction.

Oracle $\mathcal{WC}$, that repairs the network disconnections, is an encapsulation of bootstrap service [6] commonly found in peer-to-peer systems. One possible implementation of such oracle is as follows. One bootstrap process b is always present in the system. This identifier may be part of the oracle implementation and, as such, not visible to the application program using the oracle. The responsibility of this process is to maintain the greatest and smallest identifier of the system. All other processes are supplied with b's identifier. If a regular system process p does not have a left or right neighbor, it assumes that its own identifier the greatest or, respectively, smallest. Process p then sends its identifier to b.

Process b then either confirms this assumption or sends p, its current smallest or greatest identifier. This way, if the system is disconnected, the weak connectivity is restored.

Oracle $\mathcal{PD}$ encapsulates the limits between relative process speeds and maximum message propagation delay. This oracle may be implemented using a heartbeat protocol [12]. For example, if process p contains an identifier q, p sends q a heartbeat message requesting a reply. If p does not receive this reply after the time above the maximum network delay, p considers q non-existent and discards it.

Oracle $\mathcal{CD}$ may be the most difficult to implement. It appears that to implement $\mathcal{CD}$ one has to solve the strict linearization problem itself. In this sense, $\mathcal{CD}$ serves to illustrate the difficulty of the strict linearization problem rather than encode any particular oracle implementation.

Oracle optimality. This paper states the necessary and sufficient conditions for both strict and eventual linearization problem. The conditions for the eventual linearization are sharp as we use the necessary oracles to provide the algorithmic solution for the problem. For the strict linearization, there is a gap between these conditions. Specifically, our algorithmic solutions relies on $\mathcal{CD}$, which is a specific kind of the necessary non-subset splittable detector. Narrowing the gap between necessary and sufficient conditions for the solutions to the strict linearizability problem remains to be addressed in the future.

References

[1] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37:1–37:25, 2007.

[2] 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.

[3] David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In *Ad-Hoc, Mobile, and Wireless Networks: Third International Conference, (ADHOC-NOW)*, volume 3158 of *Lecture Notes in Computer Science*, pages 135–148. Springer, July 2004.

[4] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of ACM*, 43(4):685–722, 1996.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] Curt Cramer and Thomas Fuhrmann. ISPRP: a message-efficient protocol for initializing structured P2P networks. In *International Performance Computing and Communications Conference (IPCCC)*, pages 365–370, 2005.

[7] Edsger W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[8] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[9] Yuval Emek, Pierre Fraigniaud, Amos Korman, Shay Kutten, and David Peleg. Notions of connectivity in overlay networks. In Guy Even and Magnús M. Halldórsson, editors, *Structural Information and Communication Complexity - 19th International Colloquium, SIROCCO 2012, Reykjavik, Iceland, June 30-July 2, 2012, Revised Selected Papers*, volume 7355 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 2012.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[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] Mohamed G. Gouda and Tommy M. McGuire. Accelerated heartbeat protocols. In *18th International Conference on Distributed Computing Systems (ICDCS)*, pages 202–209, May 1998.

[13] Fabíola Greve and Sébastien Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Proceedings of IEEE International Conference on Dependable Systems and networks (DSN)*, pages 82–91. IEEE, June 2007.

[14] Nicholas J. A. Harvey and J. Ian Munro. Deterministic skipnet. *Inf. Process. Lett.*, 90(4):205–208, 2004.

[15] 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.

[16] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algogrithms*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

Appendix

The proofs of Lemma 6 and supporting lemmas are adopted from [17].

Define CP a subgraph of CC that contains links based on the identifiers stored in process memory. In this sense, CP captures current network connectivity information the set of processes possesses.

Lemma 7 *If a computation of $\mathcal{L}$ 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 $\mathcal{L}$ 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 $\mathcal{L}$ 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. □