Linearizing Peer-to-Peer Systems with Oracles

Rizal Mohd Nor[1,⋆], Mikhail Nesterenko[2], and Sébastien Tixeuil[3,⋆⋆]

[1] International Islamic University, Malaysia
[2] Kent State University, USA
[3] UPMC Sorbonne Universités & IUF, France

Abstract. We study distributed linearization or topological sorting in peer-to-peer networks. We define strict and eventual variants of the problem. We consider these problems restricted to existing peer identifiers or without this restriction. None of these variants are solvable in the asynchronous message-passing system model. We define a collection of oracles and prove which oracle combination is necessary to enable a solution for each variant of the linearization problem. We then present a linearization algorithm. We prove that this algorithm and a specific combination of the oracles solves each stated variant of the linearization problem.

1 Introduction

Oracles and Limits of Solvability in peer-to-peer Systems. Mohd Nor et al [17] showed that construction of structured peer-to-peer systems in asynchronous systems have fundamental limits such as inability to connect a disconnected network or discard peer identifiers that are not present in the system. These limits do not appear to be reducible to just the properties of asynchronous systems alone, such as lack of consensus [10]. That is, the limits are specific to peer-to-peer problems.

In this paper we endeavor to systematically study these limits. We intentionally pattern our work on the classic proof of impossibility of crash-robust consensus [10] and its resolution with failure detector oracles [4, 5]. That is, we identify peer-to-peer system specific oracles and isolate the source of impossibility in them, we then show the minimality of oracles by proving their necessity for solution existence and then solve the problem by providing an oracle-based algorithm.

We focus on the problem of linearization (topological sort). Let us motivate our choice of the problem. Linearization requires each process p to determine two peers whose identifiers are consequent, i.e. next to one another in topological order, with this p's identifier. This problem underlies most popular peer-to-peer systems [1, 2, 14–16, 19, 20] as more sophisticated constructions start by topologically sorting the peer-to-peer network. While being foundational for many

⋆ This work is supported in part by a Research Acculturation Grant Scheme RAGS 12-042-0042, IIUM, MOHE Malaysia.

⋆⋆ This work was supported in part by LINCS.

T. Higashino et al. (Eds.): SSS 2013, LNCS 8255, pp. 221–236, 2013.

peer-to-peer systems, linearization is similar to consensus in the following sense. Linearization is simple enough so that one can observe how the results established for this problem pertain to all peer-to-peer systems.

Our Contribution. 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.

In present work, we show that none of the four variants of the linearization problem are solvable in the asynchronous message-passing systems. We use the concept of oracles to encapsulate the impossible. We define the weak connectivity oracle that detects the system to be 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 splittable oracle does not provide information about the state of the outside system to a particular subset of processes. We then prove that a non-subset splittable oracle is necessary to solve strict linearization.

On the constructive side, we use a simple linearization algorithm [17] and show that it solves each variant of the linearization problem with a particular combination of oracles. Specifically, this 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 4.

Related Literature. Mohd Nor et al [17] provided impetus for this work. As a part of the work presented in their paper, they showed that there are limitations of achievable results in peer-to-peer systems. However, the applicability of their negative results is limited, as Mohd Nor et al considered only self-stabilizing algorithms [7, 21]. To prove impossibility of stabilization, it is sufficient to show that there exists a global state from which no program can possibly recover. However, such results may not be applicable to regular, non-stabilizing programs, as non-stabilizing programs are only required to solve the problem from a particular non-faulty initial state. Therefore, such programs may never reach the degenerate states that self-stabilizing programs have to address. Hence, proving the limits for regular programs is significantly more involved.

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. Gall et al [11] consider linearization performance bounds. 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 of N 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, the message is discarded. A process a *forwards* identifier b to process c, if a sends a 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 either. 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 variable type 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 a 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 of 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 daemon [8]. We assume two kinds of fairness: weak fairness of action execution and 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, i.e. the message is received. 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 which we call copy-store-forward algorithms. 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. Hence, 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 exist two non-intersecting sets 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. For every state s_1 of σ_1 where this oracle is enabled, this oracle is also enabled in $s_1 \cup s_2$. In other words, if the processes of S_2 in state s_2 are added to any such state 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, a process forwards each 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 identifier 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 are respectively distinguished by letters r and l.

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

Fig. 1. Linearization algorithm oracles

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 creating a link (p, q) in CC thus connecting the graph. 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.

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

Proof: To prove subset splittability of an oracle, by definition, we need to identify two non-intersecting sets of processes S_1 and S_2, a computation σ_1 on

S_1 of an arbitrary linearization algorithm $\mathcal{A}$ and a state s_2 of S_2, such that if this oracle is enabled in some state of s_1 of σ_1, it remains enabled in $s_1 \cup s_2$.

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 in state $s_1 \cap s_2$. Hence, if an action of $\mathcal{PD}$ is enabled in s_1, it 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 of S_2 be such that none of these processes stores identifiers of 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. $\square$

4 Necessary Conditions

Lemma 2. If the channel connectivity graph CC is disconnected in the initial state of copy-store-forward algorithm computation, then either CC is disconnected in every state of the computation or this computation contains an execution of a weak connectivity oracle action.

Proof: Let us consider the computation σ of an arbitrary copy-store-forward algorithm such that σ contains states where CC is at least weakly connected yet CC is disconnected in the initial state of σ. Let s_2 be the first state of σ where CC is connected. 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. Refer to Figure 2 for illustration. Since processes in the message passing system model 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. Since the algorithm is copy-store-forward, to send a message to a, process c needs to store the identifier of a in its local memory in the preceding state s_1. That is,

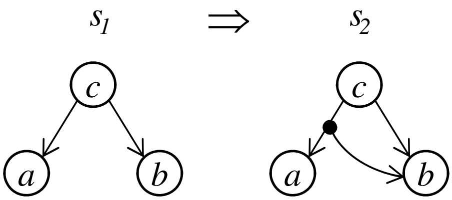

Fig. 2. Illustration to the proof of Lemma 2. Transition from state s_1 where processes a and b are disconnected to s_2 where they are connected via a link (a, b) in the incoming channel of process a, requires initial overall system connectivity, i.e. CC needs to be connected.

c has to be connected to a in CC of 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 can only be an oracle action. This action connects two disconnected processes. That is, it has to be the action of the weak connectivity oracle. Therefore, if a computation of a copy-store-forward algorithm starts from a state where CC is disconnected, the only way this computation produces a state with connected CC is through the action of a 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 such that if process $a \in S_1$ stores identifier b then $b \notin S_2$. Similarly if process $c \in S_2$ stores identifier d then $d \notin S_1$. Note that in thus formed initial state, the sets S_1 and S_2 are disconnected in the channel connectivity graph CC.

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 continue building σ_3 by sequentially executing the actions of σ_1. Computation σ_1 is produced by $\mathcal{A}$ which, 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 algorithm do not remove non-existing identifiers.

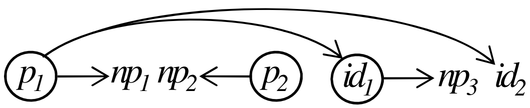

Fig. 3. Illustration to the proof of Theorem 3. In the initial state of constructed computation, two consequent processes p_1 and p_2 hold non-existent identifiers np_1 and np_2. An oracle action at p_1 adds identifiers id_1 and id_2 to process p_1. Process p_1 forwards id_2 to id_1.

Let us construct a computation σ on some set of processes. We select the initial state of σ as follows. Refer to Figure 3 for illustration. 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.

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 either. Since $\mathcal{A}$ is copy-store-forward, its actions cannot add new identifiers to the system. That is, there are no enabled actions of $\mathcal{A}$ that change its topology in the initial state of σ.

Since $\mathcal{A}$ does not use the participant detector oracle, the oracles that it does use cannot remove the non-existing identifiers either. That is, the only oracle actions that are enabled in the initial state of σ add process identifiers.

We construct σ as follows. Let p_1 be the process that has an enabled oracle action. We execute this action and consider the identifiers that the oracle action adds to p_1. The identifiers may be greater or smaller than p_1. Moreover, they may be existent or non-existent.

We consider the added identifiers that are greater than p_1. The case of smaller identifiers is similar. Process p_1 already holds $np_1 > p_1$. Since $\mathcal{A}$ is proper, process p_1 has to select two identifiers id_1 and id_2 such that $p_1 < id_1 < id_2$ and forward id_2 to id_1. Thus, p_1 eliminates id_2 from its memory. We add this forwarding action to σ. We continue this process of identifier elimination until p_1 holds only a single identifier greater than its own.

If p_1 ever forwards non-existing np_1 to some process id_1, then $p_1 < id_1 < np_1$. That is, the remaining identifier id_1 is non-existing. Therefore, once p_1 is left with a single identifier, this identifier is non-existing and p_1 remains disconnected from the higher-id processes.

Let us now consider what happens with the identifiers that p_1 forwards. The recipient identifier id_1 may be existing or non-existing. If id_1 is non-existing, the forwarded identifier id_2 is lost. Let us address the situation when id_1 is existing. Note that id_2 is greater than than id_1. Once id_2 is received by id_1, its operation depends on the value of its right non-existent identifier np_3. There may be two cases. In the first case, id_2 is greater than np_3. Since $\mathcal{A}$ is proper, id_2 is forwarded to np_3. Since np_3 is non-existing, id_2 is lost and the system remains disconnected. If id_2 is less than np_3, id_2 is definitely non-existing. Since $\mathcal{A}$ is proper, id_1 keeps id_2 and forwards np_3 to id_2. That is, np_3 is discarded. The system, however, remains disconnected. We construct the computation σ by thus processing all identifiers forwarded by p_1.

The resultant state resembles the initial state of σ in the sense that all processes are disconnected and the only actions that may be enabled are the actions of an id-adding oracle. We continue constructing σ by executing an enabled oracle action in a fair manner and then letting the algorithm handle the added identifiers. We proceed with this construction either indefinitely or until there are no more enabled oracle actions.

Thus constructed σ is a computation of $\mathcal{A}$. However, no process outputs the identifiers of its consequent processes. That is, contrary to our assumption, $\mathcal{A}$ is not a solution to the linearization problem with non-existing identifiers. $\square$

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 4(a).

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

(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.

Fig. 4. Necessary and sufficient conditions for a linearization problem solution

5 Linearization Solutions

Algorithm Description. The linearization algorithm $\mathcal{L}$ is adapted from [17]. The algorithm contains two actions: $\mathcal{REC}$ and $\mathcal{TO}$. The actions are shown in Figure 5. 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 right neighbor r of p, p forwards this identifier to r 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 id 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 correctness 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 $\mathcal{L}$ 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 actions of $\mathcal{L}$. The operation of receive action $\mathcal{REC}$ 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 the weak connectivity

```
process p

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

algorithm action
REC: message(id) is in the coming channel of p ⟶
            receive message(id)
            if id > p then
                if id < r then
                    if r < +∞ then
                        send message(r) to id
                    r := id
                else
                    send message(id) to r
            if id < p then
                if id > l then
                    if l > −∞ then
                        send message(l) to id
                    l := id
                else
                    send message(id) to l

TO: true ⟶
            if l > −∞ then
                send message(p) to l
            if r < +∞ then
                send message(p) to r
```

Fig. 5. Linearization algorithm actions

oracle $\mathcal{WC}$ and any other 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 $\mathcal{L}$ 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 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 [17].

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 4(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, every 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 is 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. We believe that to implement $\mathcal{CD}$ one has to solve the strict linearization problem itself. That is, $\mathcal{CD}$ 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 solution 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 solution to the strict linearizability problem remains to be addressed in future research.

References

1. Aspnes, J., Shah, G.: Skip graphs. ACM Transactions on Algorithms 3(4), 1–37 (2007)
2. Awerbuch, B., Scheideler, C.: The hyperring: a low-congestion deterministic data structure for distributed environments. In: SODA 2004: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 318–327. Society for Industrial and Applied Mathematics, Philadelphia (2004)
3. Cavin, D., Sasson, Y., Schiper, A.: Consensus with unknown participants or fundamental self-organization. In: Nikolaidis, I., Barbeau, M., An, H.-C. (eds.) ADHOC-NOW 2004. LNCS, vol. 3158, pp. 135–148. Springer, Heidelberg (2004)
4. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of ACM 43(4), 685–722 (1996)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
6. Cramer, C., Fuhrmann, T.: ISPRP: a message-efficient protocol for initializing structured P2P networks. In: International Performance Computing and Communications Conference (IPCCC), pp. 365–370 (2005)
7. Dijkstra, E.W.: Self-stabilization in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
8. Dubois, S., Tixeuil, S.: A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint (October 2011)
9. Emek, Y., Fraigniaud, P., Korman, A., Kutten, S., Peleg, D.: Notions of connectivity in overlay networks. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 25–35. Springer, Heidelberg (2012)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
11. Gall, D., Jacob, R., Richa, A.W., Scheideler, C., Schmid, S., Täubig, H.: Time complexity of distributed topological self-stabilization: The case of graph linearization. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 294–305. Springer, Heidelberg (2010)
12. Gouda, M.G., McGuire, T.M.: Accelerated heartbeat protocols. In: 18th International Conference on Distributed Computing Systems (ICDCS), pp. 202–209 (May 1998)
13. Greve, F., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: Proceedings of IEEE International Conference on Dependable Systems and networks (DSN), pp. 82–91. IEEE (June 2007)
14. Harvey, N.J.A., Ian Munro, J.: Deterministic skipnet. Inf. Process. Lett. 90(4), 205–208 (2004)
15. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: a scalable and dynamic emulation of the butterfly. In: PODC 2002: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, pp. 183–192. ACM, New York (2002)

16. Munro, J.I., Papadakis, T., Sedgewick, R.: Deterministic skip lists. In: SODA 1992: Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algogrithms, pp. 367–375. Society for Industrial and Applied Mathematics, Philadelphia (1992)
17. Nor, R.M., Nesterenko, M., Scheideler, C.: Corona: A stabilizing deterministic message-passing skip list. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 356–370. Springer, Heidelberg (2011)
18. Onus, M., Richa, A.W., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: ALENEX 2007: Proceedings of the Workshop on Algorithm Engineering and Experiments. SIAM (January 2007)
19. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
20. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Transactions on Networking 11(1), 17–32 (2003)
21. Tixeuil, S.: Self-stabilizing Algorithms. In: Algorithms and Theory of Computation Handbook, 2nd edn., pp. 26.1–26.45. CRC Press, Taylor & Francis Group (2009); Chapman & Hall/CRC Applied Algorithms and Data Structures