Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative

Lélia Blin, Sébastien Tixeuil

Compact Deterministic Self-Stabilizing Leader Election on a Ring: The Exponential Advantage of Being Talkative*

Lélia Blin†
Université d'Evry-Val d'Essonne, 91000 Evry, France.
UPMC Sorbonne Universités, France.
LIP6-CNRS UMR 7606, France.
lelia.blin@lip6.fr

Sébastien Tixeuil
UPMC Sorbonne Universités, France.
Institut Universitaire de France.
LIP6-CNRS UMR 7606, France.
sebastien.tixeuil@lip6.fr

Abstract

This paper focuses on *compact* deterministic self-stabilizing solutions for the leader election problem. When the solution is required to be *silent* (*i.e.*, when the state of each process remains fixed from some point in time during any execution), there exists a lower bound of $\Omega(\log n)$ bits of memory per participating node , where n denotes the number of nodes in the system. This lower bound holds even in rings. We present a new deterministic (non-silent) self-stabilizing protocol for n-node rings that uses only $O(\log \log n)$ memory bits per node, and stabilizes in $O(n \log^2 n)$ rounds. Our protocol has several attractive features that make it suitable for practical purposes. First, it assumes an execution model that is used by existing compilers for real networks. Second, the size of the ring (or any upper bound on this size) does not need to be known by any node. Third, the node identifiers can be of various sizes. Finally, no synchrony assumption, besides weak fairness, is assumed. Our result shows that, perhaps surprisingly, silence can be traded for an exponential decrease in memory space without significantly increasing stabilization time or introducing restrictive assumptions.

*A preliminary version of this paper has appeared in [12, 13].
†Additional support from the ANR project IRIS.

1 Introduction

This paper tackles the problem of designing efficient self-stabilizing algorithms for the leader election problem. *Self-stabilization* [19, 20, 37] is a general paradigm to provide recovery capabilities to distributed systems and networks. Intuitively, a protocol is self-stabilizing if it is able to recover from any transient failure, without external intervention. *Leader election* is one of the fundamental building blocks of distributed computing, as it enables a single node in the system to be distinguished, and thus to perform specific actions. Leader election is especially important in the context of self-stabilization as many protocols for various problems assume that a single leader exists in the system, even when faults occur. Hence, a self-stabilizing leader election mechanism enables such protocols to be run in networks where no leader is given a priori, by using simple stabilization-preserving composition techniques [20].

Most of the literature in self-stabilization is dedicated to improving efficiency after failures occur, including minimizing the stabilization time, *i.e.*, the maximum amount of time one has to wait before recovering from a failure. While stabilization time is meaningful to evaluate the efficiency of an algorithm in the presence of failures, it does not necessarily capture the overhead of self-stabilization when there are no faults [1], or after stabilization. Another important criterion to evaluate this overhead is the *memory space* used by each node. This criterion is motivated by two practical reasons.

First, self-stabilizing protocols for non-trivial tasks require that *some* communication carries on forever, in order to be able to detect distributed inconsistencies due to transient failures [8, 18]. The default model for writing self-stabilizing algorithms, the *state model*, assumes that the state of every node is communicated to its neighbors infinitely often. This model is also used in stabilization-preserving compilers that produce actual code [5, 15, 16, 36]. Therefore, minimizing the memory space used by each node enables the amount of information that is exchanged between nodes to be minimized.

Second, minimizing memory space also reduces the amount of memory used for the purpose of redundancy when mixing self-stabilization and replication [27, 28]. While self-stabilization permits recovery from *arbitrary* memory corruptions, replication permits withstanding *random* memory corruptions. For instance, having three copies of every bit at each node permits one randomly flipped bit to be withstood in a transparent manner. The combination of self-stabilization and replication increases the probability of masking or containing transient faults when they randomly appear (using redundancy) and recovering from them (using self-stabilization) [27, 28]. So, decreasing the memory space of a self-stabilizing protocol facilitates the use of redundancy as less memory is required to replicate the initial memory in order to tolerate a given number of random bit-flips.

A foundational result regarding memory space in the context of self-stabilization is due to Dolev *et al.* [21]. It states that, in n-node networks, $\Omega(\log n)$ bits of memory per node are required for solving global tasks such as leader election. Importantly, this bound holds even for the ring. A key component of this lower bound is that the protocol is assumed to be *silent*. (A protocol is silent if each of its executions reaches a point in time after which the states of nodes do *not* change.) The lower bound can be extended to *talkative* protocols (*i.e.,* protocols that are not silent), but only for some specific settings. For instance, the $\Omega(\log n)$ bound holds in *anonymous* (and uniform) unidirectional rings, even of prime size [11, 26]. As a matter of fact, most deterministic self-stabilizing leader election protocols [2, 4, 6, 17, 22] use $\Omega(\log n)$ bits of memory per node. Indeed, either these protocols directly compare node identifiers (and thus communicate node identifiers to neighbors), or they compute some variant of a hop-count

distance to the elected node and this distance can be as large as $\Omega(n)$ to be accurate.

A few previous works [31, 32, 35, 11] managed to obtain self-stabilizing leader election algorithms with $o(\log n)$ bits of memory per node in other models. Nevertheless, the corresponding algorithms exhibit shortcomings that hinder their relevance to practical applications.

For instance, Beauquier *et al.* [11] consider unidirectional networks where node identifiers are bounded above by $n+k$, where k is a small constant and n is the exact size of the network. Their deterministic algorithm uses a constant number of bits per node and gathers all identifiers that are below $k+1$. (There exists at least one of them and at most $k+1$ of them by hypothesis.) and elects the node with the lowest identifier as the leader. As acknowledged by the authors, such assumptions about node identifiers are unrealistic for practical purposes. In Internet-like networks, unique IP addresses typically span the entire range of addresses (using 128 bits for the IPv6 standard). By contrast, the actual size of an application-level network such as the WWW was recently estimated to around 1 billion nodes, which only requires 32 bits for unique identifiers. So, protocols requiring that the range of addresses matches the network size [11] are not suitable for practical networks.

Also, the algorithms by Mayer *et al.* [35], by Itkis and Levin [31], and by Awerbuch and Ostrovsky [7] use a constant number of bits per node. However, these algorithms only guarantee *probabilistic* self-stabilization (in the Las Vegas sense). In particular, the stabilization time is only *expected* to be polynomial in the size of the network. Moreover, these algorithms are designed for communication models that are more powerful than the classical state model used in this paper. More specifically, Mayer *et al.* [35] use the message passing model and Awerbuch and Ostrovsky [7] use the link-register model, where communication between neighboring nodes is carried out through dedicated registers. Finally, Itkis and Levin [31] use the state model augmented with *reciprocal pointers* to neighbors. In this model, not only is a node u able to distinguish a particular neighbor v (which can be done using local labeling), but this distinguished neighbor v is aware that it has been selected by u. Implementing this mutual interaction between neighbors typically requires distance-two coloring, link coloring, or two-hop communication. All these techniques increase the memory space requirement significantly [34].

It is also important to note that the communication models in [7, 31, 35] allow nodes to send different information to different neighbors, while this capability is beyond the power of the classical state model where the nodes do not have unique identifiers. The state model enriched with unique identifiers permits specific information to be sent to specific neighbors. (A node can simply prefix the information to be sent with the neighbor destination identifier.) However, this technique requires $\delta \log n$ bits of memory per node, where δ denotes the node degree, so it is not compatible with the $o(\log n)$ bits memory constraint. The ability to send different messages to different neighbors is a strong assumption in the context of self-stabilization. It enables a "path of information" that is consistent between nodes to be constructed. This path is typically used to distribute the storage of information along a path, in order to reduce the information stored at each node. However, this additional assumption precludes preserving the $o(\log n)$ bit memory constraint when using existing compilers that produce code for actual hardware [5, 15, 16, 36]. Those compilers are designed for the state model and operate by executing a local broadcast of the same message to all neighboring nodes, so sending different messages to different neighbors requires usingthe identifiers to specify the destination. So implementing the protocols in [7, 31, 35] in actual networks requires significant effort (that is, not reusing existing compilers) if memory usage is to be preserved.

To our knowledge, the only *deterministic* self-stabilizing leader election protocol for bidirectional rings using sub-logarithmic memory space in the classical state model is due to Itkis *et*

al. [32]. Their elegant algorithm uses only a constant number of bits per node, and stabilizes in $O(n^2)$ time in n-node rings. However, the algorithm relies on several restrictive assumptions. First, the algorithm works properly only if the size of the ring is *prime*. Second, it assumes that, at any time, a *single* node is scheduled for execution, that is, it assumes a *central* scheduler [25]. Such a scheduler is less general than the classical *distributed* scheduler, which allows any set of nodes to be scheduled for execution concurrently. Third, the algorithm in [32] assumes that the ring is *oriented*. That is, every node is supposed to possess a consistent notion of left and right. This orientation permits reciprocal pointers to neighbors to be simulated. Extending the algorithm by Itkis *et al.* [32] to other models such as unoriented rings of arbitrary size or state models with a distributed scheduler, is not trivial if one wants to preserve sub-logarithmic memory space at each node. For example, the existing transformers enabling protocols designed for the central scheduler to operate under the distributed scheduler require $\Theta(\log n)$ memory at each node [25]. Similarly, self-stabilizing ring-orientation protocols exist, but those with sub-logarithmic memory space either work only in rings of odd size [29], or just provide probabilistic guarantees [30]. Moreover, in both cases, the stabilization time is $\Theta(n^2)$, which is quite large.

To summarize, all existing self-stabilizing leader election algorithms designed for a practical communication model, for rings of arbitrary size, without a priori orientation, use $\Omega(\log n)$ bits of memory per node. Breaking this bound, without introducing any kind of restriction on the model, requires, beside being talkative, a completely new approach.

Our results

In this paper, we present a deterministic (non-silent) self-stabilizing leader election algorithm that operates under the distributed scheduler in non-anonymous unoriented rings of arbitrary size. Our algorithm is talkative to circumvent the $\Omega(\log n)$ lower bound on the bits of memory per node in [21]. It uses only $O(\log \log n)$ bits of memory per node, and stabilizes in $O(n \log^2 n)$ time.

Unlike the algorithms in [7, 31, 35], our algorithm is deterministic, and designed to run under the classical state model [5, 15, 16, 36]. Unlike [32], the size of the ring is arbitrary, the ring is not assumed to be oriented, and the scheduler is distributed. Moreover the stabilization time of our algorithm is significantly smaller than the one in [32]. Similarly to [7, 31, 33, 35], our algorithm uses a technique to distribute the information among nearby nodes along a sub-path of the ring. However, our algorithm does not rely on powerful communication models such as the ones used in [7, 31, 35]. Those powerful communication models make the construction and management of such sub-paths easy. The use of the classical state-sharing model makes the construction and management of the sub-paths much more difficult.

In addition to the use of sub-logarithmic memory space and a quasi-linear stabilization time, our algorithm retains several attractive features with respect to versatility. First, the size (or an upper bound for the size, [11]) does not need to be known by any node. Second, the node identifiers (or identities) can be of various sizes. (This models, *e.g.*, Internet networks running different versions of IP.) Third, no synchrony assumption besides weak fairness is assumed. (This means, a node that is continuously enabled for execution is eventually scheduled for execution.)

At a high level, our algorithm is essentially based on two techniques [3, 24, 10]. One consists of electing the leader by comparing the identities of the nodes, bitwise, which requires special care, especially when the node identities can be of various sizes. The second technique consists of maintaining and merging trees based on a parenthood relation, and verifying the absence of cycles in the graph induced by this parenthood relation. This verification is performed using

small memory space by grouping the nodes into hyper-nodes of appropriate size. Each hyper-node handles an integer encoding a distance to a root. The bits of this distance are distributed among the nodes of the hyper-nodes to maintain small memory usage per node. Difficulties arise when one needs to perform arithmetic operations on these distributed bits, especially in the context where nodes are unaware of the size of the ring. The precise design of our algorithm requires overcoming many other difficulties due to the need to maintain correct information in an environment subject to arbitrary faults.

In addition, our algorithm is ready for implementation, *i.e.*, we do not only describe a conceptual protocol, but also produce a *concrete* self-stabilizing leader election protocol. This article provides a high level description of our algorithm, a detailed description of the protocol, and a complete proof of correctness.[1] Our result shows that, perhaps surprisingly, silence can be traded for an exponential decrease in memory space without significantly increasing stabilization time or introducing restrictive assumptions.

2 Model and definitions

2.1 Protocol syntax and semantics

We consider a distributed system consisting of n processes that form a ring-shaped communication graph. The processes are represented by the nodes of this graph, and the edges represent pairs of processes that can communicate directly with each other. Such processes are said to be *neighbors*. The *distance* between two processes is the length of (*i.e.*, number of edges on) the shortest path between them in the communication graph. Let $C_n = (V, E)$ be an n-node ring, where V is the set of nodes, and E the set of edges. A node v has access to a constant unique identifier Id_v, but can only access its identifier one bit at a time, using the $\mathrm{Bit}(x, \mathsf{Id}_v)$ function, which returns the position of the x^{th} most significant bit equal to 1 in Id_v. This position can be encoded with $O(\log \log n)$ bits when identifiers are encoded using $O(\log n)$ bits, as we assume they are. A node v has access to locally unique port numbers (1 and 2 in ring-shaped networks) associated to its adjacent edges. We denote by $\mathtt{port}_u$ the port number corresponding to neighbor u of node v. We do not assume any consistency between port numbers of a given edge, and port numbers do not necessarily induce an orientation for the ring. In short, port numbers are constant throughout the execution but initialized by an adversary.

Each process contains variables and rules. A variable ranges over a domain of values. The variable var_v denote the variable var located at node v. A rule is of the form

$$\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle.$$

A *guard* is a boolean predicate over process variables. A *command* is a set of variable-assignments. A command of process p can only update its own variables. On the other hand, p can read the variables of its neighbors. This classical communication model is called the *state model* or the *state-sharing communication model*.

An assignment of values to all variables in the system is called a *configuration*. A rule whose guard is **true** in some system configuration is said to be *enabled* in this configuration.

[1] The reader is invited to consult *www-npa.lip6.fr/~blin/Election/* where a video of the dynamic execution of the protocol is presented. This video is the result of a complete implementation of the protocol. The video execution uses a randomized distributed scheduler. The initial configuration is illegitimate, and the video displays the protocol moving towards a legitimate configuration.

The rule is *disabled* otherwise. The atomic execution of a subset of enabled rules (at most one rule per process) results in a transition of the system from one configuration to another. This transition is called a *step*. A *run* of a distributed system is a maximal alternating sequence of configurations and steps. Maximality means that the execution is either infinite, or its final configuration has no rule enabled.

2.2 Schedulers

A *scheduler*, also called a *daemon*, is a restriction on the runs to be considered. The schedulers differ by different execution semantics and by different fairness requirements in the activation of the processes [25]. With respect to execution semantics, we consider the least restrictive scheduler, called the *distributed scheduler*. In a run of a distributed scheduler, any step can contain the execution of a non-empty arbitrary subset (at most one rule per process) . With respect to fairness, we use the least restrictive fair scheduler, called the *weakly fair scheduler*. In every run of the weakly fair scheduler, a rule of a correct process is executed infinitely often if it is enabled in all but finitely many configurations of the run. That is, a rule has to be executed if it is continuously enabled. A *round* is the smallest portion of an execution where every process has the opportunity to execute at least one action. In more detail, each process having at least one enabled rule at the beginning of a round r is either scheduled for execution during r, or all its rules become disabled due to neighbors' rules being executed during r.

2.3 Predicates and specifications

A predicate is a boolean function over configurations. A configuration *conforms* to some predicate R, if R evaluates to **true** in this configuration. The configuration *violates* the predicate otherwise. Predicate R is *closed* in a certain protocol P, if every configuration of a run of P conforms to R, provided that the protocol starts from a configuration conforming to R. Note that if a protocol configuration conforms to R, and the configuration resulting from the execution of any step of P also conforms to R, then R is closed in P.

A *Problem specification* prescribes the protocol behavior. The desired output of the protocol is expressed through specification variables that in turn form specification configurations. A problem specification is a the set of sequences of specification configurations.

Part of the implementation is the mapping from the protocol configurations to the specification configurations. This mapping does not have to be one-to-one. However, we only consider unambiguous protocols where each protocol configuration maps to only one specification configuration. Once the mapping between protocol and specification configurations is established, the protocol runs are mapped to specification sequences as follows. Each protocol configuration is mapped to the corresponding specification configuration. Then, stuttering, the appearance of consecutive identical specification configurations, is eliminated.

Overall, a run of the protocol satisfies the specification if its mapping belongs to the specification. Protocol P *solves* problem S under a certain scheduler if every run of P produced by that scheduler satisfies the specifications defined by S.

Given two predicates l_1 and l_2 for protocol P, l_2 is an *attractor* for l_1 if every run that starts from a configuration that conforms to l_1 contains a configuration that conforms to l_2. Such a relationship is denoted by $l_1 \triangleright l_2$. Also, the $\triangleright$ relation is transitive: if l_1, l_2, and l_3 are predicates for P, and $l_1 \triangleright l_2$ and $l_2 \triangleright l_3$, then $l_1 \triangleright l_3$. In this last case, l_2 is called an *intermediate* attractor towards l_3.

Definition 1 (Self-stabilization) *A protocol P is* self-stabilizing *[19] to specification S if there exists a predicate L for P such that:*

1. *L is an attractor for* true*,*
2. *Any run of P starting from a configuration satisfying L satisfies S.*

So, a protocol P *solves a problem S in a self-stabilizing manner* under scheduler D if from an arbitrary initial configuration of P, every run of P produced by D contains a configuration c such that P solves S starting from c.

2.4 Leader election specification

Consider a system of processes where each process' set of variables is mapped to a boolean specification variable leader. The *leader election* specification sequence consists in a single specification configuration where a unique process p maps to $\mathsf{leader}_p = true$, and every other process $q \neq p$ maps to $\mathsf{leader}_q = false$.

3 A compact leader-election protocol for rings

In this section, we describe our self-stabilizing algorithm for leader election in arbitrary n-node rings. The algorithm is later proved to use $O(\log \log n)$ bits of memory per node, and to stabilize in quasi-linear time, whenever the identities of the nodes are between 1 and n^c, for some $c \geq 1$. We first provide a general overview of the algorithm in Subsection 3.1, followed by a more detailed description in Subsection 3.2.

3.1 Overview of the algorithm

Like many existing deterministic self-stabilizing leader election algorithms, our algorithm elects the node with maximum identity among all nodes, and, simultaneously, constructs a spanning tree rooted at the elected node. The main constraint imposed by our sub-logarithmic memory usage is that we cannot exchange or even locally use complete identifiers, as the $\Omega(\log n)$ bits that are necessary to encode them do not fit. As a matter of fact, we assume that every node can access the bits of its identifier, but only a constant number of them can be simultaneously stored and/or communicated to neighbors at any given time. Our algorithm assumes however that every node is able to store the current position of a particular bit of the identifier, referred to as a *bit-position* in the sequel. Only bit-positions related to bits valued 1 in the identifier are exchanged.

3.1.1 Leader selection

Our algorithm operates in phases. At each phase, each node that remains a candidate leader reveals a new bit-position, different from the ones exchanged during previous phases, to its two neighbors. More precisely, let Id_v be the identity of node v, and assume that $\mathsf{Id}_v = \sum_{i=0}^{k} b_i 2^i$. Let $I_v = \{i \in \{0, ..., k\}, b_i \neq 0\}$ be the set of all non-zero bit-positions in the binary representation of Id_v. Let us rewrite $I_v = \{pos_1, ..., pos_j\}$ where $pos_k > pos_{k+1}$. During the algorithm execution, the candidate leaders must agree on the same bit-position pos_{j-i+1} (for $i = 1, \ldots, j$)

to be revealed; this step of the algorithm defines *phase* i. (That is, the bit-positions are communicated in decreasing order of significance in the encoding of the identifier.) In turn, this may propagate it to their neighbors, and possibly to the whole network in subsequent phases. During phase i, node v either becomes passive (that is, v stops acting as a candidate leader) or remains a candidate leader. If, at the beginning of the execution of the algorithm, more than one node is a *candidate* leader, then during each phase, some candidate leaders are expected to be eliminated, until exactly one candidate leader remains, which becomes the actual leader. More precisely, let $pos_{max}(i)$ be the most significant bit-position revealed at phase i among all candidate leader nodes. Then, among them, only those whose bit-position revealed at phase i is equal to $pos_{max}(i)$ carry on the election process. The other ones become passive.

If all identities are in $[1, n^c]$, for some constant $c \geq 1$, then the communicated bit-positions are less than or equal to $c\lceil \log n \rceil$, and thus can be represented with $O(\log \log n)$ bits. The difficulty is to implement this simple "compact" leader election mechanism in a self-stabilizing manner. In particular, the number of bits used to encode identifiers may be different for two given nodes, so there is no common upper bound for the size of identifiers. We circumvent this problem using a ranking on bit-positions that is agnostic on the size of the identifiers.

Our approach is to tie the election process to a tree construction mechanism. Ultimately, the elected process is the root of the unique constructed tree. More than one candidate indicates that several trees (rooted at those candidates) are present: the tree construction eventually merges those trees. In the case where all nodes are passive, this implies that they all have a parent and are in the same cycle: the tree construction process guarantees that at least one node detects the cycle and all nodes become candidates again (see Section 3.1.2).

An additional problem in self-stabilizing leader election is the potential presence of *impostor* leaders due to an erroneous initial configuration. An impostor leader is a node that believes (according to its local memory) that it is a leader, while *e.g.* its identifier is actually not the largest in the network. If one can store the identity of the leader at each node, then detecting an impostor is easy. (A node that has a larger identifier than the impostor can detect an error and trigger a corrective action.) Under our memory constraints, nodes cannot store the identity of the leader, nor store all the bits of their own identifier. So, detecting impostor leaders becomes non trivial, notably when an impostor has an identity whose most significant bit is equal to the most significant bit of the leader. To overcome this problem, the selection of the leader must run perpetually, causing our algorithm to be talkative.

3.1.2 Spanning tree construction

Our approach to make the above scheme self-stabilizing is to merge the leader election process with a tree construction process. Every candidate leader is the root of a tree and every passive node points to its parent, a neighbor one step closer to the root of the tree it belongs to. Whenever a candidate leader becomes passive, its tree is merged into another tree, until there remains only one tree. The main obstacle to self-stabilizing tree-construction is the possibility of an arbitrary initial configuration. This is particularly difficult if the initial configuration contains a cycle (induced by the parent variables of passive nodes) rather than a spanning forest whose roots are candidate leaders. When such a cycle exists in a ring-shaped network, there are no candidate leaders. Such a configuration is called a *leaderless* configuration. In order to break cycles that can be present in the initial configuration, we use an improved variant of the classical distance calculation [23]. In the classical approach, every node u maintains an integer variable d_u that stores the distance from u to the root of its tree. If par_u denotes the parent of u, then, when stabilization is achieved, $\mathsf{d}_{par_u} = \mathsf{d}_u - 1$ should hold. Now, if in a configuration

$\mathsf{d}_{par_u} \geq \mathsf{d}_u$, then u can execute a corrective action for its variable par_u. As the topology of the network is a ring, detecting the presence of an initial spanning cycle with this approach implies that distance variables could be as large as n, requiring $\Omega(\log n)$ bits of memory.

In order to use exponentially less memory, our algorithm uses the distance technique modulo (an estimate of) $c\lfloor \log n \rfloor$, so that distance variables remain $O(\log \log n)$ bits. More specifically, each node v maintains three variables : d_v, par_v, and dB_v. The first variable is an integer $\mathsf{d}_v \in \{0, ..., c\lfloor \log n \rfloor\}$, called the "distance" of node v. Only candidate leaders v can have $\mathsf{d}_v = 0$. At every other (passive) node v, it is expected that $\mathsf{d}_v = 1 + (\mathsf{d}_w \bmod c\lfloor \log n \rfloor)$ where w denotes the parent neighbor of v (see below).

The second variable is par_v, denoting the parent of node v when v is passive. This variable takes values in $\{\emptyset, 0, 1\}$, where $\emptyset$ denotes the fact that the node has no parent, and 0 and 1 denote the port numbers leading to the parent. When it is clear from the context, we use $\mathsf{par}_v = u$ (resp. $\mathsf{par}_v \neq u$) to denote the fact that the port number stored in par_v leads to (resp., does not lead to) neighbor u. This parent is a neighbor w such that $\mathsf{d}_v = 1 + (\mathsf{d}_w \bmod c\lfloor \log n \rfloor)$. By itself, this technique is not sufficient to detect the presence of a cycle, as the number of nodes could be a multiple of $c\lfloor \log n \rfloor$. Therefore, we also introduce the notion of *hyper-node*, defined as follows:

Definition 2 *A* hyper-node X *is a sequence* $(x_1, x_2, \cdots, x_{c\lfloor \log n \rfloor})$ *of consecutive nodes in the ring, such that* $\mathsf{d}_{x_1} = 1$, $\mathsf{d}_{x_2} = 2$,..., $\mathsf{d}_{x_{c\lfloor \log n \rfloor}} = c\lfloor \log n \rfloor$, $\mathsf{par}_{x_2} = x_1$, $\mathsf{par}_{x_3} = x_2$, ..., $\mathsf{par}_{x_{c\lfloor \log n \rfloor}} = x_{c\lfloor \log n \rfloor - 1}$ *and* $\mathsf{par}_{x_1} \neq x_2$.

Definition 3 *A hyper-leaf is a sequence* $(x_1, x_2, \cdots, x_k)$ *of consecutive nodes in the ring, such that* $\mathsf{d}_{x_1} = 1$, $\mathsf{d}_{x_2} = 2$,..., $\mathsf{d}_{x_k} = k$, $\mathsf{par}_{x_2} = x_1$, $\mathsf{par}_{x_3} = x_2$, ..., $\mathsf{par}_{x_k} = x_{k-1}$ *with* $k < c\lfloor \log n \rfloor$, $\mathsf{par}_{x_1} \neq x_2$, *and* x_k *is a leaf. (That is, the other neighbor* x' *of* x_k *is such that* $\mathsf{par}_{x'} \neq x_k$.*)*

When two hyper-nodes $X = (x_1, x_2, \cdots, x_{c\lfloor \log n \rfloor})$ and $Y = (y_1, y_2, \cdots, y_{c\lfloor \log n \rfloor})$ are such that $\mathsf{par}_{x_1} = y_{c\lfloor \log n \rfloor}$, Y is said to be the *parent* of X. By definition, there are at most $\lceil n / c\lfloor \log n \rfloor \rceil$ hyper-nodes in any given network of size n. Note that, as the last node in a hyper-leaf is a leaf, it cannot belong to a cycle. Our algorithm tries to maintain the acyclicity of the graph whose vertices are the hyper-nodes and whose edges are defined by the parent relation, which happens only when at least one hyper-leaf exists in this graph. The key to our protocol is that hyper-nodes can maintain larger (*e.g.* $c\lfloor \log n \rfloor$ bits) distance information than a single node, by distributing the information among its nodes. More precisely, our algorithm assumes that each node v maintains one bit of information, stored in variable dB_v. Let $X = (x_1, x_2, \cdots, x_{\lfloor \log n \rfloor})$ be a hyper-node. Then, the sequence $\mathsf{dB}_X = (\mathsf{dB}_{x_1}, \mathsf{dB}_{x_2}, \cdots, \mathsf{dB}_{x_{c\lfloor \log n \rfloor}})$ can be interpreted as the binary representation of a $c\lfloor \log n \rfloor$-bit integer (that is, a value between 0 and $2^{c\lfloor \log n \rfloor} - 1$). Now, this approach makes it possible to use the classical distance approach to prevent cycles, but at the hyper-node level. Part of our protocol consists of comparing the distance dB_X and the distance dB_Y for two hyper-nodes X and Y such that Y is the parent of X. If the difference between dB_X and dB_Y is not one, an inconsistency is detected and a corrective action is initiated. Since hyper-nodes include $c\lfloor \log n \rfloor$ nodes (and the fact that $2^{c\lfloor \log n \rfloor} \geq n / \log n$), dealing with distances between hyper-nodes is sufficient to detect the presence of a cycle spanning the n-node ring. In essence, the part of our algorithm that is dedicated to checking the absence of a spanning cycle generated by the parent relationship boils down to comparing distances between hyper-nodes. However, comparing such distances requires communication between nodes at distance $\Omega(\log n)$. In some initially incorrect state, those hyper-node distances may be locally consistent, so this computation must run perpetually. This is another reason why our algorithm is talkative.

3.2 General description

3.2.1 Notation and preliminaries

The variable that stores the parent node of v, denoted by par_v, actually stores the port number of the edge connecting v to its parent node, or $\emptyset$ if v has no parent. In the case of n-node rings, $\mathsf{par}_v \in \{\emptyset, 0, 1\}$ for every v. In a legitimate configuration, the subgraph of C_n induced by the parent relation must be a tree. The presence of more than one tree, or a cycle, corresponds to illegitimate configurations. We denote by N_v the set of the two neighbors of v in C_n, for any node $v \in V$.

The distance of node v to its root, denoted by d_v, takes values in $\{0, 1, ..., c\lfloor \log n \rfloor\}$. We have $\mathsf{d}_v = 0$ if v is a root of some tree (*i.e.,* a candidate leader) induced by the parent relationship, and $\mathsf{d}_v \geq 1$ otherwise (*i.e.*, if v is passive). Note that we only assume that variable d_v can hold *at least* (and not *exactly*) $(c\lfloor \log n \rfloor)+1$ different values, since nodes are not aware of n but only use an estimate of n that is computed by our algorithm. The children of a node v are the nodes whose distance is equal to $\mathsf{d}_v = 1 + (\mathsf{d}_w \bmod c\lfloor \log n \rfloor)$ and whose variables are readable by v using $\mathtt{Ch}(v)$. Then, it is possible for a node u to be the child of two nodes v and w. However, at any given time, u has a single parent. In a legitimate configuration, the overall structure that is deduced from the parent relationship is eventually a (possibly evolving) tree rooted on the node with maximum identifier and spanning the whole network.

To detect cycles, we use four variables. First, each node v maintains dB_v (introduced in the Section 2) for constructing a distributed integer stored in a hyper-node. The second variable, $\mathsf{Add}_v \in \{+, \mathrm{ok}, \emptyset\}$, is used for performing additions involving values stored distributedly on hyper-nodes. The third variable, PL_v (for pipeline), is used to send the result of an addition to the children hyper-nodes of v's hyper-node. Finally, the fourth variable, HC_v (for Hyper-node Checking), is dedicated to checking the accuracy of each hyper-node bit. Both PL_v and HC_v are either empty, or composed of a pair of variables $(x, y) \in \{1, ..., c\lfloor \log n \rfloor\} \times \{0, 1\}$.

For constructing the tree rooted at the node with highest identity, we use three additional variables. After convergence, we expect the leader to be the unique node with distance zero, and to be the root of an inward directed spanning tree of the ring, where the edges of the tree are defined by the parent relation. To satisfy the leader election specifications, we introduce the variable $\mathsf{leader}_v \in \{0, 1, 2\}$ whose value is 1 if v is the leader, 2 if the node is frozen (defined in Section 3.2.5) and 0 otherwise. Since we do not assume that the identifiers of every node are encoded using the same number of bits, simply comparing the i-th most significant bit of two nodes is irrelevant. So, we use variable $\widehat{\mathsf{B}}_v$, which represents the most significant bit-position of all the identities present in the ring known to v. This variable is also used at each node v as an estimate of $\lfloor \log n \rfloor$. Only the nodes v whose variable $\widehat{\mathsf{B}}_v$ is equal to the most significant bit of Id_v continue participating in the election. Finally, the variables Phase, Place and Check are the core of the election process. Let r be the root of the tree including node v. Then, the variable Phase_v stores the current phase number i, the variable Place_v stores the bit-position of Id_r at phase i, and the variable Check_v stores a boolean value for controlling updates to Phase_v and Place_v. To make the algorithm more readable, we introduce the variable $\mathsf{Elec}_v = (\mathsf{Phase}_v, \mathsf{Place}_v)$.

3.2.2 The Compact Leader Election algorithm CLE

Algorithm **CLE** is composed of three groups of rules as described in Figure 1:

1. The rules dedicated to the leader election and tree construction: $\mathbb{R}_{\mathtt{Root}}$, $\mathbb{R}_{\mathtt{Passive}}$ and $\mathbb{R}_{\mathtt{Update}}$.

2. The rules dedicated to the hyper-node distances computation and verification: $\mathbb{R}_{\texttt{RootdB}}$, $\mathbb{R}_{\texttt{HypAdd}}$, $\mathbb{R}_{\texttt{HypBroad}}$ and $\mathbb{R}_{\texttt{HypVerif}}$.

3. The rules dedicated to the detection, propagation and correction of errors: $\mathbb{R}_{\texttt{Error}}$ and $\mathbb{R}_{\texttt{Start}}$.

3.2.3 Leader election and tree construction

As previously mentioned, our protocol simultaneously performs leader election and the construction of a spanning tree rooted at the leader. The leader should be the node whose identifier is maximal among all nodes in the ring. Our assumptions regarding identifiers are very weak. In particular, identifiers may be of various sizes (and encoded using a different number of bits at each node), and the total number n of different identifiers is not known to the nodes. In our algorithm, we use the variable $\widehat{\mathsf{B}}_v$ to store the most significant bit-position among all identities present in the ring. We consider a node v to be inconsistent if $\widehat{\mathsf{B}}_v$ does not contain contain this value. During the algorithm execution, only nodes whose identifiers match the most significant bit-position remain candidate leaders. Moreover, only candidate leaders broadcast additional bit-positions during subsequent phases.

The comparison of bits-positions is relevant only if these bits-positions are revealed at the same phase. Hence, we force the system to proceed in phases. If, at phase i, the bit-position ξ_v of node v is smaller than the bit-position ξ_u of a neighboring node u, then node v becomes passive, and v takes u as its parent. It is simple to compare two candidate leaders when these candidate leaders are neighbors. Yet, along with the execution of the algorithm, some nodes become passive, and therefore the remaining candidate leaders can be far away, separated by passive nodes. Each passive node v ($\mathsf{d}_v > 0$) is in a subtree rooted at some candidate leader. Let us now consider one such subtree T_v rooted at a candidate leader v. Whenever v increases its phase from i to $i+1$, and sets the bit-position related to phase $i+1$, all nodes u in T_v must update their variable Elec_u in order to have the same value as Elec_v.

At each phase, trees are merged into larger trees. At the end of phase i, all the nodes in a given tree have the same bit-position, and the leaves of the tree (*i.e.*, nodes v for which the function $\texttt{Ch}(v)$ returns empty), inform their ancestors that the phase is finished. The local variable Check is dedicated to this purpose. Each leaf assigns 1 to its Check variable, and a bottom-up propagation of this control variable eventually reaches the root. In this way, the root learns that the current phase is finished. Each phase results in halving the number of trees, and therefore halving the number of candidate leaders. Remember that, at phase 1, a candidate leader node v publishes $\text{Bit}(1, \mathsf{Id}_v)$, at phase 2, it publishes $\text{Bit}(2, \mathsf{Id}_v)$, etc. So the

$$
\begin{array}{lll}
\mathbb{R}_{\texttt{Error}} \;:\; \texttt{Error}(v) & \rightarrow & \mathcal{F}reeze(v); \\
\mathbb{R}_{\texttt{Start}} \;:\; \neg\texttt{Error}(v) \wedge (\mathsf{leader}_v = 2) \wedge (\forall u \in \mathsf{N}_v.(u \notin \texttt{Ch}(v))) & \rightarrow & \mathcal{S}tart(v); \\
 & & \\
\mathbb{R}_{\texttt{Passive}} \;:\; \neg\texttt{Error}(v) \wedge \;\; \texttt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 2) \wedge \texttt{T.dB}(v) & \rightarrow & \mathcal{P}ass(v); \mathcal{D}B(v) \\
\mathbb{R}_{\texttt{Root}} \;:\; \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v = 1) \wedge \texttt{T.Inc}(v) & \rightarrow & \mathcal{I}nc(v); \\
\mathbb{R}_{\texttt{Update}} \;:\; \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v = 0) \wedge \big(\texttt{T.Down}(v) \vee \texttt{T.Up}(v)\big) & \rightarrow & \mathcal{U}pdate(v); \\
 & & \\
\mathbb{R}_{\texttt{RootdB}} \;:\; \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v = 1) \wedge (\forall u \in \texttt{Ch}(v).(\mathsf{PL}_v = \mathsf{HC}_u)) & \rightarrow & \mathcal{S}tartdB(v); \\
\mathbb{R}_{\texttt{HypAdd}} \;:\; \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 1) \wedge \texttt{Tree}(v) \wedge (\mathsf{Add}_v = \perp) \wedge \texttt{T.Add}(v) & \rightarrow & \mathcal{B}inAdd(v); \\
\mathbb{R}_{\texttt{HypBroad}} \;:\; \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 1) \wedge \texttt{Tree}(v) \wedge (\mathsf{Add}_v \neq \perp) \wedge \texttt{T.Pipe}(v) & \rightarrow & \mathcal{P}ipe(v); \\
\mathbb{R}_{\texttt{HypVerif}} \;:\; \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 1) \wedge \texttt{Tree}(v) \wedge (\texttt{T.Verif}(v) \vee \texttt{CleanHC}(v)) & \rightarrow & \mathcal{V}erif(v);
\end{array}
$$

Figure 1: *Formal description of algorithm* **CLE**.

number of phases is bounded by the number of 1s in the binary representation of the largest identifier in the network. Thus, within at most $\lfloor \log n \rfloor + 1$ phases, a single leader remains. To avoid electing an impostor leader, the (unique) newly elected leader restarts the election at the first phase. This is repeated forever. If an arbitrary initial configuration induces an impostor leader ℓ, either ℓ does not have the most significant bit-position in its identifier, or this impostor leader has its most significant bit-position equal to the most significant bit-position of the (true) leader. In the former case, the error is detected by any node whose most significant bit-position is maximum. In the latter case, the error is detected by at least one node (the true leader), because there exists at least one phase i where the bit-position of the leader is larger than to the bit-position of the impostor.

The process of leader election and spanning tree construction is slowed down by the hyper-node construction and management. When a node v changes its parent, it also changes the value of dB_v, in order to not impact the current construction of the tree. The point is that variable dB_v should be handled with extra care to remain coherent with a tree resulting from merging different trees. To handle this, every candidate leader assigns bits of the distance hyper-node for its descendants in its variable PL. More precisely, if a root v has no children, then v publishes the bit of the distance hyper-node for its future children u with $\mathsf{d}_u = 1$. If root v has children at distance one, then v publishes the bit of the distance hyper-node for its descendants u with $\mathsf{d}_u = 2$.This process is repeated by the node v until the last descendant in its hyper-node reaches $\widehat{\mathsf{B}}_v$. More precisely, the hyper-node distance of a child of a leader r is 0, so r publishes $(1, 0)$ in variable PL for the descendant at distance one, $(2, 0)$ for the descendant at distance two, until $(\widehat{\mathsf{B}}_r, 0)$ for the descendant at distance $\widehat{\mathsf{B}}_r$ (Remember that, after convergence $\widehat{\mathsf{B}}_v = c\lfloor \log n \rfloor$.) A detailed description of this mechanism is presented in Section 4.4. On the other hand, a node cannot change its parent until the tentative new parent publishes the required distance. When the hyper-node adjacent to the root is constructed, the hyper-node assignment and verification process takes care of the assignment of the bits to the nodes inside the hyper-node.

3.2.4 Hyper-nodes distance assignment and verification

Let us consider two hyper-nodes $X = (x_1, x_2, \cdots, x_k)$ and $Y = (y_1, y_2, \cdots, y_k)$, with X the parent of Y. Our technique for assigning and verifying the distance between X and Y is the following. X initiates the verification. For this purpose, X dedicates two local variables at each of its nodes: Add (to perform the increment) and PL (to broadcast the result of this increment inside X). Similarly, Y uses variable HC for receiving the result of the increment.

The increment consists of adding 1 to the hyper-node distance of X. For example, suppose that dB_X is equal to 11 (eleven). Then, its binary representation distributed among all the nodes of X, we have $\mathsf{dB}_X = (1, 0, 1, 1)$. In our data structure, the least significant bit is stored in x_k ($\mathsf{d}_{x_k} = \widehat{\mathsf{B}}_{x_k}$) and the most significant bit is stored in x_1 ($\mathsf{d}_{x_1} = 1$). The increment process starts from x_k to x_1. In our example, if 1 is added to $\mathsf{dB}_{x_4} = 1$, the result generates a carry, so x_4 publishes the result of the increment for y_4 (that is, 0) and publishes the carry for x_3. As a consequence, x_3 adds 1 to $\mathsf{dB}_{x_3} = 1$. This also generates a carry, so x_3 publishes the result of the increment for y_3 (that is, 0) and publishes the carry for x_2. Node x_2 adds 1 to $\mathsf{dB}x_2 = 0$ and, in this case the result does not generate a carry. As a consequence, the increment has been accomplished, and x_2 publishes the result of the increment for y_2 (that is, 1) and notifies the end of the increment. In turn, x_1 publishes the result of the increment for y_1 (that is, 1). The global result obtained with the publications of every node of X is $(1, 1, 0, 0)$, so dB_Y equals 12, which is coherent with $\mathsf{dB}_X = 11$. If one node in Y detects an inconsistency with the result of the publication of X, an error is detected. When the publication of the result by X for Y is

done, X plays the same rule as Y for its hyper-node child.

3.2.5 Detection, propagation and fixing of errors

An error is characterized by the presence of inconsistencies between the values of the variables at a node v and the values of the variables of its neighbors. An error at node v can have impact on its descendants, because the descendants of v have election and hyper-node distance variables that are dependent on those of v. For these reasons, after a node v detects an error, the algorithm cleans v and all of its descendants. The cleaning process is achieved in three steps. Let v denote the node that detects the error. First, v deletes its parent and becomes *frozen* (v sets its variable leader_v to 2.) Note that, a frozen node cannot take a parent. Second, we *freeze* all the descendants of v. Last, the node are cleaned in the reverse direction up to the starting node v.

This approach presents several advantages. After detection of the presence of a cycle by a node v, the cycle is broken (v deletes its parent). It is important to note that frozen nodes continue the hyper-node distance verification process, but not the election process. A frozen node cannot reach its own subtree, due to the cleaning process taking place from the leaves to the root, so a livelock of two cleaning processes is avoided.

4 Detailed description of Algorithm CLE

This section is dedicated to the detailed description of each of the algorithm's rules. In the predicate definitions below, we use $P \equiv b$ to define Boolean b, which is true if and only if predicate P is true.

4.1 Variables

The core variable of Algorithm **CLE** for each node v is leader_v. Upon stabilization, the node v whose identity is maximal has $\mathsf{leader}_v = 1$ and all other nodes u have $\mathsf{leader}_u = 0$.

The election process is carried out through the publication, using variable Elec, of the successive bit-positions of competing nodes. Variable Elec_v is a pair $(\mathsf{Phase}_v, \mathsf{Place}_v)$, whose components denote the current phase i and the corresponding bit-position to be revealed by a candidate leader during phase i, respectively. Variable Check_v is employed to control the broadcast of Elec_v. Whenever Elec_v becomes smaller (in lexicographic order) than Elec_u, for some neighbor u of v, v becomes passive. Variable Hyp_v is a tuple

$$(\mathsf{dB}_v, \mathsf{Add}_v, \mathsf{HC}_v, \mathsf{PL}_v)$$

containing hyper-node distance assignment and verification variables. Consider two hyper-nodes X and Y, such that X is the parent of Y. If v is an element of X, dB_v is one bit of the binary representation of the distance of X, Add_v is used to increment the distance, PL_v is employed to broadcast the result of this increment inside X, and HC_v is employed to receive the result of the increment in Y.

4.2 Rules $\mathbb{R}_{\texttt{Error}}$ and $\mathbb{R}_{\texttt{Start}}$

Rule $\mathbb{R}_{\texttt{Error}}$

$$\mathbb{R}_{\texttt{Error}} : \texttt{Error}(v) \rightarrow \mathcal{F}reeze(v);$$

This rule is dedicated to the detection of an error by node v. When an error is detected by v, it deletes its parent and sets leader_v to 2 (by executing the $\mathcal{F}reeze(v)$(4)). A *frozen* node is a node that executed $\mathcal{F}reeze$.

$$\texttt{Error}(v) \equiv \texttt{T.Er}(v) \vee \big((\mathsf{leader}_v = 0) \wedge (\mathsf{leader}_{\mathsf{par}_v} = 2)\big) \tag{1}$$

The predicate $\texttt{Error}(v)$ is composed of two groups. The predicate $\texttt{T.Er}(v)$ is dedicated to detecting an error in node v and the second part of the guard is dedicated to freezing the tree rooted at v after v detected an error. There exist three types of errors: *(i)* the errors that are generated by the election variables, *(ii)* the errors that are generated by the hyper-node distance verification variables, and *(iii)* the errors that are generated by the frozen nodes. As the precise description of these errors requires explaining in detail the Election and Hyper-node distance verification predicates, additional dedicated explanations are delegated to the end of the relevant subsections.

$$\texttt{T.Er}(v) \equiv \texttt{Er}_{\texttt{Elec}}(v) \vee \texttt{Er}_{\texttt{Hyper}}(v) \vee \texttt{Er}_{\texttt{Freeze}}(v) \tag{2}$$

Normally, a frozen node v (*i.e.*, such that $\mathsf{leader}_v = 2$) has either no parent or a frozen parent. Otherwise, the algorithm detects an error. We denote this error by $\texttt{Er}_{\texttt{Freeze}}(v)$:

$$\texttt{Er}_{\texttt{Freeze}}(v) \equiv (\mathsf{leader}_v = 2) \wedge \big((\mathsf{par}_v = \emptyset) \vee (\mathsf{leader}_{\mathsf{par}_v} \neq 2)\big) \tag{3}$$

Let us now explain Command $\mathcal{F}reeze(v)$. A node v that detects an error ($\texttt{T.Er}(v) = \mathit{true}$) becomes frozen ($\mathsf{leader}_v := 2$), and also deletes its parent $\mathsf{par}_v := \emptyset$. Deleting the parent is extremely important, because it break a cycle when it is detected. An unfrozen v with a frozen parent assigns $\mathsf{leader}_v = 2$ but leaves its parent unchanged. The other variables are left unchanged to permit cycle detection.

$$\mathcal{F}reeze(v) : \begin{cases} \mathsf{leader}_v := 2; \mathsf{par}_v := \emptyset & \text{if } \texttt{T.Er}(v) \\ \mathsf{leader}_v := 2; & \text{otherwise} \end{cases} \tag{4}$$

Rule $\mathbb{R}_{\texttt{Start}}$

$$\mathbb{R}_{\texttt{Start}} : \neg\texttt{Error}(v) \wedge (\mathsf{leader}_v = 2) \wedge (\forall u \in \mathsf{N}_v.(u \notin \texttt{Ch}(v))) \rightarrow \mathcal{S}tart(v);$$

A frozen node v with no child (see Function $\texttt{Ch}(v)$(22)) should reset its variables. Note that a frozen node cannot participate in the election process. A frozen tree resets its variables from the leaves to the root. When a node v resets its variables, it becomes a candidate leader ($\mathsf{leader}_v := 1$ and $\mathsf{par}_v := \emptyset$), its election variable is set to correspond to its first bit-position ($\mathsf{Elec}_v := (1, \widehat{\mathsf{B}}_v)$). The control variable $\mathsf{Check}_v = 0$ means that v starts broadcasting in its tree if has any children. The value $(1, 0)$ in Hyp_v is used to indicate that the children u of v ($\mathsf{d}_u = 1$) must take for hyper-node distance the bit 0 ($\mathsf{dB}_u = 0$). The $\mathcal{S}tart$ command consists of the following:

$$\begin{aligned} \mathcal{S}tart(v) : \quad & \mathsf{leader}_v := 1; \mathsf{par}_v := \emptyset; \\ & \widehat{\mathsf{B}}_v := \mathit{Bit}(1, \mathsf{Id}_v); \mathsf{Elec}_v := (1, \widehat{\mathsf{B}}_v); \mathsf{Check}_v := 0; \\ & \mathsf{d}_v := 0; \mathsf{Hyp}_v := (0, \bot, (\bot, \bot), (1, 0)) \end{aligned} \tag{5}$$

4.3 Detailed description of Election rules:

This subsection is dedicated to the detailed description of each of our algorithm's rules, predicates, and functions related to the leader election. Its proof of correctness is delegated to Section 5.

4.3.1 Rules and variables for leader election:

The election process is based on the repeated publication of bit-positions of the nodes identities. As nodes may have identifiers spanning a different number of bits, the node whose identity is maximum is necessarily one of those whose identity uses the most number of bits (removing useless prefix zeroes). For the purpose of publishing those bit-positions, we use variable Elec. If a node v has a neighbor u with a better election variable, v becomes passive (see details of rule $\mathbb{R}_{\texttt{Passive}}$ in Subsection 4.3.3), and joins u's tree. Only the election variables of roots (*i.e.*, candidate leaders) are useful for the election process, however roots may be separated by passive nodes, making direct comparison more difficult. Let us consider two roots r_1 and r_2 that are separated by passive nodes (see Figure 2). The passive nodes between r_1 and r_2 are either in tree $\mathtt{T}_1$ (the tree rooted at r_1) or in tree $\mathtt{T}_2$ (the tree rooted at r_2). Let us denote by ℓ_1 the leaf of tree $\mathtt{T}_1$ and by ℓ_2 the leaf of tree $\mathtt{T}_2$, between r_1 and r_2. Note that ℓ_1 and ℓ_2 are neighbors. As a consequence, if all passive nodes in $\mathtt{T}_1$ have the same election variable as r_1 and all passive nodes in $\mathtt{T}_2$ have the same election variable as r_2, then there exist two neighbors ℓ_1 and ℓ_2 able to compare the election variables of r_1 and r_2. Rule $\mathbb{R}_{\texttt{Update}}$ is dedicated to broadcasting the election variable of a root r in its tree $\mathtt{T}_r$. So, $\mathbb{R}_{\texttt{Update}}$ updates the election variable of r_1 down to ℓ_1, and that of r_2 down to ℓ_2, so that their values can be compared (see details of rule $\mathbb{R}_{\texttt{Update}}$ in Subsection 4.3.5).

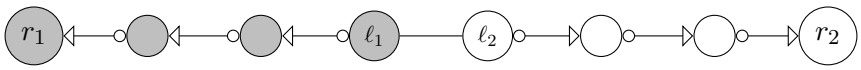
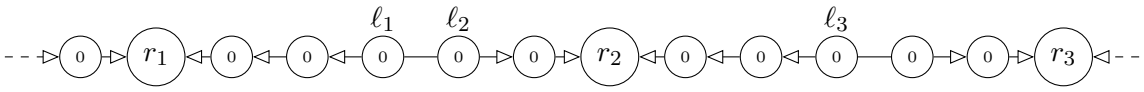


Figure 2: Separated roots

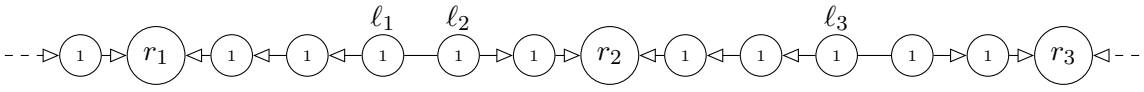
During the broadcast of the election variable from the root down the tree, when a child u of node v takes the election value of v it sets its variable Check to zero (see Figure 3(a)). If the the election variable of r_1 is equal to the election variable of r_2, ℓ_1 and ℓ_2 both set their variable Check to one. This assignment is then propagated from the leaves to the roots (see Figure 3(b)). The broadcast and convergecast are performed by rule $\mathbb{R}_{\texttt{Update}}$. If three trees $\mathtt{T}_1$, $\mathtt{T}_2$ and $\mathtt{T}_3$ have the same election variables, the two neighbors of r_2 have $\mathsf{Check} = 1$, as a consequence r_2 must publish its following bit-position to continue the election process. This action is performed by rule $\mathbb{R}_{\texttt{Root}}$, which increases the phase and publishes the corresponding bit-position (see details of rule $\mathbb{R}_{\texttt{Root}}$ in Subsection 4.3.4 and Figure 3 (b and c)). The election variables of node v are *better* than the election variables of node u if and only if:

$$(\widehat{\mathsf{B}}_v > \widehat{\mathsf{B}}_u) \vee \Big((\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_u) \wedge (\mathsf{Phase}_v = \mathsf{Phase}_u) \wedge (\mathsf{Place}_v > \mathsf{Place}_u)\Big)$$

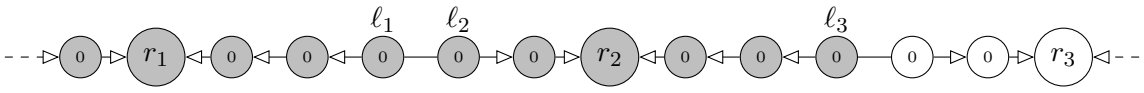
If the election variables of r_2 is better than the election variables of r_3, tree $\mathtt{T}_3$ is merged tree $\mathtt{T}_2$ (see Figure 3(d)). Rule $\mathbb{R}_{\texttt{Passive}}$ is activated when a node has a neighbor with a better election variables. The different trees merge until a single tree (rooted at the node with maximum identity) remains.



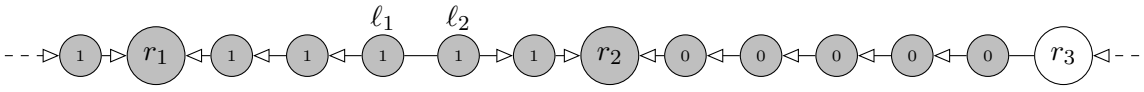
(a) Broadcast of election variables. $\mathtt{T}_1, \mathtt{T}_2$ and $\mathtt{T}_3$ have the same election variables.



(b) Convergecast: $\mathtt{T}_1, \mathtt{T}_2$ and $\mathtt{T}_3$ have the same election variables.



(c) Broadcast of the new election variables: Election variables of $\mathtt{T}_1$ and $\mathtt{T}_2$ are better than that of $\mathtt{T}_3$.



(d) Merging of $\mathtt{T}_2$ and $\mathtt{T}_3$. Election variables of $\mathtt{T}_1$ and $\mathtt{T}_2$ are better than that of $\mathtt{T}_3$

Figure 3: Comparison of election variables: big nodes denote the roots, small nodes denote the passive ones. The number inside a passive node denotes the value of variable Check. The election variables in the shaded nodes are higher than those of the non-shaded nodes.

4.3.2 States of nodes:

We first describe the predicates corresponding to a root and a passive node. A root v is a candidate leader (*i.e.*, $\mathsf{leader}_v = 1$), it doesn't have any parent (*i.e*, $\mathsf{par}_v = \emptyset$), and its distance to the root is equal to zero (*i.e.*, $\mathsf{d}_v = 0$).

Moreover, a root v has its most significant bit-position equal to the first bit-position of its identifier (*i.e.*, $\widehat{\mathsf{B}}_v = \mathrm{Bit}(1, \mathsf{Id}_v)$), and $\mathsf{Place}_v = \mathrm{Bit}(\mathsf{Phase}_v, \mathsf{Id}_v)$. Note that variable Check_v, which is dedicated to controlling the update of election variables, has value zero (see Command $\mathcal{S}tart$). Indeed, a zero denotes a broadcast phase, and a root is always in such a phase. Then, either all its descendants broadcast their election variables, or convergecast ($\mathsf{Check}_u = 1$, with u a child of the root) leading the root to publish new election variables. The bit published at a root for each hyper-node child is kept at 0.

The predicate $\mathtt{Root}$ consists of the following:

$$\begin{aligned} \mathtt{Root}(v) \;\equiv\; & (\mathsf{leader}_v = 1) \wedge (\mathsf{par}_v = \emptyset) \wedge (\widehat{\mathsf{B}}_v = \mathrm{Bit}(1, \mathsf{Id}_v)) \wedge \\ & (\mathsf{Place}_v = \mathrm{Bit}(\mathsf{Phase}_v, \mathsf{Id}_v)) \wedge (\mathsf{d}_v = 0) \wedge (\mathsf{Check}_v = 0) \wedge (\mathsf{PL}_v[1] = 0) \end{aligned} \tag{6}$$

A passive node v is not a candidate leader ($\mathsf{leader}_v = 0$), and it has a parent (par_v is equal to a port number of v.) The most significant bit-position of v is greater than or equal to the first bit-position of its identity. However, a passive node has its distance's variable different from zero :

$$\mathtt{PassNd}(v) \equiv \quad (\mathsf{leader}_v = 0) \wedge (\mathsf{par}_v \in \{0, 1\} \wedge (\widehat{\mathsf{B}}_v \geq \mathrm{Bit}(1, \mathsf{Id}_v)) \wedge (\mathsf{d}_v > 0)) \tag{7}$$

4.3.3 Detailed description of the $\mathbb{R}_{\mathtt{Passive}}$ rule:

$$\mathbb{R}_{\mathtt{Passive}} : \neg\mathtt{Error}(v) \wedge \mathtt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 2) \wedge \mathtt{T.dB}(v) \rightarrow \mathcal{P}ass(v); \mathcal{D}B(v)$$

Predicate $\texttt{T.dB}(v)$ is dedicated to monitoring assigment to variable dB. A node v may execute rule $\mathbb{R}_{\texttt{Passive}}$ if its tentative parent publishes variable dB to v's intent (see predicate $\texttt{T.dB}(15)$). More precisely, if v must adopt distance d (the distance of its parent plus one), then its parent must publish the bit dedicated to the hyper-node distance at the distance d. Rule $\mathbb{R}_{\texttt{Passive}}$ is enabled by a root or a passive node (since $\mathsf{leader}_v \neq 2$). Such a node v is enabled for rule $\mathbb{R}_{\texttt{Passive}}$ if at least one of its neighbors u has a better value for the election process (if its two neighbors qualify, v chooses the one with the best value, see Function $\texttt{Best}(v)(13)$). Moreover, to avoid the creation of an error, v makes sure that all its children have assigned v's election variable to their own (election variable).

$$\texttt{T.Best}(v) \equiv (\mathsf{leader}_{\mathsf{par}_v} \neq 2) \wedge \texttt{Best}(v) \wedge (\forall u \in \texttt{Ch}(v).(\mathsf{Elec}_u = \mathsf{Elec}_v)) \tag{8}$$

There exist three categories of nodes with better values. First, the nodes with a most significant bit-position greater than the most significant bit-position of node v.

$$\texttt{NgSupBs}(v) = \{u \in \mathsf{N}_v : (\mathsf{leader}_u \neq 2) \wedge (\widehat{\mathsf{B}}_u > \widehat{\mathsf{B}}_v)\} \tag{9}$$

Second, the neighbors with the same most significant bit-position as v (*i.e.*, $\widehat{\mathsf{B}}_u = \widehat{\mathsf{B}}_v$) that are in the same phase as v (*i.e.*, $\mathsf{Phase}_u = \mathsf{Phase}_v$), but with a greater bit-position than that of v (*i.e.*, $\mathsf{Place}_u > \mathsf{Place}_v$).

$$\texttt{NgSupBp}(v) = \{u \in \mathsf{N}_v : (\mathsf{leader}_u \neq 2) \wedge (\widehat{\mathsf{B}}_u = \widehat{\mathsf{B}}_v) \wedge (\mathsf{Phase}_u = \mathsf{Phase}_v) \wedge (\mathsf{Place}_u > \mathsf{Place}_v)\} \tag{10}$$

The last category is more intricate, and deserves further explanations. Indeed, our algorithm does not synchronize phases throughout the network, and due to our weakly fair daemon assumption, it is possible that the phase of one tree does not match the phase of another tree. Let us consider k consecutive trees in a ring, denoted by $\texttt{T}_1, \texttt{T}_2, \ldots, \texttt{T}_k$ and let us consider the following scenario represented in Figure 4. Let c be a configuration where all the trees are in the same phase i. If the election values of all trees at the same phase are equal, then all tree roots can increase their phase to $i+1$. Let us suppose that node r_1, the root of $\texttt{T}_1$, is not activated by the weakly fair daemon and remains at phase i. Now $\texttt{T}_2, \ldots, \texttt{T}_k$ are in phase $i+1$, if $\texttt{T}_2, \ldots, \texttt{T}_k$ have the same bit-position in phase $i+1$, $\texttt{T}_3, \ldots, \texttt{T}_k$ can increase their phases. But $\texttt{T}_2$ cannot increase its phase, because its leaf between $\texttt{T}_1$ and $\texttt{T}_1$ does not have the same election variable as in $\texttt{T}_2$. This process can be repeated, until no root can increase its phase anymore.

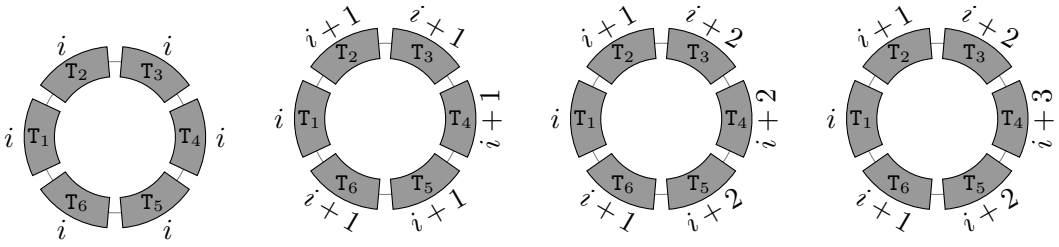


Figure 4: Phases

If the election variable of $\texttt{T}_1$ at phase $i+1$ is better than the election variable of $\texttt{T}_2$ at phase $i+1$ then the election variable of $\texttt{T}_1$ is also better than all election variables of $\texttt{T}_3, \ldots, \texttt{T}_k$

at phase $i+1$ (otherwise the phases of these trees cannot increase). As a consequence, in a normal election process, if a neighbor u of v has a smaller phase and $\mathsf{Check}=0$ then u is better with respect to the election than v. Remember that $\mathsf{Check}=0$ corresponds to the broadcast of the new election variable of the root, and $\mathsf{Check}=1$ corresponds to the converge-cast if two neighbors trees have the same election variable.

$$\texttt{NgInfPhC}(v) = \{u \in \mathsf{N}_v : (\mathsf{leader}_u \neq 2) \wedge (\widehat{\mathsf{B}}_u = \widehat{\mathsf{B}}_v) \wedge (\mathsf{Phase}_u < \mathsf{Phase}_v) \wedge (\mathsf{Check}_u = 0)\} \quad (11)$$

To summarize, the set of better values is as follows:

$$\texttt{Better}(v) = \texttt{NgSupBs}(v) \cup \texttt{NgInfPhC}(v) \cup \texttt{SupBp}(v) \quad (12)$$

For our purpose, we need the best possible value among the values that improve the election variable. The neighbors with a most significant bit-position greater than the most significant bit-position of node v have better values for the election process, so among these nodes we choose one with the maximum most significant bit-position. Then, according to the description of Function 11, nodes in `NgInfPhC` are better than nodes in `NgSupBp`. So, if $\texttt{NgSupBs}(v) = \emptyset$, we choose in `NgInfPhC` the node with the smallest phase. Finally, if $\texttt{NgSupBs}(v) = \emptyset$ and $\texttt{NgInfPhC} = \emptyset$, we choose a node in the same phase as v but with a greater bit-position. In case of several choices, we choose the one with the minimum port number. Function `Best` below directly follows from these explanations:

$$\texttt{Best}(v) = \begin{cases} \min\{\texttt{port}_u : \widehat{\mathsf{B}}_u = \max\{\widehat{\mathsf{B}}_w : w \in \texttt{NgSupBs}(v)\}\} & \text{if } \texttt{NgSupBs}(v) \neq \emptyset \\ \min\{\texttt{port}_u : \mathsf{Phase}_u = \min\{\mathsf{Phase}_w : w \in \texttt{NgInfPhC}(v)\}\} & \text{if } \texttt{NgSupBs}(v) = \emptyset \wedge \texttt{NgInfPhC}(v) \neq \emptyset \\ \min\{\texttt{port}_u : \mathsf{Place}_u = \max\{\mathsf{Place}_w : w \in \texttt{NgSupBp}(v)\}\} & \text{otherwise} \end{cases} \quad (13)$$

A node enabled by rule $\mathbb{R}_{\texttt{Passive}}$ assigns its best neighbor value, and resets its hyper-node variables to avoid inconsistencies. This command makes use of the δ^+ function, that returns the appropriate value of updating the distance of the node (defined in Equation 20).

$$\begin{array}{lll} \mathcal{P}ass(v): & \mathsf{leader}_v := 0; \mathsf{par}_v := \texttt{Best}(v); & /* \text{ leader variables } */ \\ & \widehat{\mathsf{B}}_v := \widehat{\mathsf{B}}_{\mathsf{par}_v}; \mathsf{Elec}_v := \mathsf{Elec}_{\mathsf{par}_v}; \mathsf{Check}_u = 0; & /* \text{ election variables } */ \\ & \mathsf{d}_v := \delta^+(\mathsf{par}_v); \mathsf{Hyp}_v := (0, \bot, (\bot, \bot), (\bot, \bot)) & /* \text{ hyper-node variables } */ \end{array} \quad (14)$$

Moreover, a node with a best election variable must publish the bit dedicated to the hyper-node distance (*i.e.*, dB). Let u denote the node with the best election variable. If u's distance is equal to $\widehat{\mathsf{B}}_v$, u is the last node of a hyper-node, and v becomes the first node of a new hyper-node. In this case, for the hyper-node distance bit, v checks variable PL, whose value must be equal to the hyper-node distance bit for a node at distance one. Otherwise, if u's distance is inferior to $\widehat{\mathsf{B}}_v$, v joins the hyper-node of u. In this case, v checks variable HC of u. We later detail the role of variables PL and HC in Section 4.4. This command makes use of the π^+ function, that returns the next phase (defined in Equation 19).

$$\texttt{T.dB}(v) \equiv [(\mathsf{d}_{\texttt{Best}_v} = \widehat{\mathsf{B}}_v) \wedge (\mathsf{PL}_{\mathsf{par}_v}[0] = \pi^+(\texttt{Best}(v)))] \vee [(\mathsf{d}_{\texttt{Best}_v} < \widehat{\mathsf{B}}_v) \wedge (\mathsf{HC}_{\mathsf{par}_v}[0] = \pi^+(\texttt{Best}(v)))] \quad (15)$$

Command $\mathcal{D}B(v)$ assigns a bit to the hyper-node distance bit variable, according to PL if node v has a distance equal to one, and otherwise according to HC.

$$\mathcal{D}B(v) : \begin{cases} \mathsf{dB}_v := \mathsf{PL}_{\mathsf{par}_v}[1] & \text{if } (\mathsf{d}_v = 1) \\ \mathsf{dB}_v := \mathsf{HC}_{\mathsf{par}_v}[1] & \text{if } (\mathsf{d}_v > 1) \end{cases} \tag{16}$$

Note that all remaining rules are enabled only if there exist no better neighbor ($\neg\mathtt{T.Best}(v)$ holds in all of the following rules).

In the last case of `Best`, only a node with a greatest most significant bit-position can detect an error. For example, in Figure 4(d), if the root r_4 of tree $\mathtt{T}_4$ reaches phase $i+3$, that implies that $\mathsf{Elec}_{r_2} = (\widehat{\mathsf{B}}_{r_2}, i+1, \mathsf{Place}_{r_2})$ is equal to $\mathsf{Elec}_{r_4} = (\widehat{\mathsf{B}}_{r_4}, i+1, \mathsf{Place}_{r_4})$, otherwise r_4 would not have been able to increase its phase until $i+3$ (see the explanations above in Figure 4). So, if $\mathsf{Elec}_{r_1} > \mathsf{Elec}_{r_2}$ at phase $i+1$, then $\mathsf{Elec}_{r_1} > \mathsf{Elec}_{r_4}$ at phase $i+1$. As a consequence, if r_4 has a neighbor $u \in \mathtt{NgInfPhC}(r_4)$, then r_4 detects an error if the bit-position of u is smaller than its own bit-position at the same phase as u:

$$\mathtt{Er}_{\mathtt{Place}}(v) \equiv (\mathsf{leader}_v \in \{0,1\}) \wedge (\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_{\mathtt{Best}(v)}) \wedge (\mathsf{Place}_{\mathtt{Best}(v)} < \mathrm{Bit}(\mathsf{Phase}_v, \mathsf{Id}_v)) \tag{17}$$

Let us consider the following case with two nodes u and v. Suppose that at phase i, the bit positions of u and v differ. If v has a bit position larger than a bit position of u in phase i, the identity of v is higher than the identity of u, and the comparisons between the bit positions in phases greater than i are meaningless.

A passive node v has a *coherent* parent par_v if par_v has a better value (*i.e.*, $\mathsf{par}_v \in \mathtt{Better}(v)$), or if p_v's phase is equal to that of v (and $\mathsf{Elec}_v = \mathsf{Elec}_{\mathsf{par}_v}$) or is immediately superior (see Function π^+). This last possibility corresponds to the activation of command $\mathcal{U}pdate$ by p_v (see $\mathtt{Normal}(v)$).

$$\mathtt{Normal}(v) \equiv (\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_{\mathsf{par}_v}) \wedge ((\mathsf{Elec}_v = \mathsf{Elec}_{\mathsf{par}_v}) \vee (\mathsf{Phase}_v = \pi^-(\mathsf{par}_v))) \tag{18}$$

Where $\pi^-(v)$ and $\pi^+(v)$ denote the previous and next phases of v, respectively. When a phase reaches the maximum (that is, $\widehat{\mathsf{B}}$), the algorithm starts again at phase one.

$$\pi^+(v) = \begin{cases} \mathsf{Phase}_v + 1 & \text{if } (\mathsf{Phase}_v < \widehat{\mathsf{B}}_v) \\ 1 & \text{if } (\mathsf{Phase}_v = \widehat{\mathsf{B}}_v) \end{cases} \qquad \pi^-(v) = \begin{cases} \mathsf{Phase}_v - 1 & \text{if } (\mathsf{Phase}_v > 1) \\ \widehat{\mathsf{B}}_v & \text{if } (\mathsf{Phase}_v = 1) \end{cases} \tag{19}$$

An additional constraint for a passive node v is that its distance is equal to that of its parents plus one. Recall that distance zero is for the roots. As a consequence we use the distances from one to $\widehat{\mathsf{B}}_v$ for the passive nodes. Normally, a passive node v takes as distance the distance of its parent plus one, but if its parent has distance $\widehat{\mathsf{B}}_v$, v assigns one to its distance variable (see the function δ^+).

$$\delta^+(v) = \begin{cases} \mathsf{d}_v + 1 & \text{if } (\mathsf{d}_{\mathsf{par}_v} < \widehat{\mathsf{B}}_v) \\ 1 & \text{if } (\mathsf{d}_{\mathsf{par}_v} = \widehat{\mathsf{B}}_v) \end{cases} \tag{20}$$

$$\mathtt{CohP}(v) \quad \equiv \quad (\mathsf{par}_v \in \mathtt{Better}(v)) \vee \big(\mathtt{Normal}(v) \wedge (\mathsf{d}_v = \delta^+(\mathsf{par}_v))\big) \tag{21}$$

Let us now define the set of children of a node (see Function $\mathtt{Ch}(v)$(22)). A child u of node v is not a root (*i.e.*, $\mathsf{leader}_u \neq 1$), u has a parent $\mathsf{par}_u \neq \emptyset$, and u's distance equal that of v plus

one (*i.e.*, $\mathsf{d}_u = \delta^+(v)$). More precisely, there exist two types of children for node v. The first kind is a child u with the same election variables as v (*i.e.*, $\mathsf{Elec}_u = \mathsf{Elec}_v$). The second kind is a child with the same most significant bit-position (*i.e.*, $\widehat{\mathsf{B}}_u = \widehat{\mathsf{B}}_v$) that did not already update its phase (*i.e.*, $\mathsf{Phase}_u = \pi^-(v)$).

$$\mathtt{Ch}(v) \quad = \quad \{u \in \mathsf{N}(v) : (\mathsf{leader}_u \neq 1) \wedge (\mathsf{par}_u \neq \emptyset) \wedge (\mathsf{d}_u = \delta^+(v)) \wedge \mathtt{Normal}(u)\} \tag{22}$$

4.3.4 Detailed description of the rule $\mathbb{R}_{\mathtt{Root}}$:

$$\mathbb{R}_{\mathtt{Root}} : \neg\mathtt{Error}(v) \wedge \neg\mathtt{T.Best}(v) \wedge (\mathsf{leader}_v = 1) \wedge \mathtt{T.Inc}(v) \rightarrow \mathcal{I}nc(v);$$

Let us now explain predicate $\mathtt{T.Inc}(v)$: A root v (*i.e.*, $\mathsf{leader}_v = 1$ in a guard of rule $\mathbb{R}_{\mathtt{Root}}$) can increase its phase if and only if all its children have the same election variable as v and advertise that the election broadcast is finished (*i.e.*, $\mathsf{Check}_u = 1$). The other neighbors must have the same election variable as v, or be in the next phase.

$$\begin{aligned}\mathtt{T.Inc}(v) \quad \equiv \quad & (\forall u \in \mathsf{N}_v.(\widehat{\mathsf{B}}_u = \widehat{\mathsf{B}}_v)) \wedge (\mathsf{N}_v = \{u \in \mathtt{Ch}(v) : (\mathsf{Elec}_u = \mathsf{Elec}_v) \wedge (\mathsf{Check}_u = 1)\} \\ & \cup\{u \in \mathsf{N}_v \setminus \mathtt{Ch}(v) : (\mathsf{Elec}_u = \mathsf{Elec}_v) \vee (\mathsf{Phase}_u = \pi^+(v))\big)\}\end{aligned} \tag{23}$$

A root v that is enabled for rule $\mathbb{R}_{\mathtt{Root}}$ increases its phase and assigns its bit-position variable a value that conforms to the new phase, namely v executes Command $\mathcal{I}nc(v)$:

$$\mathcal{I}nc(v) : \mathsf{Elec}_v := (\pi^+(v), \mathrm{Bit}(\pi^+(v), \mathsf{Id}_v)); \tag{24}$$

4.3.5 Detailed description of the rule $\mathbb{R}_{\mathtt{Update}}$:

$$\mathbb{R}_{\mathtt{Update}} : \neg\mathtt{Error}(v) \wedge \neg\mathtt{T.Best}(v) \wedge (\mathsf{leader}_v = 0) \wedge \big(\mathtt{T.Down}(v) \vee \mathtt{T.Up}(v)\big) \rightarrow \mathcal{U}pdate(v);$$

This rule is dedicated to the updating of trees. When a root increases its phase, all its descendants must update their election variables according to the root's election variable (that is, if a passive node v has $\mathsf{Check}_v = 1$ and its parent is in the next phase).

$$\mathtt{T.Down}(v) \quad \equiv \quad (\mathsf{leader}_{\mathsf{par}_v} \neq 2) \wedge (\mathsf{Check}_v = 1) \wedge (\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_{\mathsf{par}_v}) \wedge (\mathsf{Phase}_{\mathsf{par}_v} = \pi^+(v)) \tag{25}$$

Predicate $\mathtt{T.Up}(v)$ is employed to monitor the end of the updating. When all the descendants of a root r have updated their variables, a wave starts from the leaves to r. The purpose of this wave is to indicate to r that the updating has ended and that r can now increase its phase. More precisely, the leaf ℓ assigns $\mathsf{Check}_\ell = 1$, and when all children u of a node v have $\mathsf{Check}_u = 1$, then v assigns $\mathsf{Check}_v = 1$. Note that all neighbors u of a leaf ℓ must have the same election variable, otherwise ℓ is a better value for u, and u can join ℓ's tree.

$$\mathtt{T.Up}(v) \equiv (\mathsf{leader}_{\mathsf{par}_v} \neq 2) \wedge (\mathsf{Check}_v = 0) \wedge (\forall u \in \mathsf{N}_v.(\mathsf{Elec}_u = \mathsf{Elec}_v)) \wedge (\forall u \in \mathtt{Ch}(v).(\mathsf{Check}_u = 1)) \tag{26}$$

A passive node enabled for rule $\mathbb{R}_{\mathtt{Update}}$ either copies its parent's variables, or executes the convergecast part of the wave, namely executes Command $\mathcal{U}pdate(v)$ below:

$$\mathcal{U}pdate(v) : \begin{cases} \mathsf{Elec}_v := \mathsf{Elec}_{\mathsf{par}_v}; \mathsf{Check}_v := 0; & \text{if } \mathsf{Elec}_v \neq \mathsf{Elec}_{\mathsf{par}_v} \\ \mathsf{Check}_v := 1 & \text{otherwise} \end{cases} \tag{27}$$

It is possible for node v to use variable Check_v to detect an error: if $\mathsf{Check}_v = 1$ and there exists at least one child u such that $\mathsf{Check}_u = 0$ (see line "error of child" in predicate 28: $\mathtt{Er}_{\mathtt{Check}}$), or if its parent has $\mathsf{Check}_{\mathsf{par}_v} = 0$ (see line "error of parent" in predicate 28: $\mathtt{Er}_{\mathtt{Check}}$). Moreover, a leaf v with $\mathsf{Check}_v = 1$ detects an error if at least one of its passive neighbors has an election variable with value different than its (see line "error of leaf" in predicate 28: $\mathtt{Er}_{\mathtt{Check}}$).

$$\begin{array}{rcll} \mathtt{Er}_{\mathtt{Check}}(v) & \equiv & (\mathsf{leader}_v \neq 1) \wedge (\mathsf{par}_v \neq \emptyset) \wedge \big((\mathsf{Check}_v = 0) \wedge (\mathsf{Check}_{\mathsf{par}_v} = 1)\big) \vee & \text{/*Error of parent */} \\ & & [(\mathsf{Check}_v = 1) \wedge & \\ & & (\exists u \in \mathtt{Ch}(v).(\mathsf{Elec}_u \neq \mathsf{Elec}_v) \vee (\mathsf{Check}_u = 0)) \vee & \text{/* Error of child */} \\ & & (\exists u \in \mathsf{N}_v.\mathtt{PassNd}(u) \wedge (\mathsf{Elec}_u \neq \mathsf{Elec}_v))] & \text{/* Error of leaf */} \end{array} \tag{28}$$

4.3.6 Errors of election

A node v can only be in three states: *i)* root, *(ii)* passive, or *(iii)* frozen. If v is not in one of these three states, v detects an error.

$$\mathtt{Er}_{\mathtt{Nd}}(v) \equiv (\neg\mathtt{Root}(v) \wedge \neg\mathtt{PassNd}(v) \wedge (\mathsf{leader}_v \in \{0,1\})) \vee (\mathtt{PassNd}(v) \wedge \neg\mathtt{CohP}(v)) \tag{29}$$

To summarize the errors of the election process we use the following predicate:

$$\mathtt{Er}_{\mathtt{Elec}}(v) \equiv \mathtt{Er}_{\mathtt{Nd}}(v) \vee \mathtt{Er}_{\mathtt{Place}}(v) \vee \mathtt{Er}_{\mathtt{Check}}(v) \tag{30}$$

4.4 Detailed description of the rules of Hyper-nodes distance verification:

Let us consider two hyper-nodes X and Y, X being the parent of Y. If v is an element of X, dB_v is one bit of the binary representation of the distance of X. Variable Add is used to perform the binary addition (see details of rule $\mathbb{R}_{\mathtt{HypAdd}}$ in Subsection 4.4.1). The result of the addition is employed to check the correctness of the distance between X and Y. The last node of a hyper-node is the node that starts the addition process. Variable PL is employed to broadcast the result of the addition within X (see details of rule $\mathbb{R}_{\mathtt{HypBroad}}$ in Subsection 4.4.2). Similarly, HC is employed to receive the result of the addition in Y (see details of rule $\mathbb{R}_{\mathtt{HypVerif}}$ in Subsection 4.4.3). We use the same mechanism to verify and to assign the distance of a hyper-node. First, we describe the verification mechanism, and then the rule $\mathbb{R}_{\mathtt{RootdB}}$ that makes use of it. We remark that hyper-node distances are used to detect a cycle. In our setting, if the overlay structure induced by the parent variables par_v, for $v \in V$, forms a cycle C, then the cycle C necessarily contains all nodes. So, it is not necessary to verify the hyper-node distance if one neighbor u of v has no relationship with v (That is, $\mathsf{par}_v \neq u$ and $u \notin \mathtt{Ch}(v)$.) As a consequence, we check distances only if relevant:

$$\mathtt{Tree}(v) \equiv (\forall u \in \mathsf{N}_v.(\mathsf{leader}_u = \mathsf{leader}_v) \wedge \big((\mathsf{par}_v = u) \vee (u \in \mathtt{Ch}(v))\big)) \tag{31}$$

4.4.1 Detailed description of the rule $\mathbb{R}_{\mathtt{HypAdd}}$:

$\mathbb{R}_{\mathtt{HypAdd}} : \neg\mathtt{Error}(v) \wedge \neg\mathtt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 1) \wedge \mathtt{Tree}(v) \wedge (\mathsf{Add}_v = \bot) \wedge \mathtt{T.Add}(v) \rightarrow \mathcal{B}inAdd(v);$

Incrementing the hyper-node distance is described in Subsection 3.2.4. To make reading easier, we present an example as Table 1. The increment process is started bottom-up from non-root nodes (that is, $\mathsf{leader}_v \in \{0,2\}$) whose distance is $\widehat{\mathsf{B}}_v$ to the non-root nodes whose distance is one (see Example of hyper-node, Table 1). The $+$ sign for variable Add represents the start of the increment for the node whose distance is $\widehat{\mathsf{B}}$ (see v_k in Table 1 and predicate $\mathsf{Add}_+(v)$(32)) and the possible carry for the other nodes. More precisely, in our example, $\mathsf{Add}_{v_{k-1}} = +$ and $\mathsf{dB}_{v_{k-1}} = 1$ means (for the parent of node v_{k-1}) that the increment induces a carry (see predicate $\mathsf{Add}_+(v)$(32)). On the other hand, $\mathsf{Add}_{v_{k-2}} = +$ and $\mathsf{dB}_{v_{k-2}} = 0$ means that the increment is finished. As a consequence, v_{k-3} assigns ok to Add. Moreover, all ancestors of v_{k-3} acknowledge the end of the increment by assigning ok to Add (see predicate $\mathsf{Add}_{ok}(v)$(33)). When node v_1 assigns its variable Add, the increment process starts, and all nodes with ok do not change their bit (*i.e.*, dB remains the same), and all nodes with $+$ change their bit (see the last line in Table 1).

	Hyper-node X					
nodes	v_k	v_{k-1}	v_{k-2}	v_{k-3}	...	v_1
d	$\widehat{\mathsf{B}}$	$\widehat{\mathsf{B}}-1$	...	...	...	1
dB	1	1	0	1	1	0
Add	$+$	$+$	$+$	ok	ok	ok
PL	$(\widehat{\mathsf{B}}_v, 0)$	$(\widehat{\mathsf{B}}_v - 1, 0)$	$(..., 1)$	$(..., 1)$	$(2,1)$	$(1,0)$

Table 1: Example of increment

Predicate $\mathsf{Add}_+(v)$ verifies if there exists a carry for v, in other words if all children u have $\mathsf{dB}_u = 1$ and $\mathsf{Add}_u = +$:

$$\mathsf{Add}_+(v) \equiv (\forall u \in \mathtt{Ch}(v).((\mathsf{Add}_u = +) \wedge (\mathsf{dB}_u = 1))) \tag{32}$$

Predicate $\mathsf{Add}_{ok}(v)$ verifies if the increment is finished for v, in other words if all children u have $\mathsf{dB}_u = 1$ and $\mathsf{Add}_u = 0$, or all children u have $\mathsf{Add}(u) = ok$:

$$\mathsf{Add}_{ok}(v) \equiv (\forall u \in \mathtt{Ch}(v).((\mathsf{Add}_u = +) \wedge (\mathsf{dB}_u = 0)) \vee (\mathsf{Add}(u) = ok)) \tag{33}$$

So, v is enabled if $\mathsf{d}_v = \widehat{\mathsf{B}}_v$, or if either there exists a carry for v or the increment is finished. Moreover, we check if the variable dedicated to the publication of the addition result is empty ($\mathsf{PL}_v = (\perp, \perp)$).

$$\mathtt{T.Add}(v) \equiv [\big((\mathsf{d}_v = \widehat{\mathsf{B}}_v) \vee \mathsf{Add}_+(v)\big) \vee \mathsf{Add}_{ok}(v)] \wedge (\mathsf{PL}_v = (\perp, \perp)) \tag{34}$$

Command $\mathcal{B}inAdd(v)$ assigns $+$ or ok to variable Add_v, conforming to predicates $\mathsf{Add}_+(v)$ and $\mathsf{Add}_{ok}(v)$:

$$\mathcal{B}inAdd(v) : \begin{cases} \mathsf{Add}_v := + & \text{if } (\mathsf{d}_v = \widehat{\mathsf{B}}_v) \vee \mathsf{Add}_+(v) \\ \mathsf{Add}_v := ok & \text{if } \mathsf{Add}_{ok}(v) \end{cases} \tag{35}$$

Let us consider the errors involving variable Add. Let two nodes u and v be such that u is a child v. If $\mathsf{Add}_v = +$ and $\mathsf{Add}_u \neq +$, v detects an error. Likewise, if $\mathsf{Add}_v = +$ and $\mathsf{Add}_u = +$ but $\mathsf{dB}_u = 0$, v detects an error. Now, let us consider $\mathsf{Add}_v = ok$. Then there exist two case where v detects an error: *(i)* if $\mathsf{Add}_u = \perp$, or *ii)* if $\mathsf{Add}_u = +$ and $\mathsf{dB}_u = 1$.

$$\begin{array}{rcl}\texttt{Er}_{\texttt{Add}}(v) & \equiv & (\mathsf{leader} \neq 1) \wedge (\mathsf{Add}_v \neq \perp) \wedge (\mathsf{d}_v < \widehat{\mathsf{B}}_v) \wedge \texttt{Tree}(v) \wedge \\ & & \big[\big((\mathsf{Add}_v = +) \wedge (\mathsf{d}_v \neq \widehat{\mathsf{B}}_v) \wedge \neg \mathsf{Add}_+(v)\big) \vee \big((\mathsf{Add}_v = ok) \wedge \neg \mathsf{Add}_{ok}(v)\big)\big] \end{array} \tag{36}$$

Moreover, if during the increment a carry is proposed to node v with $\mathsf{d}_v = 1$, and the bit stored by v is 1, then the increment results in an overflow: as a consequence an error is detected.

$$\texttt{Er}_{\texttt{Overflow}}(v) \equiv \big((\mathsf{d}_v = 1) \wedge (\mathsf{dB}_v = 1) \wedge (\exists u \in \texttt{Ch}(v).(\mathsf{Add}_u = +) \wedge (\mathsf{dB}_u = 1))\big) \tag{37}$$

4.4.2 Detailed description of the rule $\mathbb{R}_{\texttt{HypBroad}}$

$\mathbb{R}_{\texttt{HypBroad}} : \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 1) \wedge \texttt{Tree}(v) \wedge (\mathsf{Add}_v \neq \perp) \wedge \texttt{T.Pipe}(v) \rightarrow \mathcal{P}ipe(v);$

			step 1		step 2		step 3		step 4		step 5		step 6		step 7	
	d	dB	Add	PL	Add	PL	Add	PL	Add	PL	Add	PL	Add	PL	Add	PL
X	1	0	ok	$(\perp,\perp)$	$\perp$	$(1,0)$	$\perp$	$(1,0)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$
	2	1	ok	$(\perp,\perp)$	ok	$(\perp,\perp)$	ok	$(1,0)$	ok	$(1,0)$	$\perp$	$(2,1)$	$\perp$	$(2,1)$	$\perp$	$(\perp,\perp)$
	3	0	$+$	$(\perp,\perp)$	$+$	$(\perp,\perp)$	$+$	$(\perp,\perp)$	$+$	$(1,0)$	$+$	$(1,0)$	$+$	$(2,1)$	$+$	$(2,1)$
	4	1	$+$	$(\perp,\perp)$	$+$	$(\perp,\perp)$	$+$	$(\perp,\perp)$	$+$	$(\perp,\perp)$	$+$	$(1,0)$	$+$	$(1,0)$	$+$	$(1,0)$
	d	dB		HC		HC		HC		HC		HC		HC		HC
Y	1	0		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(1,0)$

Table 2: Example of broadcast of the addition from the hyper-node X to the hyper-node Y.

Predicate `T.Pipe` manages variable PL (see an example in Table 2). Variable PL is used for broadcasting the result of the increment from the node with distance 1 to the node with distance $\widehat{\mathsf{B}}$ (see Table 2). Remember that PL is composed of a pair of elements: a distance, and a bit. In other words, to send the result of the increment, we assign the distance of the recipient to the current broadcast distance, and the corresponding bit in a child hyper-node. So, a node v has three tasks *(i)* publish the result of the increment, *(ii)* broadcast the result of its ancestors, and *iii)* delete variable PL after broadcasting.

$$\texttt{T.Pipe}(v) \equiv \texttt{Publish}(v) \vee \texttt{Pipe}(v) \vee \texttt{CleanPV}(v) \tag{38}$$

Let us describe the publishing process. The oldest ancestor (that is, the node with $\mathsf{d}_v = 1$) publishes first the result of the increment (see the first line in predicate 39: `Publish`). Note that the guard of Command $\mathcal{P}ipe(v)$ is true if and only if the increment is finished (see $(\mathsf{Add}_v \neq \emptyset)$ in the guard). If a node v and all its children have the result of the increment of v's parent, then the broadcast of the result is finished, and v can publish its own result (see the last two lines in predicate 39: `Publish`).

$$\begin{array}{rcl}\texttt{Publish}(v) & \equiv & \big((\mathsf{d}_v = 1) \wedge (\forall u \in \texttt{Ch}(v) \cup \{v\}.\mathsf{PL}_u = (\perp,\perp))\big) \vee \\ & & \big[(\mathsf{PL}_v[0] = \mathsf{d}_v - 1) \wedge (\mathsf{PL}_{\mathsf{par}_v} = (\perp,\perp)) \wedge \\ & & \big(((\widehat{\mathsf{B}}_v > \mathsf{d}_v > 1) \wedge (\forall u \in \texttt{Ch}(v).\mathsf{PL}_u = \mathsf{PL}_v)) \vee \\ & & (\mathsf{d}_v = \widehat{\mathsf{B}}_v) \wedge (\forall u \in \texttt{Ch}(v).\mathsf{HC}_u = \mathsf{PL}_v)\big)\big] \end{array} \tag{39}$$

The broadcast process is the following. When all children u of node v have the same result (*i.e.*, $\mathsf{PL}_u = \mathsf{PL}_v$ for $\widehat{\mathsf{B}}_v > \mathsf{d}_v > 1$, or $\mathsf{HC}_u = \mathsf{PL}_v$ for $\mathsf{d}_v = \widehat{\mathsf{B}}_v$) and v's parent stores the next result (*i.e.*, $\mathsf{PL}_{\mathsf{par}_v}[0] = \mathsf{PL}_v[0] + 1$)), then v copies the result of its parent.

$$\begin{array}{rcl} \texttt{Pipe}(v) & \equiv & (\mathsf{PL}_{\mathsf{par}_v} \neq (\bot,\bot)) \wedge (\mathsf{PL}_{\mathsf{par}_v}[0] = \mathsf{PL}_v[0]+1) \wedge \\ & & \big[((\widehat{\mathsf{B}}_v > \mathsf{d}_v > 1) \wedge (\forall u \in \texttt{Ch}[v].\mathsf{PL}_u = \mathsf{PL}_v)) \vee \\ & & \big((\mathsf{d}_v = \widehat{\mathsf{B}}_v) \wedge (\forall u \in \texttt{Ch}[v].\mathsf{HC}_u = \mathsf{PL}_v)\big)\big] \end{array} \tag{40}$$

The last task is dedicated to deleting PL after broadcasting. More precisely, when a node v had published its own variable $\mathsf{PL}_v[0] = \mathsf{d}_v$, and all its children have recorded v's variable $((\forall u \in \texttt{Ch}(v).\mathsf{PL}_u = \mathsf{PL}_v))$, v can delete PL_v.

$$\texttt{CleanPV}(v) \equiv (\mathsf{PL}_{\mathsf{par}_v} = (\bot,\bot)) \wedge (\mathsf{PL}_v[0] = \mathsf{d}_v) \wedge (\forall u \in \texttt{Ch}(v).\mathsf{PL}_u = \mathsf{PL}_v) \tag{41}$$

Command $\mathcal{P}ipe(v)$ corresponds to executing this process. Note that when a node publishes its result, it can deletes the result of the increment stored in Add. In the command, $\overline{\mathsf{dB}}_v$ denotes the opposite of dB_v (that is, $\overline{\mathsf{dB}}_v = 1$ if $\mathsf{dB}_v = 0$, and 0 otherwise).

$$\mathcal{P}ipe(v) : \begin{cases} \mathsf{PL}_v := (\mathsf{d}_v, \mathsf{dB}_v); \mathsf{Add}_v := \bot & \text{if } \texttt{Publish}(v) \wedge (\mathsf{Add}_v = ok) \\ \mathsf{PL}_v := (\mathsf{d}_v, \overline{\mathsf{dB}}_v); \mathsf{Add}_v := \bot & \text{if } \texttt{Publish}(v) \wedge (\mathsf{Add}_v = +) \\ \mathsf{PL}_v := \mathsf{PL}_{\mathsf{par}_v} & \text{if } \texttt{Pipe}(v) \wedge (\widehat{\mathsf{B}}_v > \mathsf{d}_v > 1) \\ \mathsf{HC}_v := \mathsf{PL}_{\mathsf{par}_v} & \text{if } \texttt{Pipe}(v) \wedge (\mathsf{d}_v = \widehat{\mathsf{B}}_v) \\ \mathsf{PL}_v := (\bot,\bot) & \text{if } \texttt{CleanPV}(v) \end{cases} \tag{42}$$

A non-root node v broadcasts the bits of its ancestors in its hyper-node and its own bit (a node with $\mathsf{d}_v = 1$ has no ancestors in its hyper-node). So, if v has in $\mathsf{PL}_v[0]$ a distance that is greater than its distance (*i.e.*, $\mathsf{PL}_v[0] > \mathsf{d}_v$), v detects an error. The fact that $\mathsf{PL}_{\mathsf{par}_v} = (\bot,\bot)$ and $\mathsf{PL}_v \neq (\bot,\bot)$ may only happen when v publishes its variables or the variable of its parent permits an error to be detected at v if $\mathsf{PL}_v[0] > \mathsf{d}_v$. Last, if $\mathsf{PL}_{\mathsf{par}_v} \neq (\bot,\bot)$, v must be in a broadcast process either with the same variable as its parent, or with the previous variable of its parent. If it is not the case, v detects an error $((\mathsf{PL}_v \neq \mathsf{PL}_{\mathsf{par}_v}) \wedge (\mathsf{PL}_v[0] \neq \mathsf{PL}_{\mathsf{par}_v}[0] - 1))$.

$$\begin{array}{rcl} \texttt{Er}_{\mathsf{PL}}(v) & \equiv & (\mathsf{leader}_v \neq 1) \wedge (\mathsf{PL}_v \neq (\bot,\bot)) \wedge \texttt{Tree}(v) \wedge (\mathsf{d}_v > 1) \wedge \big[(\mathsf{PL}_v[0] > \mathsf{d}_v) \vee \\ & & \big((\mathsf{PL}_{\mathsf{par}_v} = (\bot,\bot)) \wedge (\mathsf{Add}_{\mathsf{par}_v} \neq \bot)\big) \vee \\ & & \big((\mathsf{PL}_{\mathsf{par}_v} \neq (\bot,\bot)) \wedge (\mathsf{PL}_v \neq \mathsf{PL}_{\mathsf{par}_v}) \wedge (\mathsf{PL}_v[0] \neq \mathsf{PL}_{\mathsf{par}_v}[0] - 1)\big)\big] \end{array} \tag{43}$$

4.4.3 Detailed description of the rule $\mathbb{R}_{\texttt{HypVerif}}$:

$$\mathbb{R}_{\texttt{HypVerif}} : \neg\texttt{Error}(v) \wedge \neg\texttt{T.Best}(v) \wedge (\mathsf{leader}_v \neq 1) \wedge \texttt{Tree}(v) \wedge (\texttt{T.Verif}(v) \vee \texttt{CleanHC}(v)) \rightarrow \mathcal{V}erif(v);$$

The process of assigning or verifying is the following. The increment results in broadcasting with variable HC from the node with distance one to the node with distance $\widehat{\mathsf{B}}_v$ (see Table 3).

Let us consider a hyper-node X. The node v whose distance is one is the ancestor of all other passive nodes in X, so v collects the values for distance verification in variable PL_{p_v} of its parent, because par_v is not in the same hyper-node as v. For the other passive nodes (*i.e.*, such that $\mathsf{d}_v > 1$), the distance verification uses variable HC of the parent (see Line 1 in predicate $\texttt{Broad}(v)$(44)). A node collects its parent's variable if its own variable has already been collected by its children. We remark that, the node with $\mathsf{d}_v = \widehat{\mathsf{B}}_v$ has no children within

			step 1		step 2		step 3		step 4		step 5		step 6		step 7	
	d	dB	Add	PL	Add	PL	Add	PL	Add	PL	Add	PL	Add	PL	Add	PL
X	1	0	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$
	2	1	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$
	3	0	$+$	$(2,1)$	$+$	$(2,1)$	$\perp$	$(3,1)$	$\perp$	$(3,1)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$	$\perp$	$(\perp,\perp)$
	4	1	$+$	$(1,0)$	$+$	$(2,1)$	$+$	$(2,1)$	$+$	$(3,1)$	$+$	$(3,1)$	$\perp$	$(4,0)$	$\perp$	$(4,0)$
	d	dB		HC		HC		HC		HC		HC		HC		HC
Y	1	0		$(1,0)$		$(1,0)$		$(2,1)$		$(2,1)$		$(3,1)$		$(3,1)$		$(4,0)$
	2	1		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(2,1)$		$(2,1)$		$(3,1)$		$(3,1)$
	3	1		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(3,1)$
	4	0		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$		$(\perp,\perp)$

Table 3: Example of hyper-node distance verification from Hyper-node X to Hyper-node Y.

its hyper-node, so it does not check its children for this task (see Lines 2 and 3 in predicate $\texttt{Broad}(v)$(44)).

$$\texttt{Broad}(v) \equiv$$
$$\Big[\big((\mathsf{d}_v = 1) \wedge (\mathsf{HC}_v[0] \neq \mathsf{PL}_{\mathsf{par}_v}[0] \neq \perp\big) \vee \big((\mathsf{d}_v > 1) \wedge (\mathsf{HC}_v \neq \mathsf{HC}_{\mathsf{par}_v}) \wedge (\mathsf{HC}_{\mathsf{par}_v} \neq (\perp,\perp))\big)\Big] \wedge$$
$$\Big[(\mathsf{d}_v < \widehat{\mathsf{B}}_v) \wedge \Big(\forall u \in \texttt{Ch}(v).\big((\mathsf{HC}_v[0] = \mathsf{d}_v) \wedge (\mathsf{HC}_u = (\perp,\perp))\big) \vee \big((\mathsf{HC}_v[0] > \mathsf{d}_v) \wedge (\mathsf{HC}_u = \mathsf{HC}_v)\big)\Big)\Big] \tag{44}$$

To verify the hyper-node distances, all nodes v of X must check whether the bit in variable $\mathsf{HC}_v[1]$ for itself $(\mathsf{HC}_v[0] = \mathsf{d}_v)$ corresponds to its actual bit. If it is not the case, node v detects an error.

$$\texttt{Er}_{\texttt{HypDis}}(v) \equiv (\mathsf{HC}_v[0] = \mathsf{d}_v) \wedge (\mathsf{HC}_v[1] \neq \mathsf{dB}_v) \tag{45}$$

Finally, once the broadcast is finished, node v requires cleaning.

$$\begin{array}{rcl} \texttt{CleanHC}(v) & \equiv & (\mathsf{HC}_{\mathsf{par}_v} = (\perp,\perp)) \wedge (\mathsf{HC}_v[0] = \widehat{\mathsf{B}}_v) \wedge \\ & & (\forall u \in \texttt{Ch}(v) : (\mathsf{d}_u = \widehat{\mathsf{B}}_u) \vee \big((\mathsf{d}_u < \widehat{\mathsf{B}}_u) \wedge (\mathsf{HC}_u = \mathsf{HC}_v)\big) \end{array} \tag{46}$$

Command $\mathcal{V}erif(v)$ consists in selectively applying these assignments:

$$\mathcal{V}erif(v) : \begin{cases} \mathsf{HC}_v := \mathsf{PL}_{\mathsf{par}_v} & \text{if } (\mathsf{d}_v = 1) \wedge \texttt{Broad}(v) \\ \mathsf{HC}_v := \mathsf{HC}_{\mathsf{par}_v} & \text{if } (\mathsf{d}_v > 1) \wedge \texttt{Broad}(v) \\ \mathsf{HC} := (\perp,\perp) & \text{otherwise} \end{cases} \tag{47}$$

A passive node v collects the bits of its descendants and itself, so if v has in variable $\mathsf{PL}_v[0]$ a distance that is lower than its distance (*i.e.*, $\mathsf{HC}_v[0] < \mathsf{d}_v$), v detects an error. The only possibility for Child u of v to have $\mathsf{HC}_u = (\perp,\perp)$ and $\mathsf{PL}_v \neq (\perp,\perp)$ is when v publishes its variables, so if $\mathsf{HC}_v[0] \neq \mathsf{d}_v$, node v detects an error. Last, if $\mathsf{HC}_u \neq (\perp,\perp)$, node u must be in a broadcast process either with the same variable as its parent v, or with the previous variable of its parent. If it is not the case, v detects an error $((\mathsf{HC}_u \neq \mathsf{HC}_v) \wedge (\mathsf{HC}_u[0] \neq \mathsf{HC}_v[0] - 1)))$.

$$\begin{aligned}\mathtt{Er}_{\mathtt{HC}}(v) \quad \equiv \quad & (\mathsf{leader}_v \neq 1) \wedge (\mathsf{HC}_v \neq (\bot,\bot)) \wedge \mathtt{Tree}(v) \wedge \\ & \big[(\mathsf{HC}_v[0] < \mathsf{d}_v) \vee [(\mathsf{d}_v < \widehat{\mathsf{B}}_v) \wedge \\ & \big((\exists u \mathtt{Ch}(v).\mathsf{HC}_u = (\bot,\bot)) \wedge (\mathsf{HC}_v[0] \neq \mathsf{d}_v)\big) \vee \\ & \big((\exists u \mathtt{Ch}(v).\mathsf{HC}_u \neq (\bot,\bot)) \wedge (\mathsf{HC}_u \neq \mathsf{HC}_v) \wedge (\mathsf{HC}_u[0] \neq \mathsf{HC}_v[0]-1)\big)\big]\end{aligned} \tag{48}$$

4.4.4 Detailed description of rule $\mathbb{R}_{\mathtt{RootdB}}$:

$\mathbb{R}_{\mathtt{RootdB}}$:$\neg\mathtt{Error}(v) \wedge \neg\mathtt{T.Best}(v) \wedge (\mathsf{leader}_v = 1) \wedge (\forall u \in \mathtt{Ch}(v).\mathsf{PL}_v = \mathsf{HC}_u) \rightarrow \mathcal{S}tartdB(v)$;

A root r is not an element of a hyper-node, moreover the distance of its child hyper-node is zero. So, let us denote by X the child hyper-node of r. All nodes $u \in X$ must have $\mathsf{dB}_u = 0$. The child v of r broadcasts for all nodes in X the pair (distance, bit) using variable HC_v. To achieve that, r must publish all theses pairs. This process is also employed for verifying the distance of X. As a consequence, when r has published the bit for the last node in X, r restarts the publishing.

$$\mathcal{S}tartdB(v) : \mathsf{PL}_v := (\delta^+(\mathsf{PL}_v[0]), 0); \tag{49}$$

We remark that a node v that is a neighbor of u with $\mathtt{Root}(u) = \mathit{true}$ waits to become passive until u publishes $\mathsf{PL}_u = (1,0)$ (see predicate $\mathtt{T.dB}(15)$ in Subsection 4.3.3). As a root is not included in a hyper-node, the variable dedicated to performing the hyper-node distance increment Add and the variable dedicated for receiving the result of the increment remain equal to $\bot$. A root detects an error if it is not the case.

$$\mathtt{Er}_{\mathtt{MRoot}}(v) \quad \equiv \quad (\mathsf{leader}_v = 1) \wedge \neg\big((\mathsf{Add}_v = \bot) \wedge (\mathsf{HC}_v = (\bot,\bot))\big) \tag{50}$$

4.4.5 Errors of hyper-node distances

To summarize the errors of the hyper-nodes distance verification, we use the following predicate:

$$\mathtt{Er}_{\mathtt{Hyper}}(v) \equiv \mathtt{Er}_{\mathtt{Add}}(v) \vee \mathtt{Er}_{\mathtt{Overflow}}(v) \vee \mathtt{Er}_{\mathtt{PL}}(v) \vee \mathtt{Er}_{\mathtt{HypDis}}(v) \vee \mathtt{Er}_{\mathtt{HC}}(v) \vee \mathtt{Er}_{\mathtt{MRoot}}(v) \tag{51}$$

5 Correctness

In this section, we provide a detailed proof that establishes the correctness of our Algorithm.

Theorem 1 *Algorithm* **CLE** *solves the leader election problem in a self-stabilizing manner in any n-node ring, assuming the state model, and a distributed weakly-fair scheduler. Moreover, if the n node identities are in $[1, n^c]$, for some $c \geq 1$, then Algorithm* **CLE** *uses $O(\log\log n)$ bits of memory per node, and stabilizes in $O(n \log^2 n)$ rounds.*

The main difficulty for proving Theorem 1 is to establish the ability of **CLE** to detect any cycle that could be generated by the parenthood relation in the initial configuration, and whenever such a cycle is detected, to remove this cycle. Let Γ be the set of all possible configurations of the n-node ring. First, we prove that Algorithm **CLE** detects the presence of "trivial" errors,

that is, inconsistencies between neighbors. Second, we prove that, after correcting all trivial errors (possibly using a cleaning mechanism), **CLE** converges and maintains configurations free of trivial errors. The set of configurations that are free of trivial errors is denoted by $\Gamma_{\texttt{TEF}}$, where TEF stands for "Trivial Error Free". Then, we assume that configurations are in $\Gamma_{\texttt{TEF}}$ only. The core of the cycle detection proof is based on proving the correctness of the hyper-node distance verification process. This verification process is the most technical part of the algorithm, and proving its correctness is the main challenge on the way to establishing Theorem 1. Once the hyper-node distance verification process has been proved, we establish the convergence of Algorithm **CLE** from an arbitrary configuration in $\Gamma_{\texttt{TEF}}$ to a configuration without cycles, and where all hyper-node distances are correct. The set of configurations without cycles is denoted by $\Gamma_{\texttt{CF}}$ (where CF stands for "Cycle Free"). We prove that a configuration is in $\Gamma_{\texttt{CF}}$ if and only if all hyper-node distances are correct. Then, we restrict ourselves to configurations in $\Gamma_{\texttt{CF}}$, and prove the correctness of our mechanisms for detecting and removing impostor leaders. We denote by $\Gamma_{\texttt{IEF}}$ (where IEF stands for "Impostor leader Error Free") the set of configurations with no impostors. Finally, assuming a configuration in $\Gamma_{\texttt{IEF}}$, we prove that the system reaches and maintains a configuration with exactly one leader, equal to the node with maximum identity. Moreover, we establish that the structure induced by the parenthood relation is a tree rooted at the leader that spans all nodes. We denote by $\Gamma_{\texttt{LE}}$ the set of configurations where the unique leader is the node with maximum identity. Overall, we prove that **CLE** is self-stabilizing to $\Gamma_{\texttt{LE}}$.

In the details of lemmas that are presented in the sequel, we use predicates on configurations. These predicates are intermediate attractors towards a legitimate configuration (*i.e.*, a configuration with a unique leader). To establish that those predicates are indeed attractors, we use potential functions [37], that is, functions that map configurations to non-negative integers, and that strictly decrease after any algorithm step is executed.

To avoid additional notations, we use sets of configurations to define predicates; the predicate should then be understood as the characteristic function of the set (that returns *true* if the configuration is in the set, and *false* otherwise).

Let us first define predicate $\Gamma_{\texttt{LE}}$ (*Leader Election*), that serves as the definition for legitimate configurations. Let $L : \Gamma \to \mathbb{N}$ be the function defined by

$$L(\gamma) = \sum_{v \in V} \mathsf{leader}_v \ .$$

A configuration γ is legitimate for the leader election specification (*i.e.*, satisfies $\Gamma_{\texttt{LE}}$) if and only if $L(\gamma) = 1$. That is,

$$\Gamma_{\texttt{LE}} = \{\gamma \in \Gamma : L(\gamma) = 1\}.$$

For the sake of clarity we denote by $P(v, \gamma)$ the predicate P for node v in configuration γ.

We now define the trivial errors predicate:

$$\texttt{Er}_{\texttt{T}}(v) \equiv \texttt{Er}_{\texttt{Freeze}}(v) \vee \texttt{Er}_{\texttt{Nd}}(v) \vee \texttt{Er}_{\texttt{Check}}(v) \vee \texttt{Er}_{\texttt{Add}}(v) \vee \texttt{Er}_{\texttt{PL}}(v) \vee \texttt{Er}_{\texttt{HC}}(v) \vee \texttt{Er}_{\texttt{MRoot}}(v) \quad (52)$$

and the predicate $\Gamma_{\texttt{TEF}}$ (*Trivial Error Free*). Let $\psi : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\psi(\gamma, v) = \begin{cases} 1 & \text{if } \texttt{Er}_{\texttt{T}}(v, \gamma) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Let $\Psi : \Gamma \to \mathbb{N}$ be the function defined by:

$$\Psi(\gamma) = \sum_{v \in V} \psi(\gamma, v) \ .$$

Note that all nodes are legitimate with respect to $\texttt{Er}_\texttt{T}(v)$ if and only if $\Psi(\gamma) = 0$. We define

$$\Gamma_{\texttt{TEF}} = \{\gamma \in \Gamma : \Psi(\gamma) = 0\}.$$

Lemma 1 *$true \triangleright \Gamma_{\texttt{TEF}}$ in one round.*

Proof. We prove that, starting from an arbitrary initial configuration γ_0, Algorithm **CLE** reaches a configuration in $\Gamma_{\texttt{TEF}}$ in one round. Any node v with $\texttt{Er}_\texttt{T}(v, \gamma_0) = true$ is enabled by rule $\mathbb{R}_{\texttt{Error}}$ (since predicates of $\texttt{Er}_\texttt{T}$ are a subset of those of $\texttt{T.Er}$). These nodes are activated during the first round. After execution of command $\mathcal{F}reeze$ (see Formula (4)), all these nodes become frozen. In other words, upon activation, a node v with $\texttt{Er}_\texttt{T}(v, \gamma_0)$ assigns $\mathsf{leader}_v = 2$ and $\mathsf{par}_v = \emptyset$ in γ_1, but does not change other variables (see Subsection 4.2).

Remark 1: To evaluate to true, $\texttt{Er}_{\texttt{Nd}}(v)$ (see Formula (29)) and $\texttt{Er}_{\texttt{MRoot}}(v)$ (see Formula (50)) require $\mathsf{leader}_v \in \{0, 1\}$, and $\texttt{Er}_{\texttt{Freeze}}(v)$ (see Formula (3)) requires $\mathsf{leader}_v = 2$.

Remark 2: To evaluate to true $\texttt{Er}_{\texttt{Freeze}}(v)$(see Formula (3)), $\texttt{Er}_{\texttt{Check}}(v)$ (see Formula (28)), $\texttt{Er}_{\texttt{Add}}$(see Formula (36)), $\texttt{Er}_{\texttt{PL}}(v)$ (see Formula (43)) and $\texttt{Er}_{\texttt{HC}}(v)$ (see Formula (48)) require $\mathsf{par}_v \neq \emptyset$ (see predicate $\texttt{Normal}(v)$ for $\texttt{Er}_{\texttt{Add}}(v)$, $\texttt{Er}_{\texttt{PL}}(v)$ and $\texttt{Er}_{\texttt{HC}}(v)$).

Remark 3: If a node u with $\texttt{Er}_\texttt{T}(u, \gamma_0) = false$ executes a command taking into account its neighbors' variables, and its neighbors do not execute any command, then by construction of **CLE** commands, no error is generated and $\texttt{Er}_\texttt{T}(u, \gamma_1)$ remains false.

A node v with $\mathsf{leader}_v = 2$ and $\mathsf{par}_v = \emptyset$ has $\texttt{Er}_\texttt{T}(v, \gamma_1) = false$ (see Formula (52) and Remarks 1 and 2). Note that, after execution of $\mathcal{F}reeze(v, \gamma_0)$, v's variables do not change (except leader_v and par_v). Let u be a neighbor of v with $\texttt{Er}_\texttt{T}(u, \gamma_0) = false$. Only predicate $\texttt{Er}_{\texttt{Freeze}}(u)$ (see Formula (3)) included in predicate $\texttt{Er}_\texttt{T}(u)$ checks variable leader_v of its neighbors if $\mathsf{par}_u = v$. Yet, in γ_1, variable $\mathsf{leader}_v = 2$, so $\texttt{Er}_{\texttt{Freeze}}(u, \gamma_1) = false$, and thanks to Remark 3, $\texttt{Er}_\texttt{T}(u, \gamma_1)$ remains false. Thus, starting from any arbitrary initial configuration γ_0, the system reaches a configuration $\gamma_1 \in \Gamma_{\texttt{TEF}}$ in one round. □

Lemma 2 *$\Gamma_{\texttt{TEF}}$ is closed.*

Proof. We prove that, starting from a configuration where $\gamma_0 \in \Gamma_{\texttt{TEF}}$ holds, Algorithm **CLE** preserves the fact that $\texttt{Er}_\texttt{T}(v)$ (see Formula (52)) remains false for any node v in V.

We now consider the impact of the execution of each rule at node v.

$\mathbb{R}_{\texttt{Error}}$: This rule is detailed in Subsection 4.2. We remark that only three types of errors are *not* included in $\texttt{Er}_\texttt{T}$ (see Formula (52)): $\texttt{Er}_{\texttt{Place}}$(see Formula (17)), $\texttt{Er}_{\texttt{Overflow}}$(see Formula (37)), and $\texttt{Er}_{\texttt{HypDis}}$(see Formula (45)). Now, if at least one of these predicates is true at node v, then rule $\mathbb{R}_{\texttt{Error}}$ is enabled. The proof of Lemma 1 shows that in this case, after execution of command $\mathcal{F}reeze(v)$ (see Formula (4)), $\texttt{Er}_\texttt{T}(v)$(see Formula (52)) remains false, and any neighbor u of v also has $\texttt{Er}_\texttt{T}(u) = false$. Rule $\mathbb{R}_{\texttt{Error}}$ is also executed by a node v whose parent is frozen ($\mathsf{leader}_{\mathsf{par}_v} = 2$), so $\texttt{Er}_{\texttt{Freeze}}(v)$ (see Formula (3)) remains false and by Lemma 1, we obtain for a node v and any of its neighbors u that $\texttt{Er}_\texttt{T}(v)$ and $\texttt{Er}_\texttt{T}(u)$ both remain false. So, starting in a configuration in $\gamma_0 \in \Gamma_{\texttt{TEF}}$, after execution of command $\mathcal{F}reeze(v)$, predicate $\texttt{Er}_\texttt{T}(v, \gamma_1)$ remains false.

$\mathbb{R}_{\texttt{Start}}$: This rule is detailed in Subsection 4.2. Only a frozen node v with no child can execute this rule. After execution of command $\mathcal{S}tart(v)$ (see Formula (5)), v becomes a root ($\mathsf{leader}_v = 1$).

Remark 4: Predicates $\texttt{Root}(v)$ (see Formula (6)), $\texttt{Er}_{\texttt{MRoot}}(v)$ (see Formula (50)), and $\texttt{PassNd}(v)$ (see Formula (7)) are independent from the local variables of v's neighbors. By construction of **CLE**'s commands, those do not generate errors in $\texttt{Root}(v)$ (see Formula (6)), $\texttt{Er}_{\texttt{MRoot}}(v)$ (see Formula (50)) and $\texttt{PassNd}(v)$ (see Formula (7)).

Execution of $\mathcal{S}tart(v)$(see Formula (5)) assigns $\mathsf{leader}_v = 0$. Thanks to Remarks 2 and 4, we obtain that $\texttt{Er}_{\texttt{T}}(v, \gamma_1)$ remains false independently of a command execution by a neighbor of v. So, starting in a configuration in $\gamma_0 \in \Gamma_{\texttt{TEF}}$, after execution of command $\mathcal{S}tart(v)$, predicate $\texttt{Er}_{\texttt{T}}(v, \gamma_1)$ remains false.

$\mathbb{R}_{\texttt{Root}}$ and $\mathbb{R}_{\texttt{RootdB}}$: These rules are detailed in Subsections 4.3.4 and 4.4.4. They are enabled at any node v with $\texttt{Root}(v, \gamma_0) = true$ (see $\mathsf{leader}_v = 1$ in the guard of $\mathbb{R}_{\texttt{Root}}$). Executing $\mathcal{I}nc(v)$ (see Formula (24)) or $\mathcal{S}tartdB(v)$ (see Formula (49)) maintains $\mathsf{leader}_v = 1$. Thanks to Remarks 2 and 4, we obtain the fact that $\texttt{Er}_{\texttt{T}}(v, \gamma_1)$ (see Formula (52)) remains false, independently of a command execution by a neighbor of v. So, starting in a configuration in $\gamma_0 \in \Gamma_{\texttt{TEF}}$, after execution of command $\mathcal{I}nc(v)$, predicate $\texttt{Er}_{\texttt{T}}(v, \gamma_1)$ remains false.

$\mathbb{R}_{\texttt{Passive}}$: This rule is detailed in Subsection 4.3.3.

Remark 5: Executing command $\mathcal{P}ass(v)$ (see Formula (14)) deletes the hyper-nodes variables of v. As a consequence, the following predicates become false: $\texttt{Er}_{\texttt{Add}}(v, \gamma_1)$(see Formula (36)), $\texttt{Er}_{\texttt{PL}}(v, \gamma_1)$(see Formula (43)), and $\texttt{Er}_{\texttt{HC}}(v, \gamma_1)$ (see Formula (48)).

Remark 6: Predicate $\texttt{CohP}(v)$(see Formula (21)) depends on the local variables of v's parent of v. Predicate $\texttt{Er}_{\texttt{Check}}(v)$(see Formula (28)) depends on the local variable of v's parent when $\mathsf{Check}_v = 0$, and on the local variables of its neighbors when $\mathsf{Check}_v = 1$.

Remark 7: Let v and u be two neighbors such that $\mathsf{par}_v = u$ in γ_0 and γ_1. Then, command $\mathcal{F}reeze(u, \gamma_0)$ assigns $\mathsf{leader}_u = 2$ and can assign $\mathsf{par}_u = \emptyset$, which have no impact on $\texttt{CohP}(v, \gamma_1)$ (see Formula (21)) and $\texttt{Er}_{\texttt{Check}}(v, \gamma_1)$ (see Formula (28)), if u remains parent of v in γ_1.

Remark 8: A node with u with $\mathsf{leader}_u = 2$ cannot be a best neighbor of its neighbors v ($\texttt{Best}(v)$ (see Formula (13))), and u cannot involve v as its parent in rules $\mathbb{R}_{\texttt{Update}}$ (see $\texttt{T.Down}(v)$(Formula (25)) and $\texttt{T.Up}(u)$ (see Formula (26)).

From Remarks 4 and 5, to evaluate $\texttt{Er}_{\texttt{T}}(v, \gamma_1)$(see Formula (52)), we only need to check predicates $\texttt{CohP}(v)$ (see Formula (21)) and $\texttt{Er}_{\texttt{Check}}(v)$ (see Formula (28)). Let us study the impact of executing commands $\mathcal{F}reeze(u, \gamma_0)$ (see Formula (4)), $\mathcal{S}tart(u, \gamma_0)$ (see Formula (5)), $\mathcal{I}nc(u, \gamma_0)$ (see Formula (24)), $\mathcal{P}ass(u, \gamma_0)$ (see Formula (14)), and $\mathcal{U}pdate(u, \gamma_0)$ (see Formula (27)) by a node $u \in \mathsf{N}_v$. If these commands are enabled, the others commands have no impact on predicates $\texttt{CohP}(v)$ (see Formula (21)) and $\texttt{Er}_{\texttt{Check}}(v)$ (see Formula (28)). Executing $\mathcal{P}ass(v)$ (see Formula (14)) assigns $\mathsf{Check}_v = 0$ in γ_1. From Remark 6, we only need to study the neighbor u of v such that $\mathsf{par}_v = u$ in γ_1 ($\texttt{Best}(v, \gamma_0) = u$) (see Formula (13)). We remark that $\mathsf{Check}_u = 0$ and $\mathsf{Elec}_v \neq \mathsf{Elec}_u$ in γ_0 (due to $\texttt{Er}_{\texttt{Check}}(u) = false$ and $\texttt{Best}(v) = u$).

– For commands $\mathcal{F}reeze(u, \gamma_0)$ (see Formula (4)) and $\mathcal{S}tart(u, \gamma_0)$ (see Formula (5)), see Remarks 7 and 8.

– Command $\mathcal{I}nc(u,\gamma_0)$ (see Formula (24)) requires $\mathsf{Elec}_w = \mathsf{Elec}_u$ or $\mathsf{Phase}_u = \pi^-(w)$, for all $w \in \mathsf{N}_u$, which contradicts $\mathtt{Best}(v,\gamma_0) = u$ (see Formula (13)). So, rule $\mathbb{R}_{\mathtt{Root}}(u,\gamma_0)$ is not enabled.
– Since $\mathsf{Check}_u = 0$ in γ_0, rule $\mathbb{R}_{\mathtt{Update}}$ requires $\mathsf{Elec}_w = \mathsf{Elec}_u$, for all $w \in \mathsf{N}_u$, to be enabled (see predicate $\mathtt{T.Up}$ (see Formula (26))). So, $\mathbb{R}_{\mathtt{Update}}(u,\gamma_0)$ is not enabled because $\mathtt{Best}(v,\gamma_0) = u$.
– Command $\mathcal{P}ass(u,\gamma_0)$ (see Formula (14)) is enabled if and only if there exists a neighbor w of u such that $\mathtt{Best}(u\gamma_0) = w$ and $w \neq v$. As $\mathtt{Best}(v,\gamma_0) = u$ and $\mathtt{Best}(u,\gamma_0) = w$ after execution of $\mathcal{P}ass(v,\gamma_0)$ and $\mathcal{P}ass(u,\gamma_0)$, we have $\mathtt{Best}(v,\gamma_1) = u$. So, predicate $\mathtt{CohP}(v)$ remains true in γ_1. Command $\mathcal{P}ass(u)$ assigns $\mathsf{Check}_u = 0$, so v and its parent u have $\mathsf{Check}_v = \mathsf{Check}_u = 0$ in γ_1. Therefore, $\mathtt{Er}_{\mathtt{Check}}(v,\gamma_1)$ (see Formula (28)) remains false.

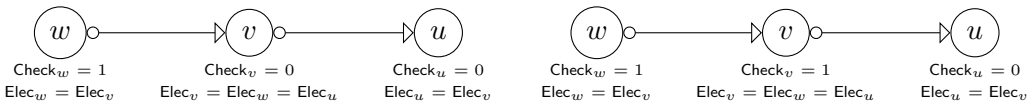
Overall, starting in a configuration in $\gamma_0 \in \Gamma_{\mathtt{TEF}}$, after execution of command $\mathcal{P}ass(v)$, predicate $\mathtt{Er}_{\mathtt{T}}(v,\gamma_1)$ remains false.

$\mathbb{R}_{\mathtt{Update}}$: This rule is detailed in Subsection 4.3.5. It is enabled by a passive node v (so $\mathtt{Er}_{\mathtt{MRoot}}(v,\gamma_0) = \mathit{false}$) and by Remark 4, we obtain $\mathtt{PassNd}(v,\gamma_1) = \mathit{true}$ (see Formula (7)) and $\mathtt{Er}_{\mathtt{MRoot}}(v,\gamma_1) = \mathit{false}$ (see Formula (50)). The execution of command $\mathcal{U}pdate$ (see Formula (27)) by v does not modify the variables used by predicates $\mathtt{Er}_{\mathtt{Add}}(v)$ (see Formula(36)), $\mathtt{Er}_{\mathtt{PL}}(v)$ (see Formula (43)), and $\mathtt{Er}_{\mathtt{HC}}(v)$ (see Formula (48)) for v and its neighbors. As a consequence, we need to check predicates $\mathtt{CohP}(v)$ (see Formula (21)) and $\mathtt{Er}_{\mathtt{Check}}(v)$ (see Formula (28)). Command $\mathcal{U}pdate(v)$ (see Formula (27)) has two possible actions.

The first case is when $\mathtt{T.Down}(v,\gamma_0) = \mathit{true}$ (see Formula (25)), and then the execution of $\mathcal{U}pdate(v,\gamma_0)$ assigns $\mathsf{Elec}_v := \mathsf{Elec}_{\mathsf{par}_v}$ and $\mathsf{Check}_v := 0$ in γ_1. Remarks 6 and $\mathsf{Check}_v = 0$ in γ_1 imply that in order to check $\mathtt{Er}_{\mathtt{T}}(v,\gamma_1)$, we need to study the impact of executing one command in γ_0, for $u = \mathsf{par}_v$ in γ_1

– For commands $\mathcal{F}reeze(u,\gamma_0)$ and $\mathcal{S}tart(u,\gamma_0)$, see Remarks 7 and 8.
– Command $\mathcal{I}nc(u)$ (see Formula (24)) requires in γ_0 that $\mathsf{Elec}_w = \mathsf{Elec}_u$ or $\mathsf{Phase}_u = \pi^-(w)$, for all $w \in \mathsf{N}_u$, which contradicts $\mathtt{T.Down}(v) = \mathit{true}$ (see Formula (25)). So rule $\mathbb{R}_{\mathtt{Root}}(u,\gamma_0)$ is not enabled.
– Rule $\mathbb{R}_{\mathtt{Update}}$ (see Formula (27)) requires $\mathsf{Elec}_w = \mathsf{Elec}_u$, for all $w \in \mathtt{Ch}(u)$, to be enabled (see predicates $\mathtt{Er}_{\mathtt{Check}}(u)$ (see Formula (28)) and $\mathtt{T.Up}(u)$ (see Formula (26))), but $v \in \mathtt{Ch}(u,\gamma_0)$ and $\mathtt{T.Down}(v,\gamma_0)$ implies $\mathsf{Elec}_v \neq \mathsf{Elec}_u$, so $\mathbb{R}_{\mathtt{Update}}(u,\gamma_0)$ is not enabled.
– Command $\mathcal{P}ass(u)$ (see Formula (14)) is enabled if and only if there exists a neighbor w of u such that $\mathtt{Best}(u) = w$ and $w \neq v$. As Elec_v in γ_1 is equal to Elec_u in γ_0, we have $\mathtt{Best}(v) = u$ after execution of $\mathcal{U}pdate(v)$ (see Formula (27)) and $\mathcal{P}ass(u)$ (see Formula (14)). Hence, predicate $\mathtt{CohP}(v,\gamma_1)$ remains true. Moreover, command $\mathcal{P}ass(u,\gamma_0)$ assigns $\mathsf{Check}_u = 0$, so both v and its parent u have $\mathsf{Check}_v = \mathsf{Check}_u = 0$ in γ_1. Therefore, $\mathtt{Er}_{\mathtt{Check}}(v,\gamma_1)$ (see Formula (28)) remains false.

The second case is when $\mathtt{T.Up}(v,\gamma_0) = \mathit{true}$ (see Formula (26)). In other words, $\mathsf{Elec}_v = \mathsf{Elec}_u$ for all $u \in \mathsf{N}_v$, and $\mathsf{Check}_v = 0$ and $\mathsf{Check}_u = 1$ for all $u \in \mathtt{Ch}(v)$. We remark that, since $\mathtt{Er}_{\mathtt{Check}}(u,\gamma_0) = \mathit{false}$, we have $\mathsf{Check}_u = 0$ in γ_0 (see Figure 5(a)). Execution of $\mathcal{U}pdate(v)$ assigns $\mathsf{Check}_v = 1$(see Figure 5(b)). Let u be the neighbor of v such that $\mathsf{par}_v = u$, and w be the neighbor such that $\mathsf{par}_w = v$ in γ_1.



(a) Configuration γ_0 (b) Configuration γ_1

Figure 5: $\texttt{T.Up}(v) = \mathit{true}$ in γ_0.

- For commands $\mathcal{F}reeze(u, \gamma_0)$ and $\mathcal{S}tart(u, \gamma_0)$, see Remarks 7 and 8. $\mathcal{F}reeze(w, \gamma_0)$ and $\mathcal{S}tart(w, \gamma_0)$ contradict the fact that $\mathsf{par}_w = v$ in γ_1, because $\mathsf{leader}_v \neq 2$ implies that command $\mathcal{F}reeze(w, \gamma_0)$ assigns $\mathsf{par}_w = \emptyset$ and command $\mathcal{S}tart(w, \gamma_0)$ also assigns $\mathsf{par}_w = \emptyset$.
- Command $\mathcal{I}nc(u, \gamma_0)$ (see Formula (24)) requires $\mathsf{Elec}_x = \mathsf{Elec}_u$, and $\mathsf{Check}_x = 1$ for all $x \in \texttt{Ch}(u, \gamma_0)$, which contradicts $\texttt{T.Up}(v, \gamma_0) = \mathit{true}$ (see Formula (26)). So, rule $\mathbb{R}_{\texttt{Root}}(u, \gamma_0)$ is not enabled. Then, command $\mathcal{I}nc(w, \gamma_0)$ is enabled by a root and does not modify its parent, so this contradicts $\mathsf{par}_w = v$ in γ_1. Hence, rule $\mathbb{R}_{\texttt{Root}}(w, \gamma_0)$ is not enabled.
- Rule $\mathbb{R}_{\texttt{Update}}$ is not enabled for u and w in γ_0, since u requires either $\mathsf{Check}_u = 1$ or $\mathsf{Check}_u = 0$ and $\mathsf{Check}_v = 1$, and w requires $\mathsf{Elec}_w \neq \mathsf{Elec}_v$ to be enabled.
- Command $\mathcal{P}ass(u, \gamma_0)$ is enabled if and only if there exists a neighbor x of u such that $\texttt{Best}(u) = x$ and $x \neq v$. As Elec_v in γ_1 is equal to Elec_u in γ_0, $\texttt{Best}(v) = u$ holds after execution of $\mathcal{U}pdate(v)$ and $\mathcal{P}ass(u)$, so predicate $\texttt{CohP}(v, \gamma_1)$ remains true. Moreover, command $\mathcal{P}ass(u, \gamma_0)$ assigns $\mathsf{Check}_u = 0$, so v and its parent u satisfy $\mathsf{Check}_v = \mathsf{Check}_u = 0$ in γ_1. Therefore, $\texttt{Er}_{\texttt{Check}}(v, \gamma_1)$ remains false. Command $\mathcal{P}ass(w, \gamma_0)$ is enabled if and only if there exists a neighbor x of u such that $\texttt{Best}(u) = x$ and $x \neq v$. Yet, this activation assigns $\mathsf{par}_w = x$ in γ_1, which contradicts our assumption that $\mathsf{par}_w = v$ in γ_1. So, rule $\mathbb{R}_{\texttt{Passive}}(w, \gamma_0)$ is not enabled.

Overall, starting in a configuration in $\gamma_0 \in \Gamma_{\texttt{TEF}}$, after execution of command $\mathcal{U}pdate(v)$, predicate $\texttt{Er}_{\texttt{T}}(v, \gamma_1)$ remains false.

Remark 9: Commands $\mathcal{B}inAdd(v)$, $\mathcal{P}ipe(v)$, and $\mathcal{V}erif(v)$ do not modify leader_v (i.e., $\mathsf{leader}_v \in \{1, 2\}$). Thanks to Remark 4, $\texttt{Er}_{\texttt{Nd}}(v, \gamma_1)$ and $\texttt{Er}_{\texttt{MRoot}}(v, \gamma_1)$ remain false after execution of one of these commands.

Remark 10: Algorithm **CLE** maintains $\texttt{Er}_{\texttt{Freeze}}(v) = \mathit{false}$, a node v has $\texttt{Er}_{\texttt{Freeze}}(v) = \mathit{true}$ if and only if $\mathsf{leader}_v = 2$ and $\mathsf{leader}_{\mathsf{par}_v} \neq 2$. If we consider a configuration $\gamma_0 \in \Gamma_{\texttt{TEF}}$, and a node v with $\mathsf{leader}_v = 2$, either v has no parent and no rule assigns a parent to a node with $\mathsf{leader}_v = 2$, or v has a parent and its parent has $\mathsf{leader}_{\mathsf{par}_v} = 2$. Then, the only rule executable by par_v is $\mathcal{S}tart$, but this rule is executable only by a node without child.

$\mathbb{R}_{\texttt{HypAdd}}$: This rule is detailed in Subsection 4.4.1. Command $\mathcal{B}inAdd(v)$ only modifies variable Add_v.

$\texttt{Er}_{\texttt{Check}}(v)$ (see Formula (28)): Command $\mathcal{B}inAdd(v)$ does not modify the variables used in $\texttt{Er}_{\texttt{Check}}(v)$, so $\texttt{Er}_{\texttt{Check}}(v)$ may become true if and only if one of its neighbors u modifies a variable used in $\texttt{Er}_{\texttt{Check}}(v)$. In other words, if u executes $\mathcal{P}ass(u)$ or $\mathcal{U}pdate(u)$. Let us consider first the case when $\mathsf{Check}_v = 0$. We need to check Check_u with $\mathsf{par}_v = u$ (Remark 6) in γ_1. If command $\mathcal{P}ass(u)$ is executed in γ_0,

then $\mathsf{Check}_u = 0$ in γ_1, and $\mathtt{Er}_{\mathtt{Check}}(v)$ remains false. Rule $\mathbb{R}_{\mathtt{Update}}$ is not enabled for u (thanks $\mathtt{Er}_{\mathtt{Check}}(v) = \mathit{false}$ in γ_0). Now we consider, $\mathsf{Check}_v = 1$, we need to check Check_u with $u \in \mathtt{Ch}(v)$ (remark 6) in γ_1. Thanks to $\mathtt{Er}_{\mathtt{Check}}(v, \gamma_0) = \mathit{false}$, $\mathbb{R}_{\mathtt{Passive}}(u, \gamma_0)$ and $\mathbb{R}_{\mathtt{Update}}(u, \gamma_0)$ are not enabled.

$\mathtt{Er}_{\mathtt{Add}}(v)$ (see Formula (48)): Command $\mathcal{B}in\mathit{Add}(v)$ modifies the variables used in $\mathtt{Er}_{\mathtt{Add}}(v)$, so we need to check variable Add_u of the child u of v in γ_1 (Remark 6) if $\mathsf{d}_v < \widehat{\mathsf{B}}_v$. Three commands may modify Add_u. The first one is $\mathcal{P}\mathit{ass}(u)$, but from Remark 5, $\mathtt{Er}_{\mathtt{Add}}(v)$ remains false. The second one is $\mathcal{B}in\mathit{Add}(u)$, but the activation of rule $\mathbb{R}_{\mathtt{HypAdd}}$ at u implies that $\mathsf{Add}_u \neq \bot$, so $\mathbb{R}_{\mathtt{HypAdd}}$ is not enabled for u. The last one is $\mathcal{P}\mathit{ipe}(u)$, and more precisely when predicate $\mathtt{Publish}(u) = \mathit{true}$ (see Formula (39)). If $\mathtt{Publish}(u) = \mathit{true}$, then either $\mathsf{d}_u = 1$ (it is not the case since u is child of v, and v is not a root), or $\mathsf{PL}_u[0] = \mathsf{d}_u - 1$. Let us study the second case. The hypothesis that $\mathbb{R}_{\mathtt{HypAdd}}$ is enabled for v implies $\mathsf{Add}_v = \bot$ and $\mathsf{PL}_v = (\bot, \bot)$, so $\mathtt{Er}_{\mathtt{PL}}(v) = \mathit{false}$ in γ_0 implies $\mathsf{PL}_u = (\bot, \bot)$ or $\mathsf{PL}_u[0] = \mathsf{d}_u - 1$. We already saw that $\mathsf{d}_u \neq 1$, so as $\mathtt{Publish}(u) = \mathit{true}$, predicate $\mathsf{PL}_u \neq (\bot, \bot)$. Now, $\mathsf{PL}_u[0] = \mathsf{d}_u - 1$ in γ_0 implies $\mathsf{Add}_u = \bot$ (Thanks to $\mathtt{Er}_{\mathtt{PL}} = \mathit{false}$ in γ_0), which contradicts the activation of $\mathbb{R}_{\mathtt{HypAdd}}$ for v (which needs $\mathsf{Add}_u \neq \bot$). As a consequence, rule $\mathbb{R}_{\mathtt{HypBroad}}$ is not enabled for u. Then, $\mathtt{Er}_{\mathtt{Add}}(v)$ remains false in γ_1.

$\mathtt{Er}_{\mathtt{PL}}(v)$ (see Formula (43)): Command $\mathcal{B}in\mathit{Add}(v)$ does not modify the variables used in $\mathtt{Er}_{\mathtt{PL}}(v)$. So, we need to check variable $\mathsf{PL}(u)$ of the parent u of v in γ_1 (Remark 6) if $\mathsf{d}_v > 1$. Only commands $\mathcal{P}\mathit{ipe}(u)$ and $\mathcal{P}\mathit{ass}(u)$ may modify PL_u. Rule $\mathbb{R}_{\mathtt{HypBroad}}$ is not enabled, because $\mathtt{Er}_{\mathtt{Add}}(v) = \mathit{false}$ in γ_0 implies $\mathsf{Add}_u = \bot$, and rule $\mathbb{R}_{\mathtt{HypBroad}}$ is not enabled at u in this case. By Remark 5, $\mathtt{Er}_{\mathtt{PL}}(v)$ remains false after execution of $\mathcal{P}\mathit{ass}(u)$.

$\mathtt{Er}_{\mathtt{HC}}(v)$ (see Formula (48)): Command $\mathcal{B}in\mathit{Add}(v)$ does not modify the variables used in $\mathtt{Er}_{\mathtt{PL}}(v)$. So, we need to check variable HC_u of the child u of v in γ_1 (Remark 6) if $\mathsf{d}_v < \widehat{\mathsf{B}}_v$. Only commands $\mathcal{P}\mathit{ass}(u)$, $\mathcal{P}\mathit{ipe}(u)$, and $\mathcal{V}\mathit{erif}(u)$ may modify HC_u. By Remark 5, $\mathtt{Er}_{\mathtt{HC}}(v)$ remains false after execution of $\mathcal{P}\mathit{ass}(u)$. Now, $\mathsf{d}_u > 1$ since $\mathsf{par}_u = v$ and v is not a root, so rule $\mathbb{R}_{\mathtt{HypVerif}}$ is enabled in two case. If $\mathsf{HC}_v \neq (\bot, \bot)$ and $\mathsf{HC}_u \neq \mathsf{HC}_v$, command $\mathcal{V}\mathit{erif}(u)$ assigns $\mathsf{HC}_u = \mathsf{HC}_v$, and $\mathtt{Er}_{\mathtt{HC}}(v)$ remains false in γ_1. Otherwise rule $\mathbb{R}_{\mathtt{HypVerif}}$ is enabled if $\mathsf{HC}_v = (\bot, \bot)$, and command $\mathcal{V}\mathit{erif}(u)$ assings $\mathsf{HC}_u = (\bot, \bot)$. So,$\mathtt{Er}_{\mathtt{HC}}(v)$ remains false in γ_1.

Overall, starting in configuration in $\Gamma_{\mathtt{TEF}}$, after execution of command $\mathcal{B}in\mathit{Add}(v)$, predicate $\mathtt{Er}_{\mathtt{T}}(v, \gamma_1)$ remains false.

$\mathbb{R}_{\mathtt{HypBroad}}$: This rule is detailed in Subsection 4.4.2. Command $\mathcal{P}\mathit{ipe}(v)$ (see Formula (42)) modifies variables PL_v and HC_v.

$\mathtt{Er}_{\mathtt{Check}}(v)$ and $\mathtt{Er}_{\mathtt{HC}}(v)$: Command $\mathcal{V}\mathit{erif}(v)$ does not modify the variables used in $\mathtt{Er}_{\mathtt{Check}}(v)$ and $\mathtt{Er}_{\mathtt{HC}}(v)$, so these cases are dealt with in the proof of closure of rule $\mathbb{R}_{\mathtt{HypAdd}}$.

$\mathtt{Er}_{\mathtt{Add}}(v)$ (see Formula (36)): Command $\mathcal{P}\mathit{ipe}(v)$ does not modify the variables used in $\mathtt{Er}_{\mathtt{Add}}(v)$, so $\mathtt{Er}_{\mathtt{Add}}(v)$ may become true if and only if its child u modifies its variable used in $\mathtt{Er}_{\mathtt{Add}}(v)$ (Remark 6). In other words, if rule $\mathbb{R}_{\mathtt{HypAdd}}$ is enabled for u, and u executes $\mathcal{B}in\mathit{Add}(u)$. Rule $\mathbb{R}_{\mathtt{HypBroad}}$ is enabled for v if and only if $\mathsf{Add}_v \neq \bot$, so $\mathbb{R}_{\mathtt{HypAdd}}$ is not enabled for u because $\mathsf{Add}_u \neq \bot$ (Thanks to $\mathtt{Er}_{\mathtt{Add}}(v) = \mathit{false}$ in γ_0).

$\mathtt{Er}_{\mathtt{PL}}(v)$ (see Formula (43)): Command $\mathcal{P}\mathit{ipe}(v)$ modifies the variables used in $\mathtt{Er}_{\mathtt{PL}}(v)$. So, we need to check variable $\mathsf{PL}(u)$ of the parent u of v in γ_1 (Remark 6) if $\mathsf{d}_v > 1$. Only commands $\mathcal{P}\mathit{ipe}(u)$ and $\mathcal{P}\mathit{ass}(u)$ may modify PL_u. By Remark 5, $\mathtt{Er}_{\mathtt{PL}}(v)$ remains false after execution of $\mathcal{P}\mathit{ass}(u)$. predicates $\mathtt{Publish}(v)$ and $\mathtt{CleanPV}$ require

$\mathsf{PL}_u = (\bot, \bot)$, so $\mathsf{Add}_u = \bot$ (thanks to $\mathtt{Er}_{\mathtt{PL}}(v) = \mathit{false}$ in γ_0). Then, rule $\mathbb{R}_{\mathtt{HypBroad}}$ is not enabled for u in this case. predicate $\mathtt{Pipe}(v)$ requires $\mathsf{PL}_u \neq \mathsf{PL}_v$, so u has a child (v) with different variable PL. Then, $\mathtt{Publish}(u)$, $\mathtt{Pipe}(u)$, and $\mathtt{CleanPV}(u)$ are false in γ_0. As a consequence, rule $\mathbb{R}_{\mathtt{HypBroad}}$ is not enabled at u in this case.

Overall, starting from a configuration in $\Gamma_{\mathtt{TEF}}$, after execution of command $\mathcal{B}in\mathit{Add}(v)$, predicate $\mathtt{Er}_{\mathtt{T}}(v, \gamma_1)$ remains false.

$\mathbb{R}_{\mathtt{HypVerif}}$: This rule is detailed in Subsection 4.4.3. Command $\mathcal{V}\mathit{erif}(v)$ (see Formula (47)) only modifies variable HC_v.

$\mathtt{Er}_{\mathtt{Check}}(v)$, $\mathtt{Er}_{\mathtt{Add}}(v)$ and $\mathtt{Er}_{\mathtt{PL}}(v)$: Command $\mathcal{V}\mathit{erif}(v)$ does not modify the variables used in $\mathtt{Er}_{\mathtt{Check}}(v)$, $\mathtt{Er}_{\mathtt{Add}}(v)$ and $\mathtt{Er}_{\mathtt{PL}}(v)$. Son, these cases are dealt with in the proof of closure of rules $\mathbb{R}_{\mathtt{HypBroad}}$ and $\mathbb{R}_{\mathtt{HypAdd}}$.

$\mathtt{Er}_{\mathtt{HC}}(v)$ (see Formula (48)): Command $\mathcal{V}\mathit{erif}(v)$ does not modify the variables used in $\mathtt{Er}_{\mathtt{HC}}(v)$. So, we only need to check variable HC_u of the child u of v in γ_1 (Remark 6) if $\mathsf{d}_v < \widehat{\mathsf{B}}_v$. Only commands $\mathcal{P}\mathit{ass}(u)$ (see Formula (14)), $\mathcal{P}\mathit{ipe}(u)$ (see Formula (42)) and $\mathcal{V}\mathit{erif}(u)$ (see Formula (47)) may modify HC_u. By Remark 5, $\mathtt{Er}_{\mathtt{HC}}(v)$ remains false after execution of $\mathcal{P}\mathit{ass}(u)$. Then, $\mathsf{d}_u > 1$ because $\mathsf{par}_u = v$ and v is not a root, so rule $\mathbb{R}_{\mathtt{HypVerif}}$ is enabled in two case. If $\mathsf{HC}_v \neq (\bot, \bot)$ and $\mathsf{HC}_u \neq \mathsf{HC}_v$, command $\mathcal{V}\mathit{erif}(u)$ assigns $\mathsf{HC}_u = \mathsf{HC}_v$, and $\mathtt{Er}_{\mathtt{HC}}(v)$ remains false in γ_1. Otherwise rule $\mathbb{R}_{\mathtt{HypVerif}}$ is enabled if $\mathsf{HC}_v = (\bot, \bot)$, and command $\mathcal{V}\mathit{erif}(u)$ assigns $\mathsf{HC}_u = (\bot, \bot)$, so $\mathtt{Er}_{\mathtt{HC}}(v)$ remains false in γ_1. Command $\mathcal{P}\mathit{ipe}(u)$ modifies HC_u if and only if $\mathsf{d}_u = \widehat{\mathsf{B}}_u$. As u is the parent of v, $\mathsf{d}_v = 1$ and $\mathtt{Er}_{\mathtt{HC}}(v)$ remains false in γ_1.

Overall, starting in a configuration in $\Gamma_{\mathtt{TEF}}$, after execution of command $\mathcal{B}in\mathit{Add}(v)$, predicate $\mathtt{Er}_{\mathtt{T}}(v, \gamma_1)$ remains false.

To conclude, starting from a configuration $\gamma_0 \in \Gamma_{\mathtt{TEF}}$, Algorithm **CLE** preserves the fact that $\mathtt{Er}_{\mathtt{T}}(v)$ (see Formula (52)) remains false for all nodes v in V. □

We now define predicate $\Gamma_{\mathtt{CF}}$ (for *Cycle Free*), which ensures that no cycles induced by parent variables remain in the network. Let $\lambda : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\lambda(\gamma, v) = |\mathsf{dB}_{P_X} - \mathsf{dB}_X - 1|$$

Where X is a hyper-node containing v, and P_X is the hyper-node parent of X (recall that dB_X is an integer whose binary representation is $\mathsf{dB}_{x_1}, \ldots, \mathsf{dB}_{x_k}$ where $k = \widehat{\mathsf{B}}$). Let $\Lambda : \Gamma \to \mathbb{N}$ be the function defined by:

$$\Lambda(\gamma) = \sum_{v \in V} \lambda(\gamma, v).$$

Let $\phi : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\phi(\gamma, v) = \begin{cases} |\mathsf{d}_{\mathsf{par}_v} - \mathsf{d}_v - 1| & \text{if} \quad \mathsf{d}_{\mathsf{par}_v} < \widehat{\mathsf{B}}_v \\ |\mathsf{d}_v - 1| & \text{if} \quad \mathsf{d}_{\mathsf{par}_v} = \widehat{\mathsf{B}}_v \end{cases}$$

Let $\Phi : \Gamma \to \mathbb{N}$ be the function defined by:

$$\Phi(\gamma) = \sum_{v \in V} \phi(\gamma, v).$$

We now define:

$$\Gamma_{\mathtt{CF}} = \{\gamma \in \Gamma_{\mathtt{TEF}} : \Phi(\gamma) = \Lambda(\gamma) = 0 \text{ and } L(\gamma) > 0\}.$$

Lemma 3 $\Gamma_{\texttt{TEF}} \triangleright \Gamma_{\texttt{CF}}$ *in* $O(n \log n)$ *rounds.*

Proof. Consider an initial configuration $\gamma_0 \in \Gamma_{\texttt{TEF}}$ such that the structure induced by the parent variables par_v, for $v \in V$, forms a cycle C. The cycle C necessarily contains all nodes, which implies that no node has an empty parent pointer. Moreover, since $\gamma_0 \in \Gamma_{\texttt{TEF}}$, for every node v, $\mathsf{leader}_v \neq 1$. Rule $\mathbb{R}_{\texttt{Passive}}$ can modify the parent pointer. Let us consider a configuration γ_0 where no node is enabled with respect to rule $\mathbb{R}_{\texttt{Passive}}$. Then, for all $u, v \in V$ ($\mathsf{Elec}_v = \mathsf{Elec}_u$). Otherwise in a cycle, there would exist at least a node v with a best neighbor u (see Functions `NgSupBs` (see Formula (9)), `NgInfPhC` (see Formula (11)), `NgSupBp` (see Formula (10)), and `Best` (see Formula (13))). Since $L(\gamma) = 0$, there is no root, so for all $v \in V$, $\mathsf{d}_v > 0$. As a consequence, rule $\mathbb{R}_{\texttt{Root}}$ (see $\mathsf{d}_v = 0$ in rule $\mathbb{R}_{\texttt{Root}}$ in Subsection 4.3.4) is not enabled. Therefore, command $\mathcal{I}nc(v)$ cannot be executed at any node v. Then, the system reaches a configuration $\gamma_{0'} \in \Gamma_{\texttt{TEF}}$, where the rule $\mathbb{R}_{\texttt{Update}}(v)$ is not enabled, for every node $v \in V$ (see Subsection 4.3.5).

Claim: $\Phi(\gamma_{0'}) = 0$.

Assume for the purpose of contradiction that $\Phi(\gamma_{0'}) \neq 0$. Then there exists at least one passive node v that detects an error between its distance and the distance of its parent (see predicates $\texttt{CohP}(v)$ (see Formula (21)) and $\texttt{Er}_{\texttt{Nd}}$ (see Formula (29))), which contradicts $\gamma_{0'} \in \Gamma_{\texttt{TEF}}$. Thus, $\Phi(\gamma_{0'}) = 0$. A direct consequence of the fact that $\Phi(\gamma_{0'}) = 0$ is that the number of hyper-nodes in $\gamma_{0'}$ is exactly $n/\texttt{Max}\widehat{\mathsf{B}}$.

We are now ready to show that, if the initial configuration γ_0 contains a cycle, then Algorithm **CLE** detects an error in $O(n \log n)$ rounds.

We remark that the assignment dB is made when a node takes its parent or changes its parent (see rule $\mathbb{R}_{\texttt{Passive}}$ in Subsection 4.3.3). Moreover, all passive nodes v have $\mathsf{dB}_v \in \{0, 1\}$ by definition of variable dB_v. This would contradict the hypothesis that $\gamma_{0'} \in \Gamma_{\texttt{TEF}}$. The distance hyper-node verification can be made. Since all nodes are passive in $\gamma_{0'}$, the only commands that can be executed by a node are those related to the distance verification between hyper-nodes, that is, commands $\mathcal{B}inAdd(v), \mathcal{P}ipe(v)$ and $\mathcal{V}erif(v)$. More specifically, the only nodes that can possibly be activated in $\gamma_{0'}$ are the nodes v such that $\mathsf{d}_v = \widehat{\mathsf{B}}_v$.

For every hyper-node $X = (x_1, x_2, \ldots, x_k)$, where $k = \texttt{Max}\widehat{\mathsf{B}}$, since the scheduler is weakly fair, predicate $\texttt{T.Add}(x_k) = true$, and x_k executes command $\mathcal{B}inAdd(x_k)$ at round 1. This yields the proper execution of the binary addition. The binary addition occurs from x_k to x_1, and every node in each hyper-node X eventually takes value "$+$" or "ok" once $\texttt{Max}\widehat{\mathsf{B}}$ rounds have elapsed. Now, if $\mathsf{dB}_{x_1} = 1$ and $\mathsf{Add}_{x_1} = +$, then an error is detected since the binary addition overflows (see predicate $\texttt{Er}_{\texttt{Overflow}}(v)$ (see Formula (37))).

Node x_1 starts the verification process that propagates from x_1 to x_k. Consider Hyper-node $X = (x_1, x_2, \ldots, x_k)$, and let us denote by Y the child hyper-node of X in the current configuration at round $\texttt{Max}\widehat{\mathsf{B}}$. Node x_1 computes the values of dB_{y_1} (see predicate `T.Pipe`, and command $\mathcal{P}ipe$). This value is broadcast from x_1 to x_k (see predicate `T.Pipe`, and command $\mathcal{P}ipe$). Node y_1 checks whether $\mathsf{PL}_{x_k}[1] = \mathsf{dB}_{y_1}$. If it is the case, then the verification process for all other nodes in Y carries on (see predicate `T.Verif` and command $\mathcal{V}erif$). Otherwise y_1 detects an error (see $\texttt{Er}_{\texttt{HypDis}}(v)$ (see Formula (46))).

Thanks to predicates `T.Add`, `T.Pipe`, and `T.Verif`, and to commands $\mathcal{B}inAdd$, $\mathcal{P}ipe$, and $\mathcal{V}erif$, all node in Y are eventually checked, after an additional $\texttt{Max}\widehat{\mathsf{B}}$ rounds. The total number of rounds for checking hyper-nodes is the following, assuming $\texttt{Max}\widehat{\mathsf{B}} = \lfloor \log n \rfloor$: there are $n/\lfloor \log n \rfloor$ hyper-nodes, and each hyper-node completes verification in $O(\lfloor \log n \rfloor)$ rounds, so the overall process takes $O(n \log n)$ rounds.

To sum-up, if the configuration γ_0 contains a cycle, then at least one node v of a hyper-node detects an error in $O(n \log n)$ rounds, and executes command $\mathcal{F}reeze(v)$ (see Formula (4)).

Finally, if $L(\gamma_0) = 0$ in an arbitrary configuration, then Algorithm **CLE** detects an error in $O(n \log n)$ rounds. Then, by Lemma 1, the node detecting the cycle resets its parent variable in one round. As a consequence, the system reaches a configuration in $\Gamma_{\texttt{CF}}$. □

Lemma 4 *$\Gamma_{\texttt{CF}}$ is closed.*

Proof. Let us consider a configuration $\gamma \in \Gamma_{\texttt{CF}}$. We already noticed in the proof of Lemma 2 that Algorithm **CLE** preserves coherent distances (i.e., $\Phi(\gamma) = 0$), and does not introduce trivial errors (i.e., $\Psi(\gamma) = 0$). Moreover, in the proof of Lemma 3, we have explained that the hyper-node distance verification correctly reports errors, if any. variable dB is only modified by command $\mathcal{P}ass$. In the sequel, we use the wording "v joins $\mathtt{T}_r$" when a node v executes $\mathcal{P}ass$, and the pointer par_v of v then points to a node in the subtree $\mathtt{T}_r$ rooted at r.

We denote by X the set of candidate leader nodes. Let us first consider a node v (in X or not), and a root $r \in X$ such that $\mathsf{d}_v \leq \lfloor \log n \rfloor$, and v joins $\mathtt{T}_r$ when r increases its phase from $i-1$ to i, for some integer i. Thanks to predicate $\mathtt{T.StartdB}$ and to command $\mathcal{S}tartdB$, r publishes first the bit for the node whose distance from r is 1. In other words, $\mathsf{PL}_r = (1, 0)$. When any node v at distance 1 joins $\mathtt{T}_r$, v sets $\mathsf{dB}_v = \mathsf{PL}_r[1]$, and then v informs r about the updating of dB by assigning $\mathsf{PL}_v = \mathsf{PL}_r$. At this point, r can publish the bit for nodes at distance 2 (i.e., $\mathsf{PL}_r = (2, 0)$), and so on until the distance reaches $\lfloor \log n \rfloor$. Now, a node v joins $\mathtt{T}_r$ only if its candidate leader parent publishes the bit that corresponds to the binary representation of the distance between v and r. In other words, for any node u, if $\mathsf{d}_u = k$ with $k < \lfloor \log n \rfloor$, then $\widehat{\mathsf{B}}_u$ must be equal to $k + 1$. This enables $\Lambda(\gamma) = 0$ to remain invariant. When k reaches $\lfloor \log n \rfloor$, a hyper-node is created. Then, a binary addition process is carried out, and computes the bit for v when v joins $\mathtt{T}_r$. This process maintains $\Lambda(\gamma) = 0$, and, as a direct consequence, $L(\gamma)$ remains greater than 0. To conclude, algorithm **CLE** maintains $\Psi(\gamma) = 0$, $\Phi(\gamma) = 0$, $\Lambda(\gamma) = 0$, and $L(\gamma) > 0$. □

We now introduce predicate $\Gamma_{\texttt{IEF}}$ (for *Impostor Error Free*), which ensures that the currently elected leader is not an impostor. Let $\xi : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\xi(\gamma, v) = |\mathtt{maxFB} - \widehat{\mathsf{B}}_v|$$

where $\mathtt{maxFB} = \max\{\mathrm{Bit}(1, \mathsf{Id}_v) : v \in V\}$. Let $\Xi : \Gamma \to \mathbb{N}$ be the function defined by:

$$\Xi(\gamma) = \sum_{v \in V} \xi(\gamma, v).$$

We show that **CLE** reaches a legitimate configuration with respect to leader election if and only if $\Xi(\gamma) = 0$.

Let $\epsilon : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\epsilon(\gamma, v) = \begin{cases} 0 & \text{if} \quad \mathsf{d}_v = 0 \wedge \mathrm{Bit}(\mathtt{minPh}, \mathsf{Id}_v) = \mathrm{Bit}(\mathtt{minPh}, l^*) \\ 1 & \text{if} \quad \mathsf{d}_v = 0 \wedge \mathrm{Bit}(\mathtt{minPh}, \mathsf{Id}_v) < \mathrm{Bit}(\mathtt{minPh}, l^*) \\ 0 & \text{otherwise} \end{cases}$$

where $\mathtt{minPh} = \min\{\mathsf{Phase}_v : v \in V\}$ and l^* is the identity of the node with the maximum identity. Let $\Upsilon : \Gamma \to \mathbb{N}$ be the function defined by:

$$\Upsilon(\gamma) = \sum_{v \in V} \epsilon(\gamma, v)$$

We define:

$$\Gamma_{\texttt{IEF}} = \{\gamma \in \Gamma_{\texttt{CF}} : \Psi(\gamma) = \Xi(\gamma) = \Upsilon(\gamma) = 0\}.$$

Lemma 5 *$\Gamma_{\texttt{CF}} \triangleright \Gamma_{\texttt{IEF}}$ in $O(n \log n)$ rounds.*

Proof. Let us first consider an initial configuration $\gamma \in \Gamma_{\texttt{CF}}$. We observed in Lemma 3 that γ being in $\Gamma_{\texttt{CF}}$ implies $L(\gamma) > 0$, and at least one node has no parent. As frozen nodes do not participate to the election process, without loss of generality, we consider that the system contains no frozen node.

We remark that, in a configuration $\gamma \in \Gamma_{\texttt{IEF}}$, a root v such that $\mathsf{Phase}_v = \widehat{\mathsf{B}}_v$ can restart the phase from the begining if and only if all the nodes are in its subtree, or all subtree are in phase $\mathsf{Phase} = \widehat{\mathsf{B}}_v$. Let us suppose the opposite, let $\mathtt{T}_v$ denote the subtree rooted at v (such that $\mathsf{Phase}_v = \widehat{\mathsf{B}}_v$). Let u be the neighbor of ℓ_v, a leaf of $\mathtt{T}_v$. If $\mathsf{Phase}_u < \mathsf{Phase}_{\ell_v}$, then u is a best neighbor of ℓ_v. If $\mathsf{Check}_{\ell_v} = 1$, then an error is detected, which contradicts the hypothesis that $\gamma \in \Gamma_{\texttt{IEF}}$. Otherwise, if $\mathsf{Check}_{\ell_v} = 0$, then $\mathtt{T}_v$ reaches the subtree containing u.

Let us denote by $\mathscr{L}^*$ the node with maximum identity, and let X denote the set of candidate leader nodes. Let us suppose that $\mathscr{L}^* \notin X$. Then, let $\mathscr{L}$ be the node with maximum identity in X. At some point in the execution, the system reaches a configuration where all sub-spanning tree merge in a unique spanning tree rooted at $\mathscr{L}$. Thus, let us suppose that γ is a configuration where the network is spanned by an unique tree rooted at $\mathscr{L}$. In this case, $\mathsf{d}_{\mathscr{L}} = 0$ and $\mathsf{d}_u > 0$, for every node $u \neq \mathscr{L}$.

Let us assume, for the purpose of contradiction, that $\Xi(\gamma) \neq 0$. If the tree is rooted at $\mathscr{L}$, then every node must have the same $\widehat{\mathsf{B}}$ as $\mathscr{L}$. Since $\mathscr{L}$ is a root, $\widehat{\mathsf{B}}_{\mathscr{L}} = \mathrm{Bit}(1, \mathsf{Id}_{\mathscr{L}})$. Hence, $\widehat{\mathsf{B}}_{\mathscr{L}} \neq \mathtt{maxFB}$ (because $\Xi(\gamma) \neq 0$). Now, $\widehat{\mathsf{B}}_{\mathscr{L}}$ cannot be larger than $\mathtt{maxFB}$, so there exists v such that $\widehat{\mathsf{B}}_v = \mathtt{maxFB}$, and $\mathtt{PassNd}(v)$ is true. This contradicts $\gamma \in \Gamma_{\texttt{TEF}}$, so we can conclude that $\Xi(\gamma) = 0$.

Since $\mathscr{L}$ and $\mathscr{L}^*$ have the same number of bits, there must exist one phase where the bit-position of $\mathscr{L}^*$ is larger than the bit-position of $\mathscr{L}$. More formally, there exists i, $1 < i \leq \lfloor \log n \rfloor + 1$, such that, for every $j < i$, we have $\mathrm{Bit}(j, \mathsf{Id}_{\mathscr{L}}) = \mathrm{Bit}(j, \mathsf{Id}_{\mathscr{L}^*})$, and, for every $k \geq i$, we have $\mathrm{Bit}(k, \mathsf{Id}_{\mathscr{L}}) < \mathrm{Bit}(k, \mathsf{Id}_{\mathscr{L}^*})$. Note that $i > 1$, because, in $\Gamma_{\texttt{TEF}}$, predicate $\mathtt{PassNd}$ must be true. The worst case with respect to time complexity is for $i = \lfloor \log n \rfloor + 1$, and the arbitrary initial configuration starts at phase 2. In this case, only $\mathbb{R}_{\mathtt{Root}}$ can be activated for $\mathscr{L}$, and only rule $\mathbb{R}_{\mathtt{Update}}$ for the other nodes (in parallel to the hyper-node distance verification). Node ℓ executes command $\mathcal{I}nc(\mathscr{L})$ $\lfloor \log n \rfloor + 1 - 2$ times. After each execution of command $\mathcal{I}nc$, every node executing this command updates Elec in a top-down manner (see predicate $\mathtt{T.Up}$ and command $\mathcal{U}pdate$). This updating process takes at most n rounds. When all nodes have the same election values, a bottom-up control process is initiated (see predicate $\mathtt{T.Up}$ and command $\mathcal{U}pdate$). This process takes at most n rounds. After that, $\mathscr{L}$ increases its phase, and the same process is repeated. At the last phase, $\mathtt{Er}_{\mathtt{Place}}(\mathscr{L}^*) = \mathit{true}$ holds. Then, an error is detected. Therefore, if the system contains a impostor leader, then an error is detected in $O(n \log n)$ rounds. □

Lemma 6 *$\Gamma_{\texttt{IEF}}$ is closed.*

Proof. Let $\gamma \in \Gamma_{\texttt{IEF}}$, with $L(\gamma) > 0$. Let X be the set of candidate leaders (i.e., for every $x \in X$, $\mathtt{Root}(x) = \mathit{true}$). Let $x \in X$, and let $\mathtt{T}_{(x,i)}$ be the subtree rooted at x during phase i. For a node $v \in \mathtt{T}_{(x,i)}$, we have *(i)* either $\widehat{\mathsf{B}}_v < \widehat{\mathsf{B}}_x$ or $\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_x$, or *ii)* $\mathrm{Bit}(j, \mathsf{Id}_v) < \mathrm{Bit}(j, \mathsf{Id}_x)$

for every $j \leq i$. Moreover, for any two candidates leaders x_1 and x_2, we have that, at phase i, $\widehat{\mathsf{B}}_{x_1} = \widehat{\mathsf{B}}_{x_2}$, and $\mathrm{Bit}(j, \mathsf{Id}_{x_1}) = \mathrm{Bit}(j, \mathsf{Id}_{x_2})$, for every $j \leq i$. Let i be such that $\mathtt{T}_{(x_1,i)}$ and $\mathtt{T}_{(x_2,i)}$ have adjacent nodes. Let $x'_1 \in \mathtt{T}_{(x_1,i)}$ and $x'_2 \in \mathtt{T}_{(x_2,i)}$ be two nodes such that x'_1 is adjacent to x'_2. If, at phase $i+1$, $\mathrm{Bit}(i+1, \mathsf{Id}_{x_1}) > \mathrm{Bit}(i+1, \mathsf{Id}_{x_2})$, then the nodes of $\mathtt{T}_{(x_1,i)}$ and $\mathtt{T}_{(x_2,i)}$, activated by predicate $\mathtt{T.Up}$, execute $\mathcal{U}pdate$. When x'_1 reaches Elec_{x_1} at phase $i+1$, x'_2 becomes passive (cf. command $\mathcal{P}ass$) and selects x'_1 as its parent. In a bottom-up fashion, every node of $\mathtt{T}_{(x_2,i)}$ joins the subtree $\mathtt{T}_{(x_1,i+1)}$ (cf. command $\mathcal{P}ass$), and, eventually, x_2 becomes passive and joins $\mathtt{T}_{(x_1,i+1)}$. By this process, for every node v in $\mathtt{T}_{(x_1,i+1)}$, we have *(i)* either $\widehat{\mathsf{B}}_v < \widehat{\mathsf{B}}_{x_1}$ or $\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_{x_1}$, and *ii)* $\mathrm{Bit}(j, \mathsf{Id}_v) < \mathrm{Bit}(j, \mathsf{Id}_{x_1})$ for every $j \leq i+1$. This process is repeated until phase $\lfloor \log n \rfloor$, where there remains a single leader in the network. □

Let us define the predicate $\Gamma_{\mathtt{NF}}$ (*No Frozen node*). We define $\mathtt{F}(\gamma)$, the set of nodes v with $\mathsf{leader}_v = 2$ and $\mathsf{par}_v = \emptyset$ in γ, $\mathtt{T}_v(\gamma)$, the tree rooted at $v \in \mathtt{F}$, and $\mathtt{T}^2_v(\gamma) = \{u \in \mathtt{T}(\gamma)_v : \mathsf{leader}_u = 2\}$. Let $\eta : \Gamma \times \mathtt{F} \to \mathbb{N}$ be the function defined by:

$$\eta(\gamma, v) = \begin{cases} n^2 - |\mathtt{T}^2_v(\gamma)| & \text{if} \quad \mathtt{T}^2_v(\gamma) \neq \mathtt{T}_v(\gamma) \\ |\mathtt{T}^2_v(\gamma)| & \text{otherwise} \end{cases}$$

Let $\kappa : \Gamma \to \mathbb{N}$ be the function defined by:

$$\kappa(\gamma) = \sum_{v \in \mathtt{F}} \eta(\gamma, v)$$

Note that all nodes are not frozen node if and only if $\kappa(\gamma) = 0$. We define:

$$\Gamma_{\mathtt{NF}} = \{\gamma \in \Gamma_{\mathtt{IEF}} : \kappa(\gamma) = 0\}.$$

Lemma 7 *$\Gamma_{\mathtt{IEF}} \triangleright \Gamma_{\mathtt{NF}}$ in $O(n)$ rounds.*

Proof. Let us consider an initial configuration $\gamma_0 \in \Gamma_{\mathtt{IEF}}$ such that $\eta(\gamma_0) \neq 0$. As $\Gamma_{\mathtt{TEF}} \subset \Gamma_{\mathtt{IEF}}$, if a node v has $\mathsf{leader}_v = 2$, then either v has no parent, or the parent u of v has $\mathsf{leader}_u = 2$ (see $\mathtt{Er}_{\mathtt{Freeze}}(v)$ (see Formula (3))). Without loss of generality, let us consider that the set $\mathtt{F}(\gamma_0)$ is composed of exactly two nodes u and v such that $\mathtt{T}^2_u \neq \mathtt{T}_u$ and $\mathtt{T}^2_v = \mathtt{T}_v$ (see Figure 6). As a consequence $\kappa(\gamma_0) = n^2 - |\mathtt{T}^2_u(\gamma_0)| + |\mathtt{T}^2_v(\gamma_0)|$.

- Let $x \in \mathtt{T}_u$ and $x \in \mathtt{T}^2_u$ be the first descendant of u such that $\mathsf{leader}_{\mathsf{par}_x} = 2$ and $\mathsf{leader}_x \neq 2$. So, x is continuously enabled by rule $\mathbb{R}_{\mathtt{Error}}$, while command $\mathcal{F}reeze(x)$ is not executed, due to the presence of $\neg\mathtt{Error}(x)$ in all the others rules (see Algorithm **CLE**).
- Let ℓ_v be the leaf of tree $\mathtt{T}_v(\gamma_0)$. So ℓ_v is continuously enabled for rule $\mathbb{R}_{\mathtt{Start}}$, while command $\mathcal{S}tart(x)$ is not executed. Yet, $\mathbb{R}_{\mathtt{Start}}$ is the only rule enabled with $\mathsf{leader}_{\ell_v} = 2$ (see Algorithm 1)).
- Let ℓ_u be the leaf of $\mathtt{T}_u$, so ℓ_u is not enabled until its parent becomes frozen. Because the rules of hyper-nodes increment require that all neighbors of ℓ_v are in the same tree as ℓ_v, that is not the case (ℓ_v is a leaf). Rule $\mathbb{R}_{\mathtt{Passive}}(\ell_u)$ requires a best neighbor, but the parent of ℓ_u is in the same tree, so it is not a best neighbor. Moreover, node v has $\mathsf{leader}_v = 2$, so v is not a best neighbor (see $\mathtt{Best}(v)$ Formula (13)).
- Node y_i not in $\mathtt{T}_v$ or $\mathtt{T}_u$ between u and ℓ_v cannot reach $\mathtt{T}_v$ or $\mathtt{T}_u$, because a node a with $\mathsf{leader}_a = 2$ cannot be a best neighbor (see $\mathtt{Best}(a)$ Formula (13)).
- A node z, neighbor of ℓ_u, can reach $\mathtt{T}_u$ if and only if $\mathtt{Best}(z) = \ell_u$, otherwise this node is not enabled. We remark that a node in $\mathtt{T}_u$ but not in $\mathtt{T}^2_u$ has no impact on κ.

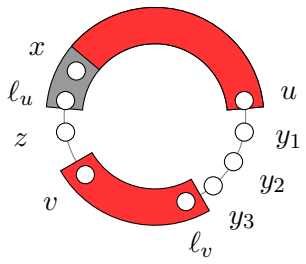


Figure 6:

Since the scheduler is weakly fair, and x is continuously enabled, there exists a configuration γ' where the scheduler activates x, and we get $|\mathtt{T}_u^2(\gamma')| > |\mathtt{T}_u^2(\gamma_0)|$, so $\kappa(\gamma') < H(\gamma_0)$. Likewise, for ℓ_v, and we obtain $|\mathtt{T}_v^2(\gamma')| < |\mathtt{T}_v^2(\gamma_0)|$, so $\kappa(\gamma') < H(\gamma_0)$. Now, if the scheduler activates x and ℓ_v in the same round, we obtain $|\mathtt{T}_u^2(\gamma')| > |\mathtt{T}_u^2(\gamma_0)|$ and $|\mathtt{T}_v^2(\gamma')| < |\mathtt{T}_v^2(\gamma_0)|$, so $\kappa(\gamma') < H(\gamma_0)$. As a consequence, while $\mathtt{F}$ is non-empty, starting from configuration γ_0, the activation of rule $\mathbb{R}_{\mathtt{Error}}$ or rule $\mathbb{R}_{\mathtt{Start}}$ result in a configuration γ' such that $\kappa(\gamma') < H(\gamma)$.

The number of rounds are bounded by the last frozen tree that disappears. Let us denote by $\mathtt{T}_v$ such a tree. Suppose that in γ_0, $\mathtt{T}_v^2$ contains only v. Then, the child u of v is continuously enabled, so the first round ends when u is active, and so on until $\mathtt{T}_v = \mathtt{T}_v^2 = n$. Now, the leaf ℓ_v of $\mathtt{T}_v^2$ is continuously enabled, so the round ends when ℓ_v is activates, and so on until v is started and $\mathtt{T}_v^2$ is empty, that is in at most n rounds. So starting in a configuration γ whith $\mathtt{F} \neq \emptyset$, the system converge in configuration $\gamma' \in \Gamma_{\mathtt{NF}}$ in $2n$ rounds. □

Lemma 8 *$\Gamma_{\mathtt{NF}}$ is closed.*

Proof. Directly by Lemmas 2, 4 and 6. □

Lemma 9 *$\Gamma_{\mathtt{NF}} \triangleright \Gamma_{\mathtt{LE}}$ in $O(n \log^2 n)$ rounds.*

Proof. We know by Lemma 4 that $\Gamma_{\mathtt{CF}}$ is closed, and we know by Lemma 6 that $\Gamma_{\mathtt{IEF}}$ is closed. Moreover the proof of Lemma 6 provides details about the election process at each phase. Let $\gamma \in \Gamma_{\mathtt{IEF}}$ at round t. Moreover, let us suppose that $\gamma \in \Gamma_{\mathtt{CF}}$. That is, $L(\gamma) > 0$. More precisely, let i, $1 \leq i \leq \lfloor \log n \rfloor + 1$, denote the smallest phase counter in the network, among all nodes. At phase i, there are at most $n/2^{i-1}$ candidate leaders (i.e., at most $n/2^{i-1}$ roots). Thus, $L(\gamma) = n/2^{i-1}$ at phase i. We have studied in the proof of Lemma 5 how Algorithm **CLE** performs the election process. After $O(n \log n)$ rounds, phase $i+1$ is completed, and the system reaches some configuration γ'. At this point, there remains at most $n/2^i$ candidate leaders. Since $L(\gamma') \leq n/2^i$, we get that:

$$L(\gamma') < L(\gamma).$$

The number of phases is upper bounded by $\lfloor \log n \rfloor + 1$. At phase $\lfloor \log n \rfloor + 1$, we reach a configuration γ'' satisfying $L(\gamma'') = 1$. A direct consequence of Lemmas 5 and 6 is that only the node $\mathscr{L}^*$ with maximum identity has $\mathsf{leader}_{\mathscr{L}^*} = 1$. Every other node v has $\mathsf{leader}_v = 0$. Moreover, for every node $v \neq \mathscr{L}^*$, we have $\mathsf{par}_v \neq \emptyset$, and the structure induced by the pointers par_v, for all $v \neq \mathscr{L}^*$ forms a spanning tree rooted at $\mathscr{L}^*$. Regarding time complexity, our algorithm takes $O(n \log n)$ rounds to detect an impostor leader, $O(n)$ rounds to clean the system after the detection of an error, and $O(n \log^2 n)$ rounds to elect the leader. Therefore, in total, Algorithm **CLE** performs $O(n \log^2 n)$ rounds to converge to the leader specification. □

Lemma 10 $\Gamma_{\texttt{LE}}$ *is closed.*

Proof. The rule $\mathbb{R}_{\texttt{Passive}}(v)$ is the only rule performed by v that modifies the distance and the leader variables of node v. Let $\mathscr{L}^*$ be the node with the maximum identity. As a direct consequence of Lemma 9, in the initial configuration, $\mathscr{L}^*$ is the only node that has $\mathsf{d}_{\mathscr{L}^*} = 0$ and $\mathsf{leader}_{\mathscr{L}^*} = 1$. In other words, $\mathscr{L}^*$ is the only elected node. Therefore, $\mathscr{L}^*$ changes the phase of the system by increasing the current phase, or by restarting from phase 1 (see predicate `T.Inc` and command $\mathcal{I}nc$). Thus, every node v satisfies $\widehat{\mathsf{B}}_v \leq \widehat{\mathsf{B}}_{\mathscr{L}^*}$. Moreover, for every phase i, $1 \leq i \leq \lfloor \log n \rfloor + 1$, every node v satisfies $\mathrm{Bit}(i, \mathsf{Id}_v) < \mathsf{Place}_{\mathscr{L}^*}$. Hence, every node can only executes the command $\mathcal{U}pdate$, and the commands related to the hyper-node distances verification. Finally, nodes never change their distance, parent, and leader variables. □

5.1 Memory requirements

Lemma 11 *Algorithm* **CLE** *use* $O(\log \log n)$ *bits of memory per node.*

Proof. Algorithm **CLE** has two types of variables: the variables that use a constant number of bits, and those that use $O(\log \log n)$ bits. Variables of the first type are:

$$\mathsf{par}_v \in \{\emptyset, 0, 1\}, \quad \mathsf{dB}_v \in \{0, 1\}, \quad \mathsf{Add}_v \in \{+, ok, \emptyset\}, \text{ and } \mathsf{leader}_v \in \{0, 1, 2\}.$$

Variables of the second type are:

$$\widehat{\mathsf{B}}_v \in \{1, ..., \lfloor \log n \rfloor\}, \quad \mathsf{PL}_v \in \{1, ..., \lfloor \log n \rfloor\} \times \{0, 1\}, \quad \mathsf{HC}_v \in \{1, ..., \lfloor \log n \rfloor\} \times \{0, 1\},$$

and

$$\mathsf{Elec}_v \in \{1, ..., \lfloor \log n \rfloor\} \times \{1, ..., \lfloor \log n \rfloor\} \times \{0, 1\}.$$

Hence, **CLE** uses $O(\log \log n)$ bits of memory per node. □

6 Conclusion

In this paper, we have shown that, in the state model, with a weakly fair distributed scheduler, one can elect a leader in a ring with a (talkative) self-stabilizing algorithm using only $O(\log \log n)$ bits of memory per node.

The techniques developed in this paper suggest that more tasks based on unique process identifiers could be solved with $o(\log n)$ bits of memory per node, when allowing the protocol to be talkative. One natural candidate is the center finding task in general graphs, that is known to require $\Omega(\log n)$ bits of memory per node if required to be silent [21]. To our knowledge, the best talkative protocol [14] for this problem does match this bound. So, the existence of a talkative solution with $o(\log n)$ bits remains open.

Finally, it is known that one cannot do the same using only $O(1)$ bits of memory per node [9]. An intriguing question is whether one can perform leader election in the same framework as in this paper, using $o(\log \log n)$ bits per node. By applying the techniques in this paper recursively, it might be possible to reduce the memory requirements to $O(\log^* n)$ bits of memory per node. However, this seems non-trivial, as self-stabilization has to be maintained at every level of the recursion.

References

[1] J. Adamek, M. Nesterenko, and S. Tixeuil. Using abstract simulation for performance evaluation of stabilizing algorithms: The case of propagation of information with feedback. In *SSS 2012*, LNCS. Springer, 2012.

[2] Y. Afek and A. Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998.

[3] Sudhanshu Aggarwal and Shay Kutten. Time optimal self-stabilizing spanning tree algorithms. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15-17, 1993, Proceedings*, volume 761 of *Lecture Notes in Computer Science*, pages 400–410. Springer, 1993.

[4] A. Arora and M. G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[5] M. Arumugam and S. S. Kulkarni. Prose: A programming tool for rapid prototyping of sensor networks. In *S-CUBE*, pages 158–173, 2009.

[6] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Trans. Dependable Sec. Comput.*, 4(3):180–190, 2007.

[7] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC*, pages 254–263. ACM, 1994.

[8] J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1):39–51, June 2007.

[9] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, January 2007.

[10] Joffroy Beauquier, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilizing census with cut-through constraint. In Anish Arora, editor, *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 70–77. IEEE Computer Society, 1999.

[11] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 199–208, 1999.

[12] L. Blin and S. Tixeuil. Brief announcement: deterministic self-stabilizing leader election with o(log log n)-bits. In *Proceedings of the 32st ACM Symposium on Principles of Distributed Computing, (PODC13)*, pages 125–127, 2013.

[13] L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election: The exponential advantage of being talkative. In *Proceedings of the 27th International Conference on Distributed Computing (DISC 2013)*, Lecture Notes in Computer Science (LNCS), pages 76–90. Springer Berlin / Heidelberg, 2013.

[14] Lelia Blin, Fadaw Boubekeur, and Swan Dubois. A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon. In *IPDPS 2015*, pages 1056–1074, 2015.

[15] Y. Choi and M. G. Gouda. A state-based model of sensor protocols. *Theor. Comput. Sci.*, 458:61–75, 2012.

[16] A. R. Dalton, W. P. McCartney, K. Ghosh Dastidar, J. O. Hallstrom, N. Sridhar, T. Herman, W. Leal, A. Arora, and M. G. Gouda. Desal alpha: An implementation of the dynamic embedded sensor-actuator language. In *ICCCN*, pages 541–547. IEEE, 2008.

[17] A. Kumar Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *TCS*, 412(40):5541–5561, 2011.

[18] S. Devismes, T. Masuzawa, and S. Tixeuil. Communication efficiency in self-stabilizing silent protocols. In *ICDCS 2009*, pages 474–481. IEEE Press, 2009.

[19] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[20] S. Dolev. *Self-stabilization.* MIT Press, March 2000.

[21] S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[22] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[23] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

[24] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election (extended absrtact). In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop, WDAG '91, Delphi, Greece, October 7-9, 1991, Proceedings*, volume 579 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 1991.

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

[26] F. E. Fich and C. Johnen. A space optimal, deterministic, self-stabilizing, leader election algorithm for unidirectional rings. In *DISC*, pages 224–239. Springer, 2001.

[27] M. G. Gouda, J. Arturo Cobb, and C. Huang. Fault masking in tri-redundant systems. In *SSS*, LNCS, pages 304–313. Springer, 2006.

[28] T. Herman and S. V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.

[29] J. Hoepman. Self-stabilizing ring-orientation using constant space. *Inf. Comput.*, 144(1):18–39, 1998.

[30] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Inf. Comput.*, 104(2):175–196, 1993.

[31] G. Itkis and L. A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *FOCS*, pages 226–239. IEEE Computer Society, 1994.

[32] G. Itkis, C. Lin, and J. Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *WDAG*, LNCS, pages 288–302. Springer, 1995.

[33] Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an mst. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011*, pages 311–320, 2011.

[34] T. Masuzawa and S. Tixeuil. On bootstrapping topology knowledge in anonymous networks. *ACM Transactions on Adaptive and Autonomous Systems*, 4(1), 2009.

[35] A. J. Mayer, Y. Ofek, R.l Ostrovsky, and M. Yung. Self-stabilizing symmetry breaking in constant-space (extended abstract). In *STOC*, pages 667–678, 1992.

[36] T. M. McGuire and M. G. Gouda. *The Austin Protocol Compiler*, volume 13 of *Advances in Information Security*. Springer, 2005.

[37] S. Tixeuil. *Algorithms and Theory of Computation Handbook*, pages 26.1–26.45. CRC Press, Taylor & Francis Group, 2009.