Content-Oblivious Leader Election in 2-Edge-Connected Networks

Jérémie Chalopin
Aix Marseille Univ, CNRS, LIS, Marseille, France

Yi-Jun Chang
National University of Singapore, Singapore

Lyuting Chen
National University of Singapore, Singapore

Giuseppe A. Di Luna
DIAG, Sapienza University of Rome, Italy

Haoran Zhou
National University of Singapore, Singapore

Abstract

Censor-Hillel, Cohen, Gelles, and Sela (PODC 2022 & Distributed Computing 2023) studied *fully-defective* asynchronous networks, where communication channels may arbitrarily corrupt messages. The model is equivalent to *content-oblivious* computation, where nodes communicate solely via pulses. They showed that if the network is 2-edge-connected, then any algorithm for a noiseless setting can be simulated in the fully-defective setting; otherwise, no non-trivial computation is possible in the fully-defective setting. However, their simulation requires a predesignated leader, which they conjectured to be necessary for any non-trivial content-oblivious task.

Recently, Frei, Gelles, Ghazy, and Nolin (DISC 2024) refuted this conjecture for the special case of oriented ring topology. They designed two asynchronous content-oblivious leader election algorithms with message complexity $O(n \cdot \mathsf{ID}_{\max})$, where n is the number of nodes and $\mathsf{ID}_{\max}$ is the maximum ID. The first algorithm stabilizes in unoriented rings without termination detection. The second algorithm quiescently terminates in oriented rings, thus enabling the execution of the simulation algorithm after leader election. In this work, we present two results:

General 2-edge-connected topologies: First, we show an asynchronous content-oblivious leader election algorithm that quiescently terminates in any 2-edge-connected network with message complexity $O(m \cdot N \cdot \mathsf{ID}_{\min})$, where m is the number of edges, N is a known upper bound on the number of nodes, and $\mathsf{ID}_{\min}$ is the smallest ID. Combined with the above simulation, this result shows that whenever a size bound N is known, any noiseless algorithm can be simulated in the fully-defective model without a preselected leader, fully refuting the conjecture.

Unoriented rings: We then show that the knowledge of N can be dropped in unoriented ring topologies by presenting a quiescently terminating election algorithm with message complexity $O(n \cdot \mathsf{ID}_{\max})$ that matches the previous bound. Consequently, this result constitutes a *strict* improvement over the previous state of the art and shows that, on rings, fully-defective and noiseless communication are computationally equivalent, with no additional assumptions.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Asynchronous model, fault tolerance, quiescent termination

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.21

Related Version *Full Version*: https://arxiv.org/abs/2507.08348 [9]

Funding This work has been partially supported by ANR project DUCAT (ANR-20-CE48-0006), by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (24-1323-A0001), and by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through projects SERICS (PE00000014).

1 Introduction

Fault tolerance is a cornerstone of distributed computing, enabling systems to remain operational despite various failures, such as node crashes or channel noise [5, 17, 32, 38]. One important category of faults, known as *alteration errors*, changes the content of messages but does not create new messages or eliminate existing ones. Traditional methods for handling such errors often rely on adding redundancy through coding techniques and assume bounds on the frequency or severity of faults. However, in many practical settings, such assumptions may not hold, limiting the effectiveness of these approaches.

Fully-defective networks. Censor-Hillel, Cohen, Gelles, and Sela [7] introduced *fully-defective* asynchronous networks, in which all links are subject to severe alteration errors: The content of every message can be arbitrarily modified, although the adversary cannot insert new messages or remove existing messages. Since messages can no longer carry meaningful information, the model is equivalent to *content-oblivious* computation, where communication is reduced to sending and receiving *pulses*. Algorithms in this setting operate entirely based on the patterns of pulse arrivals.

An impossibility result. A natural approach to designing a content-oblivious algorithm is to use unary encoding, representing messages as sequences of pulses. However, in the *asynchronous* setting, where no upper bound on message delivery time is known, there is no reliable way to detect the end of a sequence. Censor-Hillel, Cohen, Gelles, and Sela [7] established a strong *impossibility* result: Let $f(x, y)$ be any non-constant function, then any two-party deterministic asynchronous content-oblivious communication protocol, where one party holds x and the other holds y, must fail to compute f correctly.

The impossibility result implies that, in a certain sense, no non-trivial computation is possible in any network that is *not* 2-edge-connected – that is, any network containing an edge e whose removal disconnects the graph into two components. By assigning control of each component to a different party, the problem reduces to the two-party setting to which the impossibility result applies.

Algorithm simulation over 2-edge-connected networks. Interestingly, Censor-Hillel, Cohen, Gelles, and Sela [7] showed that in 2-edge-connected networks, any algorithm designed for the noiseless setting can be simulated in a fully defective network. We briefly sketch the main ideas underlying their approach below.

For the special case of an oriented ring, the challenge of termination detection can be addressed. Suppose a node wishes to broadcast a message. It can send a sequence of pulses that represents the unary encoding of the message in one direction and use the opposite direction to signal termination. To allow all nodes the opportunity to communicate, a token can be circulated in the ring, enabling each node to speak in turn.

To extend this approach to general 2-edge-connected networks, they leverage a result by Robbins [39], which states that any 2-edge-connected undirected graph can be oriented to form a strongly connected directed graph. This guarantees the existence of a *Robbins cycle*, which is an oriented ring that traverses all nodes, possibly revisiting some nodes multiple times. They designed a content-oblivious algorithm that uses a depth-first search (DFS) to construct such a cycle.

Preselected leader assumption. A key limitation of their result is that it inherently relies on a *preselected leader*. For instance, a leader is needed to generate a unique token in a ring or to select the root for the DFS. Censor-Hillel, Cohen, Gelles, and Sela [7] conjectured that the preselected leader assumption is essential for any non-trivial content-oblivious computation, leaving the following question open.

▶ **Question 1.** *In 2-edge-connected networks, is it possible to simulate any algorithm designed for a noiseless setting in the fully defective model without relying on a preselected leader?*

Content-oblivious leader election on rings. To answer the above question affirmatively, it suffices to design a content-oblivious leader election algorithm that possesses certain desirable properties, enabling it to be composed with other algorithms. Recently, Frei, Gelles, Ghazy, and Nolin [24] answered this question for the special case of *oriented rings*. They designed two asynchronous content-oblivious leader election algorithms, both with message complexity $O(n \cdot \mathsf{ID}_{\max})$, where n is the number of nodes and $\mathsf{ID}_{\max}$ is the maximum identifier. The first algorithm *stabilizes* in unoriented rings, without termination detection. The second algorithm *quiescently* terminates in oriented rings, ensuring that no further pulses are sent to a node v after the algorithm on v declares termination.

The quiescent termination property enables the execution of the general simulation algorithm discussed earlier, following their leader election algorithm, as each node can correctly identify which pulses belong to which algorithm. As a result, this finding refutes the conjecture of Censor-Hillel, Cohen, Gelles, and Sela [7] for the special case of oriented ring topologies. Given the work by Frei, Gelles, Ghazy, and Nolin [24], the next question to address is whether content-oblivious leader election is feasible for general 2-edge-connected networks.

▶ **Question 2.** *In 2-edge-connected networks, is it possible to design a quiescently terminating content-oblivious leader election algorithm?*

Existing approaches for content-oblivious leader election heavily rely on the *oriented* ring structure, making use of its two orientations to handle two types of messages. We briefly outline the key ideas behind the algorithms of Frei, Gelles, Ghazy, and Nolin [24] as follows.

A central building block of their algorithms is a stabilizing leader election algorithm on oriented rings, which operates as follows. Each node generates a token, and all tokens travel in the same direction. Each node v maintains a counter to track the number of tokens it has passed. Once the counter reaches $\mathsf{ID}(v)$, node v destroys one token and temporarily elects itself as leader. Any incoming token will cause it to relinquish leadership. All counters eventually reach $\mathsf{ID}_{\max}$ and stabilize. The node r with $\mathsf{ID}(r) = \mathsf{ID}_{\max}$ becomes the unique leader. This algorithm is not terminating, as nodes cannot determine whether additional tokens will arrive.

To achieve a quiescently terminating algorithm on oriented rings, they run the procedure in both directions sequentially, using the moment when the two counter values match as the termination condition. A node v begins the second run when its counter value from the first run reaches $\mathsf{ID}(v)$.

We stress that orientation is a key assumption of the above algorithm. In fact, they conjectured the impossibility of terminating leader election in unoriented rings [24].

To achieve a stabilizing algorithm on unoriented rings, they run the procedure in both directions concurrently. The correctness follows from the observation that the same node is selected as the leader in both executions.

1.1 Contributions

We answer Question 1 affirmatively under the assumption that an upper bound $N \geq n$ on the number of nodes $n = |V|$ is known, by presenting a content-oblivious leader election algorithm that quiescently terminates on any 2-edge-connected network.

▶ **Theorem 1.** *There is a quiescently terminating leader election algorithm with message complexity $O(m \cdot N \cdot \mathsf{ID}_{\min})$ in any 2-edge-connected network $G = (V, E)$, where $m = |E|$ is the number of edges, $N \geq n$ is a known upper bound on the number of nodes $n = |V|$, and $\mathsf{ID}_{\min}$ is the smallest ID. Moreover, the leader is the last node to terminate.*

Combined with the simulation result of Censor-Hillel, Cohen, Gelles, and Sela [7], our finding implies that in any 2-edge-connected network, any algorithm designed for the noiseless setting can be simulated in the fully-defective setting, without assuming a preselected leader. More precisely, as established in the prior work [24], a sufficient condition for achieving this objective consists of (1) quiescent termination and (2) the requirement that the leader is the final node to terminate. This is because the algorithmic simulation in [7] is initiated by the leader. Hence we answer Question 2 affirmatively, again under the assumption that an upper bound $N \geq n$ is known. This refutes the original conjecture posed by [7].

For unoriented ring topologies, the knowledge of N is not required.

▶ **Theorem 2.** *There exists a quiescently terminating leader election algorithm with message complexity $O(n \cdot \mathsf{ID}_{\max})$ in any unoriented ring, where $\mathsf{ID}_{\max}$ is the largest identifier. Moreover, the leader is the last node to terminate.*

This result refutes the conjecture of Frei, Gelles, Ghazy, and Nolin [24], demonstrating that orientation is not necessary for quiescently terminating leader election in rings.

Table 1 New and old results on content-oblivious leader election.

Topology	Guarantee	Messages	Condition	Reference
Unoriented rings	Stabilizing	$O\left(n \cdot \mathsf{ID}_{\max}\right)$	$\times$	[24]
Oriented rings	Quiescently terminating	$O\left(n \cdot \mathsf{ID}_{\max}\right)$	$\times$	[24]
Oriented rings	Stabilizing	$\Omega\left(n \cdot \log \frac{\mathsf{ID}_{\max}}{n}\right)$	$\times$	[24]
2-edge-connected	Quiescently terminating	$O\left(m \cdot N \cdot \mathsf{ID}_{\min}\right)$	Known N	Theorem 1
Unoriented rings	Quiescently terminating	$O\left(n \cdot \mathsf{ID}_{\max}\right)$	$\times$	Theorem 2

See Table 1 for a comparison of our results and the results from prior work [24].

Our algorithm for general 2-edge-connected topologies has a higher message complexity of $O(m \cdot N \cdot \mathsf{ID}_{\min})$ compared to the previous $O(n \cdot \mathsf{ID}_{\max})$ bound in terms of network size, but it reduces the dependency on identifiers from $\mathsf{ID}_{\max}$ to $\mathsf{ID}_{\min}$.

At first glance, the upper bound of $O(m \cdot N \cdot \mathsf{ID}_{\min})$ may seem to contradict the lower bound of $\Omega\left(n \cdot \log \frac{\mathsf{ID}_{\max}}{n}\right)$ established in previous work [24]. However, the two results are consistent, as the lower bound does not make any assumptions about the value of $\mathsf{ID}_{\min}$. Precisely, their bound is of the form $\Omega\left(n \cdot \log \frac{k}{n}\right)$, where k is the number of distinct, *assignable* identifiers in the network. In other words, k is the size of the ID space.

While our algorithm for Theorem 1 solves leader election in any 2-edge-connected topology, it does require a mild assumption: All nodes must *a priori* agree on a known upper bound N on the number of nodes $n = |V|$ in the network, making it *non-uniform*. This assumption is widely adopted in many existing leader election and distributed graph algorithms, as upper bounds on network size are often available or can be estimated. In contrast, the terminating algorithm from prior work [24] is *uniform*, but only works on oriented rings.

Our leader election algorithm on rings requires neither orientation nor knowledge of N, while matching the message complexity of the algorithms of Frei, Gelles, Ghazy, and Nolin [24]. It therefore constitutes a *strict* improvement over the previous state of the art.

2 Preliminaries

We present the model and problem considered in the paper and overview some basic terminology.

2.1 Content-Oblivious Model

The communication network is a graph $G = (V, E)$, where each node $v \in V$ is a computing device and each edge $e \in E$ is a bidirectional communication link. We write $n = |V|$ and $m = |E|$. Each node v has a unique identifier $\mathsf{ID}(v)$, and we define $\mathsf{ID}_{\max} = \max_{v \in V} \mathsf{ID}(v)$ and $\mathsf{ID}_{\min} = \min_{v \in V} \mathsf{ID}(v)$. Throughout the paper, we assume that G is 2-edge-connected: removing one edge does not disconnect the graph.

General 2-edge-connected topologies. For our result on general 2-edge-connected topologies, we assume no prior knowledge of the network topology, except that each node is given an upper bound $N \geq n$ on the total number of nodes. That is, the algorithm is non-uniform.

The degree of a node v, $\deg(v)$, is the number of edges incident to v. A node does not know the identifiers of its neighbors, but can distinguish its incident edges using *port numbering*, a bijection between the set of incident edges and the set $\{1, 2, \ldots, \deg(v)\}$. Therefore, we use the term port to refer to the local endpoint of an edge, uniquely identified by its port number, which is used to send and receive messages to and from a specific neighbour.

Unoriented rings. For our result on rings, we assume that the topology is an unoriented ring. In this case, each node v_i is connected to nodes v_{i-1} and v_{i+1} (indices taken modulo n) by port 0 and port 1. These local labels do not induce a consistent orientation of the ring; that is, the global assignment of labels 0 and 1 to ports is arbitrarily set by an adversary. A correct algorithm must work for any possible assignment of port labels.

Content-Oblivious communication. In the content-oblivious setting, nodes communicate by sending *content-less pulses*. We consider the *asynchronous* model, where node behavior is *event-driven*: A node can act only upon initialization or upon receiving a pulse. Based on the pattern of previously received pulses, each node decides its actions, which may include sending any number of pulses through any subset of its ports. We emphasize that when a node receives a pulse, it knows the port through which the pulse arrived. We restrict our attention to the *deterministic* setting, in which nodes do not use randomness in their decision-making.

Time. There is no upper bound on how long a pulse may take to traverse an edge, but every pulse is guaranteed to be delivered after a finite delay. An equivalent way to model this behavior is by assuming the presence of a scheduler. At each time step, as long as there are pulses in transit, the scheduler arbitrarily selects one and delivers it.

We assume that upon receiving a pulse, all actions triggered by that pulse are performed *instantaneously*, so that the system can be analyzed in *discrete* time steps. We focus only on specific, well-defined moments in the execution of the algorithm – namely, the moments just

before the scheduler selects a pulse to deliver. At these times, all the actions resulting from earlier pulse deliveries have been completed, and the system is in a stable state, awaiting the next step. We do not analyze the system during any intermediate period in which a node is still performing the actions triggered by a pulse arrival.

Counting the number of pulses. We write $[s] = \{1, 2, \ldots, s\}$. In the algorithm description and analysis for Theorem 1, for each node $v \in V$ and each port number $i \in [\deg(v)]$, we use the following notation to track the number of pulses sent and received so far:

- $\sigma_i(v)$ denotes the number of pulses sent on port i of node v.
- $\rho_i(v)$ denotes the number of pulses received on port i of node v.

2.2 Leader Election

We now define the *leader election* problem.

▶ **Definition 3** (Leader election). *The task of leader election requires that each node outputs exactly one of* Leader *and* Non-Leader*, with the additional requirement that exactly one node outputs* Leader*.*

More precisely, each node maintains a binary internal variable isLeader to represent its current output, taking the value Leader or Non-Leader. The value of isLeader may change arbitrarily many times during the execution of the algorithm, but it must eventually stabilize to a final value that remains unchanged thereafter. The final outputs of all nodes must collectively satisfy the conditions of Definition 3.

Due to the asynchronous nature of our model, an algorithm can *finish* in various ways. We hence classify algorithms into three types by the increasing strength of termination guarantees.

▶ **Definition 4** (Stabilization). *A node stabilizes at time t if its output does not change after time t. An algorithm is stabilizing if it guarantees that all nodes eventually stabilize.*

▶ **Definition 5** (Termination). *A node terminates at time t if it explicitly declares termination at time t. After declaring termination, the node ignores all incoming pulses and can no longer change its output. An algorithm is terminating if it guarantees that all nodes eventually terminate.*

▶ **Definition 6** (Quiescent termination). *A node quiescently terminates at time t if it terminates at time t and no further pulses arrive at the node thereafter. An algorithm is quiescently terminating if it guarantees that all nodes eventually quiescently terminate.*

Ideally, we aim to design leader election algorithms that are quiescently terminating, as this property allows a second algorithm to be executed safely after leader election. In contrast, if the algorithm guarantees only stabilization or termination, nodes may be unable to distinguish whether an incoming pulse belongs to the leader election process or the subsequent algorithm. This ambiguity can affect the correctness of both algorithms. For this reason, prior work [24] establishes a general algorithm simulation result only for oriented rings, but not for unoriented rings.

2.3 DFS and Strongly Connected Orientation

We describe a DFS tree T_G of the graph $G = (V, E)$ that is *uniquely determined* by the port numbering and node identifiers. Based on this DFS tree, we describe an edge orientation of G that transforms it into a strongly connected directed graph $\vec{G}$. Both the DFS tree and the orientation play a critical role in the analysis of our algorithm for Theorem 1.

DFS. Throughout this paper, we write r to denote the node with the smallest identifier. Consider a DFS traversal rooted at r such that when a node selects the next edge to explore among its unexplored incident edges, it chooses the one with the *smallest* port number. Let $T_G = (V, E_T)$ be the resulting DFS tree. The edges in E_T are called *tree edges*, and the remaining edges $E_B = E \setminus E_T$ are called *back edges*. The following observation is folklore.

▶ **Observation 7.** *For any edge $\{u, v\} \in E_B$, either u is an ancestor of v, or v is an ancestor of u.*

Proof. When a node u explores an incident edge $e = \{u, v\}$ during the DFS, there are two possibilities: Either v is an ancestor of u, or v has not been visited yet. In the first case, $\{u, v\} \in E_B$. In the second case, v is discovered for the first time, so $\{u, v\} \in E_T$ is added as a tree edge in E_T. ◀

▶ **Definition 8** (Strongly connected orientation)**.** *The directed graph $\vec{G} = (V, \vec{E}_T \cup \vec{E}_B)$ is defined by*

$$\vec{E}_T = \{(u, v) \mid \{u, v\} \in E_T \text{ and } u \text{ is the parent of } v \text{ in } T_G\},$$
$$\vec{E}_B = \{(u, v) \mid \{u, v\} \in E_B \text{ and } v \text{ is an ancestor of } u \text{ in } T_G\}.$$

A directed graph is *strongly connected* if for any two nodes u and v in the graph, there is a directed path from u to v. We have the following observation. Due to page limit, the proof of Observation 9 is deferred to the full version [9] of the paper.

▶ **Observation 9.** *If G is 2-edge-connected, then $\vec{G}$ is strongly connected.*

Shortest paths. Throughout the paper, for any two nodes $u \in V$ and $v \in V$, we write $P_{u,v}$ to denote any shortest path from u to v in $\vec{G}$, whose existence is guaranteed by Observation 9. We allow $u = v$, in which case $P_{u,v}$ is a path of zero length consisting of a single node $u = v$.

3 Non-Uniform Leader Election for 2-Edge-Connected Topologies

In this section, we provide a high-level proof sketch for Theorem 1; the pseudocode is deferred to Section B, and the complete proof is deferred to the full version [9] of the paper.

▶ **Theorem 1.** *There is a quiescently terminating leader election algorithm with message complexity $O(m \cdot N \cdot \mathsf{ID}_{\min})$ in any 2-edge-connected network $G = (V, E)$, where $m = |E|$ is the number of edges, $N \geq n$ is a known upper bound on the number of nodes $n = |V|$, and $\mathsf{ID}_{\min}$ is the smallest ID. Moreover, the leader is the last node to terminate.*

Our algorithm aims to elect the node r with the smallest identifier as the leader. It operates in two phases. In the first phase, each node maintains a counter that is incremented in an almost synchronized manner using the *α-synchronizer* of Awerbuch [4]. A node v declares itself the leader once its counter reaches $\mathsf{ID}(v)$. This first phase alone yields a simple *stabilizing* leader election. To ensure *quiescent termination*, the second phase is executed, in which the elected leader announces its leadership through a DFS traversal.

3.1 Synchronized Counting

Counting. To achieve an almost synchronized counting, we employ the α-synchronizer [4], as follows. Each node maintains a counter variable $\mathsf{Count}(v)$, which is initialized to 0 at the start of the algorithm. A node increments its counter by one after receiving a new pulse from each of its neighbors. Formally, the counter value of a node v is given by

$$\mathsf{Count}(v) = \min_{j \in [\deg(v)]} \rho_j(v).$$

Each node broadcasts a pulse to all its neighbors upon initialization and immediately after each counter increment. This process is *non-blocking*: In the absence of an exit condition, nodes will continue counting together indefinitely. Furthermore, no node can advance significantly faster than its neighbors: For any pair of adjacent nodes, their counter values differ by at most one. A faster-counting node must wait for a pulse from a slower-counting neighbor before it can increment its counter again.

Stabilizing leader election. A node v elects itself as the leader once $\mathsf{Count}(v) = \mathsf{ID}(v)$, after which node v *freezes* its counter and does not increment it further. This action caps the counter value of any neighbor of v at most $\mathsf{ID}(v) + 1$. More generally, in a graph of $n \leq N$ nodes, the counter value of any node is upper bounded by $\mathsf{ID}(v) + N - 1$, due to the diameter upper bounded by $N - 1$.

To ensure that the algorithm elects the node r with the smallest identifier as the *unique* leader, it suffices for any two identifiers to differ by at least N. This can be achieved by first multiplying each identifier by N. This is where the *a priori* knowledge of N comes into play.

At this stage, we have a simple stabilizing leader election algorithm. However, it remains insufficient for our purposes: We also want all non-leader nodes to recognize their non-leader status and terminate in a quiescent manner.

3.2 DFS Notification

We proceed to describe the second phase of the algorithm. The leader must find a way to notify all other nodes to settle as non-leaders. This seems challenging because the leader can only send additional pulses, which appear indistinguishable from those in the first phase. As we will see, assuming the network is 2-edge-connected, there is a simple and effective approach: Just send many pulses to overwhelm the listener.

Notifying a neighbor. Let u be a neighbor of the leader r, and let the edge $e = \{u, r\}$ correspond to port i at r and port j at u. The leader r notifies u of its leadership as follows:

1. Wait until $\rho_i(r) \geq \mathsf{ID}(r) + 1$.
2. Send pulses to u until $\sigma_i(r) = \mathsf{ID}(r) + N + 2$.

Since the graph G is 2-edge-connected, there exists a path P from u to r that avoids e, implying

$$\mathsf{Count}(u) \leq \mathsf{Count}(r) + \mathsf{length}(P) \leq \mathsf{ID}(r) + N - 1.$$

Eventually, u receives $\mathsf{ID}(r) + N + 2$ pulses from r, so there must be a moment where

$$\rho_j(u) \geq \mathsf{Count}(u) + 3,$$

triggering anomaly detection. Under normal conditions in the first phase, u would expect

$$\rho_j(u) \leq \sigma_i(r) = \mathsf{Count}(r) + 1 \leq \mathsf{Count}(u) + 2,$$

so this deviation signals a violation.

Additionally, the waiting condition $\rho_i(r) \geq \mathsf{ID}(r) + 1$ ensures that $\mathsf{Count}(u) \geq \mathsf{ID}(r)$. Thus, u not only recognizes that it is not the leader but can also deduce the exact value of $\mathsf{ID}(r)$ using

$$\mathsf{ID}(r) = N \cdot \left\lfloor \frac{\mathsf{Count}(u)}{N} \right\rfloor,$$

since all node identifiers are integer multiples of N.

DFS. Intuitively, the leader r is able to notify a neighbor u because the counter value of node u is constrained by a *chain* P, whose other end is *anchored* at the leader r, whose counter has already been frozen. To extend this notification process to the entire network, we perform a DFS, using the same notification procedure to explore each new edge. When a node finishes exploring all its remaining incident edges, it can notify its parent by sending pulses until a total of $\mathsf{ID}(r) + N + 2$ pulses have been sent through this port since the start of the algorithm. When a node u explores a back edge $\{u, v\}$, the node v replies using the same method. When a node decides which new incident edge to explore, it prioritizes the edge with the smallest port number, so the resulting DFS tree is T_G, as defined in Section 2.3.

Quiescent termination. To see that the algorithm achieves quiescent termination, observe that by the end of the execution, for each edge e, the number of pulses sent in both directions is exactly $\mathsf{ID}(r) + N + 2$. Therefore, once a node observes that it has both sent and received exactly $\mathsf{ID}(r) + N + 2$ pulses along each of its ports and has completed all local computations, it can safely terminate with a quiescent guarantee.

Correctness. A similar chain-and-anchor argument can be applied to analyze the correctness of the DFS-based notification algorithm. We prove by induction that for each node u that has been visited by DFS, its frozen counter value satisfies

$$\mathsf{Count}(u) \leq \mathsf{ID}(r) + \mathsf{length}(P_{u,r}).$$

Recall from Section 2.3 that $P_{a,b}$ is a shortest path from a to b in the directed version $\vec{G}$ of G defined in Definition 8.

Suppose during the DFS, a node x is about to explore an incident edge $e = \{x, y\}$ using the notification procedure, and that y has not yet been visited. Let w be the first node on the path $P_{y,r}$ that is either x or an ancestor of x in the DFS tree T_G, then we have

$$\mathsf{length}(P_{y,r}) = \mathsf{length}(P_{y,w}) + \mathsf{length}(P_{w,r}).$$

We now apply the chain-and-anchor argument using the chain $P_{y,w}$ and anchor w. This choice of chain is valid because the path $P_{y,w}$ cannot include the edge e, as e is oriented from x to y in $\vec{G}$. This is precisely why the analysis is carried out in $\vec{G}$ rather than in G.

By the induction hypothesis, the frozen counter value at the anchor satisfies $\mathsf{Count}(w) \leq \mathsf{ID}(r) + \mathsf{length}(P_{w,r})$. It follows that

$$\begin{aligned}\mathsf{Count}(y) &\leq \mathsf{Count}(w) + \mathsf{length}(P_{y,w}) \\ &\leq \mathsf{ID}(r) + \mathsf{length}(P_{w,r}) + \mathsf{length}(P_{y,w}) \\ &= \mathsf{ID}(r) + \mathsf{length}(P_{y,r}) \\ &\leq \mathsf{ID}(r) + N - 1.\end{aligned}$$

Therefore, the chain-and-anchor argument works, allowing y to be successfully notified. Moreover, the frozen counter value of y also satisfies the induction hypothesis.

In addition, we must ensure that the DFS procedure proceeds correctly and no event is triggered unexpectedly. Once again, the chain-and-anchor argument implies that the condition

$$\mathsf{Count}(u) \leq \mathsf{ID}(r) + \mathsf{length}(P_{u,r}) \leq \mathsf{ID}(r) + N - 1 < \mathsf{ID}(u)$$

holds for all nodes $u \in V \setminus \{r\}$ at all times. This ensures that no node other than r can be mistakenly elected as a leader. It also guarantees that pulses sent by nodes still in the first

phase cannot accidentally trigger a notification in the second phase. Specifically, if u is still in the first phase, then for every port $j \in [\deg(u)]$, we have

$$\sigma_j(u) = \mathsf{Count}(u) + 1 \leq \mathsf{ID}(r) + N,$$

which remains below the threshold of $\mathsf{ID}(r) + N + 2$ pulses required to confirm receipt of a notification.

Remark. One might wonder why the notification must follow a DFS instead of being performed in parallel. A key reason is that doing so can break the chain-and-anchor argument by eliminating anchors. For instance, in a ring topology, if the node r with the smallest identifier simultaneously sends multiple pulses to *both* neighbors, all of the pulses might be interpreted as part of the first phase, with no detectable anomaly, regardless of how many pulses are sent.

Both our leader election algorithm and the simulation algorithm of [7] utilize DFS, but for different purposes. They perform a DFS from a pre-selected leader to find a cycle, while we use DFS to disseminate the identity of the elected leader to all nodes and to achieve quiescent termination. Coincidentally, both approaches leverage the strongly connected orientation of 2-edge-connected graphs. In our case, the orientation $\vec{G}$ is used solely in the analysis to facilitate the chain-and-anchor argument. In their case, it guarantees the existence of a Robbins cycle, which is critical to their algorithm.

4 Uniform Leader Election for Unoriented Rings

In this section, we prove our main theorem on rings.

▶ **Theorem 2.** *There exists a quiescently terminating leader election algorithm with message complexity $O(n \cdot \mathsf{ID}_{\max})$ in any unoriented ring, where $\mathsf{ID}_{\max}$ is the largest identifier. Moreover, the leader is the last node to terminate.*

The proof is given by our uniform algorithm, whose pseudocode is given in Algorithm 1.

We introduce a new action, **RcvPulse$_i$**: Wait until a pulse is received from port i, where $i \in \{0, 1, q\}$. When $i = q$, the procedure waits for the first pulse on any port and stores the corresponding port number in the variable q. We also use **SendPulses$_i(k)$**: send k pulses through port i, where $i \in \{0, 1\}$.

We define the local variable **Diff(v)**, which stores the difference between the number of pulses node v received on port 1 and the number of pulses received on port 0.

Algorithm description. Our algorithm is decomposed into different phases that we describe below.

- The *initialization*. First, each node doubles its identifier ID (Line 1). The purpose of this action is to ensure that no two nodes have consecutive identifiers. This allows us to distinguish the different phases of the algorithm during the execution.
- The *competing* phase (Lines 2 to 5 in blue). Every node starts by exchanging pulses with both its neighbors in a pseudo-synchronous way (sending them pulses and waiting for their responses). The number of times a node repeats this step is equal to the value of its identifier ID.

- The *solitude-checking* phase (Lines 7 to 9 in orange). Once a node has finished the competing phase, it wants to know if it was the last one competing. To do so, it sends two pulses on port 0, receives one on port 1 and waits for another pulse. If this last pulse arrives on port 1, it means that its own pulse has made a complete traversal of the cycle and that all other nodes stopped competing. Otherwise (i.e., if it gets a pulse on port 0 before getting two pulses on port 1), it means that there is still another node executing the competing phase.
- The *global termination* phase (Lines 11 to 12 in green). If a node detects that all other nodes stopped competing (i.e., if $q = 1$ on Line 10), then it knows it will be the leader. Before terminating, it sends a pulse on port 0 to inform the other nodes that the execution is over. This termination pulse will traverse the entire ring and come back through port 1. At this point, every other node will eventually be in state Non-Leader and there will be no pulses in transit. Then the node can enter the state Leader and terminate.

Algorithm 1 Uniform leader election on unoriented rings for node v.

```
1  ID(v) ← 2 * ID(v)
2  for i ← 1 to ID(v) do
3  |  SendAll(1)                                          // send 1 pulse on both ports
4  |  RcvPulse_0                                         // wait for a pulse from port 0
5  |  RcvPulse_1                                         // wait for a pulse from port 1
6  end
   /* Check if I am alone, i.e., if I have the largest identifier                    */
7  SendPulses_0(2)                                       // send 2 pulses on port 0
8  RcvPulse_1                                            // wait for a pulse from port 1
9  RcvPulse_q                  // wait for first pulse from any port; store port in q
10 if q = 1 then
   |  /* I am the last one competing                                                 */
11 |  SendPulses_0(1)                  // send termination pulse on port 0 - I'm leader
12 |  RcvPulse_1                      // wait for termination pulse to return from port 1
13 |  return Leader
14 else
   |  /* There is someone with a larger identifier                                   */
15 |  SendPulses_1(2)                          // send 2 pulses on port 1 - balance line 7
16 |  RcvPulse_0                                          // wait for a pulse from port 0
17 |  RcvPulse_1                                          // wait for a pulse from port 1
18 |  Diff ← 0
   |  /* Diff = pulses from port 1 minus pulses from port 0                          */
19 |  repeat
20 |  |  RcvPulse_q                    // wait for pulse from any port; store port in q
21 |  |  Diff ← Diff + 2 * q − 1                          // update difference counter
22 |  |  SendPulses_{1−q}(1)                          // forward pulse to opposite port
23 |  until |Diff| ≥ 3
24 |  return Non-Leader
25 end
```

- The *rebalancing* phase (Lines 15 to 17 in violet). If the node is not alone (i.e., if $q = 0$ on Line 10), we want the node to relay pulses between its competing neighbors so that these neighbors act as if there was no node between them. To do so, we have to ensure that it sent the same number of pulses in both directions. So the node sends two pulses on port 1 (to balance the pulses sent on port 0 on Line 7). At this point, on each port, the node has sent one more pulse than it has received. So it waits for one pulse on each port before entering the relaying phase. For its neighbors, the node behaves as if it has executed two more iterations of the competing phase: it has sent two pulses on each port and has received two pulses on each port. Since no two nodes have consecutive identifiers (recall that each node doubled its ID at Line 1), no other node can terminate its competition phase before receiving both these pulses.

- The *relaying* phase (Lines 18 to 23 in red). In this phase, the node just relays the pulses it gets on port 0 to port 1 and vice-versa. It also keeps a counter (Diff) to store the difference between the numbers of pulses received on port 0 and on port 1. When this counter reaches 3 or -3, then it means that one node has successfully passed the solitude test and sent a termination pulse. The node can then enter the state Non-Leader and terminate.

4.1 Correctness

We say that a node is *active* if it is executing an instruction between Lines 2 and 17. A node is a *relaying node* if is executing an instruction between Lines 18 and 23.

Consider two rings: R of size n and R' of size $n-1$, where R' is obtained by removing a node v from R and connecting its neighbours w and u. We say that an execution on R is indistinguishable from an execution on R' if there exists an execution on R' in which w and u return from waiting on the RcvPulse actions at exactly the same times as they do in R.

In such executions, w and u do not perceive any difference between the execution of the algorithm on R and on R'. This, in turn, implies that all other nodes also observe an execution, when viewed in terms of RcvPulse events, that could occur in either R or R'.

▶ **Proposition 10.** *Consider a ring of size $n \geq 2$ with consecutive nodes v_1, v_2 and v_3 (with $v_1 = v_3$ if $n = 2$) such that v_2 is the node with the smallest identifier on the entire ring. As long as v_2 does not stop, the execution of the algorithm on this ring is indistinguishable for any node on the ring from a reduced ring of size $n-1$, where v_2 has been removed and v_1 is directly connected with v_3.*

Without loss of generality, let us assume that at v_2, the port number to v_1 is 0 and the port number to v_3 is 1. Let ID_1, ID_2, and ID_3 be the respective identifiers of nodes v_1, v_2, v_3. To prove Proposition 10, we first establish several observations and lemmas.

▶ **Observation 11.** *When a node ends the ith iteration of the loop of the competing phase, it has sent i pulses to each of its neighbors and has received i pulses from each of them.*

▶ **Lemma 12.** *Consider two neighboring nodes, v_a and v_b, executing the competing phase. Given their respective indices, i_a and i_b, in the loop at Lines 2–5, we have $|i_a - i_b| \leq 1$. Moreover, if $i_b \leq i_a$ and v_b has not executed the sends of its iteration i_b then $i_b = i_a$.*

Proof. Suppose w.l.o.g. that node v_a exchanges pulses with v_b on port 1, while node v_b exchanges pulses with v_a on port 0.

If node v_a executed x times Line 5 (i.e., $x + 1 \geq i_a \geq x$), then node v_b executed $y \geq x$ times Line 3 (i.e., $i_b \geq y$). From this $i_b \geq i_a - 1$. By a symmetric argument we have $i_a \geq i_b - 1$ and this shows the first part of the lemma. Assume now that v_b has not executed its send, this means that $y = (i_b - 1)$ and thus $x \leq y \leq i_b - 1$ therefore since $i_a \leq x + 1 \leq i_b$. For the hypothesis we have $i_a = i_b$. ◀

▶ **Lemma 13.** *Among nodes v_1, v_2 and v_3, node v_2 is the first to exit the competing phase. Moreover, v_1 (resp. v_3) cannot exit its competing phase before it has received two pulses sent by v_2 after v_2 has exited the competing phase (sent at Line 7 for v_1 and at 15 for v_3).*

Proof. By Lemma 12 and Observation 11, we have that when v_2 exits the loop, it has sent ID_2 pulses to both its neighbors and thus v_1 and v_3 are in iteration ID_2 or $\mathsf{ID}_2 + 1$ of the loop. Since their identifiers are at least $\mathsf{ID}_2 + 2$, they are still in their competing phase. Note that, by Lemma 12 and Observation 11, they cannot complete iteration $\mathsf{ID}_2 + 2$ before they have received at least $\mathsf{ID}_2 + 2$ pulses from v_2. The last two of these pulses have to be sent by v_2 either at Line 7 or at Line 15. ◀

▶ **Lemma 14.** *Node v_2 eventually executes Line 10, i.e., it receives a pulse on port 1 and then another pulse on some port q. Moreover, the second pulse delivered arrives on port $q = 0$ and v_2 enters the rebalancing phase.*

Proof. Any node $v \neq v_2$ in the ring has an ID that is at least $\mathsf{ID}_2 + 2$. Consequently, by Lemma 13 and Lemma 12, every node eventually completes ID_2 iterations of the competing phase and every node $v \neq v_2$ enters iteration $\mathsf{ID}_2 + 1$. Thus, in particular, v_1 and v_3 send pulses to v_2 in iteration $\mathsf{ID}_2 + 1$. Consequently, v_2 will eventually receive a pulse from v_3 on port 1 (i.e., it will not be blocked at Line 8). Moreover, since v_1 has sent a pulse `p` to v_2 at iteration $\mathsf{ID}_2 + 1$, v_2 will eventually receive a pulse on some port q (i.e., it will execute Line 9). As long as v_2 does not send a pulse to v_3, v_3 will not complete iteration $\mathsf{ID}_2 + 1$ of the competing phase, and it will thus not send another pulse to v_2 before `p` is delivered. Consequently, $q = 0$ and v_2 enters the rebalancing phase. ◀

Notice that if v_2 is only a local minimum among v_1, v_2, and v_3, rather than a global one, it is still possible to adapt the proof to obtain a weaker variant of the lemma. In this variant, if v_2 executes Line 10, it must do so with $q = 0$.

From Lemma 14 we have:

▶ **Corollary 15.** *Node v_2 does not enter the global termination phase and thus it cannot terminate the algorithm in the* Leader *state.*

The following observation follows from the fact that the second minimum identifier in the ring is at least $\mathsf{ID}_2 + 2$. Recall that the reduced ring is obtained by removing node v_2 and directly connecting v_1 and v_3 while respecting the original port labeling.

▶ **Observation 16.** *When the algorithm is executed on the reduced ring, v_1 and v_3 send at least $\mathsf{ID}_2 + 2$ pulses to each other (and they receive these pulses) during their competing phase.*

▶ **Lemma 17.** *When node v_2 completes the rebalancing phase, v_2 has sent exactly $\mathsf{ID}_2 + 2$ pulses to node v_1 and $\mathsf{ID}_2 + 2$ pulses to node v_3. Moreover, at the end of the rebalancing phase, v_2 has received exactly $\mathsf{ID}_2 + 2$ pulses from v_1 and $\mathsf{ID}_2 + 2$ pulses from v_3.*

Proof. Once v_2 terminates its competing phase it has sent/received ID_2 pulses to/from each of its neighbors. At Line 7, it sends 2 pulses to v_1 and at Line 15 it sends 2 pulses to v_3. Moreover, at Lines 8 and 17, it has received two pulses from v_3 and at Lines 9 and 16 it has received two pulses from v_1 (recall, Lemma 14, that if Line 16 is executed then $q = 0$ at Line 9). ◀

▶ **Observation 18.** *From Observation 16 and Lemma 17, v_1 and v_3 cannot distinguish during the first $\mathsf{ID}_2 + 2$ iterations of their competing phase if they are in the original ring or in the reduced ring.*

We are now ready to prove Proposition 10.

Proof of Proposition 10. By Observation 18, before node v_2 completes its rebalancing phase, nodes v_1 and v_3 are not able to distinguish this execution from the one on the reduced ring. By Lemma 14, node v_2 completes its rebalancing phase and it eventually enters the relaying phase. It is easy to see that, from this point onward, v_2 behaves as an asynchronous link between v_1 and v_3 as long as v_2 does not exit the loop at Lines 19–23. ◀

From Proposition 10, it follows that as long as no node terminates and there are at least two active nodes, the one with the minimum identifier will enter the relaying phase, and the other active nodes will behave as if this node were not present in the ring.

Observe that the global minimality of ID_2 is used only in the proof of Lemma 14, to establish that v_2 must enter the relaying phase, all the other lemmas, when needed, they only require the identifier of v_2 to be a local minimum among v_1, v_2, v_3. As a matter of fact, a node v' may have an identifier that is a local minimum over a sufficiently long sequence of nodes, allowing it, in some executions, to enter the relaying phase before the global minimum. This does not affect the indistinguishability claimed in Proposition 10 that remains valid for v' and its two neighbours.

We now show that no node terminates before the one with the maximum identifier exits its competing phase. Moreover, we show that this node eventually enters the global termination phase, triggering the algorithm's termination with the election of the node with the maximum identifier. We will show that each node terminates when it has received three more pulses on one port than the other. In the next lemma, we characterize when such an event can happen.

▶ **Lemma 19.** *During the execution, if at some point, some node v has sent three more pulses on port q than it has sent on port $1-q$, or if v has sent three more pulses on port q than it has received on port q, then either v has executed Line 11, or v is a relaying node and v has previously received three more pulses on port $1-q$ than on port q.*

In any case, when this happens, the node v does not send any pulses afterwards.

Proof. During the competing phase, on each port, each node alternately sends a pulse on each port and receives a pulse on each port. At the end of the competing phase, node with identifier ID has sent and received ID pulses on each port. At Line 7, it sends two pulses on port 0. Then, if the condition at Line 10 is true, it sends a third pulse on port 0 (i.e., it executes Line 11) and stops sending pulses afterwards.

Suppose now that the condition at Line 10 is not satisfied. Then it means that at Line 9, $q = 0$ and at this point, v has sent $\mathsf{ID}+2$ pulses on port 0, has sent ID pulses on port 1, has received $\mathsf{ID}+1$ pulses on port 0, and has received $\mathsf{ID}+1$ pulses on port 1. After the rebalancing phase (Lines 15 to 17), v has sent and received $\mathsf{ID}+2$ pulses on each port.

After that, it just relays the pulses in the order they arrive and stores the difference between the number of pulses sent on each port in the variable Diff. Thus, after each iteration of the loop of the relaying phase, the number of pulses it has sent on a port q is precisely the number of pulses it has received on port $1-q$. Thus, it will send three more pulses on port q than on port $1-q$ only if it has received three more pulses on port $1-q$ than on port q. When this happens, $|\mathsf{Diff}| = 3$ and node v exits the loop and terminates. ◀

▶ **Corollary 20.** *If a node v terminates the algorithm (i.e., executes Line 13 or 24), then there is a node v' that has executed Line 11, and v' will eventually terminate in the* Leader *state.*

Proof. By contradiction, suppose that some node v terminates and that no node has executed Line 11 before. Then v necessarily terminates the algorithm at Line 24, and thus at some time t, v has received three more pulses on some port q than on port $1-q$. Among all such nodes, consider the node v such that this time t is minimal. Let v' be the neighbor of v behind port q. By Lemma 19, v is a relaying node, and v has received three more pulses from v' than it has sent to v'. Consequently, there exists a time $t' < t$ where v' has sent at least three more pulses to v than it has received from v. By Lemma 19 and since we assumed that v' has not executed Line 11, it implies that at time t', v' has received three more pulses on one port than on the other. But this contradicts the definition of v and t. ◀

By iteratively applying Corollary 15 and Proposition 10, and by observing that, by Corollary 20, no node terminates early, we obtain the following corollary:

▶ **Corollary 21.** *The node v with the maximum identifier eventually becomes the only competing node. Moreover, any other node eventually becomes a relaying node and cannot terminate the algorithm in the* Leader *state.*

We now show that the node with the maximum identifier eventually enters the Leader state and that, after this point, no more pulses are sent in the network, and every other node eventually enters the Non-Leader state.

▶ **Lemma 22.** *In a ring containing only one competing node v, the condition at Line 10 is true for v. Moreover, v will eventually execute Lines 11, 12, and 13, terminating the algorithm in the* Leader *state. Furthermore, when v executes Line 13, there are no pulses in transit and for every other node v', we have $|\mathsf{Diff}(v')| = 3$ and thus v' eventually terminates in the* Non-Leader *state.*

Proof. Suppose that there is a moment t where the ring contains only one competing node v. Then, by Proposition 10, Corollary 15, and Corollary 20 applied iteratively to all nodes except v, node v cannot distinguish the original ring from a ring containing only v.

Thus during the competing phase, v sends and receives $\mathsf{ID}(v)$ pulses to itself in each direction. In the solitude-checking phase, v sends two pulses on port 0 that are eventually delivered to itself on port 1. Once v has received both pulses, the condition at Line 10 is true and v enters the global termination phase. It sends another pulse to itself on port 0 (Line 11) that is eventually delivered on port 1 and v can then execute Lines 12 and 13 terminating in the Leader state.

Note that by Corollary 15, when v exits its competing phase, every other node v' is a relaying node. Consequently, at this point, it has sent and received the same number of pulses on each port by Observation 11. Therefore, at this point, there are no pulses in transit in the network and, by Lemma 17, for each node $v' \neq v$, we have $\mathsf{Diff}(v') = 0$.

Once v has executed Line 7, there are two pulses in transit in the network that are moving in the same direction and each node $v' \neq v$ relays them (through the same port) before they reach v on port 1. So when v executes Line 9, for each node $v' \neq v$, we have $|\mathsf{Diff}(v')| = 2$. Then, when v executes Line 11, it sends a third pulse in the same direction that will eventually reach v on port 1, at which point there are no pulses in transit. In the meantime, every node $v' \neq v$ will have relayed it and its variable $\mathsf{Diff}(v')$ will satisfy $|\mathsf{Diff}(v')| = 3$. Thus, after v' has relayed this third pulse, v' will exit the loop in the relaying phase and it will terminate in the Non-Leader state. ◀

Proof of Theorem 2. It suffices to show that Algorithm 1 is a correct leader election algorithm for all unoriented rings and it has a message complexity of $O(n\mathsf{ID}_{\max})$.

Correctness follows directly from Corollary 21 and Lemma 22. Regarding the message complexity, take the node $v_{\max}$ with maximum identifier $\mathsf{ID}_{\max}$, by Corollary 21 and Lemma 22, this node will execute Lines 1–13, sending $4\mathsf{ID}_{\max} + 3$ pulses (the factor 4 comes from the fact that we initially double each identifier and send pulses in both directions) in the process. Consider now any other node v_j with identifier $\mathsf{ID}_j \neq \mathsf{ID}_{\max}$, by Corollary 15 and Lemma 17, v_j first executes Lines 1–9 and Lines 15–17, the number of pulses sent at these lines is $4\mathsf{ID}_j + 4$, and then it executes Lines 19–23, but here it will only send a pulse when it receives one. Therefore, Lemma 22 and the fact that $v_{\max}$ sends $4\mathsf{ID}_{\max} + 3$ pulses show that v_j cannot send more than $\delta_j = (4\mathsf{ID}_{\max} + 3) - (4\mathsf{ID}_j + 4)$ pulses while relaying. The total number of pulses sent by Algorithm 1 is therefore at most $n(4\mathsf{ID}_{\max} + 3)$. ◀

5 Conclusions and Open Questions

In this paper, we demonstrate that non-uniform quiescently terminating content-oblivious leader election is achievable in general 2-edge-connected networks, and that uniform quiescently terminating content-oblivious leader election is achievable in unoriented rings.

Previously, such leader election was only known to be possible in ring topologies – specifically, a quiescently terminating algorithm in oriented rings and a stabilizing algorithm in unoriented rings [24]. Consequently, we remove the need for a preselected leader in the general algorithm simulation result by Censor-Hillel, Cohen, Gelles, and Sela [7], trading this requirement for the much weaker assumption of non-uniformity. For the special case of ring topologies, we provide a definitive answer on the computational equivalence of fully-defective communication and noiseless communication.

Several intriguing open questions remain. The foremost is whether *uniform* content-oblivious leader election algorithm exists for general 2-edge-connected topologies. So far, this has only been established for oriented rings [24] and later for unoriented rings (Theorem 2). Our current algorithm requires an upper bound N on the number of nodes n, as node identifiers must be spaced N apart to ensure that no node other than the leader r satisfies the Leader exit condition $\mathsf{Count}(v) = \mathsf{ID}(v)$. This reliance on N seems inherent to our approach, where counters of adjacent nodes may differ by one.

Another important open question is whether the efficiency of our algorithms can be improved. Our algorithm for general 2-edge-connected topologies is highly sequential due to its use of DFS. A common way to measure *time complexity* of an asynchronous algorithm is to assume that each message takes one unit of time to be delivered [3]. Under this model, the time complexity of our algorithm is $O(m \cdot N \cdot \mathsf{ID}_{\min})$, matching its message complexity, as the DFS traverses the m edges sequentially, and the traversal of each edge requires sending $O(N \cdot \mathsf{ID}_{\min})$ pulses. It remains unknown whether we can make the algorithm less sequential.

Much less is known about message complexity lower bounds for content-oblivious leader election. The only known result is a lower bound of $\Omega\left(n \cdot \log \frac{k}{n}\right)$, where k is the number of distinct, assignable identifiers in the network, due to Frei, Gelles, Ghazy, and Nolin [24]. Currently, there remains a significant gap between this lower bound and the upper bounds, both in rings and in general 2-edge-connected networks. A key open question is whether the message complexity for general 2-edge-connected networks is inherently higher than that of ring topologies. If so, how can we leverage the structural properties of the hard instances to establish such a lower bound?

References

1 Dan Alistarh, Joel Rybicki, and Sasha Voitovych. Near-optimal leader election in population protocols on graphs. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 246–256, 2022. doi:10.1145/3519270.3538435.

2 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/S00446-005-0138-3.

3 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004.

4 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

5 Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys (CSur)*, 25(2):171–220, 1993. doi:10.1145/152610.152612.

6 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 119–129, 2020. doi:10.1145/3357713.3384312.

7 Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. *Distributed Computing*, 36(4):501–528, 2023. doi:10.1007/S00446-023-00452-2.

8 Keren Censor-Hillel, Ran Gelles, and Bernhard Haeupler. Making asynchronous distributed computations robust to noise. *Distributed Computing*, 32:405–421, 2019. doi:10.1007/S00446-018-0343-5.

9 Jérémie Chalopin, Yi-Jun Chang, Lyuting Chen, Giuseppe A. Di Luna, and Haoran Zhou. Content-oblivious leader election in 2-edge-connected networks, 2025. doi:10.48550/arXiv.2507.08348.

10 Yi-Jun Chang, Tsvi Kopelowitz, Seth Pettie, Ruosong Wang, and Wei Zhan. Exponential separations in the energy complexity of leader election. *ACM Transactions on Algorithms*, 15(4), 2019. doi:10.1145/3341111.

11 Imrich Chlamtac and Shay Kutten. On broadcasting in radio networks-problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 2003.

12 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proceedings of the 24th International Conference on Distributed Computing*, DISC'10, pages 148–162, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-15763-9_15.

13 Artur Czumaj and Peter Davies. Leader election in multi-hop radio networks. *Theor. Comput. Sci.*, 792(C):2–11, November 2019. doi:10.1016/j.tcs.2019.02.027.

14 Artur Czumaj and Peter Davies. Exploiting spontaneous transmissions for broadcasting and leader election in radio networks. *Journal of the ACM (JACM)*, 68(2):1–22, 2021. doi:10.1145/3446383.

15 Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

16 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018. doi:10.1007/S00446-016-0281-Z.

17 Elena Dubrova. *Fault-Tolerant Design*. Springer, Berlin, 2013. doi:10.1007/978-1-4614-2113-9.

18 Fabien Dufoulon, Janna Burman, and Joffroy Beauquier. Beeping a deterministic time-optimal leader election. In *32nd International Symposium on Distributed Computing (DISC 2018)*, pages 20:1–20:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.20.

19 Yuval Emek and Eyal Keren. A thin self-stabilizing asynchronous unison algorithm with applications to fault tolerant biological networks. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 93–102, 2021. doi:10.1145/3465084.3467922.

20 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing (PODC)*, pages 137–146, 2013. doi:10.1145/2484239.2484244.

21 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

22 Orr Fischer and Merav Parter. Distributed CONGEST algorithms against mobile adversaries. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 262–273. ACM, 2023. doi:10.1145/3583668.3594578.

23 Klaus-Tycho Förster, Jochen Seidel, and Roger Wattenhofer. Deterministic leader election in multi-hop beeping networks. In *Proceedings of 28th International Symposium on Distributed Computing (DISC)*, pages 212–226. Springer, 2014.

24 Fabian Frei, Ran Gelles, Ahmed Ghazy, and Alexandre Nolin. Content-Oblivious Leader Election on Rings. In Dan Alistarh, editor, *38th International Symposium on Distributed Computing (DISC 2024)*, volume 319 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2024.26.

25 Ran Gelles. Coding for interactive communication: A survey. *Foundations and Trends® in Theoretical Computer Science*, 13(1–2):1–157, 2017. doi:10.1561/0400000079.

26 Mohsen Ghaffari and Bernhard Haeupler. Near optimal leader election in multi-hop radio networks. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 748–766. SIAM, 2013. doi:10.1137/1.9781611973105.54.

27 Yael Hitron and Merav Parter. Broadcast CONGEST algorithms against adversarial edges. In Seth Gilbert, editor, *Proceedings of the 35th International Symposium on Distributed Computing (DISC)*, volume 209 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.23.

28 Yael Hitron and Merav Parter. General CONGEST compilers against adversarial edges. In Seth Gilbert, editor, *Proceedings of the 35th International Symposium on Distributed Computing (DISC)*, volume 209 of *LIPIcs*, pages 24:1–24:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.24.

29 Yael Hitron, Merav Parter, and Eylon Yogev. Broadcast CONGEST algorithms against eavesdroppers. In Christian Scheideler, editor, *Proceedings of the 36th International Symposium on Distributed Computing (DISC)*, volume 246 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.27.

30 Yael Hitron, Merav Parter, and Eylon Yogev. Secure distributed network optimization against eavesdroppers. In Yael Tauman Kalai, editor, *Proceedings of the 14th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 251 of *LIPIcs*, pages 71:1–71:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITCS.2023.71.

31 Abhishek Jain, Yael Tauman Kalai, and Allison Bishop Lewko. Interactive coding for multiparty protocols. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 1–10, 2015. doi:10.1145/2688073.2688109.

32 Israel Koren and C. Mani Krishna, editors. *Fault-Tolerant Systems*. Morgan Kaufmann, San Francisco, 2nd edition, 2020. doi:10.1016/B978-0-12-818105-8.

33 Merav Parter. A graph theoretic approach for resilient distributed algorithms. In Alessia Milani and Philipp Woelfel, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, page 324. ACM, 2022. doi:10.1145/3519270.3538453.

34 Merav Parter and Eylon Yogev. Distributed algorithms made secure: A graph theoretic approach. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1693–1710. SIAM, 2019. doi:10.1137/1.9781611975482.102.

35 Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 107–116. ACM, 2019. doi:10.1145/3293611.3331620.

36 Andrzej Pelc. Reliable communication in networks with byzantine link failures. *Networks*, 22(5):441–459, 1992. doi:10.1002/NET.3230220503.

37 Sridhar Rajagopalan and Leonard Schulman. A coding theorem for distributed computation. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing (STOC)*, pages 790–799, 1994. doi:10.1145/195058.195462.

38 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach*. Springer, Cham, 2018. doi:10.1007/978-3-319-94141-7.

39 Herbert Ellis Robbins. A theorem on graphs, with an application to a problem of traffic control. *The American Mathematical Monthly*, 46(5):281–283, 1939.

40 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS 1989*, volume 349 of *Lecture Notes in Comput. Sci.*, pages 304–313. Springer, 1989. doi:10.1007/BFB0028994.

41 Leonard J Schulman. Communication on noisy channels: A coding theorem for computation. In *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*, pages 724–733. IEEE Computer Society, 1992. doi:10.1109/SFCS.1992.267778.

42 Leonard J Schulman. Deterministic coding for interactive communication. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC)*, pages 747–756, 1993. doi:10.1145/167088.167279.

43 Leonard J Schulman. Coding for interactive communication. *IEEE Transactions on Information Theory*, 42(6):1745–1756, 1996. doi:10.1109/18.556671.

44 Yuichi Sudo and Toshimitsu Masuzawa. Leader election requires logarithmic time in population protocols. *Parallel Processing Letters*, 30(01):2050005, 2020.

45 Robin Vacus and Isabella Ziccardi. Minimalist leader election under weak communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 406–416, 2025. doi:10.1145/3732772.3733559.

A Additional Related Work

Fault-tolerant distributed computing involves two main challenges. The first is message corruption, which can occur due to factors such as channel noise. The second is the presence of faulty nodes and edges, which can arise from issues like unreliable hardware or malicious attacks.

To address message corruption, a common approach is to introduce additional redundancy using coding techniques, known as *interactive coding*. The noise affecting the messages can be either random or adversarial, and there is usually an upper bound on the level of randomness or adversarial behavior. The study of interactive coding was initiated by Schulman [41, 42, 43] in the two-party setting and later extended to the multi-party case by Rajagopalan and Schulman [37]. For a comprehensive overview of interactive coding, see the survey by Gelles [25].

For networks with n nodes, the maximum fraction of corrupted messages that a distributed protocol can tolerate is $\Theta(1/n)$ [31]. If more than this fraction of messages are corrupted, the adversary can fully disrupt the communication of the node that communicates the fewest messages. Censor-Hillel, Gelles, and Haeupler [8] presented a distributed interactive coding scheme that simulates any asynchronous distributed protocol while tolerating an optimal corruption of $\Theta(1/n)$ of all messages. A key technique underlying their algorithm is a content-oblivious BFS algorithm.

The celebrated impossibility theorem of Fischer, Lynch, and Paterson [21] states that achieving consensus in an asynchronous distributed system is impossible for a deterministic algorithm when one or more nodes may crash. Similar impossibility results on the solvability of consensus have been found when processes are correct but may experience communication failures, such as message omissions, insertions, and corruptions [40].

If the number of Byzantine edges is bounded by f, reliable communication can only be achieved if the graph is at least $2f$-edge-connected [15, 36].

A series of recent studies has focused on resilient and secure distributed graph algorithms in the synchronous setting [35, 34, 28, 27, 29, 33, 22, 30]. These works have developed compilation schemes that transform standard distributed algorithms into resilient and secure versions. For instance, in a $(2f+1)$-edge-connected network, any distributed algorithm in the CONGEST model can be adapted to remain resilient against up to f adversarial edges [28].

Content-oblivious computation has also been studied in the synchronous setting. In the *beeping model* introduced by Cornejo and Kuhn [12], during each communication round, a node can either beep or listen. If the node listens, it can distinguish between silence or the presence of at least one beep. Several leader election algorithms have been developed for the beeping model and its variations [13, 18, 23, 26, 45].

Several additional distributed models have been designed with limited communication capabilities to capture various real-world constraints. These include radio networks [11], population protocols [2], and stone-age distributed computing [20]. Leader election has been extensively studied in these models [1, 6, 13, 14, 10, 16, 19, 26, 44, 45].

B Pseudocode for Theorem 1

In this appendix, we present the pseudocode for leader election in general 2-edge-connected networks. The algorithm for Theorem 1 is presented in Algorithm 2, with the DFS notification subroutine detailed in Algorithm 3. There are five main variables used in our algorithm:

State(v): This variable represents the current state of v. We say a node v is in the *synchronized counting* phase if $\mathsf{State}(v) = \perp$. Otherwise, the node is in the *DFS notification* phase, and $\mathsf{State}(v) \in \{\mathsf{Leader}, \mathsf{Non\text{-}Leader}\}$ indicates the output of v. At the start, every node is in the synchronized counting phase. Once a node advances to the DFS notification phase, it fixes its output and cannot go back to the synchronized counting phase.

LeaderID(v): This variable stores the identifier of the leader as known by node v.

Parent(v): This variable stores the port number through which v connects to its parent in T_G.

Count(v): This variable implements the counter used in the synchronized counting phase.

$\mathcal{P}(v)$: This variable represents the remaining unexplored ports of v in the DFS notification phase. Precisely, $\mathcal{P}(v)$ is the set of ports on which v has not yet received $\mathsf{LeaderID} + N + 2$ pulses.

For ease of presentation, we define the following two actions:

SendPulsesUntil$_i(k)$: Send pulses along port i until a total of k pulses have been sent from the start of the algorithm (i.e., $\sigma_i = k$).

SendAll(1): Send one pulse along each port.

Events. For ease of analysis, several positions in Algorithm 2 and Algorithm 3 are annotated with labels: $\mathsf{StartDFS}(v)$ (node v initiates the DFS), $\mathsf{SendExplore}_i(v)$ (node v begins sending an explore-notification on port i), $\mathsf{ReceiveExplore}_i(v)$ (node v confirms receipt of an explore-notification from port i), $\mathsf{SendDone}_i(v)$ (node v begins sending a done-notification on port i), and $\mathsf{ReceiveDone}_i(v)$ (node v confirms receipt of a done-notification from port i). These labels are treated as *events* in the analysis in the full version [9] of the paper.

Algorithm 2 Leader election algorithm for node v.

```
1  ID(v) ← ID(v) · N          // Forcing all IDs to be integer multiples of N
2  State ← ⊥; LeaderID ← ⊥; Parent ← ⊥
3  Count ← 0
4  SendAll(1)
5  while State = ⊥ do                                   // Synchronized counting
6      on event ρ_i is incremented for some i ∈ [deg(v)] do
7          if ρ_i = min_{j∈[deg(v)]} ρ_j then           // Counter increment condition
8              Count ← Count + 1
9              SendAll(1)
10             if Count = ID(v) then                    // Leader exit condition
11                 State ← Leader
12                 LeaderID ← ID(v)
13                 Event: StartDFS(v)
14         else if ρ_i − σ_i > 1 then                   // Non-Leader exit condition
15             State ← Non-Leader
16             Parent ← i
17             LeaderID ← ⌊Count/N⌋ · N
18             wait until ρ_i = LeaderID + N + 2
19             Event: ReceiveExplore_i(v)
20     end
21 end
22 Notify(State, LeaderID, Parent)                      // Algorithm 3
```

Algorithm 3 Notify(State, LeaderID, Parent) for node v.

```
1  P = [deg(v)]                                              // The set of all ports
2  if State = Non-Leader then
3  |  P ← P \ {Parent}
4  while P ≠ ∅ do
5  |  j ← min(P)    // Traversal order:  prioritizing smaller port numbers
6  |  Event: SendExplore_j(v)
7  |  wait until ρ_j ≥ LeaderID + 1
8  |  SendPulsesUntil_j(LeaderID + N + 2)                   // Continue DFS on port j
9  |  while j ∈ P do
10 |  |  on event ρ_h = LeaderID + N + 2 for some h ∈ P do
11 |  |  |  if j = h then                                   // DFS returned from port j
12 |  |  |  |  P ← P \ {j}
13 |  |  |  |  Event: ReceiveDone_j(v)
14 |  |  |  else                  // Incoming DFS from port h via a back edge
15 |  |  |  |  P ← P \ {h}
16 |  |  |  |  Event: ReceiveExplore_h(v)
17 |  |  |  |  Event: SendDone_h(v)
18 |  |  |  |  SendPulsesUntil_h(LeaderID + N + 2)
19 |  |  end
20 |  end
21 end
22 if State = Non-Leader then
23 |  Event: SendDone_Parent(v)
24 |  SendPulsesUntil_Parent(LeaderID + N + 2)          // Return DFS on parent port
```