Deterministic Leader Election in Multi-Hop Beeping Networks
[Extended Abstract]*

Klaus-Tycho Foerster, Jochen Seidel, and Roger Wattenhofer

Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland
{foklaus,seidelj,wattenhofer}@ethz.ch

Abstract. We study deterministic leader election in multi-hop radio networks in the beeping model. More specifically, we address explicit leader election: One node is elected as the leader, the other nodes know its identifier, and the algorithm terminates at some point with the network being quiescent. No initial knowledge of the network is assumed, i.e., nodes know neither the size of the network nor their degree, they only have a unique identifier. Our main contribution is a deterministic explicit leader election algorithm in the synchronous beeping model with a run time of $O(D \log n)$ rounds. This is achieved by carefully combining a fast local election algorithm with two new techniques for synchronization and communication in radio networks.

1 Introduction

Distributed computing and wireless communication are prime application areas for randomization, as randomized algorithms are often both simpler and more efficient than their deterministic counterparts. However, in some cases the randomized algorithm is only of Monte Carlo nature, i.e., with some probability the algorithm fails. This is a problem if the randomized algorithm is used as a starting point for other (deterministic and Las Vegas) algorithms, as the algorithm as a whole can also not provide any guarantees anymore. A classic example for such a basic problem is leader election, which is often used to as a first step for other wireless algorithms. We would argue in this paper that leader election deserves to be understood deterministically as well, and we present a new algorithm that solves leader election in the wireless beeping model – our algorithm is slower than the fastest known randomized algorithm, but the overhead is bearable.

The beeping model has emerged as an alternative to the traditional radio network model. The beeping model is binary, in a synchronous time step nodes can only choose to beep or not to beep. If a node is beeping, it does not get any feedback regarding other nodes. On the other hand, if a node is silent, it will

* The full version of this paper is available at http://disco.ethz.ch/publications/DISC2014-leader.pdf

learn whether all its neighbors are also silent, or whether at least one neighbor is beeping. The beeping model was introduced to the distributed computing community by Cornejo and Kuhn [7] shortly after it was implemented [8].

In this model, we deterministically solve leader election: All the nodes in the multi-hop network have to agree on a single leader. As leader election is impossible without nodes having unique identifiers [1], we assume that each node is equipped with a unique ID. We want our algorithm to be uniform, i.e., apart from their ID, nodes have no knowledge about any global or local network properties (e.g., the network size, or their degree).

Our main result is an algorithm that deterministically solves the leader election problem in $O(D \log n)$ time, where D is the diameter of the network and n is the number of nodes. Once a leader is elected, all nodes in the network know the leader's ID, and the network is quiescent. We achieve this task by carefully combining several methods.

1.1 Overview

First, we describe a `Campaigning` algorithm (Section 3) that can be compared to one iteration of a real word political campaign: Every node is equipped with a candidate leader and attempts to convince its neighborhood that this candidate would make a great leader. The idea is that, if enough campaigns are performed, everyone will be convinced of the same leader, since her influence spreads at least one hop per iteration. In other words, we would like to perform multiple campaigns, one after another.

As it turns out, in the beeping model ensuring that the next algorithm starts synchronized is a non-trivial task. We thus develop a technique that allows us to sequentially execute algorithms (Section 4) and apply it to the `Campaigning` algorithm (Section 4.3).

The third method establishes a "back-channel" (Section 5) that directs messages towards a specific node, in our case the current candidate leaders. This allows the last remaining candidate to detect its election and turn the network quiescent. Our main result is now obtained by executing the `Campaigning` algorithm multiple times sequentially, while at the same time using the back-channel to notify the global leader when its successful election is detected. Lastly, we briefly sketch how our algorithm can be extended to include a simple synchronized wake-up protocol (Section 6).

1.2 Related Work

Leader election is one of the fundamental problems in distributed computing, often used as the first step for solving a myriad of other problems in networks. As such, the problem was studied over decades in various communication and network models [17].

In radio networks, communication takes usually place in synchronous rounds, and nodes may either transmit or listen in every round. If a node transmits, it

cannot hear incoming messages, but the message is sent to all its neighbors at once. If a node listens, it receives messages from all its neighbors, but the message obtained depends on the model of collision detection. Should collision detection be available, then a node can separate between no message sent, exactly one message sent, or a collision of multiple messages. With no collision detection available nodes can only distinguish between exactly one message sent to it or just noise.

Leader election in radio networks was first considered in single-hop radio networks, followed by the study of multi-hop radio networks. We start with a short coverage of the single-hop case:

For deterministic algorithms in single-hop radio networks, the run time highly depends on the availability of collision detection: With collision detection, it is $\Theta(\log n)$ [3,12,13,19], while without collision detection, it is $\Theta(n \log n)$ [6]. A similar case can be made for randomized algorithms in single-hop radio networks: With collision detection, the expected run time is $\Theta(\log \log n)$ [20]. The expected run time goes to $O(\log n)$ if w.h.p. is desired. Should no collision detection be available, then the run time increases to $\Theta(\log n)$ in the expected case [2,16], and to $\Theta(\log^2 n)$ w.h.p. [14].

We would argue that the study of leader election in multi-hop radio networks can be divided into the following fields for related work to our results. One can consider (1) radio networks with or (2) without collision detection, and (3) the beeping model. Second, the used algorithms can be either deterministic or randomized. We refer to [4,10,15] for an extended overview of these areas.

For deterministic algorithms, Kowalski and Pelc [15] displayed the discrepancy between models with and without collision detection. They showed that if collision detection is available, the runtime is $\Theta(n)$, while without collision detection, there is a lower bound of $\Omega(n \log n)$. Their $O(n)$ algorithm with collision detection relies on a careful combination of multiple innovative techniques, e.g., remote token elimination and distributed fuzzy-degree clustering. In contrast to the model in this paper, they require messages of logarithmic size, collision detection, and the knowledge of an upper bound polynomial in the number of identifiers. Our algorithm can be simulated therein since their model is strictly stronger. Asymptotically, we achieve a better run time for graphs with a diameter $D \in o(n/\log n)$, cf. [10].

For randomized algorithms in radio networks without collision detection, Chlebus, Kowalski, and Pelc [4] broke the $\Omega(n \log n)$ barrier: They present a randomized algorithm with $O(n)$ expected time and prove a lower bound of $\Omega(n)$. Furthermore, they give a deterministic algorithm for the model without collision detection with a run time of $O(n \log^{3/2} n \sqrt{\log \log n})$. They use logarithmic size messages and also assume that an upper bound on the network size is known.

Finally, Ghaffari and Haeupler [10] considered randomized leader election in the beeping model. Their algorithm runs in $O((D + \log n \log \log n) \cdot \min(\log \log n, \log n/D))$ time. To choose the random starting set of candidates, they rely on knowledge of n, while we assume our algorithms to be uniform. To cope with overlapping transmissions, they present a sophisticated technique using super-

imposed codes. We deem our overhead of $O(\log n)$ in the worst case bearable. To the best of our knowledge, no results are published for deterministic leader election in the beeping model.

The authors of [10] also consider a variant of the beeping model in which only a subset $S \subseteq V$ of the nodes wakes up in round 0 [11]. We adapt our algorithm to this setting in Section 6. The difficulty is to allow nodes that are being woken up by neighbors to synchronize their execution with that of nodes that are already awake. The related wake-up problem, where nodes may also activate spontaneously and no collision detection is available was studied in its own right, for single-hop [9] as well as multi-hop [5] networks. In [18] the goal is to activate the whole network if exactly one node is active initially.

2 Preliminaries

Network Model. The network is modeled as a connected undirected graph $G = (V, E)$ with node set V and edge set E. We denote by D the diameter of G, and by n the number of nodes in V. All nodes $u \in V$ have a unique identifier (ID), denoted by $\mathrm{id}(u)$, from the range $\{1, 2, \ldots, O(n^\gamma)\}$, with $\gamma \geq 1$ being a constant. We denote by $l(v)$ the *length* of u's identifier in bits, i.e., $l(u) = \lceil \log_2(\mathrm{id}(u)) \rceil$. The *neighborhood* $\mathcal{N}(u)$ of u is the set $\{u\} \cup \{v : (u, v) \in E\}$. In a similar fashion, the *d-neighborhood* $\mathcal{N}(u, d)$ of a node u contains all nodes with a distance of at most d to u, e.g., $\mathcal{N}(u, 1) = \mathcal{N}(u)$.

Beeping Model. We consider one of the most basic communication models, the synchronous beeping model: All nodes start synchronized[1] in round 0, and communication between nodes proceeds in synchronous rounds, where messages are transmitted via the edges of the network. In every round, each node may choose to either *beep* or *listen* to incoming messages. If a node v beeps, the beep will be transmitted to all nodes in $\mathcal{N}(v)$. Otherwise, if v listens, then the message received by v in a round is defined as follows: (i) if no node in $\mathcal{N}(v)$ beeps, then v receives a 0 (*silence*), and (ii) if one or more nodes in $\mathcal{N}(v)$ beeps, then v receives a 1 (*beep*).

Uniform Algorithms. We only consider uniform algorithms. That is, unless mentioned otherwise, the input for a node v consists of only $\mathrm{id}(v)$ (but not the value of γ). Note that neither n, nor D, nor any upper bounds on those network parameters, can be inferred from the value $\mathrm{id}(v)$ (or $l(v)$) of a single node v. Nodes also do not have any knowledge about the network topology, e.g., the IDs of their neighbors, or even their own degree. Moreover, we require that in every network the algorithm reaches a quiescent state, i.e., a state in which no node transmits beeps anymore.

[1] In Section 6 we also handle the case in which only a subset of the nodes wakes up.

3 Convincing Your Neighbors

In this section, we give an algorithm called `Campaigning` that can be compared to a political campaign at a word of mouth level. Everyone is convinced that either she herself is a good candidate, or that she knows the name of a good candidate. If you know a better candidate than all of your neighbors and their neighbors, you will try to convince your direct neighbors. However, if they are aware that a better candidate is out there — they will ignore your conversion attempts. Some candidates might reach a good deal of local followers, but only a globally best candidate can guarantee to spread her sphere of influence all the time.

The algorithm `Campaigning` can be seen as one iteration of this process, where nodes exchange information only with their local neighborhood. The general idea is that after D iterations are performed, *"There can be only one!"*[2], and all nodes will be convinced of the same leader. Hence, the candidates of the different nodes do not have to be unique, e.g., the algorithm works with just one candidate for all nodes or n different candidates.

We have to reach a state where nodes can transmit information to their neighbors, without other nodes disturbing them, since beeps do not encode relevant further information. Particular challenges arise from the facts that the algorithm has to be uniform, i.e., that n is not known, and that we are confined to the restricted beeping model. E.g., one cannot just "beep the identifier" and then proceed with another part of the algorithm, since any receiving node will hear all its neighbors – and cannot distinguish if all sent a beep or just one.

The main idea is to first reach local consensus on the longest identifier, then to agree locally on the highest identifier, and finally, to let those with the locally highest identifier transmit their identifier to their neighbors. To reach a state of local consensus, we turn some nodes into buffer-nodes that no longer participate. Therefore, we divide our algorithm into three separate procedures `campaign_longest_id`, `campaign_highest_id`, and `campaign_transmit_id`.

We first give an overview of the three procedures in Subsection 3.1, followed by a detailed mode of operations for `Campaigning` in Subsection 3.2. The full version also contains a pseudo code description of our algorithm. We conclude by stating correctness and run time results in Subsection 3.3.

3.1 Overview of the Procedures

Every node v gets as input an id, referred to as *campaigning-identifier*, that is stored in v's variable id_{in}. Also, all nodes start in an *active role*, but can change to be *passive* or *inactive* during the algorithm. Active nodes might convince their neighbors at the end and passive nodes might receive a new candidate, but it can be necessary to turn nodes inactive to let them act as local separators.

After the first procedure, `campaign_longest_id`, exactly those nodes v with the *longest* id_{in} in their 2-neighborhood are still active. If a node v is not active,

[2] Connor MacLeod, 1985. In *Highlander*

but has an active neighbor w, then v turns passive, since it is interested in the campaigning-identifier of w. Nodes not fulfilling either of these requirements turn inactive. Furthermore, to separate clusters of active nodes with campaigning-identifiers of different length, the procedure creates buffers of inactive nodes between them. Thus, inside a cluster, all active campaigning-identifiers are of equal length, allowing each cluster to agree on a common starting time for the following procedure.

The second procedure `campaign_highest_id` mimics the first procedure, but now for the *highest* instead of the *longest* identifier. After `campaign_highest_id`, exactly those nodes v with the highest id_{in} in their 2-neighborhood remain active. The buffer of inactive nodes is extended to separate active nodes with different campaigning-identifiers. Hence, in the third procedure `campaign_transmit_id`, all still active nodes can convince their passive neighbors unhindered.

3.2 Details of the Algorithm

In this subsection, we describe the algorithm `Campaigning` and each of its three procedures for a node $v \in V$. We describe the algorithm from the perspective of a single node v. The input campaigning-identifier for v is stored in id_{in}, and the length of id_{in} in bits is stored in the variable l_{in}. Furthermore, v initializes the variables $role \leftarrow active$, $l_{out} \leftarrow l_{in}$, and $\mathrm{id}_{out} \leftarrow \mathrm{id}_{in}$. Then, the node v executes `campaign_longest_id`, `campaign_highest_id`, `campaign_transmit_id`, and the output of node v is id_{out}. Should a node become inactive at any time, i.e., if $role = inactive$, then the algorithm immediately terminates and the value currently stored in id_{out} is returned as v's output.

Each procedure consist of *phases*, which are divided into three rounds each. For ease of notation, we call the rounds in one phase *slots*, i.e., slot 0, slot 1, and slot 2. Conceptually, the first two slots 0 and 1 of each phase are used to transmit data, while slot 2 will exclusively be used for notification signals from active nodes. Recall that v *hears* a beep only if some node $u \in \mathcal{N}(v)$, $u \neq v$ transmits a beep, i.e., v does not hear beeps of itself.

`campaign_longest_id`. The length of `campaign_longest_id` may vary; at the end of the procedure, node v stores the number of elapsed phases in l_{out} if at the end of the procedure v is active or passive. Node v starts by beeping in slots 0 and 1 for the first $l_{in} - 1$ phases. Then, v listens in slot 0, and beeps received in slot 0 are relayed in slot 1. If v relays at least one beep, it turns passive. Should a beep be heard in the next slot 2, the node v turns inactive. Otherwise, already in phase l_{in} there was no beep to relay. In that case, if a (relayed) beep is received in slot 1, then v turns inactive. Else v beeps in slot 2 of that phase and finishes the procedure as active. Should after phase l_{in} a beep be heard in slot 1, the passive relaying node v turns inactive as well. Should there be a phase where the passive node v hears no beeps in slot 0,1, it either *i*) turns inactive if no beep is heard in slot 2, or *ii*) finishes the procedure as passive if a beep is heard in slot 2.

campaign_highest_id. This procedure consists of l_{out} phases, and we denote the current phase of node v by p.

If v is passive at the beginning of phase p, then beeps heard in slot 0 are relayed in slot 1. Should no beep be heard in slot 0, but a beep is heard in slot 1, v turns inactive. Also, if no beep is heard in slot 2 of phase $l_{out}(v)$, then v turns inactive.

We denote by the positions $1, \ldots, l_{in}$ the bits of id_{in}, starting from the most significant bit. If v is active at the beginning of phase p, then v beeps in slots 0 and 1 if position p in id_{in} is a 1 bit. Else, when a beep is heard in slot 0 or 1, v turns passive. If the current phase is l_{out} and v is still active, then v beeps in slot 2.

campaign_transmit_id. Much like `campaign_highest_id`, this procedure consists of l_{out} phases. An active node v uses the l_{out} phases to transmit the l_{out} bits of id_{in}, whereas passive nodes store the l_{out} received bits in id_{out}.

3.3 Convincing via `Campaigning`

We can now state some important properties of the algorithm `Campaigning`, which will be used in the next sections to prove our main result. For formal proofs please refer to the full version of this paper; here we restrict ourselves to presenting the necessary key ideas. We begin with the following correctness lemma, which essentially states that nodes may only adopt identifiers from their neighborhood, i.e., identifiers spread only locally and no new identifiers are created.

Lemma 1. *Let v be a node that just finished algorithm* `Campaigning`$(id_{in}(v))$. *Then $id_{out}(v) \leq \max_{w \in \mathcal{N}(v,1)} \mathrm{id}_{in}(w)$ and $\exists x \in \mathcal{N}(v,1)$ s.t. $id_{out}(v) = id_{in}(x)$.*

The proof to Lemma 1 consists of a careful case distinction based on the node's *role* in the `Campaigning` algorithm. In Theorem 2, we show that the influence of a potential leader will spread one hop per round. This is crucial for the whole leader election process, since it will be extended later on to show that D executions of the algorithm suffice to convince all nodes of the leader.

Theorem 2. *Execute algorithm* `Campaigning`$(\mathrm{id}_{in}(v))$ *for $\forall v \in V$. Let $v' \in V$ be a node with $\mathrm{id}_{in}(v') = \max_{w \in \mathcal{N}(v',3)} \mathrm{id}_{in}(w)$. Then for all nodes $u \in \mathcal{N}(v',1)$ holds: $\mathrm{id}_{out}(u) = \mathrm{id}_{in}(v')$.*

The above theorem is established by observing that a node v with a locally highest campaigning-identifier (i.e., the highest id_{in} in $\mathcal{N}(v,3)$) remains active, its neighbors do not turn inactive, and thus the campaigning-identifier is propagated one hop. Finally, Theorem 3 states that the run time of `Campaigning` depends only on the largest campaigning-identifier length in the 1-neighborhood.

Theorem 3. *Execute algorithm* `Campaigning`$(\mathrm{id}_{in}(v))$ *for $\forall v \in V$. The run time for each node v is $O(\max_{w \in \mathcal{N}(v,1)} l_{in}(w))$ rounds.*

This is true since the maximum run time of a node is completely determined after `campaign_longest_id` has finished. Recall that all identifiers are at most in $O(n^\gamma)$, and hence the run time is bounded by $O(\log n)$ rounds. Since Lemma 1 ensures that no new identifiers are created in the network, we obtain the following corollary.

Corollary 4. *Let* $\max_{v \in V} l_{in}(v) \in O(\log n)$*. It holds that the run time of algorithm* `Campaigning`$(\mathrm{id}_{in}(v))$ *is* $O(\log n)$ *rounds for* $\forall v \in V$*.*

4 Convincing Your Network

We would like to apply the campaigning method presented in the previous section to propagate the highest ID further. In other words, we need to execute `Campaigning` multiple times in succession. This task would be easy if there was some kind of global synchronization in order to guarantee that all nodes can start the next invocation of the campaigning algorithm at the same time. However, since the node labels have different lengths, so does each campaign. To overcome this obstacle, we design a generic approach to sequentially execute arbitrarily many algorithms in the beeping model. The key ingredient in our approach is the following *balanced counter* technique.

4.1 Balanced Counters

We present a method that enables the network to manage a balanced counter for every node u. At every node u, our balanced counter technique stores an integer value denoted by *counter*. To manipulate *counter* the two methods `increment` and `reset`, which instruct the counter to increment its value by one or reset it to zero, respectively, are provided. Our goal is to satisfy the following *balancing property*: For any two neighboring nodes u, v *participating*, i.e., not currently resetting their counters, the *counter* values of u and v shall differ by at most 1.

Note that transmitting the whole counter value in every round is not feasible due to the limited nature of the communication means the nodes have at their disposal. However, it turns out that transmitting the *counter* value modulo 3 suffices to ensure the balancing property. The transmission technique we use requires three reserved rounds, and allows a node to determine whether their neighbors have a lower counter value than themselves. The idea is now that nodes refrain from incrementing the counter as long as there are neighbors that are still behind.

We describe the balanced counter technique from the perspective of some node u using a state machine. Each node may be in one of the following states: Count, Reset-Notify, or Reset-Wait, and we denote u's current state by *state*. If *state* = Count, then u is considered to be a *participating* node, and either `increment` or `reset` may be invoked at u. In the other two states those operations are not available to u. The only allowed state transitions for node u are

1. COUNT → RESET-NOTIFY if no node $v \in \mathcal{N}(u)$ is in RESET-WAIT,
2. RESET-NOTIFY → RESET-WAIT if no node $v \in N(u)$ is in COUNT, and
3. RESET-WAIT → COUNT if no node $v \in N(u)$ is in RESET-NOTIFY.

Communication of the balanced counter technique is subdivided into phases indexed by the positive integers. Each individual phase consists of 6 rounds; to avoid confusion we use the term slot to refer to the individual rounds within a phase. The role of the first three slots $(0, 1, 2)$ is to transmit the counter increments, whereas the last three slots $(3, 4, 5)$ are used to transmit the node's current state. We now give a detailed description of the balanced counter technique; the full version also includes a pseudo-code description.

Initially, the state of u is COUNT, and $counter = 0$. In each phase, the operation at node u is as follows:

1. If $state =$ COUNT, then u beeps in slot $counter \pmod 3$ and in slot 3;
2. If $state =$ RESET-NOTIFY, then u beeps in slot 4; and
3. If $state =$ RESET-WAIT, then u beeps in slot $counter \pmod 3$ and in slot 5.

Node u listens in all slots in which it does not beep.

`Increment`. The purpose of this operation is to increment *counter* by one without violating the balancing property. When `increment` is invoked at node u, then u waits for the first phase in which no beep is received in slot $counter - 1 \pmod 3$ (note that u never transmits in slot $counter - 1 \pmod 3$). Node v increments *counter* by 1 at the end of that phase and returns from the `increment` operation.

`Reset`. The purpose of this operation is to reset node u's value of *counter* to zero in accord with neighboring nodes $v \in N(u)$, while allowing nodes v to proceed participating before invoking `reset` themselves. Specifically, when `reset` is invoked at node u, then u successively transitions (1) from COUNT to RESET-NOTIFY, thereby setting $counter \leftarrow 0$, (2) from RESET-NOTIFY to RESET-WAIT, and eventually (3) from RESET-WAIT back to COUNT. In this process u respects the aforementioned restrictions for state transitions, utilizing the transmissions in slots 3 to 5. In particular, the aforementioned transition (i), $1 \leq i \leq 3$, is consummated in the first phase in which no beep is received in slot $2 + i$.

An inductive argument can be used to establish the following lemma; a formal proof is presented in the full version of this paper.

Lemma 5. *The balanced counter technique satisfies the balancing property.*

4.2 Balanced Executions

Consider two algorithms $\mathcal{A}$ and $\mathcal{B}$ that shall be simulated sequentially. To achieve our goal, we intend to simulate the execution of $\mathcal{A}$ and $\mathcal{B}$ in the network. In $\mathcal{A}$'s simulation, the balanced counter is used as a round counter. Since the round counter satisfies the balancing property, it is ensured that the simulations performed by neighboring nodes progress at the same rate. When at some node u

the simulation of $\mathcal{A}$ terminates, the round counter is reset by u. Node u then waits until its round counter returns to the COUNT state and thereupon starts the simulation of $\mathcal{B}$.

One needs to ensure that when round r of $\mathcal{A}$ (or $\mathcal{B}$) is simulated at node u, then u can determine whether one of its neighbors transmitted a beep in round $r-1$ of the simulation. To that end, we extend each phase of the counter technique by three additional slots and reserve the first 6 slots (0–5) for the balanced counter technique. Consider a phase p and a node u currently simulating algorithm $\mathcal{A}$, and denote by r the *counter* value for node u at the beginning of phase p. The three new slots (6–8) are used to transmit and receive the beeps emitted during the simulation as follows.

Assuming that $\mathcal{A}$ did not terminate in round $r-1$, the goal in phase p is to simulate $\mathcal{A}$'s round r. Node u simulates round r of algorithm $\mathcal{A}$ utilizing slot $r \pmod 3 + 6$ to replace $\mathcal{A}$'s access to the communication channel, where beeps received in slot $(r-1 \pmod 3 + 6)$ replace the beeps received by v in the simulation if node u listened in round $r-1$ of $\mathcal{A}$. Moreover, in slot $r-1 \pmod 3$, node u re-transmits a beep if u beeped in the last simulated round $r-1$ under $\mathcal{A}$. If u incremented the counter to the value r in the current phase, i.e., the counter progressed from $r-1$ to r, then v invokes `increment` again. Note that `increment` may delay incrementing r for several phases; in that case, the same round r of $\mathcal{A}$ is simulated in phase p multiple times, and if the beeps received in slot $r-1$ change, then so does the simulated execution of $\mathcal{A}$'s round r.

Otherwise, if $\mathcal{A}$ terminated its execution in the previous round $r-1$, the goal is to safely start the simulation of the next algorithm $\mathcal{B}$ at node u. To that end, node u invokes the `reset` operation. However, the simulated execution of the next algorithm $\mathcal{B}$ (possibly using u's output of $\mathcal{A}$ as input) only starts once u continues participating in the balanced counter, i.e., when $state = \text{COUNT}$.

In the full paper we explain how to exploit the balanced counter property to obtain the following correctness lemma. It essentially states that the balanced execution technique behaves as if global synchronization was used to start the algorithms one after the other.

Lemma 6. *Let $A = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ be a finite sequence of algorithms. Denote, for every $v \in V$, by $\hat{o}_1(v)$ the output produced at v by $\mathcal{A}_1$ when executed on G. For $i > 1$ and for every $v \in V$, denote by $\hat{o}_i(v)$ the output produced at v by $\mathcal{A}_i$ when executed on G, where the input to every $u \in V$ for $\mathcal{A}_i$ is specified as $\hat{o}_{i-1}(u)$.*

It holds that for every node v, the output $o(v)$ produced at v when using the balanced execution technique for A is $o(v) = \hat{o}_k(v)$.

4.3 Leader Election through Campaigning

We now have the tools available to design a non-quiescent leader election algorithm. Utilizing the balanced execution technique, every node executes the `Campaigning` algorithm sequentially, again and again. For every node u, the input to the first invocation of `Campaigning` is $id(u)$, and the input to every following invocation of `Campaigning` is the output of the previous one. In the

following we refer to this basic protocol as the `Restless-LE` (for leader election) algorithm. It is immediate from the design of `Restless-LE`, that the network will never reach a quiescent state — for instance, the balanced counter technique never ceases to transmit. The following lemma states that `Restless-LE` obtains the desired result after at most D invocations of `Campaigning`.

Lemma 7. *If the network G executes* `Restless-LE`*, then for every node $u \in V$, the output produced at u by the D-th invocation of* `Campaigning` *is* $\max_{v \in V} \mathrm{id}(v)$.

Utilizing the balanced execution technique, Lemma 7 can be obtained by inductively applying Lemma 1 and Theorem 2 for D times. Simulating D invocations of `Campaigning` takes $O(D \log n)$ rounds, as is stated in Theorem 8. The proofs to both Lemma 7 and Theorem 8 appear in the full version.

Theorem 8. *If the network G executes* `Restless-LE`*, then for every node $u \in V$, the D-th invocation of* `Campaigning` *terminates after $O(D \log n)$ rounds.*

Note that the network never reaches quiescence since the balanced counter technique continues to beep even after the D-th invocation of `Campaigning` has terminated. Moreover, without knowledge of D, node u has no means to decide when sufficiently many campaigns have been run.

5 Terminating & Achieving Quiescence

It seems that in the previous section we robbed Peter to pay Paul: We obtained `Restless-LE` which finds a leader in time $O(D \log n)$, but now our algorithm does not achieve quiescence, nor does a node know when to terminate. These two flaws could be considered a major drawback if one wishes to use the leader election algorithm as a foundation for another algorithm, since it is unclear when the latter can be started. To overcome this obstacle we implement an overlay network protocol that executes concurrently to the `Campaigning` invocations. The overlay network we establish on top of the original communication graph resembles the layers of an onion with the elected leader at its core. Utilizing the overlay network, we then describe how candidates detect whether the leader election process has terminated. Causing all non-elected nodes to terminate is now achieved by sending a broadcast message.

In order to form the overlay network, each node u keeps track of one additional variable *depth* taking values from the set $\{0, 1, 2\}$, initially set to 0. We say that a path $p = (u_1, \ldots, u_k)$, $(u_{i-1}, u_i) \in E$ for $2 \leq i \leq k$, is a *downward overlay path* if for all $i \geq 2$ it holds that $\mathit{depth}(u_i) = \mathit{depth}(u_{i-1}) + 1 \pmod 3$, and we denote the length k of a path p by $\mathrm{length}(p)$. Conversely, we say that p is an *upward overlay path* if p reversed is a downward overlay path. One can think of downward overlay paths as leading away from the network's core, whereas upward overlay paths lead towards it. Note that initially, all overlay paths consist of only a single node. The general idea is to relay beeps along upward and downward overlay paths. Before extending the `Restless-LE` algorithm to utilize

the overlay network, thus obtaining the quiescent terminating leader election algorithm `Quiescent-LE` in Section 5.1, we describe the operation of our overlay network technique in more detail. Note that in the full version of this paper we include a pseudo-code representation of the overlay network technique.

Every round r of the leader election algorithm is replaced by phases consisting of 10 slots, one single slot and three triplets of slots. The single slot is reserved to execute the non-terminating leader election algorithm we obtained in Section 4.3. For clarity, we refer to the first slot triplet as *control* slots, to the second triplet as *up channel* slots, and to the last triplet as *down channel* slots. The control slots, up channel slots, and down channel slots are numbered from 0 to 2 (e.g., up channel slot 2 is the last slot in the second triplet of slots in a phase). While the role of the control slots is to establish the overlay network, the up and down channel slots are used to transmit beeps to nodes with smaller and higher depth, respectively.

More specifically, in every phase p, node u listens in the up channel $\mathit{depth}-1 \pmod 3$ and in the down channel $\mathit{depth}+1 \pmod 3$. If a beep is received in one of those slots, then in the following phase $p+1$, u beeps in the up channel depth or in the down channel depth, correspondingly. The overlay network further provides the two operations `beep_depth` and `join`, which are implemented as follows. When `beep_depth` is invoked by node u, then u transmits a beep in control slot depth. The corresponding `join` operation causes u to listen in the three control slots; node u then sets $\mathit{depth} \leftarrow i+1 \pmod 3$, where i denotes the index of the first control slot in which a beep was received, thereby joining the overlay network of one of its neighbors that invoked `beep_depth`.

5.1 Quiescent Leader Election

In this section, we describe the `Quiescent-LE` algorithm that utilizes the overlay network technique in conjunction with the `Restless-LE` algorithm. Formation of the overlay network is tightly coupled with `Restless-LE` and the invocations of `Campaigning` therein. Namely, whenever an invocation of `Campaigning` at node u returns a new ID x, node u joins the overlay network of a neighbor that transmitted x to u. Nodes that are currently being convinced of a new leader emit a signal into the upward channel of neighboring nodes, thus ensuring that no candidate terminates unless a consensus on the leader's ID has been reached.

In particular, for a node u in phase p, denote by σ the state in which the last `Campaigning` invocation that terminated for node u was upon termination. Denote further by $\mathit{last\_role}$, $\mathit{last\_in}$, and $\mathit{last\_out}$ the values of the corresponding variables role, $\mathrm{id}_{\mathrm{in}}$ and $\mathrm{id}_{\mathrm{out}}$ in σ. In phase p a node u is called a *candidate* if $\mathit{last\_in} = \mathrm{id}(u)$, and we say that node c is the candidate of u if $\mathit{last\_in} = \mathrm{id}(c)$. The idea is now to utilize the overlay network so that nodes may join the overlay network of their corresponding candidate. This is accomplished by setting the depth variable accordingly whenever the value of $\mathit{last\_out}$ changes.

In `Quiescent-LE`, the operation of node u is as follows (please refer to the full version for a pseudo-code description). If $\mathit{last\_role} = \mathit{active}$, then u invokes `beep_depth`, thus allowing nodes $v \in \mathcal{N}(u)$ to set their depth. Correspondingly,

u invokes `join` if its candidate has changed (i.e., if $last\_role = passive$), in order to assign a new value to its depth variable. In any case, if $last\_role \neq active$, then node u beeps in all three up channel slots in contrast to the normal up channel operation. A candidate that has not received a beep through the up channel for 18 consecutive rounds emits a signal in the down channel, thus instructing nodes to terminate.

Theorem 9. *The uniform algorithm* `Quiescent-LE` *terminates after* $O(D \log n)$ *rounds at every node. Every node returns the same output* $\max_{v \in V} \mathrm{id}(v)$.

The proof to our main result, namely, the above Theorem 9, is based on the concept of coalitions that form around potential leader nodes. All nodes inside a coalition share the same potential leader and form an onion layer network with the potential leader at its core. Eventually, the coalition Z corresponding to the highest identifier z overrules every other. In particular, coalition Z extends its borders by one step in every `Campaigning` invocation. This is crucial in order to ensure proper formation of the onion layer network, which in turn guarantees that the leader node (with identifier z) can safely issue the terminating signal. We refer to the full version of this paper for an extensive proof.

6 Synchronized Wake-Up Protocol

Note that one may also study a variant of the beeping model (see, e.g., [11]) in which only a subset $S \subseteq V$ of the nodes wakes up in round 0. Nodes in $V \setminus S$ are initially *asleep*, and wake up only if they receive a beep from one of their neighbors. In particular, such a node is no longer considered asleep. We briefly discuss how our algorithm can be extended to include a wake-up protocol.

Every original slot in a phase of `Quiescent-LE` is replaced by two slots, where the first slot takes the role of the corresponding original slot, and in the second slot a node is always silent. Additionally, the phase is preceded by another two slots, referred to as wake-up slots. Consider an asleep node u. As soon as u receives a beep, it enters an intermediate *snooze* state, and if u receives a beep in the next round as well, then it turns *awake*. Otherwise, snoozing nodes turn awake after receiving two beeps consecutively. A node that just turned awake enters the protocol after the wake-up slots, thus aligning its execution with awake neighbors. That is, the first round in which u participates corresponds to the first original slot of `Quiescent-LE`. Note that in particular, due to the balanced execution technique, node u postpones the progress of awake neighboring nodes. Lastly, a node u that is awake beeps in both wake-up slots whenever u starts a phase of `Quiescent-LE` that coincides with the beginning of a balanced execution phase, and remains silent in the wake-up slots otherwise.

7 Conclusion

We described a deterministic uniform leader election algorithm in the beeping model that achieves quiescence after $O(D \log n)$ rounds. There are three main ingredients to our algorithm:

1. A `Campaigning` algorithm that propagates the locally highest identifier one hop per invocation.
2. A technique to sequentially execute arbitrarily many algorithms in the beeping model, based on a simple balanced counter approach.
3. An overlay network, based on the onion layer principle.

Our algorithm is obtained by using the sequential execution technique (2.) to execute the `Campaigning` method (1.) multiple times, one after the other. In its first invocation, the algorithm essentially creates a 2-hop independent set containing at least one node. The independent nodes are potential leaders and transmit their identifier to their neighbors. In subsequent invocations, potential leaders correspond to clusters of nodes with the same campaigning-identifier. When clusters touch, the cluster C having the larger campaigning-identifier wins, and the neighboring clusters shrink as bordering nodes join C. This yields a non-quiescent uniform algorithm `Restless-LE` for leader election, where the leader is not informed about her successful election.

If the diameter D was known to all nodes, then termination could be achieved by stopping after the D^{th} invocation of `Campaigning`. However, we want our algorithm to be uniform. We create an onion layer overlay network (3.) in order to achieve uniformity and quiescence. Potential leaders form the core of an onion, and nodes in a cluster are layered according to their distance to the core. Since the cluster of the eventual leader grows in each step, eventually all nodes will be part of a single cluster. The onion layer principle can now be used to establish a communication channel from outer layers towards the core and vice versa. When the cluster stops growing, the leader is informed about her successful election, in turn allowing her to issue a termination signal to all nodes. Lastly, we explain how the algorithm can be extended to handle the synchronous wake-up situation described in [11].

References

1. Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC*, pages 82–93. 1980.
2. Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *J. Comput. Syst. Sci.*, 45 (1): pages 104–126, 1992.
3. John Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Transactions on Information Theory*, 25 (5): pages 505–515, 1979.
4. Bogdan S. Chlebus, Dariusz R. Kowalski, and Andrzej Pelc. Electing a leader in multi-hop radio networks. In *OPODIS*, pages 106–120. 2012.
5. Marek Chrobak, Leszek Gasieniec, and Dariusz R. Kowalski. The wake-up problem in multihop radio networks. *SIAM J. Comput.*, 36 (5): pages 1453–1471, 2007.
6. Andrea E. F. Clementi, Angelo Monti, and Riccardo Silvestri. Distributed broadcast in radio networks of unknown topology. *Theor. Comput. Sci.*, 302 (1-3): pages 337–364, 2003.
7. Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *DISC*, pages 148–162. 2010.

8. Roland Flury and Roger Wattenhofer. Slotted programming for sensor networks. In *IPSN*, pages 24–34. 2010.
9. Leszek Gasieniec, Andrzej Pelc, and David Peleg. The wakeup problem in synchronous broadcast systems. *SIAM J. Discrete Math.*, 14 (2): pages 207–222, 2001.
10. Mohsen Ghaffari and Bernhard Haeupler. Near optimal leader election in multi-hop radio networks. In *SODA*, pages 748–766. 2013.
11. Mohsen Ghaffari and Bernhard Haeupler. Near optimal leader election in multi-hop radio networks. *CoRR*, abs/1210.8439v2, April 2014.
12. Albert G. Greenberg and Schmuel Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *J. ACM*, 32 (3): pages 589–596, 1985.
13. Jeremiah F. Hayes. An adaptive technique for local distribution. *Communications, IEEE Transactions on*, 26 (8): pages 1178–1186, 1978.
14. Tomasz Jurdzinski and Grzegorz Stachowiak. Probabilistic algorithms for the wake-up problem in single-hop radio networks. *Theory Comput. Syst.*, 38 (3): pages 347–367, 2005.
15. Dariusz R. Kowalski and Andrzej Pelc. Leader election in ad hoc radio networks: A keen ear helps. *Journal of Computer and System Sciences*, 79 (7): pages 1164 – 1180, 2013.
16. Eyal Kushilevitz and Yishay Mansour. An $\Omega(D \log(N/D))$ lower bound for broadcast in radio networks. *SIAM J. Comput.*, 27 (3): pages 702–712, 1998.
17. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
18. Andrzej Pelc. Activating anonymous ad hoc radio networks. *Distributed Computing*, 19 (5-6): pages 361–371, 2007.
19. Boris S. Tsybakov and V.A. Mikhailov. Free synchronous packet access in a broadcast channel with feedback. *Probl Inf Transm*, 14 (4): pages 259–280, 1978.
20. Dan E. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM J. Comput.*, 15 (2): pages 468–477, 1986.