HAL
open science

The Computational Power of Beeps

Seth Gilbert, Calvin Newport

To cite this version:

Seth Gilbert, Calvin Newport. The Computational Power of Beeps. DISC 2015, Toshimitsu Masuzawa; Koichi Wada, Oct 2015, Tokyo, Japan. 10.1007/978-3-662-48653-5_3 . hal-01199811

HAL Id: hal-01199811

https://hal.science/hal-01199811v1

Submitted on 16 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Computational Power of Beeps

Seth Gilbert[1,⋆] and Calvin Newport[2,⋆⋆]

[1] National University of Singapore. seth.gilbert@comp.nus.edu.sg
[2] Georgetown University. cnewport@cs.georgetown.edu

Abstract. We study the quantity of computational resources (state machine states and/or probabilistic transition precision) needed to solve specific problems in a single hop network where nodes communicate using only beeps. We begin by focusing on randomized leader election. We prove a lower bound on the states required to solve this problem with a given error bound, probability precision, and (when relevant) network size lower bound. We then show the bound tight with a matching upper bound. Noting that our optimal upper bound is slow, we describe two faster algorithms that trade some state optimality to gain efficiency. We then turn our attention to more general classes of problems by proving that once you have enough states to solve leader election with a given error bound, you have (within constant factors) enough states to simulate correctly, with this same error bound, a logspace TM with a constant number of unary input tapes: allowing you to solve a large and expressive set of problems. These results identify a key simplicity threshold beyond which useful distributed computation is possible in the beeping model.

1 Introduction

The beeping model of network communication [1–3, 10, 14, 20] assumes a collection of computational *nodes*, connected in a network, that interact by *beeping* in synchronous rounds. If a node decides to beep in a given round, it receives no feedback from the channel. On the other hand, if a node decides to listen, it is able to differentiate between the following two cases: (1) no neighbor in the network topology beeped in this round, and (2) one or more neighbors beeped.

Existing work on this model provide two motivations. The first concerns digital communication networks (e.g., [10, 12]). Standard network communication (in which nodes interact using error-corrected packets containing many bits of information) requires substantial time, energy, and computational overhead (at multiple stack layers) to handle the necessary packet encoding, modulation, demodulation, and decoding. Beeps, on the other hand, provide an abstraction capturing the simplest possible communication primitive: a detectable burst of energy. In theory, beep layers could be implemented using a fraction of the complexity required by standard packet communication, establishing the possibility of *micro-network* stacks for settings where high speed and low cost are crucial.

⋆ Supported in part by NUS FRC T1-251RES1404
⋆⋆ Supported in part by NSF grant CCF 1320279

The second motivation for the beeping model concerns a connection to biological systems (e.g., [3, 19, 20]). Network communication in nature is often quite simple; e.g., noticing a flash of light from nearby fireflies or detecting a chemical marker diffused by nearby cells. Therefore, understanding how to achieve distributed coordination using such basic primitives can provide insight into how such coordination arises in nature (see [19] for a recent survey of this approach).

A Key Question. As detailed below, existing work on the beeping model seeks to solve useful problems as *efficiently* as possible in this primitive network setting. In this paper, by contrast, we focus on solving useful problems as *simply* as possible (e.g., as measured by factors such as the size of the algorithm's state machine representation), asking the key question: is it possible to solve problems with both simple communication *and* simple algorithms? Notice, the answer is not *a priori* obvious. It might be the case, for example, that complexity is conserved, so that simplifying the communication model requires more complex algorithms. Or it might be the case that simple algorithms coordinating with beeps are sufficient for even complex tasks. Given the above motivations for studying beeps, answering this question is crucial, as it will help us probe the feasibility of useful networked systems—whether constructed by engineers or evolution—that are truly simple in both their communication methods and control logic.

Our Answers. Consider a collection of n nodes connected in a *single hop* topology (i.e., the network graph is a clique). We model the randomized process executing on each node as a probabilistic state machine. The two parameters describing the complexity of these algorithms are: (1) an upper bound on the number of states (indicated by integer $s \geq 1$); and (2) an upper bound on the precision of the probabilistic transitions (indicated by integer $q \geq 2$, where we allow probabilistic transitions to be labeled with probability 0, 1, or any value in the interval $[\frac{1}{q}, 1 - \frac{1}{q}]$). We ask how large these values must grow to solve specific problems. Our motivating premise is that smaller values imply simpler algorithms. (Notice, by considering both s and q, we can capture the trade-off between memory and probabilistic precision, a question of standalone interest; c.f., [16]).

We begin by considering *leader election*, a fundamental primitive in distributed systems. We prove that for a given error bound $\epsilon \in [0, 1/2]$ and probabilistic precision q, any algorithm that solves leader election with probability $1 - \epsilon$ requires $s = \Omega(\log_q (1/\epsilon))$ states. Given a lower bound $\widetilde{N}$ on the size of the network, this lower bound *reduces* to $s = \Omega(\log_q (1/\epsilon)/\widetilde{N})$ states. Thus, the more nodes in the network, the fewer states each node needs to solve the problem.

This lower bound leverages a reduction argument. We begin by defining and lower bounding a helper problem called $(1, k)$-*loneliness detection*, which requires an algorithm to differentiate between $n = 1$ and $n \geq k$ (but has no requirements for intermediate network sizes). This bound uses an indistinguishability argument regarding how nodes move through a specified state sequence. We then show how to transform a solution to leader election for size lower bound $\widetilde{N}$, to

solve $(1, \widetilde{N})$-loneliness detection—allowing our loneliness bound to carry over to leader election.

We then turn our attention to leader election upper bounds. We begin by proving our lower bound tight by showing, for every network size lower bound $\widetilde{N} \geq 1$, how to solve leader election with $s = O(\log_q (1/\epsilon)/\widetilde{N})$ states. The key idea behind this algorithm is to have nodes work together to implement a distributed timer. The more nodes in the network, the longer the distributed timer runs, and the longer the distributed timer runs, the higher the probability that we succeed at leader election. In this way, increasing the network size reduces the states required to hit a specific error bound. A shortcoming of this new algorithm, however, is that its expected running time is exponential in the network size. With this mind, we then describe two faster algorithms (their time is polylogarithmic in the relevant parameters) that require only the minimum precision of $q = 2$. The cost for their efficiency, however, is a loss of state optimality in some circumstances.

The first algorithm requires $s = O(\log (1/\epsilon))$ states and solves leader election with probability at least $1 - \epsilon$, for any network size n. It terminates in $O(\log (n + 1/\epsilon) \log (1/\epsilon))$ rounds, with probability at least $1 - \epsilon$. The key idea behind this algorithm is to test a potentially successful election by having the potential leader(s) broadcast with probability $1/2$ for $\log (1/\epsilon)$ rounds, looking for evidence of company. It is straightforward to see that a single such test fails with probability no more than $(1/2)^{\log (1/\epsilon)} = \epsilon$. The problem, however, is that as the network size grows, the number of such tests performed also increases, making it more likely that one fails. We neutralize this problem in our analysis by showing that the test failure probabilities fall away as a geometric series in the test count—bounding the cumulative error sum as the network grows.

The second algorithm requires only $s = O(1)$ states, and yet, for every network size n, it solves leader election with high probability in n when run in a network of that size. It requires only $O(\log^2 n)$ rounds, with high probability. The key idea driving this algorithm is to harness the large amount of total states in the network to implement a distributed timer that requires $\Theta(\log n)$ time to countdown to 0, when executed among n nodes. This duration is sufficient for the nodes to safely reduce contention down to a single leader.

After studying leader election, we turn our attention to more general classes of distributed decision problems. Leveraging our leader election algorithms as a key primitive, we show how to simulate a logspace decider Turing Machine (TM) with a constant number of unary inputs (all defined with respect to the network size n). Perhaps surprisingly, this algorithm requires only $O(\log (1/\epsilon))$ states to complete the simulation with probability $1 - \epsilon$, and only $O(1)$ states to achieve high probability in n. (Notice that this is not enough states for an individual node to store even a single pointer to the tape of the simulated machine.) Our simulation uses the same general strategy first highlighted in the study of population protocols [4]: simulate a counter machine with a constant number of counters that hold values from 0 to $O(n)$, and then apply a transformation due to Minsky [17] to simulate a logspace TM with this machine. Due to the

differences between the beeping and population protocol models, however, our counter machine simulation strategies are distinct from [4].

Implications. The results summarized above establish that the $\log(1/\epsilon)$ state threshold for leader election with bounded error is (in some sense) a fundamental simplicity threshold for solving useful problems with beeps. It is striking that if you have *slightly less* than this much memory, even the basic symmetry breaking task of leader election is impossible, but if you instead have *slightly more*, then suddenly you can solve large classes of complicated problems (i.e., everything solvable by a logspace TM). If you are satisfied with high probability solutions (which is often the case), then this treshhold reduces even more all the way down to $O(1)$. Given these results, we tentatively claim a positive answer to the key question posed above: *complexity is not destiny; you can solve hard problems simply in simple network models.*

Before proceeding into the technical details of our paper, we will first take the time to place both our model and our results in the context of the several different areas of relevant related work. Among other questions, we want to understand the relationship of our bounds to existing beep results, and how the beeping model compares and contrasts to similar settings.

Comparison to Existing Beep Results. The algorithmic study of beeping networks began with Degesys et al. [12], who introduced a continuous variant of the beeping model, inspired by the pulse-coupled oscillator framework. They studied biologically inspired strategies for solving a *desynchronization* problem. Follow-up work generalized the results to multihop networks [11, 18]. Cornejo and Kuhn [10] introduced the discrete (i.e., round-based) beeping model studied in this paper. They motivated this model by noting the continuous model in [11, 12, 18] was unrealistic and yielded trivial solutions to desynchronization, they then demonstrated how to solve desynchronization without these assumptions. Around this same time, Afek et al. [3] described a maximal independent set (MIS) algorithm in a strong version of the discrete beeping model. They argued that something like this algorithm might play a role in the proper distribution of sensory organ precursor cells in fruit fly nervous system development. Follow-up work [1, 2, 20] removed some of the stronger assumptions of [3] and improved the time complexity. In recent work, Förster et al. [14] considered deterministic leader election in a multihop beeping network.

To place this paper in this context of the existing work on the beeping model, it is important to note that the above-cited papers focus primarily on two goals: minimizing time complexity and minimizing information provided to nodes (e.g., network size, max degree, global round counter). They do not, however, place restrictions on the amount of states used by their algorithms. Accordingly, these existing results require either: the ability to store values as large as $\Theta(n)$ [1–3, 10, 20], or uniques ids [14] (which in our framework would require a machine with n different initial states, or equivalently, n different machines). In this paper, we prove that the algorithmic complexity threshold for solving many useful problems

is actually much lower: $O(1)$ states are sufficient for high probability results and $O(\log(1/\epsilon))$ states are sufficient for fixed error bound results.[3] We argue the direction pursued in this paper (how complex must algorithms become to solve useful problems with beeps) complements the direction pursued in existing papers (how fast can algorithms solve useful problems with beeps). Answers to both types of queries is necessary to continue to understand the important topic of coordination in constrained network environments.

Comparison to the Radio Network Model. The standard radio network model allows nodes to send large messages, but assumes concurrent transmissions lead to message loss (that may or may not be detectable). The key difference between the radio network model and the beeping model is that in the former you can recognize the case where exactly one node broadcast (e.g., because you receive a message). This capability, which the beeping model does not offer (a single beeper looks like multiple beepers), is powerful. It allows, for example, algorithms that can solve leader election with deterministic safety using only a constant amount of state, when run in network of size at least 2. If you assume receiver collision detection, these solutions require only polylogarithmic expected time.[4] These results violate our lower bounds for leader election with beeps (where the state size grows toward infinity as you drive the error bound toward 0)—indicating that the communication limitations in the beeping model matter from a computability perspective.

Comparison to the Stone Age Computing Model. It is also important to place our results in the context of other simplified communication/computation models. Consider, for example, the stone age distributed computing model introduced by Emek and Wattenhofer [13]. This model assumes state machines of constant size connected in a network and executing asynchronously. The machines communicate with a constant-size message alphabet and when transitioning can distinguish between having received 0, 1, or $\geq b$ messages of each type, for some constant parameter $b \geq 1$. For $b = 1$, this model is essentially an asynchronous version of the beeping model. To this end, nodes in our model can simulate nodes in the stone age model with $b = 1$ indefinitely using a constant number of states. For $b > 1$, however, any such simulation likely becomes impossible in the beeping model with a constant number of states. As noted in our discussion of the radio

[3] Notice, direct comparisons between many of these results is complicated by the variety of possible assumptions; e.g., synchronous versus asynchronous starts, multihop versus single hop, small versus large probability precision.

[4] For example: divide rounds into pairs of even and odd rounds. In even rounds, nodes broadcast a simple message with constant probability. If a node ever succeeds in broadcasting alone, all other nodes become *heralds*. They stop competing in even rounds and begin competing in odd rounds. When the winner (who is now the only non-herald in the network) eventually hears a message in an odd round, it elects itself leader. If we assume collision detection, we can reduce contention fast in the even rounds with basic knockout protocols; e.g., if you choose to listen and detect a collision you are knocked out and just wait to become a herald.

network model, the ability to safely recognize the case of exactly one message being sent provides extra power beyond what is achievable (without error) using only beeps.

Comparison to the Population Protocol Model. Another relevant simplified communication/computation setting is the well-studied population protocol model [4–9] . This model describes nodes as state machines of constant size that interact in a pairwise manner—transforming both states asymmetrically. In the basic version of the model, a fair scheduler chooses pairs to interact. A version in which the scheduler is randomized adds more power. There are similarities in the goals pursued by the beeping and population protocol models: both seek (among other things) to understand the limits of limited state in distributed computation. The core difference between the two settings is the role of the algorithm in communication scheduling. In the beeping model, algorithms must reduce contention and schedule communication on their own. In the population protocol model the scheduler ensures fair and reliable interactions. Imagine, for example, a continuous leader election problem where every node has a *leader* bit, and the problem requires in an infinite execution that: (1) every node sets *leader* to 1 an infinite number of times; and (2) there is never a time at which two nodes both have *leader* set to 1. This problem is trivial in the population protocol: simply pass a leader token around the network. In the beeping model, by contrast, it is impossible as it essentially requires nodes to solve leader election correctly an infinite number of times—a feat which would require an unachievable error bound of 0. It follows that in some respects these two models are studying the impact of limited state on different aspects of distributed computation.

2 Model

We model a collection of n probabilistic computational agents (i.e., "nodes") that are connected in a single hop network and communicate using a unary primitive; i.e., *beeps*. They execute in synchronous rounds in which each node can either beep or receive. Receiving nodes can distinguish between the following two cases: (1) no node beeped; (2) one or more nodes beeped. We characterize these agents by s (a bound on the number of states in their state machine), and q (a bound on the precision allowed in probabilistic transitions, with larger values enabling more accurate transition probabilities). In more detail:

Node Definition. We specify the algorithm executing on each node as a probabilistic state machine $M = (Q_r, Q_b, q_s, \delta_\perp, \delta_\top)$, where: Q_r and Q_b are two disjoint sets of states corresponding to receiving and beeping, respectively; q_s is the start state; and $\delta_\perp$ and $\delta_\top$ are the probabilistic transition functions[5] for the cases where the node detects silence and where the node beeps/detects a beep, respectively. Some problems have all nodes execute the same state machine, while others include multiple machine types, each corresponding to a different initial value.

[5] Transition functions map the current state to a distribution over states to enter next.

Executions. Executions proceed in synchronous rounds with all nodes in their machine's start state. At the beginning of round r, for a node u running a machine $(Q_r, Q_b, q_s, \delta_\perp, \delta_\top)$, if its current state q_u is in Q_b, then u emits a beep, otherwise it receives. If at least one node beeps in r, then *all* nodes either beep or detect a beep in this round. Therefore, each node u applies the transition function $\delta_\top$ to its current state q_u and selects its next state according to the distribution $\delta_\top(q_u)$. If no node beeps in r, then each node u applies the transition function $\delta_\perp$, selecting its next state from the distribution, $\delta_\perp(q_u)$.

Parameters. We parameterize state machines with two values. The first, indicated by $s \geq 1$, is an upper bound on the number of states allowed (i.e., $|Q_r| + |Q_b| \leq s$). The second, indicated by $q \geq 2$, bounds the precision of the probabilistic transitions allowed by the δ functions. In more detail, for a given q, the probabilities assigned to states by distributions in the range of δ must either be 0, 1, or in the interval, $[\frac{1}{q}, 1 - \frac{1}{q}]$. For the minimum value of $q = 2$, probabilistic transitions can occur only with probability 1/2. As q increases, smaller probabilities, as well as probabilities closer to 1, become possible. Finally, we parameterize an execution with n—the number of nodes in the network.

3 Leader Election

The first computational task we consider is leader election: eventually, one node designates itself leader. An algorithm state machine that solves leader election must include a final *leader state* q_ℓ that is terminal (once a node enters the state, it never leaves). Entering this state indicates a node has elected itself leader. For a given error bound $\epsilon \in [0, 1/2]$, we say an algorithm *solves* leader election with respect to ϵ if when executed in a network of any size, it satisfies the following two properties: (1) *liveness*: with probability 1, at least one node eventually enters the leader state; and (2) *safety*: with probability at least $1 - \epsilon$, there is never more than 1 node in the leader state. We also consider algorithms for leader election that are designed for networks of some minimal size $\widetilde{N}$. In this case, the algorithm must guarantee liveness in every execution, but it needs to guarantee safety only if the network size n is at least $\widetilde{N}$. Our goal is to develop algorithms that use a minimum number of states to solve leader election for a given error bound ϵ, probability precision q, and, when relevant, network size minimum $\widetilde{N}$.

Roadmap. In Section 3.1, we present a lower bound for leader election. In Section 3.2, we present a universal algorithm template, followed by three specific instantiations in Sections 3.3, 3.4, and 3.5. Due to space constraints, proofs are deferred to the full version of this extended abstract [15].

3.1 Leader Election Lower Bound

Here we analyze the number of states required to solve leader election given a fixed ϵ, q, and network size lower bound $\widetilde{N}$. Our main result establishes that the number of states, s, must be in $\Omega(\lceil \frac{\log_q(1/\epsilon)}{\widetilde{N}} \rceil)$.

To prove this result, we begin by defining and bounding a helper problem called $(1, k)$-*loneliness detection*, which requires an algorithm to safely distinguish between $n = 1$ and $n \geq k$. The bound leverages a probabilistic indistinguishability argument concerning a short execution of the state machine in both the $n = 1$ and $n = k$ cases. We then show that loneliness detection captures a core challenge of leader election by demonstrating how to transform a leader election algorithm that works for $n \geq \widetilde{N}$ into a solution to $(1, \widetilde{N})$-loneliness detection. The bound for the latter then carries over to leader election by reduction.

$(1, k)$-*Loneliness Detection.* The $(1, k)$-loneliness detection problem is defined for some integer $k > 1$ and error bound ϵ. It assumes all nodes run the same state machine with two special terminal final states that we label q_a (indicating "I am alone") and q_c (indicating "I am in a crowd"). The *liveness* property of this problem requires that with probability 1, every node eventually enters a final state. The *safety* property requires that with probability at least $1 - \epsilon$, the following holds: if $n = 1$, then the single node in the system eventually enters q_a; and if $n \geq k$ then all nodes eventually enter q_c. Crucial to this problem definition is that we do not place any restrictions on the final states nodes enter for the case where $1 < n < k$.

The following bound shows that it becomes easier to break symmetry, i.e., easier to solve loneliness detection, as the threshold for detecting a crowd grows. Put another way: a big crowd is easier to detect than a small crowd.

Lemma 1. *Fix some integer $k > 1$. Let $\mathcal{L}$ be an algorithm that solves $(1, k)$-loneliness detection with error bound ϵ and probability precision q using s states. It follows that $s = \Omega(\frac{\log_q(1/\epsilon)}{k})$.*

Reducing Loneliness Detection to Leader Election. We now leverage the above result on $(1, k)$-loneliness detection to prove a lower bound for leader election under the guarantee that the network size $n \geq \widetilde{N}$. The proof proceeds by reduction: we show how to transform such a leader election solution into a loneliness detection algorithm of similar state size.

Theorem 2. *Fix some network size lower bound $\widetilde{N} \geq 1$. Let $\mathcal{A}$ be an algorithm that solves leader election with error bound ϵ and probability precision q using s states in any network where $n \geq \widetilde{N}$. It follows that $s \in \Omega(\frac{\log_q(1/\epsilon)}{\widetilde{N}})$.*

3.2 The Universal Leader Election Algorithm

We now turn our attention to leader election upper bounds. The three results that follow adopt a template/subroutine approach. In more detail, Figure 3.1 describes what we call the *universal leader election* algorithm. This algorithm, in turn, makes calls to a "termination subroutine." Different versions of this subroutine can be plugged into the universal algorithm, yielding different guarantees. Notice, this universal algorithm is parameterized with probability precision q and error bound ϵ, which it uses to define the useful parameter $\hat{q} = \min\{q, (1/\epsilon)\}$.

This algorithm (as well as one of our termination subroutines) uses $1/\hat{q}$, not $1/q$, as its smallest transition probability (intuitively, there is little advantage in using a probability too much smaller than the bound ϵ).

The basic operation of the algorithm is simple. Every node is initially active. Until the termination subroutine determines that it is time to stop, nodes repeatedly execute the knockout loop (lines 7–25). In each iteration of the loop, each active node beeps with probability $1 - 1/\hat{q}$ and listens otherwise. If a node ever hears a beep, it is knocked out, setting $ko = true$ and $active = false$. In any silent iteration where no node beeps, they execute the termination subroutine to decide whether to stop. Once termination is reached, any node that remains active becomes the leader.

Algorithm 1 Universal Leader Election

```
1:  active ← 1
2:  ko ← 1
3:  q̂ ← min{q, (1/ε)}
4:  done ← [Term. Subroutine](active, ko)
5:  ko ← 0
6:
7:  while (not done) do
8:
        ▷ Returns 0 with prob 1/q̂, else 1
9:      participate ← random_bit(1/q̂)
10:     chan ← ⊤
11:
        ▷ Knock Out Logic
12:     if active ∧ participate then
13:         beep()
14:     else
15:         chan ← recv()
16:     end if
17:     if active ∧ not participate then
18:         if chan = ⊤ then
19:             active ← 0
20:             ko ← 1
21:         end if
22:     end if
23:
        ▷ Termination Detection Logic
24:     if chan = ⊥ then
25:         done ← [Term. Subroutine](active, ko)
26:         ko ← 0
27:     end if
28: end while
29:
    ▷ Become Leader if Still Active
30: if active then
31:     leader ← 1
32: else
33:     leader ← 0
34: end if
35: return(leader)
```

Termination Subroutines. The goal of the termination subroutine is to decide whether leader election has been solved: it returns *true* if there is a leader and *false* otherwise. The termination subroutine is called simultaneously by all the nodes in the system, and it is passed two parameters: the value of *active*, which indicates whether or not the calling node is still contending to become leader, and *ko*, which indicates whether or not it has been knocked out in the main loop since the last call to the subroutine. We fix $R = 4\log_{\hat{q}}(\max(n, 1/\epsilon))$: a parameter, which as we will later elaborate, captures a bound on the calls to the subroutine needed before likely termination. We consider the following properties of a termination detection routine, defined with respect to ϵ and R:

1. *Agreement*: Every node always returns the same value.
2. *Safety*: Over the first R invocations, the probability that it returns true in any invocation with more than 1 active node is at most $\epsilon/2$.
3. *Eventual Termination*: If it is called infinitely often with only one active node, then eventually (with probability 1), it returns true.

4. *Fast Termination*: If it is called with only one active node, and with at least one node where $ko = true$, then it returns true.

Universal Leader Election Analysis. We now observe that the universal leader election algorithm is correct when combined with a termination subroutine that satisfies the relevant properties from above. To do so, we first determine how many rounds it takes until there is only one active node, and hence one possible leader. We say that an iteration of the knockout loop (lines 7–25) is *silent* if no node beeps during it. (Notice that the termination routine is only executed in silent iterations of the knockout loop.) We first bound how long it takes to reduce the number of active nodes:

Lemma 3. *Given probability $\epsilon \leq 1/2$ and parameter $R = 4\log_{\hat{q}}(\max(n, 1/\epsilon))$: after R silent iterations of the knockout loop (lines 7–25), there remains exactly one active node, with probability at least $1 - \epsilon/2$.*

Let T be a termination subroutine that satisfies Agreement and Eventual Termination. In addition, assume that T satisfies safety in networks of size at least $\widetilde{N}$. We can now show that the universal leader election algorithm is correct with termination subroutine T:

Theorem 4. *If termination subroutine T uses s states and precision q, then the universal algorithm solves leader election with error ϵ, $s + O(1)$ states, and q precision (guaranteeing safety only in networks of size $n \geq \widetilde{N}$).*

While the preceding theorem can be used to show the feasibility of solving leader election, it does not bound the performance. For that, we rely on termination subroutines that ensure fast termination:

Theorem 5. *If termination subroutine T satisfies Fast Termination instead of Eventual Termination, and if it uses s states and q precision, and if it runs in time t, then the universal algorithm solves leader election with error ϵ with $s + O(1)$ states and q precision (guaranteeing safety only in networks of size $\geq \widetilde{N}$). Furthermore, it terminates in $O(t\log_{\hat{q}}(n + 1/\epsilon))$ rounds, with probability at least $1 - \epsilon$.*

3.3 Optimal Leader Election

Here we define a termination subroutine that, when combined with the universal leader election algorithm, matches our lower bound from Theorem 2. In more detail, fix an error bound ϵ and probability precision q. Fix some lower bound $\widetilde{N} \geq 1$ on the network size. We describe a termination detection subroutine that we call *StateOptimal*$(\widetilde{N})$ that requires $O(\lceil \frac{\log_q(1/\epsilon)}{\widetilde{N}} \rceil)$ states, and guarantees Agreement, Termination, and Safety in any network of size $n \geq \widetilde{N}$.

There are two important points relevant to this leader election strategy. First, for $\widetilde{N} = 1$, it provides a general solution that works in every size network. Second, the state requirements for this algorithm are asymptotically optimal

according to Theorem 2. As will be clear from its definition below, the cost of this optimality is inefficiency (its expected time increases exponentially with n). We will subsequently identify a pair of more efficient solutions that gain efficiency at the cost of some optimality under some conditions.

The StateOptimal($\widetilde{N}$) *Termination Detection Subroutine.* The *StateOptimal*($\widetilde{N}$) subroutine, unlike the other subroutines we will consider, ignores the *active* and *ko* parameters. Instead, it runs simple distributed coin flip logic among *all* nodes. In more detail, recall from the definition of the universal algorithm that $\hat{q} = \min\{q, (1/\epsilon)\}$. The subroutine consists of $\delta = \lceil \frac{c \log_{\hat{q}}(1/\epsilon)}{\widetilde{N}} \rceil$ rounds, defined for some constant $c \geq 1$ we will bound in the analysis. In each round, each node beeps with probability $1 - 1/\hat{q}$. At the end of the δ rounds, each node returns 1 if all δ rounds were silent, otherwise it returns 0.

Analysis. It is straightforward to determine that all nodes return the same value from this subroutine (i.e., if any node beeps or detects a beep, all nodes will return 0). It is also straightforward to verify that implementing this subroutine for a given δ requires $\Theta(\delta) = \Theta(\lceil \frac{\log_{\hat{q}}(1/\epsilon)}{\widetilde{N}} \rceil) = \Theta(\lceil \frac{\log_{q}(1/\epsilon)}{\widetilde{N}} \rceil)$ states (we can replace the $\hat{q}$ with q in the final step because once q gets beyond size $1/\epsilon$, the function stabilizes at 1). Eventual termination is also easy to verify, as every call to the subroutine has a probability strictly greater than 0 of terminating.

To show safety, we observe that the routine returns true only if all n nodes are silent for all δ rounds. The probability of this happening is exponentially small in (δn) and hence it is not hard to show that every R invocations, the probability that the subroutine returns true in any invocation with more than one active node is at most $\epsilon/2$.

Lemma 6 (Safety). *Over the first R invocations, the probability that the subroutine returns true in any invocation with more than 1 active node is at most $\epsilon/2$.*

Combined with Theorem 4, this yields the following conclusion:

Theorem 7. *For any network size lower bound $\widetilde{N}$, error parameter ϵ and precision q, the universal leader election algorithm combined with the StateOptimal($\widetilde{N}$) subroutine, solves leader election with respect to these parameters when run in a network of size $n \geq \widetilde{N}$, and requires only $s = \Theta(\lceil \frac{\log_{q}(1/\epsilon)}{\widetilde{N}} \rceil)$ states.*

3.4 Fast Leader Election with Sub-Optimal State

The leader election algorithm from Section 3.3 can solve the problem with the optimal number of states for any combination of system parameters. It achieves this feat, however, at the expense of time: it is straightforward to determine that this algorithm requires time exponential in the network size. Here we consider a termination subroutine that trades state optimality for a solution that is fast (polylogarthmic in $1/\epsilon$ rounds) and simple to define (it uses the minimal probabilistic precision of $q = 2$). Furthermore, its definition is independent of

the network size n, yet it still works for every possible n. For the purpose of this section, we assume that $q = \hat{q} = 2$. As we show below, this subroutine uses $\Theta(\log(1/\epsilon))$ states. This is suboptimal when high precision (i.e., larger q) is available, and when there is a lower bound $\widetilde{N}$ on the size of the network.

The Fixed Error *Termination Detection Subroutine.* This termination subroutine consists of a fixed schedule of $\lceil \log(2/\epsilon) \rceil + 2$ rounds. During the first round, any node that calls the subroutine with parameter *ko* equal to 1 beeps while all other nodes receive. If no node beeps, then the subroutine is aborted and all nodes return false.

Assume this does not occur, i.e., at least one node beeps in the first round. For each of the $\lceil \log(2/\epsilon) \rceil$ rounds that follow, every node with parameter $active = 1$, will flip a fair two-sided coin. If it comes up heads, it will beep, otherwise it will receive. Each node with $active = 1$ will start these rounds with a flag *solo* initialized to 1. If such a node ever detects a beep during a round that it receives, it will reset *solo* to 0 (as it just gained evidence that it is not alone).

The final round is used to determine if anyone detected a non-solo execution. To do so, every node with $active = 1$ and $solo = 0$ beeps. If no node beeps in this final round, then all nodes return true. Otherwise, all nodes return false.

Analysis. We proceed as before, observing that all nodes return the same value from this subroutine since all observe the same channel activity in the first and last rounds. It is also straightforward to verify that implementing this subroutine requires $O(\log(1/\epsilon))$ states to count the rounds and record *solo*. Fast termination follows directly from a case analysis of the algorithm.

Lemma 8 (Fast Termination). *If the* Fixed Error *subroutine is called with only 1 active node and with at least 1 node where ko = true, then it returns true.*

Safety requires a little more care, showing that the failure probabilities over R invocations can be bounded by $\epsilon/2$, since the error probability depends on the number of active nodes.

Lemma 9 (Safety). *Over the first R invocations of the subroutine, the probability that it returns true in any invocation with more than one active node is at most $\epsilon/2$.*

Combined with Theorem 5, these properties yield the following conclusion:

Theorem 10. *For error parameter ϵ, the universal leader election algorithm combined with the Fixed Error subroutine, solves leader election with respect to ϵ in every size network, using only $s = \Theta(\log(1/\epsilon))$ states and $q = 2$. With probability at least $1 - \epsilon$, it terminates in $O(\log(n + 1/\epsilon)\log(1/\epsilon))$ rounds.*

3.5 Fast Leader Election with $O(1)$ States and High Probability

The final termination detection subroutine we consider requires only a constant number of states, and when executed in a network of size n, for any $n > 1$, it

solves leader election with high probability in n. At first glance, this result may seem to violate the lower bound from Section 3.1, which notes that the state requirement grows with a $\log(1/\epsilon)$ factor as ϵ decreases. The question is why a constant number of states is sufficient here even though this term grows with n. The answer lies in the fact that ϵ is here a function of n, such that for any fixed n, it is true that $\widetilde{N} \geq n$, and therefore the $\widetilde{N}$ factor in the denominator of our lower bound swamps the growth of the $\log n$ factor in the numerator.

The Constant State *Termination Detection Subroutine.* The subroutine here is identical to the *Fixed Error* subroutine, except the length of subroutine is not fixed in advance (no node has enough states to count beyond a constant number of rounds—which is not enough for our purposes). Instead, we dynamically adapt the length of the subroutine to a sufficiently large function of n using a distributed counting strategy.

In more detail, during the first round, any node that called the subroutine with parameter *ko* equal to 1 beeps while all other nodes receive. If no node beeps, then subroutine is aborted and all nodes will return value false (as is true for *Fixed Error*). Assuming the subroutine has not aborted, the nodes then proceed as follows: We partition rounds into even and odd pairs. During the odd numbered rounds, we proceed as in *Fixed Error*: every node with parameter $active = 1$, flips a fair coin; if it comes up heads, it will beep, otherwise it will receive; each node with $active = 1$ will start these rounds with a flag *solo* initialized to 1; if such a node ever detects a beep during a round that it receives, it will reset *solo* to 0 (as it just gained evidence that it is not alone).

During the even rounds, the nodes run a repeated knockout protocol for $O(1)$ iterations, for some fixed constant bounded in the analysis. In more detail, each node (regardless of whether or not it has *active* equal to true) begins the subroutine with a flag $attack = 1$ and a counter $count = 0$. In each even round, each node with $attack = 1$ flips a fair coin and beeps if it comes up heads; otherwise it listens. Any node that listens in an even round and hears a beep sets $attack = 0$. If there is an even round in which no node beeps, then all nodes increment *count* and reset $attack = 1$. This continues until *count* grows larger than the fixed constant mentioned above, When this occurs, all nodes move to the final round, which is identical to the final round in *Fixed Error*. That is: every node with $active = 1$ and $solo = 0$ beeps. If no node beeps in this final round, then all nodes return true. Otherwise, all nodes return false.

Analysis. The Liveness and Fast Termination properties follow from the same arguments used in our analysis of *Fixed Error*. The main difficulty in analyzing this subroutine is proving Safety. To do so, we first bound how long the subroutine is likely to run on any given invocation:

Lemma 11. *For any constant c, there exists a $c' > c$ and a constant bound for count, such that the main body of the subroutine runs for at least $c\log(n)$ rounds but no more than $c' \log n$ rounds, with high probability.*

Lemma 12 (Safety). *Over the first R invocations of the subroutine, the probability that it returns true in any invocation with more than one active node is at most $1/n^c$, for a constant c we can grow with our constant bound on count.*

We can then show that the subroutine guarantees safety. Combined with the Theorem 5, these properties yields the following conclusion:

Theorem 13. *For any network size n, the universal leader election algorithm combined with the Constant State termination detection subroutine, solves leader election with high probability in n using $s = O(1)$ states and $q = 2$. Also with high probability in n, it terminates in $O(\log^2 n)$ rounds.*

4 Solving General Distributed Decision Problems

In this section, we use a combination of our fast leader election algorithms as a key primitive in constructing an algorithm that simulates a logspace (in n) decider Turing Machine (TM) with a constant number of unary input tapes (of size $O(n)$ each). The simulation has error probability at most ϵ, requires only the minimum probabilistic precision ($q = 2$), and uses $s = O(\log(1/\epsilon))$ states. If high probability in n is sufficient, then the state size can be reduced to $s = O(1)$. In other words, once you have enough states to solve leader election, you can also solve a large class of expressive problems. Formally:

Theorem 14. *For any problem solvable by a logspace TM with a constant number of unary input tapes, there exist constants $c, d \geq 1$, such that for any error probability $\epsilon \in [0, 1/2]$ and network size $n \geq 1$, we can solve the problem in the beeping model in a network of size n with probability at least $1 - \epsilon$ using $s = c \log(1/\epsilon)$ states, precision $q = 2$, and an expected running time of $O(n^d \log^2(n + 1/\epsilon))$ rounds. For high probability correctness, $s = O(1)$ states are sufficient.*

Our strategy follows the outline originally identified in [4], where it was used to simulate a TM using a population protocol in the randomized interaction model. We first simulate a simple counter machine with a constant number of counters that can take values of size $O(n)$. We then apply a classical computability result due to Minsky [17] which shows how to simulate a logspace TM (with unary input tapes) using a counter machine of this type. The counter machine simulation in the beeping model, combined with Minsky's TM simulation, yields a TM simulation in the beeping model. See the full version of this extended abstract [15] for the details of our simulation, its analysis, and a discussion of its implications.

References

[1] Afek, Y., Alon, N., Bar-Joseph, Z., Cornejo, A., Haeupler, B., Kuhn, F.: Beeping a maximal independent set. In: Proceedings of the Symposium on Distributed Computing (DISC) (2011)

[2] Afek, Y., Alon, N., Bar-Joseph, Z., Cornejo, A., Haeupler, B., Kuhn, F.: Beeping a maximal independent set. Distributed Computing 26(4), 195–208 (2013)
[3] Afek, Y., Alon, N., Barad, O., Hornstein, E., Barkai, N., Bar-Joseph, Z.: A biological solution to a fundamental distributed computing problem. Science 331(6014), 183–185 (2011)
[4] Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006)
[5] Angluin, D., Aspnes, J., Eisenstat, D.: Stably computable predicates are semilinear. In: Proceedings of the Symposium on Principles of Distributed Computing (PODC). pp. 292–299 (2006)
[6] Angluin, D., Aspnes, J., Eisenstat, D.: Fast computation by population protocols with a leader. Distributed Computing 21(3), 183–199 (2008)
[7] Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. Distributed Computing 21(2), 87–102 (2008)
[8] Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)
[9] Chatzigiannakis, I., Spirakis, P.G.: The dynamics of probabilistic population protocols. In: Proceedings of the Symposium on Distributed Computing (DISC) (2008)
[10] Cornejo, A., Kuhn, F.: Deploying wireless networks with beeps. In: Proceedings of the Symposium on Distributed Computing (DISC) (2010)
[11] Degesys, J., Nagpal, R.: Towards desynchronization of multi-hop topologies. In: Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems, 2008 (SASO) (2008)
[12] Degesys, J., Rose, I., Patel, A., Nagpal, R.: Desync: self-organizing desynchronization and tdma on wireless sensor networks. In: Proceedings of the International Conference on Information Processing in Sensor Networks (2007)
[13] Emek, Y., Wattenhofer, R.: Stone age distributed computing. In: Proceedings of the Symposium on Principles of Distributed Computing (PODC) (2013)
[14] Förster, K., Seidel, J., Wattenhofer, R.: Deterministic leader election in multi-hop beeping networks - (extended abstract). In: Proceedings of the Symposium on Distributed Computing (DISC) (2014)
[15] Gilbert, S., Newport, C.: The computational power of beeps. Full version available online at: http://people.cs.georgetown.edu/∼cnewport/pubs/Beeps-Full.pdf. Also available on arXiv.
[16] Lenzen, C., Lynch, N., Newport, C., Radeva, T.: Trade-offs between selection complexity and performance when searching the plane without communication. In: Proceedings of the Symposium on Principles of Distributed Computing (PODC) (2014)
[17] Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall (1967)
[18] Motskin, A., Roughgarden, T., Skraba, P., Guibas, L.J.: Lightweight coloring and desynchronization for networks. In: Proceedings of the of the Conference on Computer Communication (INFOCOM) (2009)
[19] Navlakha, S., Bar-Joseph, Z.: Distributed information processing in biological and computational systems. Communications of the ACM 58(1), 94–102 (2014)
[20] Scott, A., Jeavons, P., Xu, L.: Feedback from nature: an optimal distributed algorithm for maximal independent set selection. In: Proceedings of the Symposium on Principles of Distributed Computing (PODC) (2013)