Desynchronization: The Theory of Self-Organizing Algorithms for Round-Robin Scheduling

Ankit Patel, Julius Degesys, Radhika Nagpal
Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02139
{abpatel,degesys,rad}@eecs.harvard.edu

Abstract

The study of synchronization has received much attention in a variety of applications, ranging from coordinating sensors in wireless networks to models of fireflies flashing in unison in biology. The inverse problem of desynchronization*, however, has received little notice. Desynchronization is a powerful primitive: given a set of identical oscillators, applying a desynchronization primitive spreads them throughout the period, resulting in a round-robin schedule. This can be useful in several applications: medium access control in wireless sensor networks, designing fast analog-to-digital converters, and achieving high-throughput traffic intersections. Here we present two biologically-inspired algorithms for achieving desynchronization:* DESYNC *and* INVERSE-MS*. Both algorithms are simple and decentralized and are able to self-adjust to the addition and removal of agents. Furthermore, neither requires a global clock or explicit fault detection. We prove convergence, compute bounds for the running time, and assess the various tradeoffs. To our knowledge, the theory of self-organizing desynchronization algorithms is presented here for the first time.*

1 Introduction

Fireflies have fascinated scientists and mathematicians alike with the phenomenon of *mutual synchronization*. In Southeast Asia, thousands of fireflies synchronize their flashing despite being spread over many miles. Mathematical models of this peculiar phenomenon remained elusive until the seminal works of Peskin [7] and Mirollo and Strogatz [6]. They were able to define a general class of local "flashing" algorithms that lead to global synchrony. Due to their simplicity and implicit fault-tolerance, these models have been easily extended to applied domains that require robust and accurate synchronization, such as wireless sensor networks [4, 5, 10].

Despite the popularity of synchronization, the "inverse" problem of spreading identical oscillators into a round-robin schedule has received surprisingly little attention. Here we define a new primitive, *desynchronization*, that addresses this inverse problem. Our motivation stems from biology: cells, acting as oscillators, control animal gaits and regulate heart valves through desynchronization. Many problems in multi-agent systems also require desynchronization. For example, in a wireless sensor network, nodes can avoid collisions and message loss by desynchronizing their transmission times. Similarly, sensor nodes can desynchronize their sampling times to distribute the energy cost while still providing efficient coverage.

In this paper we present two self-organizing, self-maintaining desynchronization algorithms: DESYNC and INVERSE-MS. DESYNC is inspired by particle diffusion and uses a simple local "spreading" rule to establish a global desynchronization. In contrast, INVERSE-MS is adapted from classic models of firefly synchronization [6]. INVERSE-MS is very fast but also brittle: it requires specialized assumptions that strongly limit its applicability. On the other hand, DESYNC is slower but more robust to message loss, message delay, and node birth and death.

Both algorithms require no centralized control, no global clock, and only a small, constant amount of state per agent. For both algorithms, we prove convergence, and analytically compute and compare convergence rates as a function of the number of nodes. Both algorithms are self-maintaining: when the number of nodes changes, they automatically adapt to reestablish a perfect round-robin schedule without any explicit fault detection. The simplicity of the self-organizing desynchronization algorithms allows them to work in a wide variety of settings. We describe several potential applications and explore the constraints involved. We also briefly discuss an implementation of DESYNC on TinyOS Telos motes and its application to TDMA, the full details of which are available elsewhere [1].

The main contribution of this paper is the design and analysis of two new self-organizing desynchronization algorithms. To the best of our knowledge, the detailed the-

ory of self-organizing desynchronization algorithms is presented here for the first time.

The paper is organized as follows. Section 2 motivates the need for a desynchronization primitive, examines potential applications, and discusses previous work. Section 3 defines the general desynchronization framework and sections 4 and 5 present the DESYNC and INVERSE-MS algorithms. Section 6 compares the performance of the algorithms with respect to several metrics, and helps guide the application-dependent choice of an algorithm. A real implementation of the DESYNC algorithm in a sensor network testbed is presented in Section 7. Conclusions and future work are in Section 8.

2 Motivation and Background

Resource scheduling is defined as the ability to organize a schedule amongst competing agents that provides fair, responsive, and exclusive access to a shared resource. The resource scheduling problem rears its head in many settings, some of which are discussed below. The simplest solution is to use a *round-robin schedule*: it guarantees every contending agent access to the resource before any other agent gets access again. If each agent's slot (uncontested access to the resource for a period of time) is equally long, then a round-robin schedule provides an equitable solution.

Round-robin scheduling requires agents to agree on (1) the access order and (2) the time of the accesses. Most systems today come to this agreement by allowing all agents access to a global clock and computing a static access order ahead of time. But a centralized mechanism can be costly and error-prone in a dynamic distributed setting. We discuss some real applications where a decentralized mechanism is desirable.

TDMA in Wireless Networks. In a wireless network, neighboring nodes share a resource: the communication channel. Two nodes transmitting at the same time causes a collision and leads to message loss. Time Division Multiple Access (TDMA) is a protocol where nodes agree upon a round-robin schedule and take turns sending data. This approach provides collision-free message transmission and fully utilizes the bandwidth under high-load conditions. However, traditional implementations suffer from two problems: (1) they require that nodes have access to a common notion of global time, and (2) the schedule is fixed and therefore bandwidth is wasted if only a fraction of the nodes need to transmit. Access to a global clock and renegotiating schedules to recover bandwidth are both costly, especially when message loads are unpredictable. As a result, most systems use simpler contention-based protocols (e.g. 802.11) despite their poor performance under high load [9].

Analog-to-Digital Converters. In signal processing, an analog-to-digital converter (ADC) creates a digital representation of an analog signal by sampling the signal at a uniform rate. One method to design a fast ADC is to take n identical ADC units and interleave their sampling, thus creating a single sampler that is n times faster. Thus, fast sampling is achieved by desynchronizing the ADC units. Typical implementations have a single global clock that produces the base frequency and each ADC unit can be staggered by adding increasing numbers of delay elements. However such a circuit must be extremely precisely designed to achieve uniform sampling - any errors in design time can not be corrected in real-time [2].

Traffic Intersections. Here the agents are cars and the resource is the intersection. Signal lights act as locks on the intersection to prevent collisions. An autonomous intersection management system that allows the cars to desynchronize their use of the intersection could eliminate the need for traffic lights. Thus, desynchronization could allow for more efficient interleaving of the traffic streams, thereby increasing throughput and reducing the wait time at intersections [3].

At present, each of these applications invokes a global leader, in the form of a clock or a lock. But a centralized mechanism has many disadvantages: high implementation complexity, lack of robustness, and bottlenecks under high loads. Thus, the potential benefits of a decentralized desynchronization algorithm are significant.

3 Desynchronization Framework

In this section we present a general framework for exploring decentralized algorithms for desynchronization. The framework models nodes as *pulse-coupled oscillators*, an idea introduced by Peskin [7] and later developed by Mirollo and Strogatz in the context of cardiac/firefly synchronization [6].

Suppose there are n nodes that can communicate with each other. Each node is an oscillator with the same fundamental frequency ω and period $T = 1/\omega$. Let the nodes be labeled in clockwise order (e.g. Figure 1(a)). Throughout the paper, node labels are always 1-based and taken modulo n. Let $\phi_i \in \mathbb{S} = [0, 1]$ denote the phase of node i, where $1 \leq i \leq n$. For example, if $\phi_2 = 0.75$, then node 2 is 75% of the way through its cycle. The *phase neighbors* of node i are the nodes $i \pm 1$. The *system state* is a column vector $\vec{\phi} = [\phi_i] \in \mathbb{S}^n$, representing the phases of all n oscillators.

The *desynchronized state*, or *desynchrony*, is any system state $\vec{\phi}^*$ in which all n oscillators are evenly spread out in phase, oscillating at the same frequency, as shown in Fig-

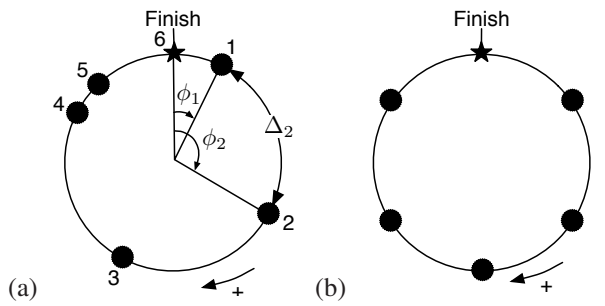

Figure 1. Desynchronization Framework. (a) Nodes move clockwise on the circle at frequency $1/T$, where T is the period. Nodes fire when they reach the finish line (red star). ϕ_i denotes the phase of node i and Δ_i represents the phase difference between consecutive oscillators. (b) The state of perfect desynchrony, where all nodes are evenly spaced in phase, shown here for $n = 6$ nodes.

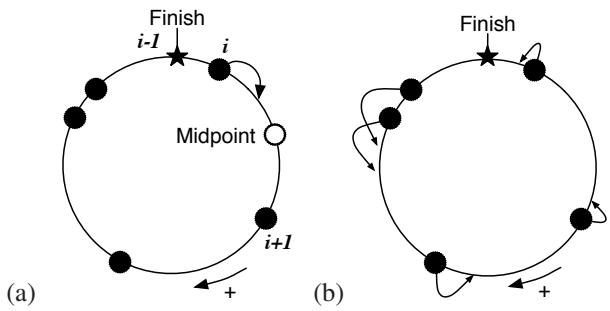

Figure 2. Desynchronization Algorithms. (a) DESYNC. After hearing a back neighbor $i-1$ fire, node i jumps towards the midpoint of its phase neighbors (open circle). (b) INVERSE-MS. After hearing any node fire, all nodes jump backwards. Nodes closer to reaching the finish line jump further back, while nodes earlier in their cycles take smaller jumps.

ure 1(b). Formally, we define desynchrony in terms of the phase differences between neighboring oscillators by introducing a new set of *delta-phase* variables: $\Delta_i \equiv \phi_i - \phi_{i-1} \pmod 1$, as shown in Figure 1(a). Let d be the mapping from $\vec{\phi}$ to $\vec{\Delta}$; that is, $\vec{\Delta} = d(\vec{\phi})$. In these variables, the desynchronized state is simply expressed as $\Delta_i^* = \frac{1}{n}$ for all i. Or, in vector form,

$$\vec{\Delta}^* = \frac{\mathbf{1}_n}{n} = \begin{pmatrix} \frac{1}{n} & \frac{1}{n} & \dots & \frac{1}{n} \end{pmatrix}^T. \tag{1}$$

We can imagine nodes as beads moving clockwise on a ring with constant velocity, as shown in Figure 1(a). When a node crosses the finish line ($\phi = 1$), it *fires*, and resets ($\phi = 0$). Firing is a node's way of communicating its phase to other nodes. Upon hearing the firings of other nodes, a node may respond by "jumping" forwards or backwards in phase. From node i's perspective, the general algorithm reads

$$\phi_i' = f_i(\Phi; \alpha). \tag{2}$$

Or in vector form,

$$\vec{\phi}' = f(\Phi; \alpha), \tag{3}$$

where $f : \mathbb{S}^n \to \mathbb{S}^n$ is the *jump function*, ϕ_i' is the phase of node i after the jump, and Φ is the *total phase history* of all oscillators. The jump function f proceeds as follows: (1) examine Φ to determine which node i is next to fire, (2) move all nodes forward in phase until node i reaches the finish line, and (3) execute jumps f_j for all nodes $j \neq i$. In summary, f represents exactly one move-fire-jump event, with node i firing and all nodes $j \neq i$ jumping. Applying f a total of n times would execute exactly one round of n firings.

Equation (3) is the most general form of a desynchronization algorithm: each node jumps as a function of the histories of some (or all) of the nodes. The parameter $\alpha \in \mathbb{R}$ is a *jump size parameter* that controls how far a node jumps. When $\alpha = 0$ there is no jumping, and when $\alpha > 0$, a node jumps a nonzero amount. Since α is fixed ahead of time, we may sometimes suppress writing it for convenience.

Our main question is: *How should an individual node jump so that the whole system of identical nodes (all executing the same program) is driven to desynchrony?* Or, mathematically speaking: *What constraints on f are sufficient to guarantee that a system of nodes can achieve desynchrony?* In the following sections, we will introduce two algorithms, DESYNC and INVERSE-MS, that help to answer these questions.

4 DESYNC Algorithms

The DESYNC algorithm proceeds as follows. When node i reaches the end of its cycle ($\phi_i = 1$), it fires, thus notifying all other nodes that it is beginning a new cycle. After node i fires, it waits for node $i-1$ to fire. When node $i-1$ fires ($\phi_{i-1} = 1$), node i jumps to a new phase ϕ_i' according to jump function f, as shown in Figure 2(a).

All DESYNC algorithms have the following three characteristics: (1) nodes only use firing information from their two phase neighbors, (2) nodes jump only once per period, and (3) nodes stop jumping when the system reaches desynchrony.

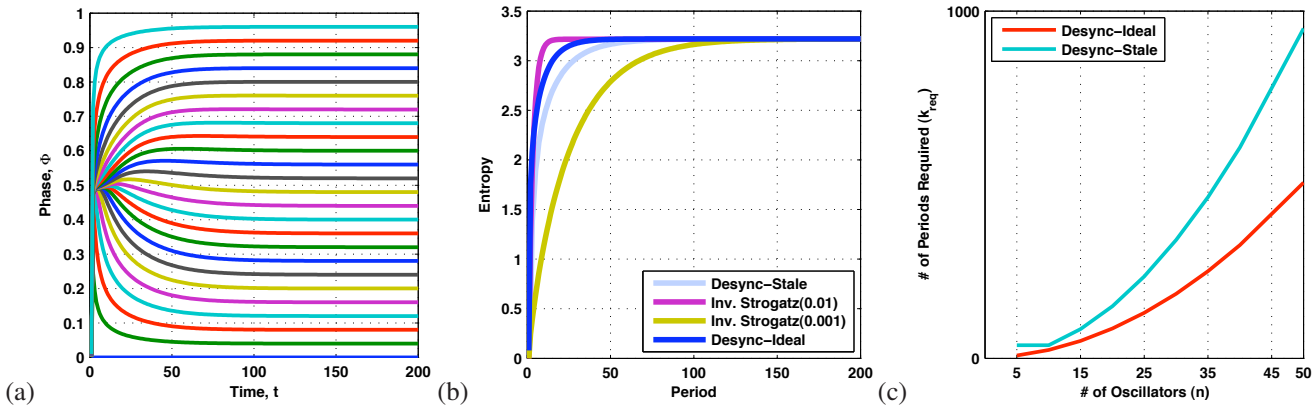

Figure 3. Phase vs. Time and convergence rates for several DESYNC variants ($n = 25$ nodes, 200 rounds, near-synchronized initial condition randomly chosen within $T/100$ fraction of phase, $\alpha = 0.95$). (a) Phase vs. Time (periods) for DESYNC-IDEAL . (b) Convergence rates as measured by entropy of phase configurations. DESYNC-IDEAL is the fastest DESYNC variant, followed by DESYNC-STALE. Stale information slows down convergence, as expected. INVERSE-MS is very fast for $\alpha = 0.01$ since all n nodes can jump at once, at the cost of steady state period lengthening. For $\alpha = 0.001$, DESYNC variants are much faster than INVERSE-MS. (c) Number of periods required to achieve ϵ-desynchrony for $\epsilon = 10^{-3}$. Both DESYNC-IDEAL and DESYNC-STALE require $O(n^2)$ periods.

4.1 Preliminaries

First, we note that the state of desynchrony is invariant to an overall shift in phase: for all constants C, $d(\vec{\phi}^* + C) = d(\vec{\phi}^*) = \vec{\Delta}^*$. Thus, it is natural to follow a system's desynchronization dynamics in the delta-phase variables, $\vec{\Delta} = d(\vec{\phi})$. It is important to note that a vector $\vec{\Delta}$ specifies only the relative distances between nodes and gives no information about the absolute phase of any node. It will be useful to interpret $\vec{\Delta}$ as a probability distribution. This is possible because $\Delta_i \geq 0$ for all i and $\sum_i \Delta_i = 1$. Thus, $\vec{\Delta} \in \mathcal{P}_n$, where $\mathcal{P}_n$ is the set of all probability distributions on n states.

In the next subsection, we will introduce the DESYNC algorithms. The inspiration for the DESYNC algorithms comes from the natural process of chemical diffusion. Given an initially non-uniform concentration gradient, the diffusion process smooths out sharp peaks and local disparities, until the gradient becomes completely flat. For our purposes, the vector $\vec{\Delta}$ plays the role of the concentration gradient and the desynchronization algorithm plays the role of diffusion. Eventually, the local smoothing of $\vec{\Delta}$ yields the uniform distribution, corresponding to perfect desynchrony.

Given this intuition, we will now introduce the DESYNC algorithms. An important design consideration is how quickly phase information is propagated to other nodes. Firings serve as messages to other nodes and if they are not received in time, a node may make a jump with "stale" information. Hence, we have designed two variants of the DESYNC algorithm: DESYNC-IDEAL and DESYNC-STALE. DESYNC-IDEAL works under idealized conditions where phase information is propagated instantaneously. On the other hand, DESYNC-STALE can handle stale information about the phases of other oscillators. For each variant, we present the algorithm, prove convergence, and compute rates of convergence.

4.2 DESYNC-IDEAL

The first DESYNC algorithm assumes that nodes can always access the exact positions of their phase neighbors. In this algorithm, node i fires and then waits for node $i-1$ to fire (when $\phi_{i-1} = 0$). At this point, node i takes a "global snapshot" of the system, acquiring the *exact* phases of nodes $i \pm 1$. Node i then computes the midpoint of its phase neighbors, $\text{mid}(\phi_{i-1}, \phi_{i+1}) = (\phi_{i-1} + \phi_{i+1})/2 = \phi_{i+1}/2$ (since $\phi_{i-1} = 0$). It then jumps a fraction α of the way towards that midpoint, as shown in Figure 2(a). All together, the algorithm, called DESYNC-IDEAL, reads

$$\begin{aligned} \phi_i' &= f_{\text{IDEAL},i}(\vec{\phi}; \alpha) \\ &= (1-\alpha) \cdot \phi_i + \alpha \cdot \text{mid}(\phi_{i-1}, \phi_{i+1}) \end{aligned} \quad (4)$$

Note that here a firing's only purpose is to trigger a node to take a snapshot and make a jump. We will make more substantial use of firings in the next section when we define a desynchronization algorithm that works in more realistic settings. But for now, we prove that the DESYNC-IDEAL algorithm achieves desynchrony.

Theorem 1 (DESYNC-IDEAL Convergence) *For all initial conditions and $\alpha \in (0, 2)$, n oscillators whose dynamics are governed by* DESYNC-IDEAL *will be driven to desynchrony.*

Proof To prove convergence, we will cast the DESYNC-IDEAL algorithm as a linear dynamical system and show that desynchrony is the unique globally stable fixed point of the dynamics. We compute the dynamics in the delta-phase variables Δ_i because this leads to simpler proofs. By equation (4), we know that node $i-1$'s firing equalizes the phase distances between node i and its neighbors $i \pm 1$ so that

$$\Delta_i' = (1-\alpha)\Delta_i + \alpha\frac{\Delta_i + \Delta_{i+1}}{2} \tag{5}$$

In matrix form, this can be written

$$\vec{\Delta}' = A_{\text{IDEAL}}\vec{\Delta} = PJ_{\text{IDEAL}}\vec{\Delta}, \tag{6}$$

where P is the permutation matrix that accounts for the relabeling of nodes (i.e. node 1 becomes node 2, node 2 becomes node 3, and so on). The matrix J_{IDEAL} represents the jump in equation (5):

$$J_{\text{IDEAL}} = \begin{pmatrix} 1-\frac{\alpha}{2} & \frac{\alpha}{2} & & & \\ \frac{\alpha}{2} & 1-\frac{\alpha}{2} & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} \tag{7}$$

It is easy to see that $A_{\text{IDEAL}}\vec{\Delta}^* = \vec{\Delta}^*$ and so the state of desynchrony is a fixed point, or equivalently, $\vec{\Delta}^*$ is an eigenvector of A_{IDEAL} with trivial eigenvalue $\lambda_1 = 1$. To prove global convergence to this fixed point, we must show that the non-trivial eigenvalues of A_{IDEAL} lie within the unit circle in the complex plane. It is straightforward to show that the characteristic polynomial of A_{IDEAL} is

$$\rho_n(\lambda;\alpha) = \lambda^n - \frac{\alpha}{2}\lambda^{n-1} - \frac{\alpha}{2}\lambda - (1-\alpha). \tag{8}$$

Since ρ_n is a *stable polynomial* (see Appendix Definition A), we are guaranteed by Theorem 5 in the Appendix that all of its nontrivial eigenvalues lie inside the unit circle. Thus, DESYNC-IDEAL converges to desynchrony, irrespective of initial conditions. □

4.3 DESYNC-STALE

The DESYNC-IDEAL algorithm is an idealization because in most realistic settings a node cannot take global snapshots to get information about its phase neighbors. This is especially true in settings where message delays or unreliable links make it difficult to acquire the most recent updates on neighboring phases. Instead, nodes must use their firings to communicate phase information. Since nodes may jump in response to other nodes' firings, information from the last firings may not always be accurate. However, we show that if a node's jump size α is restricted, nodes can still use this "stale" information to desynchronize.

Here we introduce DESYNC-STALE, a DESYNC variant that relaxes the assumption of perfect phase information. We proceed by slightly modifying the DESYNC-IDEAL algorithm in (4) to use a midpoint estimate rather than the actual midpoint.

In general, node i uses its memory to estimate the midpoint between its phase neighbors $i \pm 1$, before making a jump towards that midpoint. From node i's perspective, the DESYNC-STALE algorithm reads

$$\begin{aligned} \phi_i' &= f_{\text{STALE},i}(\Phi;\alpha) \\ &= (1-\alpha)\cdot\phi_i + \alpha\cdot\text{mid}(\phi_{i-1}, \tilde{\phi}_{i+1}), \end{aligned} \tag{9}$$

where $\tilde{\phi}_{i+1}$ is a stale estimate of the true ϕ_{i+1}. Remember that node i executes a jump according to (9) only after it hears node $i-1$ fire. This means that node i always has accurate information about ϕ_{i-1}. But, it may have stale information about ϕ_{i+1}, since node $i+1$ may have jumped when node i fired.

To illustrate DESYNC-STALE and clarify the origin of "stale" information, we begin with a simple detailed example. Consider a network of three nodes—A, B, and C—initialized with phases $\phi_A = 0.6$, $\phi_B = 0.7$, and $\phi_C = 0.9$. Each node begins with no knowledge of the other nodes' phases. Node C fires first. Node B hears this firing and observes that its own phase $\phi_B = 0.8$. It uses this information to record that node C is 0.2 ahead of it. Node B then fires, and nodes A and C record B's relative position accordingly. Finally, when node A fires ($\phi_A = 0$), node B knows that it is 0.1 ahead of node A and 0.2 behind node C. Node B uses these two pieces of information to make its jump towards the midpoint, $\text{mid}(\phi_A, \phi_C) = (0 + 0.3)/2 = 0.15$.

The key point is that *Nodes A and C are not aware of this jump*, since node B has no means of communicating its new location until its next firing. Thus, nodes A and C are left with "stale" knowledge, $\tilde{\phi}_B$, of node B's true position, ϕ_B. Later on, when it is node A's turn to jump, it can only use this "stale" estimate of node B's phase to compute a midpoint. This is in contrast to DESYNC-IDEAL, where node A could simply see the exact phases of both its neighbors. Despite this "stale" information, we shall now prove that DESYNC-STALE will still achieve desynchrony.

Theorem 2 (DESYNC-STALE Convergence) *For all initial conditions and $\alpha \in (0, 1)$, n nodes whose dynamics are governed by* DESYNC-STALE *will be driven to desynchrony.*

Proof The strategy is analogous to that of Theorem 1, except that we must augment the state of our system to

keep track of the estimate $M = \tilde{\Delta}_{i+1}$. Thus we redefine the state of the system as the length $n+1$ vector $\vec{x} = (\Delta_1, \ldots, \Delta_n, M)^T$. We compute the dynamics in the delta-phase variables $\Delta'_i = \phi'_{i-1} - \phi'_i$ by taking the discrete difference of (9). In matrix form, the dynamical system is

$$\vec{x}' = A_{\text{STALE}}\vec{x} = P_{\text{STALE}} J_{\text{STALE}} \vec{x}, \tag{10}$$

where P_{STALE} is the permutation matrix that cyclically relabels nodes 1 through n and leaves M alone. The matrix J_{STALE} represents the jump in equation (9):

$$J_{\text{STALE}} = \begin{pmatrix} 1-\frac{\alpha}{2} & & \cdots & & \frac{\alpha}{2} \\ \frac{\alpha}{2} & 1 & \cdots & & -\frac{\alpha}{2} \\ & & \ddots & & \\ & & & 1 & \\ 1 & & & & 0 \end{pmatrix} \tag{11}$$

Here all blank entries are zero. The structure of this matrix represents both the jump and the storage of $Delta_1$ into the memory M for use in the next jump. It is easy to see that the length $n+1$ vector $\vec{x}^* = (\Delta_1^*, \ldots, \Delta_n^*, M^*) = (1/n, \ldots, 1/n)$ is a fixed point of A_{STALE} and we need only show global convergence. Computing the characteristic polynomial of A_{STALE} yields

$$\sigma_n(\lambda; \alpha) = \lambda^{n+1} - \frac{\alpha}{2}\lambda^2 - (1-\alpha)\lambda - \frac{\alpha}{2} \tag{12}$$

This is a stable polynomial and so by Theorem 5 in the Appendix we have that all the non-trivial eigenvalues lie inside the unit circle. Hence, DESYNC-STALE converges to desynchrony, irrespective of initial conditions. □

4.4 Convergence Rates to Desynchrony

For real applications, how quickly a desynchronization algorithm converges is important. Both DESYNC-IDEAL and DESYNC-STALE are linear algorithms and so can be represented as multiplications by some matrix A. Since the largest eigenvalue of A is always 1, it is the second largest eigenvalue, $\lambda_*(A)$, that tells us how quickly the system is driven to desynchrony [8]. From λ_*, we can compute how many rounds the algorithm requires to become desynchronized, as a function of n and α.

Definition Let $|\vec{\Delta}|$ denote the *desynchronization accuracy* of state $\vec{\Delta} \in \mathcal{P}_n$, defined as the sum of the absolute deviations from perfect desynchrony. Mathematically, $|\vec{\Delta}| = |\vec{\Delta} - \vec{\Delta}^*|_{L^1} = \sum_{i=1}^n |\Delta_i - \frac{1}{n}|$. We say that a system of nodes is *ϵ-desynchronized* if $|\vec{\Delta}| < \epsilon$.

Using techniques from linear algebra and Markov Chain theory, we can prove the convergence rates of both algorithms.

Theorem 3 (DESYNC Rates of Convergence) *A system of n nodes whose dynamics are governed by* DESYNC-IDEAL *will achieve ϵ-desynchrony in $O(n^2 \ln(\frac{1}{\epsilon})/\alpha)$ rounds for $n > 2$ and $\alpha \in (0,2)$. Furthermore, we conjecture that a system of n nodes whose dynamics are governed by* DESYNC-STALE *will achieve ϵ-desynchrony in $O(n^2 \ln(\frac{1}{\epsilon})/\alpha)$ rounds for $n > 2$ and $\alpha \in (0,2)$.*

Proof See Appendix. □

Figure 3 shows results from simulations, confirming Theorems 1-3. In each run, all nodes start out with randomized phases, tightly clustered in a $T/100$ size region, simulating a near-synchronous (worst-case) initial condition. Figure 3(a) shows that, despite both DESYNC algorithms desynchronizing in $O(n^2)$ rounds, DESYNC-STALE converges slower than DESYNC-IDEAL by a constant factor. Thus, for practical purposes, DESYNC-IDEAL can be significantly faster. This is expected since DESYNC-STALE uses stale information, which can lead to overshoot and oscillatory behavior, thus slowing down convergence. Figure 3(b) shows that the system entropy converges to $\mathcal{H}_{\text{max}}$ at an exponential rate. Figure 3(c) shows the number of periods, $k_{\text{req}}(n)$, required to reach accuracy $\epsilon = 10^{-3}$ as a function of n, for $\alpha = 0.95$. The resulting convergence times confirm Theorem 3.

5 Inverse-MS Algorithm

There exists another class of desynchronization algorithms that are fundamentally different from the DESYNC family. In this framework — inspired by the firefly synchronization work of Mirollo and Strogatz [6] — when a node fires, *all* of the other nodes jump instead of just one. This implies that a node is affected by all other nodes, not just its phase neighbors. Within this class, we examine the INVERSE-MS algorithm, adapted and specialized from a bio-synchronization algorithm proposed in [6].

Inverse-MS Algorithm. Assume that node n is the next node to fire. After node n fires, *all* other nodes $i \neq n$ jump from phase ϕ_i to ϕ'_i where

$$\phi'_i = f_{\text{INVMS},i}(\vec{\phi}; \alpha) = \begin{cases} 1 & i = n \\ (1-\alpha)\phi_i & i \neq n \end{cases} \tag{13}$$

where $\alpha \in (0,1)$ is a jumpsize parameter (see Figure 2(b)). Note that all nodes (except the firing node n) jump at each firing. After jumping, nodes are relabeled according to the rule $i \to i+1$. In particular, node $n-1$ becomes node n and node n becomes node 1.

This algorithm does not produce the same type of desynchrony as DESYNC. Instead, we will have to adopt a less constraining definition of desynchronization in our analysis.

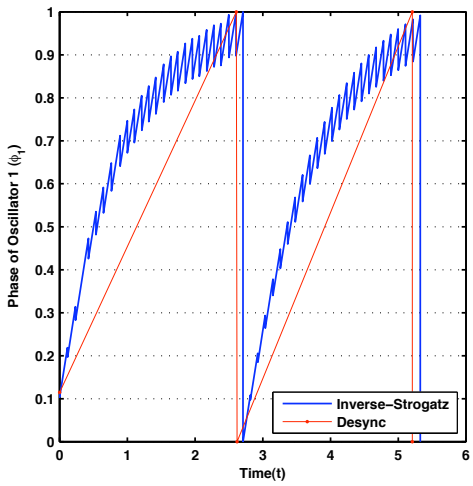

Figure 4. Comparison of steady states for DESYNC and INVERSE-MS algorithms. DESYNC is strongly desynchronized and has a static equilibrium with no jumping (red), while INVERSE-MS is weakly desynchronized and has a dynamic equilibrium, where each node is always jumping backwards (blue). Nonetheless, both algorithms achieve a steady period.

Definition A set of nodes is *weakly desynchronized* if the *time between consecutive nodes' firings are equal.*

Weak desynchronization allows oscillators to accelerate arbitrarily during their cycle, so long as the time between consecutive node firings is identical for all nodes, say some constant $S \in (0, 1)$. Of course, *strong desynchronization* — when $S = T/n$ — is a special case of weak desynchronization. Figure 4 highlights the difference between strong and weak desynchronization by showing the phase trajectory of a single oscillator under both the DESYNC and INVERSE-MS algorithms.

Theorem 4 (INVERSE-MS Convergence) *For all initial conditions and for $\alpha \in (0, 1)$, n nodes whose dynamics are governed by* INVERSE-MS *will be driven to (weak) desynchrony in $O(\frac{1}{n\alpha}\ln(\frac{n}{\epsilon}))$ rounds. However, the steady state period is lengthened to*

$$T^*(n, \alpha) = \frac{n\alpha T}{1 - (1 - \alpha)^n} \approx n\alpha T,$$

where the last approximation is for large n. Furthermore, the time between consecutive firings is $S \approx \alpha T$ for large n.

Proof We will show that INVERSE-MS can be expressed as a linear dynamical system and we will solve for the fixed point. We will also show that the fixed point is weakly desynchronous, and that for small α, it approaches strong desynchrony.

As with the DESYNC algorithms, it will be more convenient to do the analysis in the delta-phase variables, Δ_i. As before, assume that node n fires. Using equation (13) and accounting for the relabeling $i \to i + 1$, we have that

$$\Delta_i' = \begin{cases} \alpha T + (1 - \alpha)\Delta_n & i = 1 \\ (1 - \alpha)\Delta_{i-1} & i \neq 1 \end{cases} \tag{14}$$

To solve for the fixed point $\vec{\Delta}^*$, we use repeated substitution: $\Delta_n^* = (1 - \alpha)\Delta_{n-1}^* = \ldots = (1 - \alpha)^{n-1}\Delta_1^* = (1 - \alpha)^{n-1}[(1 - \alpha)\Delta_n^* + \alpha]$. Solving, we get

$$\Delta_i^* = \frac{(1 - \alpha)^{i-1}\alpha T}{1 - (1 - \alpha)^n}, \tag{15}$$

which approaches T/n as $\alpha \to 0$, by L'Hospital's Rule. Thus, the fixed point of INVERSE-MS deviates from strong desynchrony ($\Delta_i^* = T/n$) as a function of n, α, and i. But the time between consecutive firings, $S = \Delta_1^*$, is equal for all nodes, and so Δ^* is *weakly desynchronized*. In fact, $T^* = T^*(n, \alpha) = nS = n\Delta_1^*$, corresponding to n identically sized slots. Substituting the expression for Δ_1^* from equation (15), we have the desired result for $T^*(n, \alpha)$. The proof of convergence rate is given in the Appendix. □

Comparison to DESYNC. The INVERSE-MS algorithm's convergence is much faster than DESYNC for certain values of α, since all n nodes jump after any single node's firing (Figure 3(b)). However, INVERSE-MS's fast convergence comes at a price: (1) the steady-state period T^* is greater than the intrinsic oscillation period T (period lengthening), (2) every node must listen to every other node's firings, and (3) every node is always jumping backwards, even in steady-state (dynamic equilibrium). Note that DESYNC does not suffer from any of these problems, since each node's firing only affects its two phase neighbors and the steady state is a static equilibrium. Figure 4 compares the steady-state trajectory of a DESYNC node and an INVERSE-MS node over two periods.

Some applications may require an accurate and precise end-state period (e.g., time-interleaved analog-to-digital converters). For large n, the period lengthening ratio $L(n, \alpha) = T^*/T \approx n\alpha$ is linear in the number of nodes, n, and the jump-size parameter, α. A compensation strategy by setting $\alpha = 1/n$ is undesirable since it depends on the number of nodes, which maybe difficult to estimate accurately in settings where nodes come and go frequently. Nevertheless, if n is known and fixed ahead of time, α-compensation yields a fast desynchronization algorithm.

Properties	INVERSE-MS	DESYNC-IDEAL	DESYNC-STALE
Convergence Rate	$O(\log n/n)$	$O(n^2)$	$O(n^2)$
Period Distortion	$O(n\alpha)$	None	None
Robust to Message Loss	X	✓	✓
Add/Remove Nodes	✓	✓ ✓	✓ ✓
Converges on Multi-hop Topology[a]	X	✓	✓

Application	INVERSE-MS	DESYNC-IDEAL	DESYNC-STALE
ADC	X	✓	✓
Multi-core Processor	✓	✓	✓
Wireless TDMA	X[b]	X	✓
Intersection Traffic	X	✓	✓

Table 1. Overview of tradeoffs involved in DESYNC and INVERSE-MS algorithm. X=Inappropriate/Bad, ✓=Good, ✓✓=Excellent. (a) Preliminary results for multi-hop networks are based on simulations of a simple extension of the DESYNC algorithm. We are currently unable to find a way to extend INVERSE-MS to multi-hop. Full discussion is left for future work. (b) We are not yet able to extend INVERSE-MS to work in a TDMA setting.

6 Choosing the Right Algorithm for your Application

Self-organizing desynchronization is a powerful primitive that has many potential applications. The choice of desynchronization algorithm depends greatly on the constraints posed by the application environment. For example, message loss in wireless networks makes it impossible for nodes to observe all firings. Therefore, using an algorithm that is robust to missed firings is important. In other settings, the need to establish a precise period may outweigh all other concerns. To help guide the choice of a desynchronization algorithm, we present a brief overview of the tradeoffs (summarized in Table 1).

Convergence Rates. The rate of convergence is at odds with period distortion and robustness to message loss. A fast convergence rate means a quick response time to node membership changes, but it can drastically affect the periodicity of nodes' firings (INVERSE-MS only, see Section 5). A slow convergence rate means slower system response time to nodes entering and leaving, but allows for better periodicity and robustness to message loss. In simulations, INVERSE-MS is the fastest of all algorithms (see Figure 3) with the extra speed coming at the cost of a large period distortion.

Message Loss. In many settings, including wireless networks, message loss is an unavoidable reality. Therefore, robustness to missed firings is important. Here, the best option is to use a DESYNC variant, where missed firings affect only immediate neighbors and the jumpsize parameter α can be set to a low value in order to protect against lost messages. In contrast, INVERSE-MS is ill-suited for handling message loss. A missed firing in INVERSE-MS means that many nodes fail to jump, thereby altering several nodes' periods. Furthermore, achieving good desynchronization in networks with lossy links depends strongly on the algorithm's ability to generalize to a multi-hop topology. Preliminary results suggest that INVERSE-MS does not converge in multi-hop networks.

Adding and Removing Nodes. In settings where nodes enter and leave frequently (e.g. to conserve energy, returning when triggered by an asynchronous event), the system needs to respond quickly and reliably. Here INVERSE-MS suffers a major drawback: its period is a function of the number of nodes and so predicting it is difficult. In extreme situations, where nodes enter and leave incessantly, the period may become completely inaccurate. In DESYNC variants, on the other hand, a node's arrival or departure takes longer to propagate to the rest of the network, and so nodes are able to maintain a stable period throughout the process, at the cost of a slower convergence

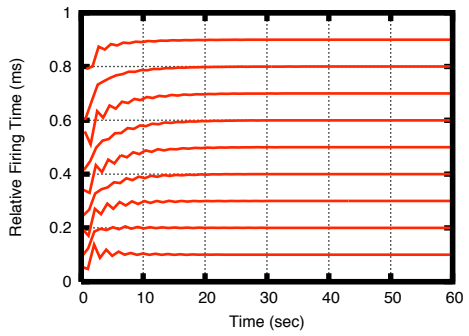

Figure 5. Relative Firing Times: 10 Telos motes placed within communication range of one another running a simple implementation of the DESYNC algorithm. Each node had a firing timer that had a default frequency of 1 second. Upon expiration of the timer, each node broadcasted a message to all other nodes. Shown here is the firing times of each node relative to an arbitrarily selected node whose firing was clamped to the horizontal axis. Note that the nodes achieve desynchronization within 15 periods.

rate.

7 Implementation

To further highlight the simplicity of the algorithm, we implemented the DESYNC algorithm along with a TDMA variation. The full details and results of these implementations can be found in [1]. The algorithms were implemented on several Telos wireless sensor motes running the TinyOS operating system. In the tests, 4-20 nodes were all placed within communication distance of one another on a table. Each node was initialized with a common timer period of 1 second. Whenever a node's timer expired, it broadcasted a message to the rest of the nodes indicating its firing. Nodes recorded the times of the nodes that fired immediately before it and after it relative to their own clocks. Once both firings were heard, the nodes calculated the midpoint and set their timer to fire a period later. The spacings of the firings of a typical run with 10 nodes are shown in Figure 5. Qualitatively, desynchronization is achieved quite quickly. For the ad-hoc TDMA protocol, we added code that had nodes send as much data as they could during their time slots. Table 2 shows abbreviated results of the performance of this algorithm. Of particular interest is the ability of the DESYNC-TDMA algorithm to share the bandwidth fairly and fully amongst the nodes. Furthermore, there is a graceful linear degradation with increasing numbers of nodes. In short, the DESYNC algorithm's simplicity and efficacy is a great advantage for any application that requires the desynchronization of fully-connected devices.

Nodes:	4	10	20
Total Throughput (Kbps):	**60.8**	**57.9**	**53.0**
(normalized, %)	(96.8)	(92.2)	(84.3)
Max Individual (Kbps):	15.2	5.8	2.8
Min Individual (Kbps):	15.2	5.6	2.4
Message Loss (%):	**0.2**	**0.3**	**0.5**

Table 2. DESYNC-TDMA's performance for varying numbers of nodes over 60-second runs. A comparison to other protocols can be found in [1]. In our experiments, the maximum rate at which a single node could transmit was 62.8 Kbps.

8 Conclusions and Future Work

As ad-hoc networks become more popular, algorithms for achieving desynchronization in a decentralized, self-organized manner will be increasingly important. In this paper we have designed, analyzed, and implemented two such algorithms: DESYNC and INVERSE-MS. The choice of algorithm depends on particular constraints posed by the application setting. INVERSE-MS is a good choice when the number of nodes is constant, all connections are reliable, the topology is fully-connected, and message delays/losses are negligible. In short, INVERSE-MS is fast but not robust. On the other hand, DESYNC is slower, but much more robust. It can handle an arbitrary number of nodes, unpredictable node arrivals and departures, and some message loss (reliability is only needed for the links between phase neighbors). The simplicity of both algorithms allows for wide applicability and we presented a successful implementation of DESYNC on a real sensor network testbed [1].

The desynchronization algorithms presented in this paper all rely on a single-hop, all-to-all topology. Thus, an important problem for future work is assessing performance on multi-hop networks. Computing an ideal desynchronization schedule that maximizes the sum of the slot sizes of all nodes seems hard, since the problem seems closely related to graph coloring and task scheduling. However, computing reasonable desynchronization schedules may be tractable. Our preliminary results suggest that: (a) DESYNC converges in multi-hop networks (though multiple steady states are possible), and (b) the optimal schedule is related to minimal graph coloring, and so is likely to be NP-hard. In contrast, INVERSE-MS fails to converge after the removal of just one edge from the topology, highlighting its

strong dependence on regular topologies.

References

[1] J. Degesys, I. Rose, A. Patel, and R. Nagpal. Desync: Self-organizing desynchronization and tdma in wireless sensor networks. In *Proc. Conference on Information Processing in Sensor Networks*, Jan 2007.

[2] V. Divi. Scalable blind calibration of timing skew in high-resolution time-interleaved adcs. In *IEEE International Symposium on Circuits and Systems*, 2006.

[3] K. Dresner and P. Stone. Sharing the road: Autonomous vehicles meet human drivers. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, Hyderabad, India, January 2007.*, 2004.

[4] Y. Hong and A. Scaglione. Time synchronization and reach-back communications with pulse-coupled oscillators for uwb wireless ad hoc networks. In *"Proc. IEEE Conference on Ultra Wideband Systems and Technologies"*, Nov 2003.

[5] D. Lucarelli and I. Wang. Decentralized synchronization protocols with nearest neighbor communication. In *Proc. Conference on Embedded Networked Sensor Systems (SenSys)*, Nov 2004.

[6] R. Mirollo and S. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal of Applied Math*, 50(6):1645–62, Dec 1990.

[7] C. S. Peskin. *Mathematical Aspects of Heart Physiology*. Courant Institute of Mathematical Sciences, New York University, New York, 1975.

[8] J. S. Rosenthal. Convergence rates for markov chains. *SIAM Review*, 37(3):387–405, 1995.

[9] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2002.

[10] G. Werner-Allen, G. Tewari, A. Patel, R. Nagpal, and M. Welsh. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proc. Conference on Embedded Networked Sensor Systems (SenSys)*, Nov 2005.

A Appendix

Here we present theory regarding convex polynomials and the proofs of the convergence rates of all analyzed algorithms.

Definition A: Stable polynomials A polynomial $\rho(\lambda) = \lambda^n - a_{n-1}\lambda^{n-1} - \cdots - a_0$ is **stable** if: (1) $\lambda = 1$ is a single ("trivial") root of $\rho(\lambda)$ and (2) the trailing coefficients $\{a_{n-1}, \ldots, a_0\}$ are nonnegative and less than 1.

Theorem 5 (Nontrivial Eigenvalues of Convex Polynomials) *The nontrivial roots of a stable polynomial lie inside the unit circle in the complex plane.*

Proof Let $\rho(\lambda)$ be a stable polynomial. Since $\rho(1) = 0$, we have that the sum of the trailing coefficients must be one, a fact that we will use later on. Consider any nontrivial root λ of ρ. We will show by contradiction that $|\lambda| < 1$. Suppose $|\lambda| > 1$. Then by rearranging ρ and applying the Triangle Inequality we have that $|\lambda|^n \leq a_{n-1}|\lambda|^{n-1} + \cdots + a_1|\lambda| + a_0 < \left(\sum_{i=0}^{n-1} a_i\right)|\lambda|^{n-1} = |\lambda|^{n-1}$, a contradiction since $|\lambda| > 1$. Hence $|\lambda| \leq 1$. Now suppose that $|\lambda| = 1$, i.e., $\lambda = e^{i\theta}$ with $0 < \theta < 2\pi$. (Note that $\theta = 0$ corresponds to the trivial root $\lambda = 1$.) Rearranging ρ as before gives us that $\lambda^n = a_{n-1}\lambda^{n-1} + \cdots + a_0$. Since the trailing coefficients are nonnegative, less than one, and sum to one, the right hand side is a convex combination of the points $\{1, \lambda, \ldots, \lambda^{n-1}\}$, all of which are on the boundary of the unit circle. Since the unit circle is a convex set, any convex combination of at least two distinct boundary points lies in the interior of the circle. But this contradicts the fact that the left hand side λ^n is on the unit circle, and hence it must be that $|\lambda| < 1$ for all nontrivial roots $\lambda \neq 1$.

Proof of Theorem 3:DESYNC-IDEAL Convergence Rate Note that the matrix A_{IDEAL} is stochastic (i.e. its entries are nonnegative and its columns sum to one). Hence we can associate a Markov Chain $\mathcal{X}_t$ with state space $\Omega = \{1, 2, \ldots, n\}$ with transition matrix A^T_{IDEAL}, thus linking the convergence rate of DESYNC-IDEAL with the mixing time of the chain. This chain takes $O(n)$ time to complete one "circuit" from state 1 back to state 1. Using the Central Limit Theorem and the symmetry of the ring, we can show that it takes $O(n^2 \ln(1/\epsilon)/\alpha)$ circuits multiplied by $O(n)$ time per circuit to mix. This implies that the number of periods require for DESYNC-IDEAL to reach ϵ-desynchrony is $O(n^2 \ln(1/\epsilon)/\alpha)$. □

Conjecture in Theorem 3:DESYNC-STALE Convergence Rate The nth power of A_{STALE} is very similar to the transition matrix of a symmetric random walk on a ring of n nodes, modulo a few edges. This suggests that $O(n^2 \ln(\frac{1}{\epsilon})/\alpha)$ periods are required to reach ϵ-desynchrony. MATLAB computations of eigenvalues in conjunction with simulations confirm these bounds. □

Proof of Theorem 4:INVERSE-MS Convergence Rate Equation 13 can be written in matrix form as $\vec{\Delta}' = C\vec{\Delta}$, where C represents the jump-and-permute steps of the INVERSE-MSalgorithm. The matrix J has non-trivial eigenvalues of magnitude $1 - \alpha$ and thus solving the equation $(1-\alpha)^F < \epsilon$ for the number of firings F yields $F > \frac{1}{\alpha}\ln(\frac{n}{\epsilon})$. Since there are n firings per round, the number of rounds require to reach accuracy ϵ is F/n, which is $O(\frac{1}{n\alpha}\ln(\frac{n}{\epsilon}))$, as desired. □