Project Report
LSP-217

Medium Access Control for Large-Scale Multi-Robot Teams: FY17 Line-Supported Autonomous Systems Program

P. Deutsch
H.P. Romero
B.E. Shrader

22 December 2017

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based on work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

Massachusetts Institute of Technology
Lincoln Laboratory

Medium Access Control for Large-Scale Multi-Robot Teams: FY17 Line-Supported Autonomous Systems Program

P. Deutsch
H.P. Romero
B.E. Shrader
Group 65

Project Report LSP-217

22 December 2017

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

Communication networks are needed to support the vision of embedding sensing and computing capabilities into large numbers of small devices. Commercial communication systems can often fulfill this need but are not typically hardened for defense applications. These networks need to operate in contested environments, where they are vulnerable to high-power signal jamming. A long-standing technical challenge to large communication networks is medium access control, which assigns time, frequency, and space resources to users or links. This is particularly challenging in decentralized networks.

This report summarizes work on a Line-funded program in developing medium access control algorithms and protocols to support low-rate, timely exchange of short messages among low-power devices in a 100-node-scale decentralized network. First, a literature review and idealized performance analysis of standard and novel approaches to medium access are conducted. Based on the results of this analysis, a novel medium access control algorithm is selected for further study. This algorithm is the Desync algorithm originally proposed in [1]: it initializes as a contention scheme and then adapts toward a conflict-free time-division scheme. An implementation of this algorithm is developed and outlined in this report; it includes minor modifications to the original algorithm and a specification of operation as a medium access control protocol. Experimental studies demonstrate the performance of this novel protocol and provide a comparison with standard approaches.

This page intentionally left blank.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF TABLES

This page intentionally left blank.

1. INTRODUCTION

New applications of sensing and computing in small devices are driving the need for new forms of wireless communication networks. Commercial applications, from environmental monitoring of water levels, agriculture, and wildlife; to monitoring and control of infrastructure, including lighting systems, power systems, transportation infrastructure, and manufacturing facilities, are all in development. Government and defense applications are also envisioned, including swarms of small autonomous vehicles to perform intelligence, surveillance, and reconnaissance missions; and large sensor networks to monitor military bases. All of these applications require low-rate communication of short messages, connecting large networks of low-power devices.

Commercial communication systems are being developed to support this need, but they are unlikely to support all requirements for defense applications. Updates to existing standards, such as ZigBee [2] and Bluetooth [3], including Bluetooth Low Energy and Bluetooth Mesh, are being made, while new standards and systems, such as Long-Term Evolution Machine-Type Communication (LTE MTC) [4], LoRaWAN [5], and RPMA [6] are competing to support communication for the "Internet of Things." With this variety of options, some of these commercial systems should fulfill the needs of defense applications in benign environments. But military systems that operate in contested environments will undoubtedly require modified or custom systems, for two key reasons. First, military systems can be subject to jammers that operate at radiated power levels that are orders of magnitude higher than anything encountered by commercial systems, and low-power devices are especially vulnerable to these conditions. Second, all of the commercial communication systems listed above rely on access points or network coordination nodes, which introduce a single point of failure that is vulnerable to jamming or even kinetic attacks. Clearly, there is a need to invest in the development of communication systems for low-power devices operating in contested environments.

One of the primary challenges in designing a large communication network is the classic problem of medium access control (MAC), which assigns time, frequency, space, and other shared resources in a way that limits interference among users of the same network. The medium access scheme and its design are closely coupled with the operation of the physical signaling to support digital communication. Yet some basic assumptions relevant to the application of interest allow the MAC problem to be decoupled and considered in isolation. First, it's assumed that omni-directional antennas are used, which is a reasonable choice for inexpensive, small devices that need to communicate in all directions and don't have the flexibility of pointing an antenna beam. This assumption has implications for how signals and interference radiate to other users. Second, users in the network are assumed to share a single common channel; while frequency division as a medium access technique and spread spectrum as an anti-jam technique are certainly relevant, the basic problem of multiple users sharing a single common channel still arises in these settings.

Medium access control for wireless networks is an old problem, dating back to at least the 1970s, and standard, well-tested techniques are available. An available set of options for medium access control had been identified and compared by 1990, and these are shown in Fig. 1. As the figure suggests, there are two basic approaches to medium access: contention and conflict-free. In contention schemes, there is little coordination among users, resulting in high levels of interference,

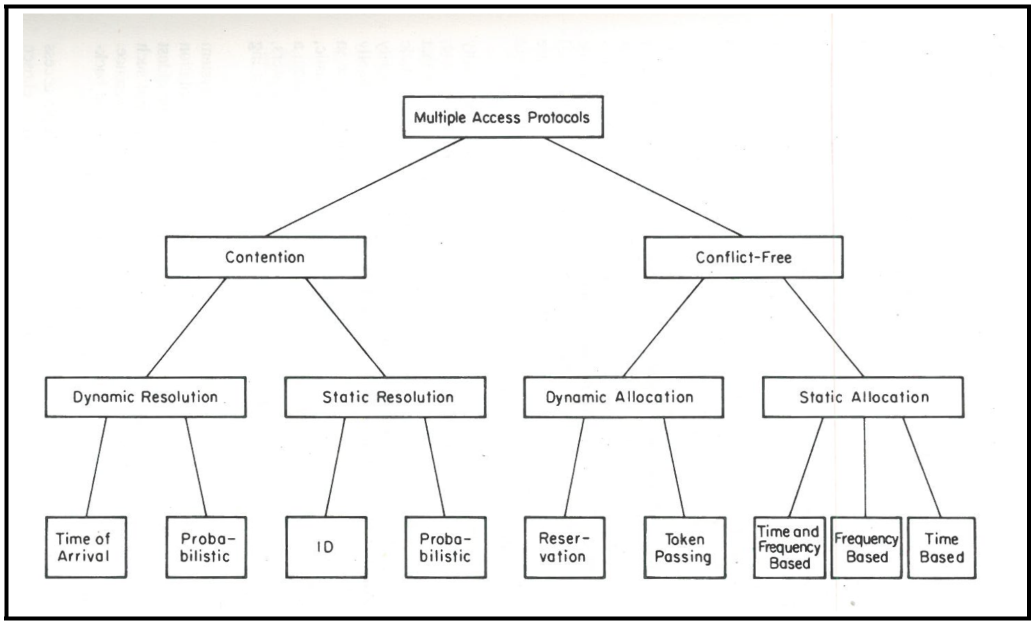


Figure 1. Classification of approaches to multiple access. From [7], published in 1990.

failed transmissions, and the need to *resolve* conflicts through some static or dynamic scheme. Contention schemes require little overhead for coordination, but consume communication resources with failed attempts. By contrast, conflict-free schemes carefully coordinate or allocate resources, leading to efficient use of the channel, but high overhead. Moreover, conflict-free schemes typically require time-synchronization; in modern systems, even military systems, this is often provided by the Global Positioning System (GPS), which is vulnerable to jamming in contested environments. In addition to these two approaches, some systems adopt a hybrid approach: a contention scheme is used to "request" resources and a conflict-free scheme is used to service those requests. Since the 1970s, hundreds of schemes have been developed and implemented, and hundreds of thousands of papers have been written on the topic of medium access control. Progress has been made in many different directions, including collision avoidance for contention schemes [8], power control, rate adaptation, "cognitive radio" schemes that adapt to changes in spectrum availability, and "demand-based" schemes that adapt to different traffic loads and latency requirements. Yet the basic approaches described in Fig. 1 remain the same.

The proliferation of small electronic devices and the vision of future applications have prompted questions about medium access control that have not previously been answered, even at the level of basic research. The number of users operating within a single collision domain is expected to be significantly larger than what's observed in current systems. The constraints of low-power devices have renewed interest in the metric of energy consumed per bit of communication. Similarly, the need for timely communication of short packets for status reports and control have given rise to the new metric of the "age of information" [9].

In this context we revisit the problem of medium access control for large networks of autonomous devices. This report is organized as follows: Section 2 contains a literature review and performance analysis of schemes of interest. As a result of this review, a novel MAC algorithm is selected for implementation, which is described in detail in Section 3. The results of experimental studies are presented in Section 4, followed by a discussion and conclusion.

This page intentionally left blank.

2. PERFORMANCE ANALYSIS

Medium access schemes can be broadly categorized into conflict-free scheduling schemes and contention-based schemes. The former require more overhead, but ultimately can support greater rates over relatively static networks. The latter require less overhead, and thus are better at adapting to changing conditions in the data to send or in the propagation and interference environment. However, the latter pay a large price in total available throughput should the network not be as dynamic as originally envisioned.

In this project we explore a candidate idea for a relatively lightweight decentralized schedule-based medium access control protocol. To assess its value, we compare it to contention-based medium access schemes and attempt to determine whether the extra complication of attempting to converge to a good distributed schedule is worth the effort.

2.1 CONFLICT-FREE SCHEDULING: MODEL

First, we begin with some definitions. In schedule-based medium access, each communication node is a assigned a transmission time during which it can transmit. A good schedule is one which both avoids interference, while also allotting each node as much time, or degrees of freedom, as is possible. To provide a framework for analysis, let $G = (V, E)$ be a graph consisting of a set of vertices V representing the communicating nodes and the set of edges E between those vertices. Edges represent pairs of nodes that would destructively interfere or collide if they were to transmit at the same time; thus edges are constructed between neighboring nodes that communicate directly, and edges are also placed between nodes that are not direct neighbors, but whose transmissions will prevent reception at their intended receiving nodes. In this graph, let C be the circle of unit length. A *schedule* is an assignment to each node $v \in V$ of an arc $a(v) \subseteq C$ such that $a(v) \cap a(u) = \emptyset$ for every $(u, v) \in E$. An equivalent representation of a schedule is given by the pair $(\phi(v), l(v))$ for each node, where $\phi(v)$ and $l(v)$ are the midpoint and length, respectively, of the arc $a(v)$.

If C represents a frame in a time-division multiple-access scheme, then a schedule corresponds to a collision-free transmission/reception allocation for the collective network. For ad-hoc networks, we will be interested in decentralized approaches to generating schedules. To assess the usefulness of these approaches, and to compare and contrast the schedules, we first consider characterizing a notion of "good" schedules.

Of particular interest are schedules for which the lengths, $l(v)$, of the arcs $a(v)$ are maximal. That is, for each $v \in V$, if the other arcs $(a(u) : u \neq v)$ are held fixed, then $a(v)$ can not be increased in size while maintaining that $(a(u) : u \neq V)$ be a schedule. In general, there is a trade-off: in any schedule, some arc lengths might be large at the expense of other arc lengths being small.

2.1.1 Max-min Fair Schedules

Consider a schedule that is maximally fair. That is, consider only a maximal schedule for which every arc has the same length, a. Then $a \leq 1/\phi_c(G)$, where $\chi_c(G)$ is the circular chromatic number of the underlying graph G [10]; the circular chromatic number may be distinct from the chromatic number. This upper bound is tight, in that a schedule exists that meets this bound with equality.

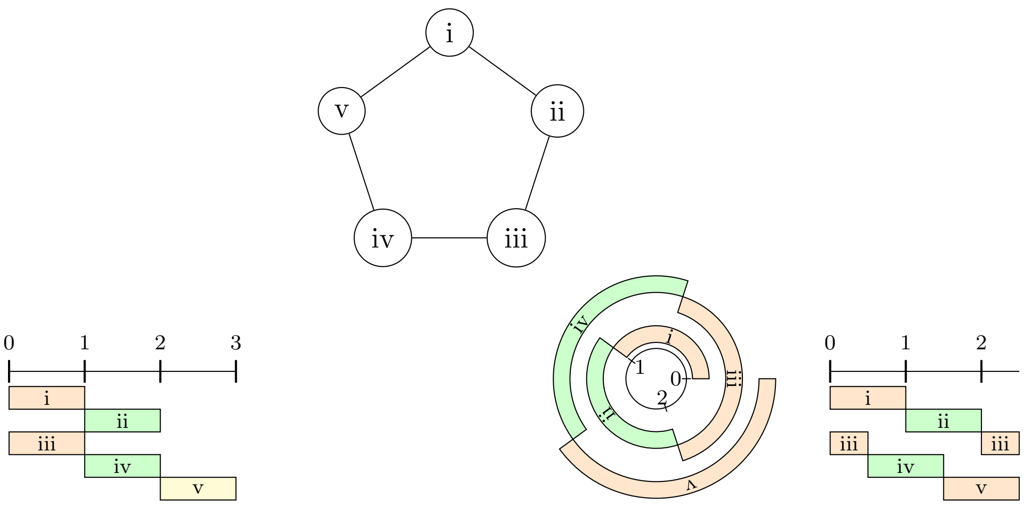


Figure 2. For the cyclic graph C_5 (top), the chromatic number is 3 (left), while the circular chromatic number is 2.5 (right).

To understand the relationship between the chromatic and cyclic chromatic number, consider first the following manner of paraphrasing graph coloring. Rather than mapping each node to a color, suppose we map each node to a unit length interval $I(v) = (i_v, i_v + 1)$, where i_v is an integer. Then graph coloring is equivalent to determining an interval mapping I satisfying $I : V \mapsto [0, r]$, for some $r > 0$, so that if $(u, v) \in E$, then $I(u) \cap I(v) = \emptyset$. The chromatic number is the minimum r for which this is possible. As this procedure is carried out on an interval, we call this interval graph coloring.

Consider a variation where, rather than mapping nodes to subintervals of an interval, we map nodes to arcs of a circle. That is, suppose that each node is mapped to a unit length arc $A(v)$ on a circle of length r. Then a cyclic graph coloring, if it exits, is an arc mapping for which, if $uv \in E$, then $A(u) \cap A(v) = \emptyset$. The minimum such r that permits a circular graph coloring is the circular chromatic number. See Figure 2 for an illustration of the two concepts. If the circle length is fixed to unity, rather than the arc length, then the analogous formulation of circular graph coloring is precisely that of determining a max-min schedule.

Let $\chi(G)$ be the chromatic number of G. Then the circular chromatic number is known to be within the interval $\chi(G) - 1 < \chi_c(G) \leq \chi(G)$. These interval bounds are tight. As an interval can be wrapped around to form a circle, a coloring on the interval, the minimum of which is given by the chromatic number, can be mapped to a coloring on the circle. Moreover, it is clear that it must be tight for the complete graph K_n. The lower bound is a tight infimum as well. For example, the cyclic graph C_n with n nodes has circular chromatic number $\chi_c(C_n) = n/\lfloor n/2 \rfloor$, while $\chi(C_n)$ is 2 when n is even and 3 if n is odd. When n is even, the chromatic and circular chromatic numbers coincide: $\chi_c(C_n) = \chi(C_n) = 2$. When n is odd, the circular chromatic number is $2n/(n-1)$, which is strictly smaller than the chromatic number and arbitrarily close to the lower bound $\chi(C_n)-1 = 2$ for sufficiently large odd n.

2.1.2 Locally Fair Schedules

An alternative notion of good schedules was developed in [11]. Let $d_C(x,y) = \min\{(x-y) \bmod 1, (y-x) \bmod 1\}$, which satisfies $d_C(x,y) \leq 1/2$. A *desynchronization configuration* is a schedule where there are are least two nodes which achieve the minimum $\min_{u:(v,u)\in E} d_C(\phi(v), \phi(u))$. When the phases are distinct, one might think of a desynchronization configuration as a schedule that is locally fair.

Each desynchronization configuration gives rise to a class of configurations, obtained by adding a constant phase offset to all nodes. To avoid talking of this caveat, we assume that $\phi_1 = 0$ without loss of generality. Every graph has at least one desynchronization configuration, the trivial solution being $\phi_v = 0$ for all $v \in V$. This state is the synchronization configuration: it is desirable for establishing a common clock, but as a medium access scheme, it would assure that there is maximum interference. Hence, we will be interested in algorithms for which this configuration is a repelling, rather than attracting, steady state. Is there always a non-trivial desynchronization configuration?

For a few structured graphs, we can answer this question by explicitly describing all of the non-trivial desynchronization configurations. The complete graph has only one non-trivial desynchronization configuration, the "round-robin" or "splay" state wherein $\phi_v = v/n$ for any enumeration of the nodes as $V = \{0, \ldots, n-1\}$. By contrast, the cyclic graph C_n has $\lfloor n/2 \rfloor + 1$ desynchronization configurations, where $\phi_v = (vm)/n \mod 1$, for any choice of $m \in \{0, \ldots, \lfloor n/2 \rfloor\}$. To see this, observe that on the cyclic graph, a desynchronization configuration must satisfy $d_C(\phi(k \mod n), \phi((k+1) \mod n) = x$ for all $k \in \{0, \ldots, n-1\}$ as

$$d_C\big(\phi(k \mod n), \phi((k+1) \mod n)\big) = d_C\big(\phi((k+1) \mod n), \phi((k+2) \mod n)\big)$$

for any $k \in V$. Moreover, nx must be an integer m, as necessarily

$$\Big(\sum_{k=1}^{n} d_C\big(\phi(k \bmod n), \phi((k+1) \bmod n)\big)\Big) \bmod 1 = 0 \ .$$

As $d_c(\cdot,\cdot) \leq 1/2$, $x \leq 1/2$, and so $x = m/n$ for some $m \in \{0, \ldots, \lfloor n/2 \rfloor\}$.

2.2 CONFLICT-FREE SCHEDULING: ALGORITHMS

Scheduling algorithms vary from purely decentralized solutions to purely centralized solutions. For the former, as exemplified by random schedules with random backoff, each node needs to know nothing about the neighbors, size of the network, or incoming packet rates. For the latter, as exemplified by a graph coloring (or circular graph coloring), global knowledge of the network is needed.

Beyond the knowledge of graph structure needed, a relevant metric of scheduling algorithms is the time to convergence, which should be faster than speed by which the underlying topology changes. As an extreme example, consider the graph coloring approach. Exact solutions to the problem can be prohibitively complex to compute. By the time a coloring schedule is computed, the topology of the graph may have changed, rendering the computed schedule outdated. As similar topologies could have very different optimal graph colorings, the outdated schedule could be ineffective. As an analogy to a notion of the physical layer, we wish for any scheduling algorithm to have a convergence time that remains less than the "coherence time" of the network topology.

One idea for determining a schedule in a decentralized manner is to use ideas from pulse-coupled oscillators [1]. Pulse-coupled oscillators are mathematical models that have been used to study how emergent coordinated behavior can arise as the result of locally coupling [12]. The most studied coordinated behavior is that of synchronization, where the network nodes evolve to a common phase, or clock, and results in this field are plentiful. That locally coupling can be used as a robust means of a global clock in wireless networks is discussed in [13].

In addition to synchronization, another sense of emergent coordinated behavior is that the nodes' phases converge to a state where the phase difference to each neighbor is constant, but non-zero. Any one of these convergent states can be thought as a common schedule. One notion of a good schedule is the notion of desynchronization proposed in [1]. There, a local coupling algorithm with fine updates is proposed that aims to achieve a desynchronized state. In [14], a good common schedule is sought after by means of a local coupling algorithm with coarse updates.

The desynchronization method of [1], which we refer to interchangeably as *fine desynchronization* or Desync, updates as follows. Each node has clock $\phi(t)$, that in the absence of coupling, revolves around a circle each unit of time. The clock "fires" a pulse when $\phi(t) = 0$ to its neighbors. In the presence of coupling, each node's clock is affected by the firing pattern of neighboring nodes. Let the jth node have clock $\phi_j(t) = (t + \nu_j) \mod 1$. Let $\tilde{\phi}_i(t) = \phi_j(t) - \phi_i(t)$ for each neighbor i of node j. Let the "previous" node be node the i_P, which was the last to fire before node i fired. That is, $\tilde{\phi}_{i_P}(t) = \min\{\tilde{\phi}_i(t) : \tilde{\phi}_i(t) > 0\}$. Let the "next" node be the node i_N, which is the next to fire after the node i fired. That is, $\tilde{\phi}_{i_N}(t) = \min\{\tilde{\phi}_i(t) : \tilde{\phi}_i(t) < 0\}$. Then, when node i_N fires at time t_N, node i updates its clock offset ν_i so that $\phi_i(t_N) = \alpha\phi_i(t_N) + (1-\alpha)(\tilde{\phi}_{i_P}(t_N) - \tilde{\phi}_{i_N}(t_N))/2$ for some parameter $\alpha \in (0, 1)$.

2.2.1 All-to-all Topologies

The all-to-all topology was the initial setting of [1]. Some work has been done to further investigate desynchronization primitives in this particular setting. In [15], a faster algorithm is proposed, by demonstrating that neighboring-phase algorithm is implicitly optimizing a convex objective, though this objective is dependent on the initial phases and all-to-all topology. Yet, given this observation, we conclude that the neighboring-phase algorithm is but one of many possible gradient descent algorithms on a convex objective, and others may be faster. Other algorithms that incorporate all neighboring phase information, rather than the two closest, have also been proposed [16–18], but only [17] attempts to analyze the algorithm in a topology outside of the all-to-all topology. In [19], the robustness of the neighboring-phase algorithm is demonstrated in the face of noisy observations of each neighbor's phase.

2.2.2 Ring Topology

The idea that Desync implicitly minimizes a convex objective can be extended from the all-to-all topology to the ring topology. In this extension, we unroll the phases from their positions on the circle to locations on an interval of integer length. This can be shown as follows:

- Repeat the following for the clockwise ($d = +$) and counterclockwise ($d = -$) directions. Unroll phases in the direction d so they are ordered on an interval $[0,\, r^{(d)}]$, where $\phi(1)$ is represented twice: first as the initial integer 0, and second as the ending initial $r^{(d)}$. Let the unrolled vector be $\phi_u^{(d)}$.
- Each iteration of Desync preserves the order on this interval.
- Each iteration of Desync decreases the convex objective

$$C(d) = \|D\phi_u^{(d)}\|^2 \ , \tag{1}$$

  where D is the matrix that returns the adjacent pair-wise differences of its input matrix, and where d can be either the clockwise or counterclockwise direction.
- As the $C(d)$ is stationary only at a vector of constants, this must be the steady-state solution.

An invariant of the algorithm is the "unrolled length" of the initial phases. If the initial phases have an unrolled length of r initially, then the algorithm will maintain this unrolled length. This permits a immediate forecast of which steady-state Desync will settle into form its initial phases. Such an analysis extends that of [17], which is restricted to an initial draw of phases that need not be unrolled: i.e., they are already ordered on the interval $[0, 1]$. There are two possible unrolled lengths: one for a clockwise unrolling and another for a counterclockwise unrolling.

Let $\pi(v)$ be the order of the node v in the clockwise direction. That is, $\phi(\pi^{-1}(i)) \leq \phi(\pi^{-1}(i + 1))$ for every $i \in [n - 1]$. Then π is a permutation on the enumeration $[n]$ of the nodes and the described unrolled lengths correspond to number of cyclic descents (for the clockwise direction) or cyclic ascents (for the counter clockwise direction) of the permutation π. The cyclic ascents and cyclic descents of a permutation are connected to the descents and ascents of that permutation [20].

The minimum of the number of cyclic descents and the number of cyclic ascents prescribes the final state of the Desync algorithm. When the phases of the Desync algorithm are drawn independently and identically at uniform form the unit diameter circle, then the resultant permutation π on the phases is drawn uniformly at random from all possible permutations on n elements. Hence, the probability that the final state of the Desync algorithm amounts to the probability that the minimum of the number of cyclic ascents or descents are a particular value. As we now describe, theory from enumerative combinatorics provides this count precisely.

Let $A_{n,i}$ be the number of permutations of length n that have $i-1$ descents. As a permutation with $i - 1$ descents has $n - i$ ascents, the number of permutations of length n that have $i - 1$ ascents is $A_{n,n-i}$. The quantities $A_{n,i}$ are the Eulerian numbers and satisfy the recursion $A_{n+1,i} = iA_{n,i} + (n - i + 2)A_{n,i-1}$ with boundary conditions $A_{n,0} = 0$ and $A_{n,n} = A_{n,1} = 1$ for all $n \geq 1$.

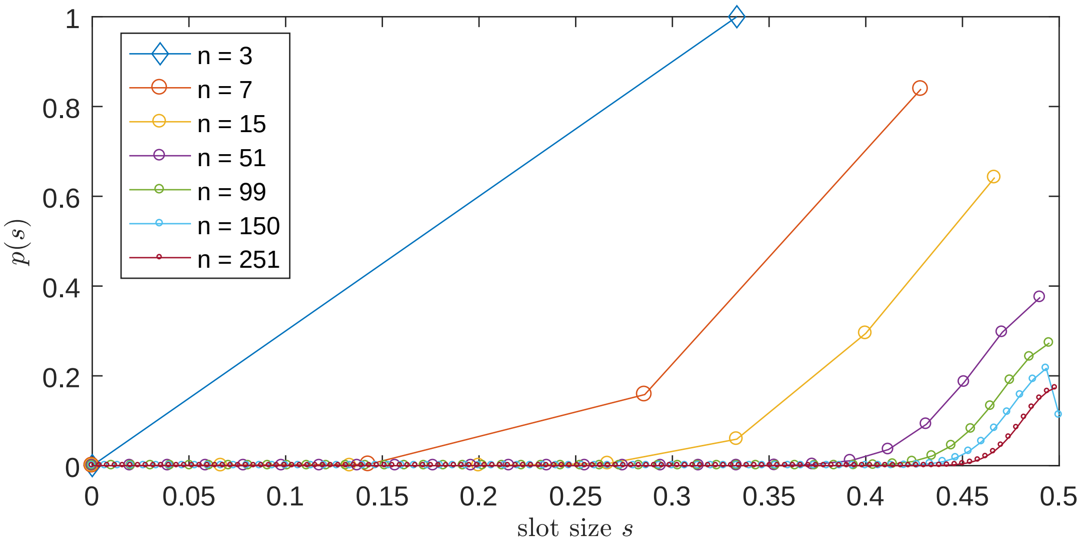


Figure 3. Probability mass function of the slots sizes achieved by Desync, after convergence to steady state, on the cyclic graph C_n of n nodes.

Collectively, the table $(A_{n,i} : n \geq 1, 1 \leq i \leq n)$ is known as the Eulerian table. Now, let $A^{(c)}_{n,i}$ be the number of permutations of length n that have i cyclic descents. Then the number of permutations of length n that have i cyclic ascents is $A^{(c)}_{n,n-i}$. Appealingly, the relationship between cyclic descents and descents is simple: $A^{(c)}_{n,i} = nA_{n-1,i}$ for all $1 \leq i \leq n-1$ [21].

Suppose that

$$S^{(n,L)} = \sum\nolimits_{i=1}^{n} d_C(\phi(i), \phi(\text{mod }(i+1, n))) \tag{2}$$

is the sum of the successive phase differences on the n-node cyclic network of the phase values of Desync after L rotations of the first node. Then, provided that the initial phases are drawn uniformly and independently at random from the unit circumference circle, $P(S^{(n,L)} = s) \to p_n(s)$ as $L \to \infty$ where

$$p_n(s) = \begin{cases} \frac{A^{(c)}_{n,s}+A^{(c)}_{n,n-s}}{n!} = \frac{A_{n-1,s}+A_{n-1,n-s}}{(n-1)!} & s < n/2 \\ \frac{A^{(c)}_{n,s}}{n!} = \frac{A_{n-1,s}}{(n-1)!} & s = n/2 \end{cases} \tag{3}$$

for each $s \in \{0, \ldots, \lfloor n/2 \rfloor\}$. A few of the predictions provided by the above theory are depicted in Figure 3. In Table 1, we compare the number of predicted vs. simulated results for the cyclic network with seven nodes.

2.2.3 Arbitrary Topologies

No theoretical guarantees on the convergence of the Desync algorithm are known. An algorithm close to Desync is proposed in [17] and, on a class of "circulant symmetric" networks, its convergence to only one particular desynchronized configuration is demonstrated provided that

TABLE 1

Simulated runs vs. predictions for the cyclic network with 7 nodes. The empirical count of (2) was computed over $50 \times 7! = 252,000$ independent runs of Desync, each with random initial phases, until the first node has traveled around the circle 50 times. The combinatorial count is the prediction $50 \times 7! \times p_7(s)$ of (3).

s	0	1	2	3
empirical count	0	705	40041	211254
combinatorial count	0	700	39900	211400

the initial are ordered in a particular way. Convergence to other configurations is not discussed when the initial phases are alternatively ordered. Nor are other network topologies considered. An alternative decentralized scheduling algorithm is provided in [14], where a non-optimal schedule can be guaranteed to be achieved in few iterations. Though there are strong theoretical guarantees with this approach, the schedule itself may be far off of what Desync might achieve. Let $d_i^{(c)}$ be the degree of node i in the conflict graph. The distributed synchronization protocol in [14] provides a throughput at node i of

$$\frac{1}{2(\hat{d}_i^{(c)}+1)} \text{ where } \hat{d}_i^{(c)} = \max_{j \in \mathcal{N}_i \cup \{i\}} d_j^{(c)}.$$

We let $d_{\max} = \hat{d}_i^{(c)}$ be the maximum degree among all nodes in the conflict graph. Then the expected max-min throughput is

$$\frac{1}{2}\left(\frac{1}{d_{\max}+1}\right) . \tag{4}$$

2.3 CONTENTION-BASED MEDIUM ACCESS

To provide context for the proposed lightweight Desync, we compare the algorithm to a simple, and prolific, medium access mechanism: ALOHA. By ALOHA, we refer to medium access protocols that tolerate collisions for the benefit of not having to bother to determine an "optimal" schedule.

2.3.1 Slotted ALOHA

In slotted ALOHA, one divides an interval of time τ into m slots. Each of K users contending for access to the medium to transmit to a common receive transmits at any one given slot with probability p, i.i.d. between slots. The probability that a given node waits n slots between transmissions is $(1-p)^{n-1}p^n$ and the probability that it transmits n times over m slots is $\sum_{k_1+\cdots+k_n=m} \prod_{i=1}^{n} (1-p)^{k_i-1} p^n = \binom{m}{n}(1-p)^{m-n}p^n$.

In a multiple access topology with K transmitters and one receiver, the probability that there is a successful transmission in the link between the kth transmitter and the receiver is $p(1-p)^{K-1}$. This probability is maximized through the choice of $p = 1/K$, yielding a per-user throughput of

$1/K(1-1/K)^{K-1}$. The expected number of successful transmissions in a single slot collectively between the transmitters and receivers at this maximal transmission probability is $(1-1/K)^{K-1}$, which is approximately 1/e for large K.

In the prior analysis, the roles of the receiver and transmitter were fixed. If the nodes are all transceivers, so that they listen when they don't transmit, then the analysis changes. Suppose that the receiver itself can transmit, so as to broadcast to the K devices that had been only transmitters. Its broadcast link is successful with probability $p(1-p)^K$. Moreover, any one of the K links between the transmitters and receivers is successful with the same probability. Hence, the maximal per-transceiver successful link probability is this case is $1/(K+1)(1+1/(K+1))^K$, and the sum throughput is $(1-1/(K+1))^K$.

In a general topology, where all nodes are transceivers, then the probability that any given link (i,j) has a successful transmission is given by $p_i(1-p_i)^{d_j}$, where d_j is the degree of the node j in the contention graph, assuming that directed transmission is possible. Let $\mathcal{N}_{i,k}$ be the number of nodes that are reachable in k hops from node i. If the transmission from node i is to be a broadcast to all neighbors $\mathcal{N}_{i,1}$, and if nodes interfere if and only if they are within two hops of each other, then node i has a successful transmission with probability

$$\underbrace{p_i}_{\text{transmission}} \underbrace{\left(\prod\nolimits_{j\in\mathcal{N}_{i,1}}(1-p_j)\right)}_{\text{reception}} \underbrace{\left(\prod\nolimits_{k\in\mathcal{N}_{i,2}\setminus\mathcal{N}_{i,1}}(1-p_k)\right)}_{\text{contention}} .$$

If all the nodes have the same transmission probability, then this per-transceiver expected throughput is $p(1-p)^{d_{i,2}}$, where $d_{i,2}$ are the number of nodes within two hops of node i. Thus, if the nodes were to know the maximum degree of the contention graph of the network, then a slotted ALOHA scheme would expect an average max-min throughput of

$$\begin{aligned}\max_p \min_i p(1-p)^{d_{i,2}} &= \max_p p(1-p)^{d_{\max}} \\ &= \frac{1}{d_{\max}+1}\left(1-\frac{1}{d_{\max}+1}\right)^{d_{\max}} \\ &\approx \frac{1}{\mathrm{e}}\left(\frac{1}{d_{\max}+1}\right) , \end{aligned} \tag{5}$$

where $d_{\max}$ is the maximum degree of the contention graph.

2.3.2 Unslotted ALOHA

To connect slotted and unslotted ALOHA, consider choosing the probability p so that the rate of transmissions over any time period t is λt. That is, τ is a frame duration, that is subdivided into m slots, then the slot size is τ/m and the probability of transmission for that slot is the rate $p=\lambda(\tau/m)$. Then the probability that a node attempts to transmit n times with this fixed-rate transmission probability is

$$\binom{m}{n}(1-p)^{m-n}p^n = \frac{(\lambda\tau)^n}{n!}\left(\prod_{i=0}^{n-1}\frac{m-i}{m}\right)(1-\lambda\tau/m)^{m-n}$$
$$\to \frac{(\lambda\tau)^n}{n!}\mathrm{e}^{-\lambda\tau} \tag{6}$$

as $m \to \infty$. Thus, for a fixed transmission rate, the natural generalization of a slotted ALOHA system to an unslotted ALOHA system allows the nodes to have exponential service times with rate parameter λ rather than geometric service times.

Let τ be the packet duration and consider again the multiple access topology with K fixed transmitters and one receiver. Then the probability of a successful link from the kth transmitter to the receiver is the product of the probability that the kth transmitter emits one packet that occupies the interval $[0, \tau)$, which is $(\lambda\tau)\mathrm{e}^{-\lambda\tau}$, and the probability that the $(K-1)$ other nodes emit no packets that start at any time in the potential collision interval $[-\tau, \tau)$, which is $\mathrm{e}^{-(K-1)\lambda(2\tau)}$. That is, with $\eta = \lambda\tau$, the expected per-link throughput is $\eta\mathrm{e}^{-\eta(1+2(K-1))}$. The choice of $\eta = 1/(2K-1)$ yields the maximum expected per-link throughput of $1/(\mathrm{e}(2K-1))$ and maximum expected sum throughput of $1/\mathrm{e}\,(K/(2K-1)) \approx 1/(2\mathrm{e})$.

Now, consider a general topology, and allow each node to be a transceiver and to have its own packet duration and transmission service rate. Assuming a two-hop interference model, the probability of a successful broadcast transmission from node i to its one-hop neighbors is

$$\underbrace{(\lambda_i\tau_i)\mathrm{e}^{-\lambda_i\tau_i}}_{\text{transmission}}\underbrace{\left(\prod\nolimits_{j\in\mathcal{N}_{i,1}}\mathrm{e}^{-\lambda_j 2\tau_i}\right)}_{\text{reception}}\underbrace{\left(\prod\nolimits_{k\in\mathcal{N}_{i,2}\setminus\mathcal{N}_{i,1}}\mathrm{e}^{-\lambda_k 2\tau_i}\right)}_{\text{contention}} .$$

If all the nodes have the transmission service rate, so that $\lambda_i = \lambda$ for all $i \in V$, then this per-transceiver expected throughput is $\eta_i\mathrm{e}^{-\eta_i(1+2d_{i,2})}$, where $\eta_i = \lambda\tau_i$ and $d_{i,2}$ are the number of nodes within two hops of node i. Observe that, for each node i, this per-node throughput is maximized by choosing $\eta_i = \frac{1}{1+2d_{2,i}}$. Thus, if the nodes are aware of the number of neighbors in their contention graph, then a slotted ALOHA scheme could expect an average max-min throughput of

$$\max_{\eta_1,\ldots,\eta_n}\min_i \eta_i\mathrm{e}^{-\eta_i(1+2d_{i,2})}$$
$$= \min_i \max_{\eta_i} \eta_i\mathrm{e}^{-\eta_i(1+2d_{\max})}$$
$$= \frac{1}{1+2d_{\max}}\frac{1}{\mathrm{e}} \approx \frac{1}{2\mathrm{e}}\left(\frac{1}{d_{\max}+1}\right) , \tag{7}$$

where $d_{\max}$ is the maximum degree of the contention graph. Observe that the choice of $\eta_i = \frac{1}{1+2d_{2,i}}$ not only maximizes the average max-min throughput, but the average sum throughput as well, and that this choice allows users in sparser neighborhoods to transmit longer packets. In particular, with λ as any arbitrary positive transmission rate, the ith node transmits packets of length $\tau_i = \frac{1}{1+2d_{i,2}}\frac{1}{\lambda}$.

Unslotted ALOHA could be thought of as achieving, with contention, a fraction $1/(2\mathrm{e})$ of what can be achieved by a centralized greedy coloring solution. In the same vein, the max-min throughput (7) that is available without convergence to a schedule, but with contention, is 1/e of the max-min throughput (4) that is available after convergence to a schedule in a decentralized manner, but without contention. Further, both the unslotted ALOHA and the coarse desynchronization methods permit nodes within sparse portions of the network to transmit at a higher rate than nodes within denser portions of the network.

Based on these analogies, one can decompose the factor of 1/(2e) in the average max-min throughput (7) of unslotted ALOHA into two components. The factor of 1/2 can be thought of as the price to pay for moving from a centralized to a decentralized solution, while the factor of 1/e can be thought of as the price to pay for moving from a schedule-based solution to a contention-based schedule.

3. IMPLEMENTATION

3.1 OVERVIEW

Desync is an entirely new concept for a link-layer protocol. Unlike traditional TMDA MAC protocols, it requires little overhead, no coordination between nodes, and no time synchronization between nodes. Nodes successfully share the available bandwidth through a process of "desynchronizing" themselves in the given transmission frame. This desynchronization is facilitated through the use of a "fire" message, which is sent during a node's transmission time in the frame. As each node hears other nodes' fire messages, it adjusts its fire time. Ultimately, the algorithm converges to a "desynchronized" state where neighboring nodes have non-overlapping timeslots within the frame.

A conceptual example is shown in Fig. 4. In the first frame, each node has an initial fire time, i.e., A, B, C, D, E. Once each node hears the fire messages of the nodes immediately before and after its fire message, the node makes an adjustment to its own fire time so that the fire time in the next frame, i.e., A', B', C', D', E', is approximately halfway between the fire times of the nodes that fired immediately before and after it. For example, B' is approximately halfway between A and C; C' is approximately halfway between B and D. These adjustments continue in each frame. In the case of a fully connected mesh network, eventually the fire times will be equidistant within the frame.

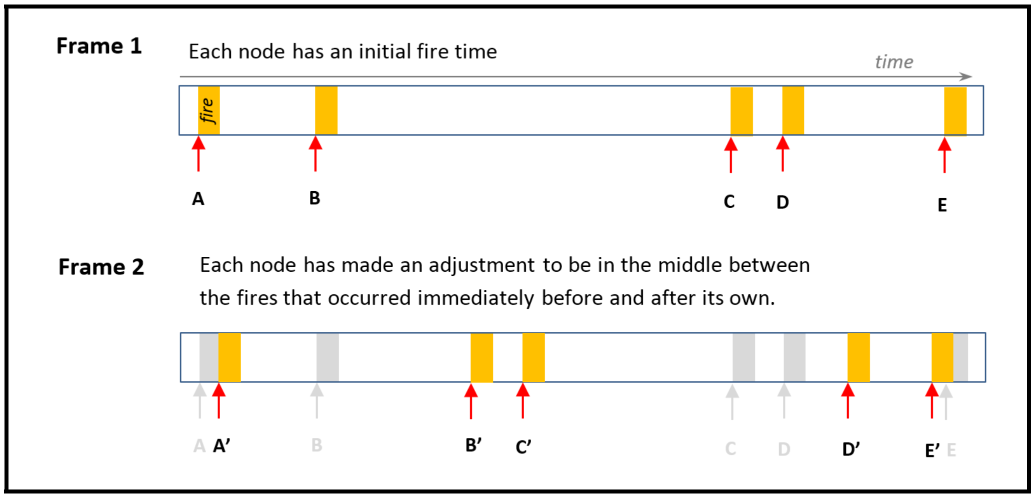

Frame 1
Each node has an initial fire time
time
fire
A
B
C
D
E
Frame 2
Each node has made an adjustment to be in the middle between the fires that occurred immediately before and after its own.
A
A'
B
B'
C'
C
D
D'
E'
E

Figure 4. Desync example.

The adjustment to a node's fire time is based on several parameters, as follows:

T	frame duration; default value is 1.0 seconds
α	an exponential decay factor; typical value is 0.95
$prevFireTime$	fire time immediately before a node's own fire time
$nextFireTime$	fire time immediately after a node's own fire time

When a node adjusts its fire time, it uses the following formula:

$$myFireTime_{new} = T + (1-\alpha) \times myFireTime_{current} + \alpha \times \frac{prevFireTime + nextFireTime}{2} \quad (8)$$

Each node also establishes its data slot based around its fire time. The slot start and slot end are essentially calculated to be in the middle of the time between a node's fire and its neighboring fires.

$$\text{Slot start:} \frac{myFireTime + prevFireTime}{2} \quad \text{Slot end:} \frac{myFireTime + nextFireTime}{2} \quad (9)$$

An example is shown in Fig. 5. There are a few important points to make about the slot. First, the fire time is not necessarily in the middle of the slot. Before the nodes are fully desynchronized (i.e., the algorithm has converged), the fire time will be skewed from the center of the slot. The slot of node C is an example: the fire time is skewed towards Node D. Second, since the edges of the slot are calculated based on the fire times, each node's slot will abut the slot immediately next to it. For example, the end of the slot for node B coincides with the start of the slot for Node C. Finally, because the slots are defined based on the fire times, they will always be mutually exclusive and will not overlap. This allows a node to start to use the slot, without collisions, before the algorithm has converged.

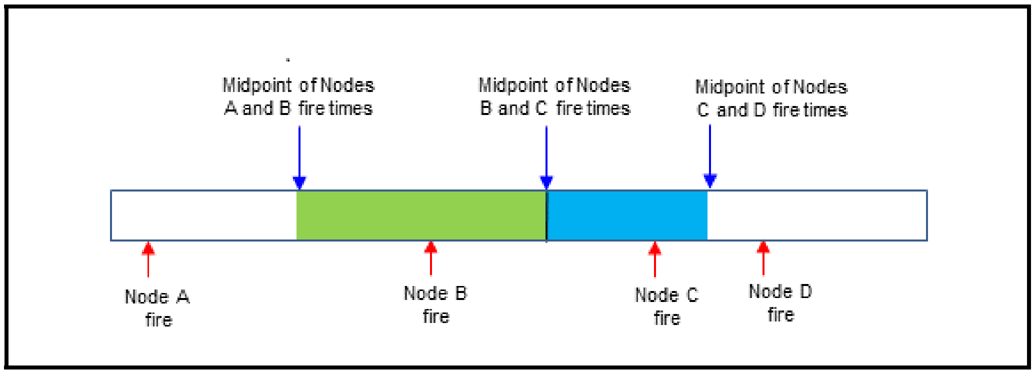

Midpoint of Nodes A and B fire times
Midpoint of Nodes B and C fire times
Midpoint of Nodes C and D fire times
Node A fire
Node B fire
Node C fire
Node D fire

Figure 5. Slot start and end.

More details about the algorithm are available in [1, 11]. The purpose of this section is to describe the implementation of the Desync algorithm in our work. This document addresses various aspects of the protocol for which the details of the design were decided specifically for this project. These are details that are not specified in [1, 11] and are thus left to interpretation. The specific areas for which we have provided additional design are outlined in this section.

3.2 ARBITRARY NETWORK TOPOLOGY

The algorithm described in [1] was originally intended for fully connected mesh networks, and direct application of this algorithm to other network topologies can present challenges. As an example, consider the network shown in Fig. 6. This is a linear network of 4 nodes. Each node can hear only the nodes immediately next to them, e.g., Node C can only hear Node B and D. Suppose the initial fire times randomly chosen are in the order A, B, C, D as shown in the figure. Node A will try to distance its fire time as far from Node B's as possible and Node D will do the same with Node C. Meanwhile, Node B will center its fire time between A and C and Node C will center itself between Node B and D. The resulting state is shown in the figure. Clearly this is not a state that will work. Because Node A does not know about Node C, they each end up with the same fire time and thus have overlapping slots. Likewise Nodes B and D have the same fire time and overlapping slots. When A and C transmit, interference will occur at Node B. When B and D transmit, interference will occur at Node C.

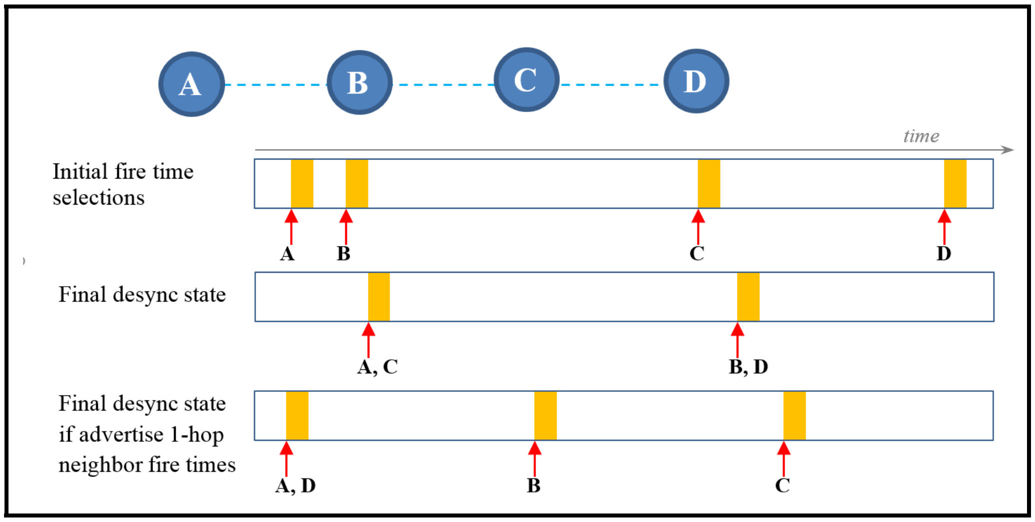

A
B
C
D
time
Initial fire time selections
A
B
C
D
Final desync state
A, C
B, D
Final desync state if advertise 1-hop neighbor fire times
A, D
B
C

Figure 6. Example of Desync in a line network.

The solution to correct this situation is to advertise the fire times of 1-hop neighbors. If this occurs, then Node A would know about Node C and Node D would know about Node B. The resulting fire times are shown in Fig. 6. Nodes A and D have the same fire time and have overlapping slots, while Node B and C have distinct slots. This state will work because Nodes A and D will not cause interference when they transmit at the same time.

Advertising 1-hop neighbor fire times presents two challenges:

- How does the algorithm work when 1-hop neighbors are included?
- How should 1-hop fire times be advertised?

When 1-hop fire times are advertised, the algorithm as described must change because there is no guarantee that the "previous" and "last" fire times that a node hears are the fire times that occur immediately before and after its own. Figure 7 shows an example from the perspective of Node B. In this example, the fire message that Node B hears immediately before its fire message is that of Node A, and the fire message it hears immediately after its fire message is that of Node C. However, Node B needs to use the fire times of Node F and Node D as its previous and next fire times for the slot adjustments. Node B will not know about Node F or Node D until it receives the fire messages from Node E and Node C, respectively. Thus, Node B cannot make adjustments based on when it receives the fire message immediately after its own as described in the original algorithm.

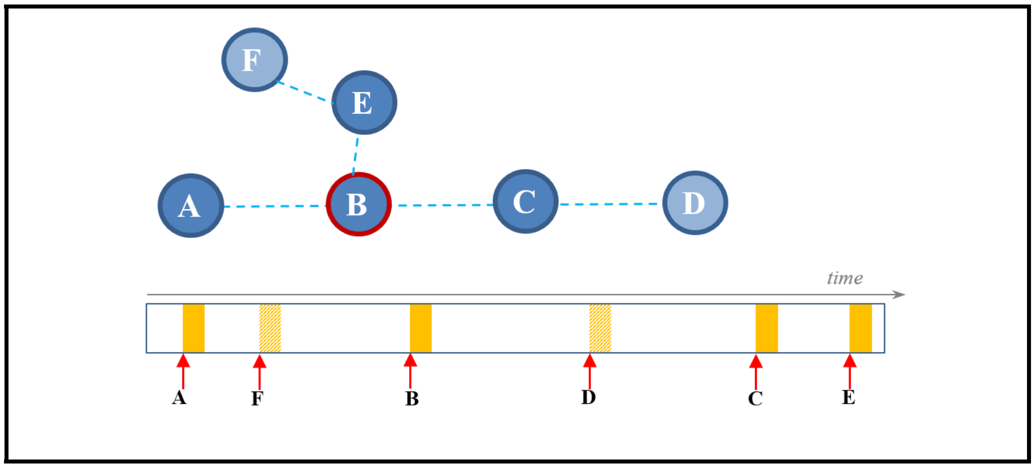


Figure 7. Example of fire times in a non-mesh network.

3.2.1 Algorithm Design

In order to take into account the 1-hop neighbor fire times, the Desync algorithm works as follows.

- During the first two frames at startup, nodes send a fire message but do not calculate slot start or end and do not make any adjustments to their fire times other than simply moving the fire time out by the time period (length of frame):

$$myFireTime_{new} = T + myFireTime_{current}. \tag{10}$$

- The fire message that each node sends includes its own fire time and the fire times of its 1-hop neighbors. Details of how this is done is described in a later section; for now just assume this is done and that it works.
- As nodes receive fire messages from neighbor nodes, each node builds a list of all known fire times. This list includes the fire times of both 1-hop and 2-hop neighbors, and there is no differentiation as to whether a fire time is associated with a 1-hop or 2-hop neighbor.

- After two frames have elapsed, a node sends its fire message, then calculates its slot start and end for the next frame, and finally makes adjustments to its fire time for the next frame. These calculations are done as follows:
  - Scan the list of known fire times and identify the fire times that occur immediately before and immediately after the node's own fire time. Let's call these previous fire and next fire, respectively. Note that these can be 1-hop or 2-hop neighbor fire times.
  - Slot start and end times are calculated as Eqn. (8).
  - The next fire time is calculated as Eqn. (9).
- After making fire time and slot start/end calculations, delete old fire times from the list of known fire times. A fire time is considered "old" and is deleted if it is less than the node's fire time used for the adjustments. Note that known fire times greater than the node's fire time, including the time used as the "next fire" remain on the list until the next adjustment.

Figure 8 shows the calculations that are performed in each frame. Notice when the next fire time calculations are performed during this process. Instead of performing the calculations when a node receives the fire message immediately after its own, a node performs the calculations at its fire time, thus allowing it to gather known fire times over the previous frame. Of course the other difference is that the calculations are based on the list of all known fire times that a node has accumulated since the last time it made an adjustment and not just the fire time a node hears immediately before and after its own.

Another item to note is that the adjustments are made based on the node's own fire time and the fire times heard in the previous frame(s) and that calculations of the slot start/end and fire time are for the next frame not the current frame. The reason that the information from the previous frame is used instead of using the current frame is that the node will not have heard the full set of information for the current frame when the adjustment is made.

Finally, one very important aspect of this process is that nodes wait two full frames at initialization before making any adjustments. This allows the node to fully discover the 2-hop neighbors so that the first adjustment that it makes after its initial fire time will be an adjustment based on full network knowledge. If nodes adjusted fire times after just one frame, these would be based on partial information about the 2-hop network and would be poor adjustments that could potentially cause the convergence to a desynchronized state to take longer. Waiting a full two frames to have complete 2-hop neighbor information allows the first adjustment to be a smart adjustment.

3.2.2 Advertising Neighbor Fire Times

In order for the algorithm to work, a node must advertise the fire times of its 1-hop neighbors as part of its fire message. The details of how this is done are discussed later. For now, the reader can just accept that at a high level, each node is able to advertise the fire times of its 1-hop neighbors as part of its fire message and that it does so by advertising an integer number of symbols (based on the symbol rate) to express the 1-hop neighbor fire times.

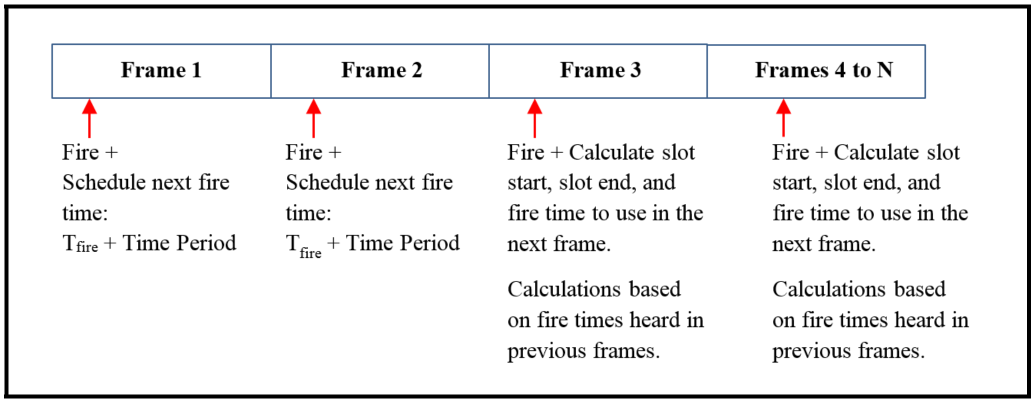

Frame 1
Frame 2
Frame 3
Frames 4 to N
Fire + Schedule next fire time: T_{fire} + Time Period
Fire + Schedule next fire time: T_{fire} + Time Period
Fire + Calculate slot start, slot end, and fire time to use in the next frame.
Calculations based on fire times heard in previous frames.
Fire + Calculate slot start, slot end, and fire time to use in the next frame.
Calculations based on fire times heard in previous frames.

Figure 8. Fire time and slot calculations in non-mesh networks.

3.2.3 Tracking Neighbor Fire Times

Each node must maintain two fire time lists. First, a list of 1-hop neighbor fire symbol offsets is maintained. This is a list of the 1-hop neighbor fire information that the node needs to advertise in its fire message.

- The values are integer number of symbols that represent the difference in time between the node's own fire time and the 1-hop neighbor's fire time.
- Values are added to the list when a node receives a fire message. The value added is the fire time advertised as the fire time of the node that sent the message.
- The list is cleared immediately after a node sends its fire message.
- Thus, the processing is as follows: send fire message, immediately clear list of 1-hop neighbor fire symbol offsets, gather new list during the next frame, and repeat.

Additionally each node maintains a list of all known fire times for 1-hop and 2-hop neighbors. This is summarized as follows:

- The values are doubles representing the fire time of each node relative to the node's own clock.
- There is no differentiation or identification of whether the fire times are for 1-hop or 2-hop neighbors.
- Values are added to the list when a node receives a fire message. The values added include the fire time advertised as the fire time of the node that sent the message as well as the fire times that the node advertised for its 1-hop neighbors.
- The list is cleared of values immediately after a node sends its fire message; however, not all values are removed. The values removed are those that are less than and excluding the fire

time that was used as "next fire" during the slot end and fire adjustment calculations. Thus, the "next fire" and any fire times greater than it will remain on the list until the next cycle.

- The processing is as follows: send fire message, immediately remove fire times that are no longer needed from list of known fire times, add more fire times to the list during the next frame, and repeat.

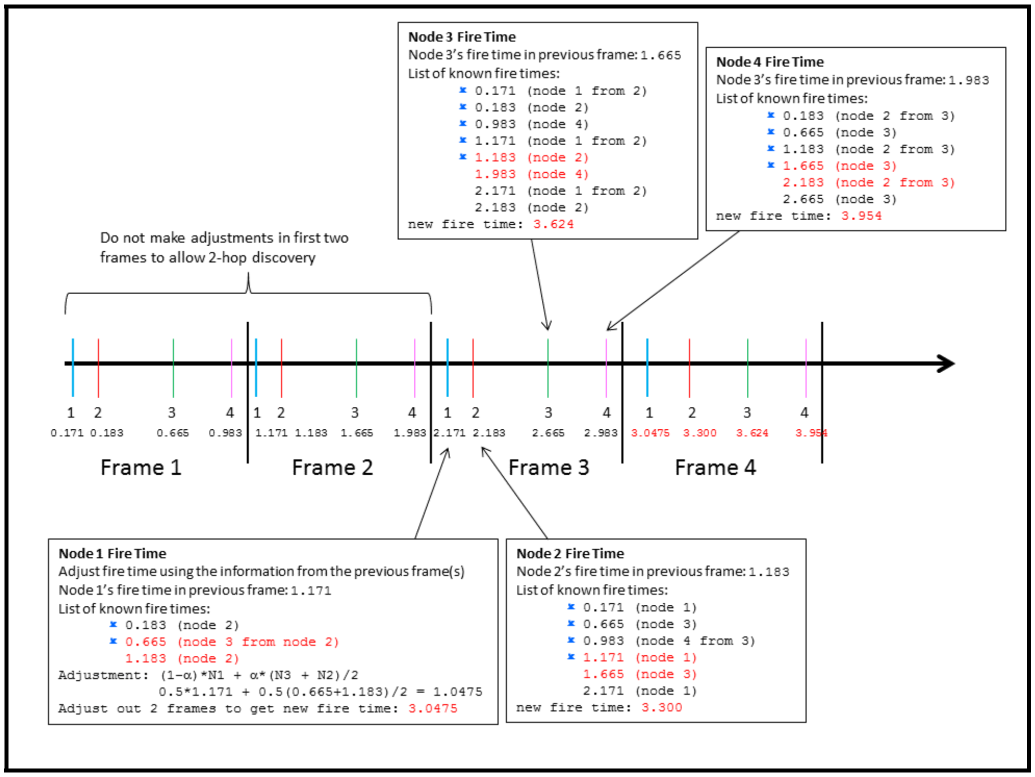
Figures 9 and 10 show an example of how the algorithm works using a four-node line topology. In Figure 9, during the first frame, the nodes have an initial fire time. When the fire message is sent, the nodes do not set a slot start and slot end but do adjust their fire time by simply moving out one time period which in this case is 1.0 second. During these two frames, each node accumulates a list of known fire times. In the third frame, the nodes calculate slot start and end (not shown in the figure) and also adjust their fire times. Taking Node 3 as an example, in frame 3, Node 3 uses its fire time from frame 2 (the previous frame), which is 1.665, as the basis for adjustments. Node 3 scans its list of known fire times and determines that the fire times immediately before and after its fire time of 1.665 are 1.183 and 1.983, respectively. These fire times are highlighted in red in the figure. Node 3 makes the adjustment to its fire time using the formula previously shown and determines its new fire time for Frame 4 to be 3.624.

Node 1 provides an example of the use of two-hop neighbor information. When Node 1 adjusts its fire time in frame 3, it uses the fire times of Node 3 and Node 2. Node 3 is a 2-hop neighbor and Node 1 knows about its fire time because it was advertised by Node 2.

The figure provides a good example of why the node's fire time from the previous frame is used to adjust fire times and calculate the slot start/end instead of using the node's fire time in the current frame. Again using Node 3 adjusting its fire time in frame 3 as an example, if Node 3 used the fire time for the current frame, 2.665, as the basis for the calculations, it has no fire time value to use as the "next fire" because it has yet to receive any fire messages after its own. (After all, the adjustment is being made just after Node 3 sent its own fire message.)

When each node clears its list of known fire times, it deletes the fire times that it considers "old." Old fire times are those that are less than the node's fire time that was used for the adjustment. For Node 3, this will be the fire time less than 1.665. The times that are deleted from each nodes list as old after the fire adjustment is made are shown in the figure with a blue x in front of them. Notice that the fire time used for "next" is not removed as old.

Figure 10 shows the adjustments for frame 4 and the resulting fire times for frame 5. Looking at the adjustments made in frame 4 provides an example to show why only "old" fire times are deleted from the list of known fire times. Again using Node 3 as an example, in Figure 9, the fire times that remain on the list of known fire times are 1.983, 2.171, and 2.183. In Figure 10, in frame 4, Node 3 uses 2.183 as the "previous fire" when making its fire time adjustments. Though Node 3 only needed the last time on the list of known fire times, there are cases in which this may not be true. In particular, in a two-node network, the time used as "next fire time" in one frame is also used as the "previous fire time" in the next frame. Thus, the times that are deleted from the list exclude the time used for "next" and greater times.



Node 3 Fire Time
Node 3's fire time in previous frame: `1.665`
List of known fire times:

```
        ⌘ 0.171 (node 1 from 2)
        ⌘ 0.183 (node 2)
        ⌘ 0.983 (node 4)
        ⌘ 1.171 (node 1 from 2)
        ⌘ 1.183 (node 2)
          1.983 (node 4)
          2.171 (node 1 from 2)
          2.183 (node 2)
new fire time: 3.624
```

Node 4 Fire Time
Node 3's fire time in previous frame: `1.983`
List of known fire times:

```
        ⌘ 0.183 (node 2 from 3)
        ⌘ 0.665 (node 3)
        ⌘ 1.183 (node 2 from 3)
        ⌘ 1.665 (node 3)
          2.183 (node 2 from 3)
          2.665 (node 3)
new fire time: 3.954
```

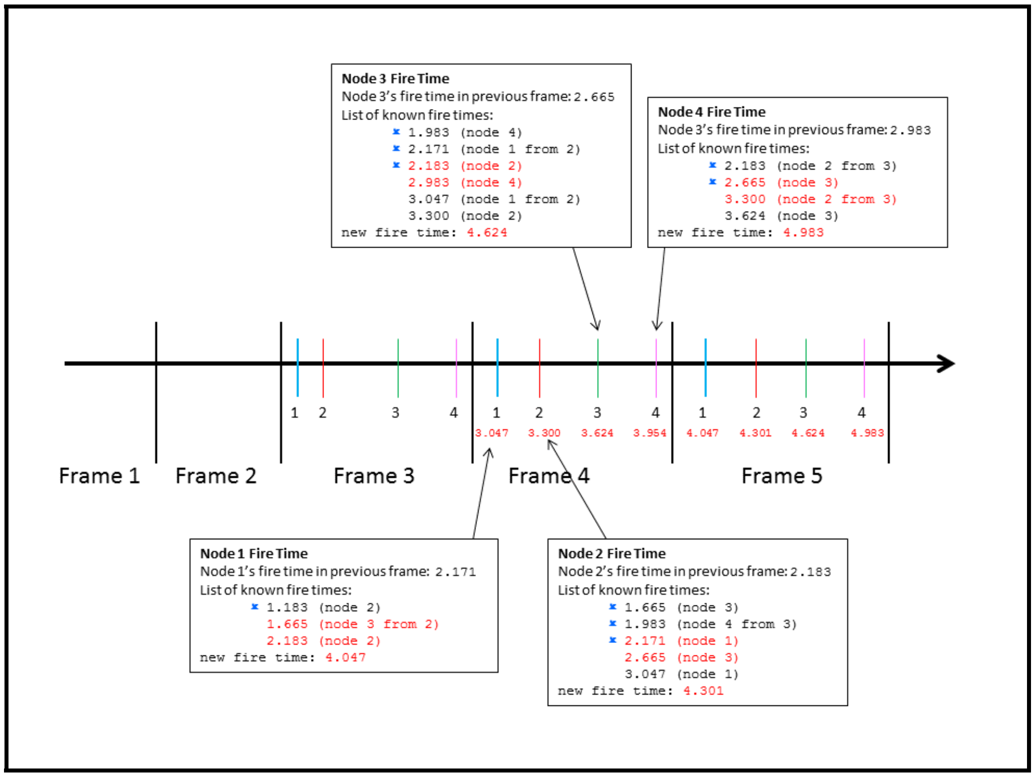
Node 1 Fire Time
Adjust fire time using the information from the previous frame(s)
Node 1's fire time in previous frame: `1.171`
List of known fire times:

```
        ⌘ 0.183 (node 2)
        ⌘ 0.665 (node 3 from node 2)
          1.183 (node 2)
Adjustment: (1-α)*N1 + α*(N3 + N2)/2
            0.5*1.171 + 0.5(0.665+1.183)/2 = 1.0475
Adjust out 2 frames to get new fire time: 3.0475
```

Node 2 Fire Time
Node 2's fire time in previous frame: `1.183`
List of known fire times:

```
        ⌘ 0.171 (node 1)
        ⌘ 0.665 (node 3)
        ⌘ 0.983 (node 4 from 3)
        ⌘ 1.171 (node 1)
          1.665 (node 3)
          2.171 (node 1)
new fire time: 3.300
```

Figure 9. Example of Desync in non-mesh networks - Part 1.



```
Node 3 Fire Time
Node 3's fire time in previous frame: 2.665
List of known fire times:
      ✗ 1.983 (node 4)
      ✗ 2.171 (node 1 from 2)
      ✗ 2.183 (node 2)
        2.983 (node 4)
        3.047 (node 1 from 2)
        3.300 (node 2)
new fire time: 4.624
```

```
Node 4 Fire Time
Node 3's fire time in previous frame: 2.983
List of known fire times:
      ✗ 2.183 (node 2 from 3)
      ✗ 2.665 (node 3)
        3.300 (node 2 from 3)
        3.624 (node 3)
new fire time: 4.983
```

```
Node 1 Fire Time
Node 1's fire time in previous frame: 2.171
List of known fire times:
      ✗ 1.183 (node 2)
        1.665 (node 3 from 2)
        2.183 (node 2)
new fire time: 4.047
```

```
Node 2 Fire Time
Node 2's fire time in previous frame: 2.183
List of known fire times:
      ✗ 1.665 (node 3)
      ✗ 1.983 (node 4 from 3)
      ✗ 2.171 (node 1)
        2.665 (node 3)
        3.047 (node 1)
new fire time: 4.301
```

Figure 10. Example of Desync in non-mesh networks - Part 2.

The subsection below on fire messages discusses the details of the fire message. More information about how the neighbor fire times are calculated is included in that section.

The advertisement of the 1-hop neighbor fire times is optional. For example, in a mesh network, it is not necessary to advertise 1-hop neighbor fire times. When 1-hop neighbor fire times are not advertised, the algorithm operates in the same fashion as shown in Figures 9 and 10. That is, there is no special processing related to whether or not 1-hop neighbor fire times are advertised.

Finally, when a node adjusts its fire time, it is possible that it does not have enough information to do so (for example, a fire message from a neighbor was dropped in a lossy environment). When this situation occurs, a node adjusts its fire (and slot start/end) by simply moving them out by the time period.

3.3 802.15.4 PHYSICAL AND MAC LAYERS

In order to test the Desync protocol, we will use 802.15.4 physical layer [2]. The 802.15.4 physical layer provides several operating modes. Table 2 shows the values used for data and fire messages. Note that a more robust transmission mode is available as an option for the fire messages in which they are sent at a lower data rate and a lower symbol rate.

TABLE 2

802.15.4 Physical Layer Attributes

Attribute	**Data Messages**	**Fire Messages** Standard	**Fire Messages** Robust
Data rate (kbits/sec)	250	250	40
Frequency (MHz)	913	913	913
Bandwidth (kHz)	600	600	600
Modulation	QPSK	QPSK	BPSK
Symbol rate (ksymbols/sec)	62.5	62.5	40
Transmit power (Watts)	0.2	0.2	0.2

Because we are using the 802.15.4 physical layer, the packets must use the 802.15.4 physical layer format. Figure 11 shows the 802.15.4 physical packet format, which consists of a 6-byte header and a variable size payload. The maximum payload size is 127 bytes because the physical header length field is 7 bits, which allows a maximum value of 127. Note that the payload at the physical layer includes the MAC header.

Even though Desync is a MAC layer protocol, we have chosen to reuse the 802.15.4 MAC layer header for Desync so that it can be compared to performance results against 802.15.4 based on the same amount of packet overhead. Figure 12 shows the 802.15.4 MAC header format, which has a total of 9 bytes.

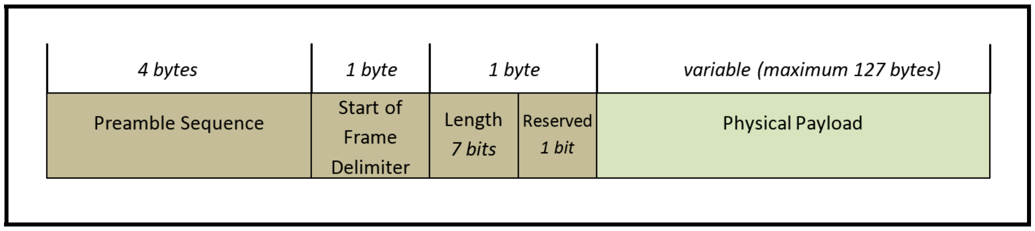


Figure 11. 802.15.4 Physical header format.

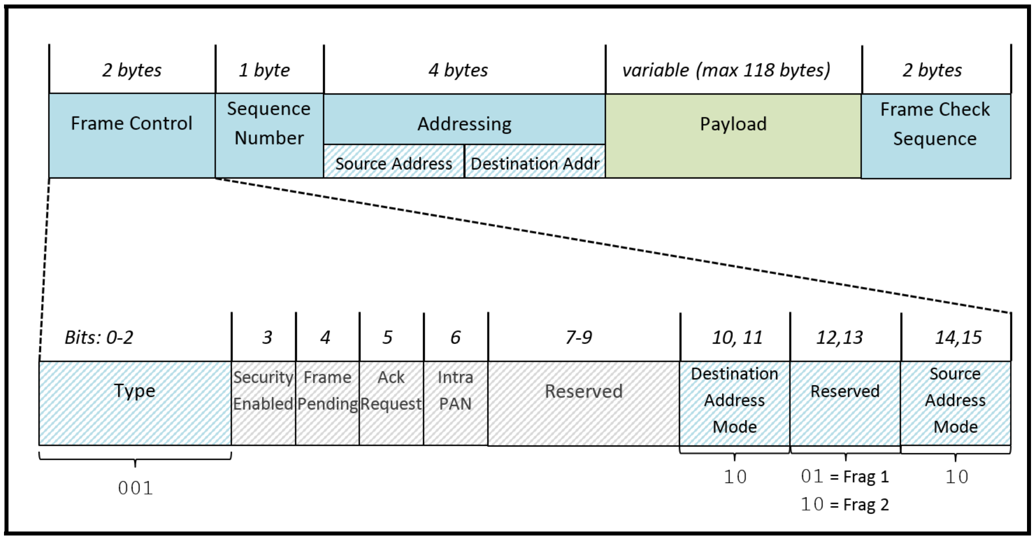


Figure 12. 802.15.4 MAC header format.

The **Frame Control** field is a total of 2 bytes with the following sub-fields:

- Type field (bits 0-1) is used to indicate the type of packet. A value of 001 is used for data packets. Note that the first bit is 0.
- The Security Enabled, Frame Pending, Ack Request, Intra PAN, or Reserved fields (bits 3-9) will not be used in this effort.
- Source (bits 14,15) and Destination (bits 10,11) Address Mode fields are used to indicate the type of addressing used in the MAC header. A value of 10 is used to indicate that the short (2 byte) addresses are used.
- Reserved field (bits 12, 13) will be used by Desync to indicate if the packet is a fragment and if so which fragment. A value of 01 is used for fragment 1, and 10 is used for fragment 2.

The **Sequence Number** field is a 1-byte field with a unique number that serves as a sequence for the packet. This field will be used by Desync to identify fragments that belong to the same packet.

The **Addressing** field of the MAC header is a field that has several options that can be set by the user. We have chosen the following options, which result in our MAC header having just 4 bytes for addressing:

- Short addresses (2 bytes each) instead of long addresses (8 bytes each)
- No use of PAN (Personal Area Network) identifiers in the addressing. PAN IDs can be 0 or 2 bytes, and we have chosen to not include them in the MAC header because we are not deploying more than a single PAN in our network.

The **Frame Check Sequence** is a 2-byte CRC over the contents of the MAC portion of the frame.

3.4 SLOT FORMAT

As described above, each node determines its slot based on its fire time and the fire times of the other nodes in the network. This slot is then used to transmit the fire message and data packets. The details of how the slot is actually used to transmit data are not specified in [1], and thus we have designed a slot format to support data transport. Figure 13 shows this slot format. The fire message is at the start of the slot, a guardband is at the end of the slot, and data packets are sent during the remaining portion of the slot. There is support for fragmentation of data packets to ensure that data is transmitted during as much of the slot time as possible.

There are several specific items to explain in regards to how this slot format was chosen, including the use of a guardband at the end of the slot, packet fragmentation to ensure that the slot utilization is maximized, and sending the fire message at the start of the slot instead of the middle of the slot.

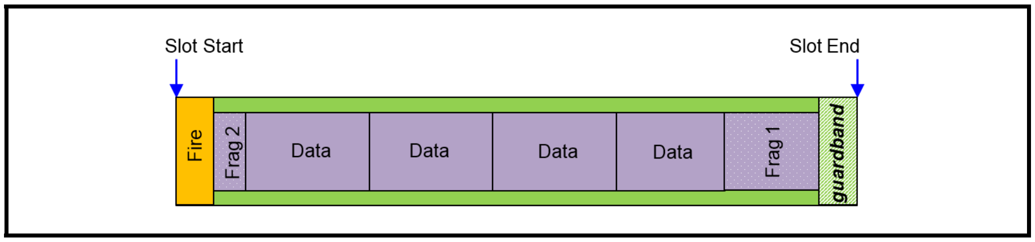


Figure 13. Slot format.

3.4.1 Guardband

As explained earlier, the slot boundary calculations result in slots that abut each other; that is, the end of one slot coincides with the start of the next slot. If a node is able to transmit up to the end of its slot and then the next node is allowed to transmit at the start of its slot (which is the same time as the previous slot end), interference can occur between the two transmissions. Figure 14 shows an example. Node B transmits at the end of its slot. The transmission fits in the slot, but there is propagation delay before the transmission from Node B reaches Node A such that the transmission end at Node A is after the slot end for Node B. Node C begins a transmission immediately at the start of its slot. Again, there is propagation delay before the transmission from Node C reaches Node A. Because Node B is further from Node A than Node C, there is interference in the transmissions at Node A.

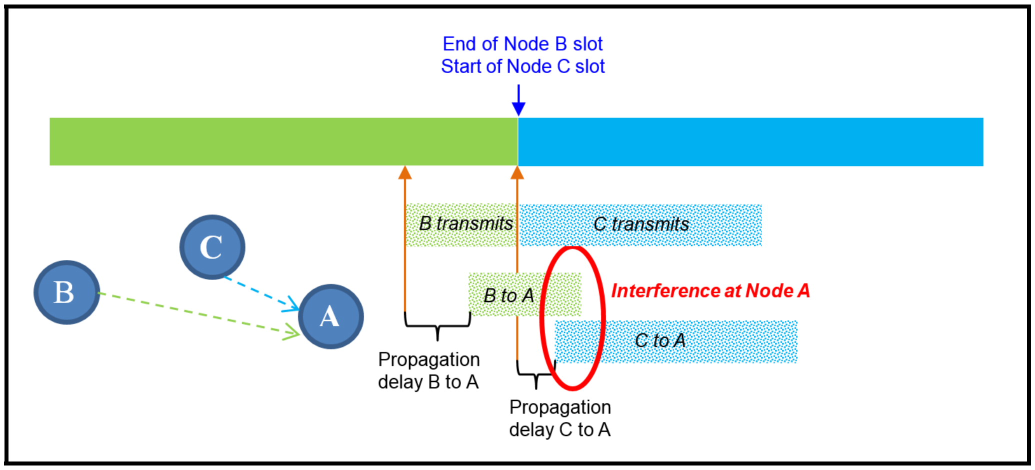


Figure 14. Interference when slot end and slot start coincide.

In order to avoid this type of interference, there needs to be a guardband at the end of each slot during which the node cannot transmit. The purpose of the guardband is to prevent a node's transmission from going beyond the end of its slot. Thus, the guardband needs to be sized based on the maximum possible propagation delay between any two nodes. Figure 15 shows the same example but with the addition of a guardband. Because Node B will consider the time in the slot for the guardband as time during which it cannot transmit, its transmission will never interfere with that of Node C.

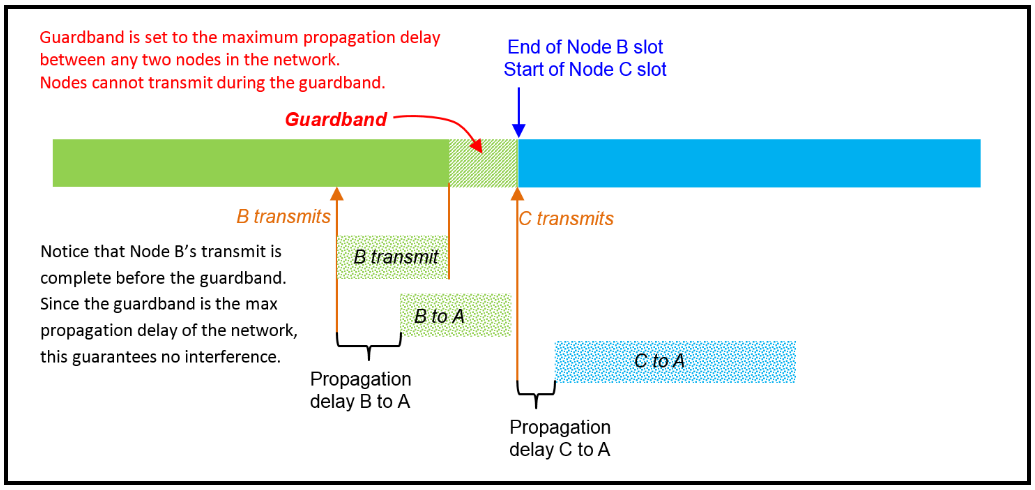

Guardband is set to the maximum propagation delay
between any two nodes in the network.
Nodes cannot transmit during the guardband.
End of Node B slot
Start of Node C slot
Guardband
B transmits
C transmits
Notice that Node B's transmit is complete before the guardband. Since the guardband is the max propagation delay of the network, this guarantees no interference.
B transmit
B to A
C to A
Propagation delay B to A
Propagation delay C to A

Figure 15. Guardband added to slot end to prevent interference.

3.4.2 Packet Fragmentation

One aspect of the slot format is to ensure that the slot time is used efficiently for data transmission. It is not very likely that packets will perfectly fit into the slot. It would be nearly impossible to predict packet size and slot size in order to maximize slot usage, and without the ability to do so, slot usage could result in a worst case scenario in which almost half of the slot goes unused. For example, suppose the slot size allows 1.99 packets to fit. This would mean that only 1 packet could actually be sent in the slot, wasting just about half of the slot time. The solution is to fragment the packets.

Figure 16 shows an example of the fragmentation. For the slot in frame N, there is enough time remaining in the slot to send a portion of a data packet, thus 100% of the slot is used in frame N. In the next frame, $N + 1$, the node transmits the second fragment of the packet followed by other data packets. Fragment 2 is the first data that is transmitted in the frame and takes priority over any other data packets. In frame $N + 1$, there is not enough time left in the slot after the last data packet is sent to allow the next packet to be fragmented, so that portion of the slot would be unused. The smallest size packet that can be transmitted as a fragment is 16 bytes. This equates to 1 byte of data payload with 6 bytes PHY header and 9 bytes of MAC header. If the amount of time remaining in the slot is too small to transmit 16 bytes, then that portion of the slot will go unused. Otherwise, the slot will end with the transmission of fragment 1 of a packet.

3.4.3 Fire Message at Slot Start

Unfortunately, this plan for ensuring that the slot is used as much as possible has one problem. It does not take into account the fact that the fire message is supposed to be sent in the middle of

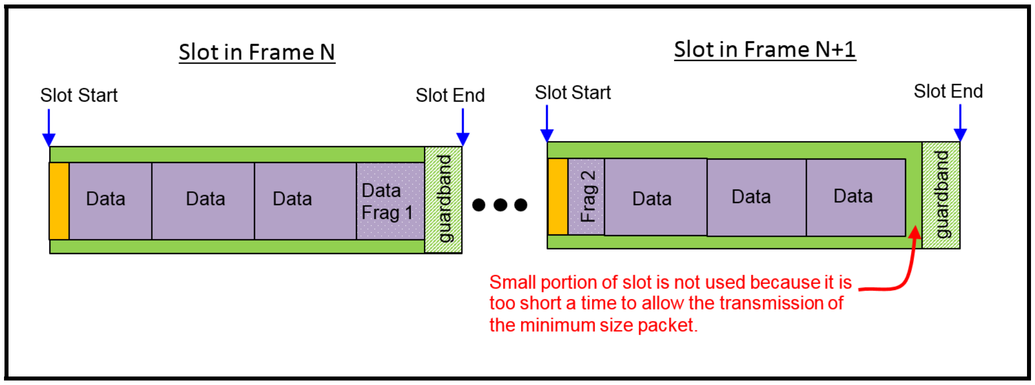


Figure 16. Data packet fragmentation.

the slot. The fire message needs to be sent at a specific time so that receiving nodes can determine how to establish their fire time and their slot boundaries. This adds some complexity to the plan for using fragmentation to ensure that the slot is used most efficiently. Figure 17 shows an example. In the figure, data messages are sent when the slot starts, but after several are sent, the next data message will not fit into the slot time available before the fire message needs to be sent, which results in wasted time in the slot. One solution is to perform fragmentation of the packet in the middle of the slot as shown in Figure 17. Though this solution results in packing data into the slot, it also results in additional fragmentation. Potentially there can be 4 fragments in each frame, as shown in Figure 17. Each packet that is fragmented suffers the penalty of an additional set of PHY and MAC headers; so though this works, a better solution would be to avoid fragmenting in the middle of the slot.

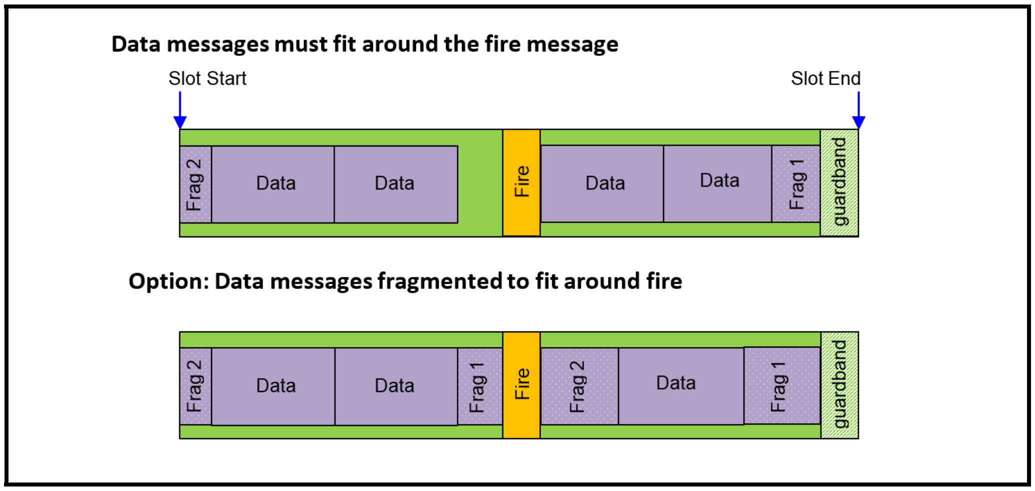


Figure 17. Data transmission with fire message.

Our solution is to move the transmission of the fire message to the start of the slot and have the fire message carry information as to the amount of time it was moved in the slot so that other nodes can determine its fire time.

3.5 FIRE MESSAGES

The fire message is a unique message that is transmitted once per frame by each node during the nodes slot time. We initially had some ideas that perhaps we could eliminate the fire message as a unique message and simply piggyback on data messages transmitted in the slot. Ultimately we decided against this for several reasons. First, the fire message needs to act as an advertisement for the specific time in the slot that is the fire time, and it could become difficult to guarantee that a data message is sent at the fire time. What happens if there is time in the slot before the fire message but not enough time to send a full packet? Do we let that time go unused or fragment the packet? If we fragment the packet, what happens if the rest of the packet does not fit into the slot that remains after the fire time? Though there are ways to solve these issues, they tend to over-complicate the protocol. Secondly, what happens if a node does not have any data to send during its slot? That situation requires that there be a unique fire message. Finally, a node may need to advertise the fire times of its 1-hop neighbors, which also requires a unique fire message. Because a unique fire message is required to support the case when a node has no data to send and the case when a node must advertise its 1-hop neighbors, it seemed most reasonable to then always use a unique fire message.

The format for the fire message is shown in Fig. 18. The fire message is a minimum of 8 bytes and a maximum of 127 bytes. It consists of the required 6 bytes of 802.15.4 physical layer header; 2, 3, or 4 bytes of payload; and an optional field for advertising 1-hop neighbor fire times. The payload is divided into three fields:

- 1 bit Type Flag that is always set to a value of 1
- Bits for advertising the node's own fire time as a Symbols Offset (15 bits, 23 bits or 31 bits)
- Variable-sized field with 1-hop neighbor symbol offsets

The fire message will always end on a byte boundary, and thus the bits for symbol offset are the number of bits needed to bring the fire payload to the byte boundary for 2, 3, or 4 bytes total.

The Type Flag field is used to indicate that the message is a fire message. Since a data message will have the first 3 bits as "001," the fire message must have a 1 in the first bit so that the receiver will know that it is a fire message.

The Symbol Offset field is used to indicate how far in the slot the node's own fire message was moved. As explained above, the fire message is supposed to be sent at the fire time that is in the middle of the slot. As explained previously, we have moved the fire message to the start of the slot and use the content of the fire message to convey the amount of time associated with this shift. The Symbol Offset field is used to carry the number of symbols based on the symbol rate

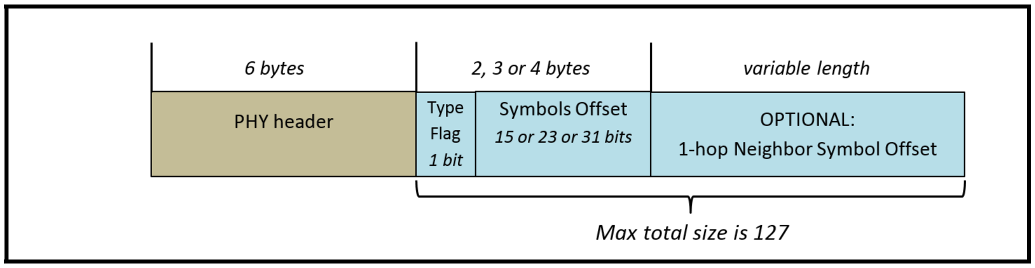


Figure 18. Fire message format.

of the transmission that the fire message was moved in the frame. The number of symbols is used instead of the absolute time because the number of symbols is an integer and thus can be a more accurate representation of the time difference in the available bits.

The fire time offset is determined as follows:

1. Calculate the time difference between the start of the slot and the fire time:
   ΔT_{fire} = (Fire Time) - (Slot Start Time)
2. Calculate the integer number of symbols that represents this amount of time offset:
   Symbol Offset = Integer($\Delta T_{fire} *$ Symbol Rate)
3. Calculate the non-integer amount of time associated with the remainder of the symbol calculation:
   Time Remainder = ΔT_{fire} - (Symbol Offset)/(Symbol Rate)
4. Schedule the fire message to be sent at the slot start, plus this non-integer time remainder:
   Fire Send Time = Slot Start Time + Time Remainder

Figure 19 shows an example to illustrate these values. The process for determining the fire time and fire message offset is as follows. First, $\Delta T_{fire} = 1.02826 - 1.003456 = 0.024808$ seconds. Second, the integer number of symbols that represent this time offset is Integer$(0.024808 * 62500) = 1550$. Third, the remainder is $0.024808 - 1550/62500 = 0.000008$ seconds. Finally, the fire message is scheduled to be sent at $1.003456 + 0.000008 = 1.003464$ seconds.

There are a few items to note about these calculations. First, with a symbol rate of 62500, the Time Remainder will always be less than 16 microseconds, since it is based on a fractional portion of a symbol and thus will always be less than 1 symbol (1 symbol is 16 microseconds). Also, the portion of the slot for the Time Remainder will be unused slot time. With a maximum value of 16 microseconds, this amount of unused slot time will have little impact. For example, suppose there are 20 nodes sharing a 1.0 second frame. The slot size will converge to 0.05 seconds. Sixteen microseconds is just 0.032% of the total slot time.

Finally, during the startup phase, a node does NOT offset its fire message to the start of its slot because there is no slot defined during startup until a node makes it first adjustment and slot

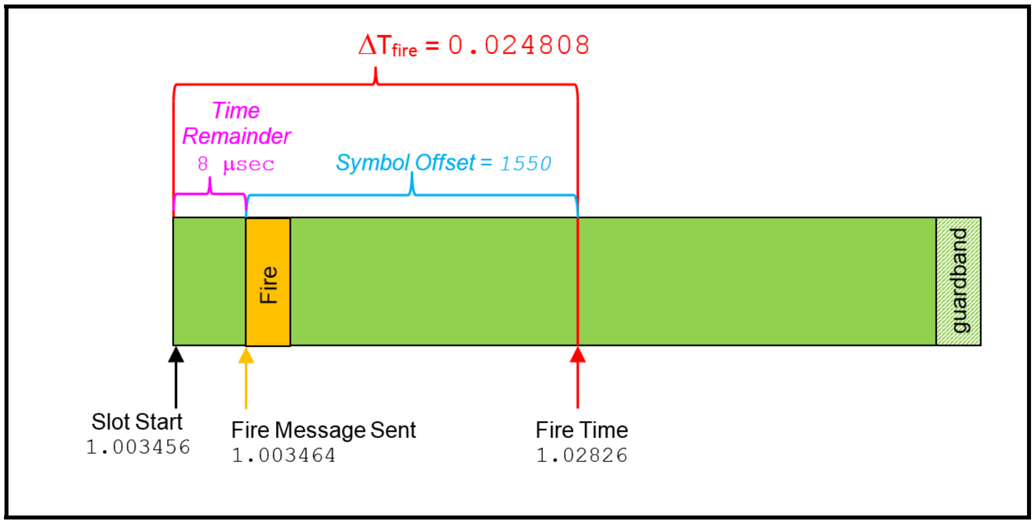


Figure 19. Fire message offset.

start/end calculations. Once the slot is established, even if the slot boundaries are still changing, the node will offset its fire message to the start of the slot.

The number of bits needed for the symbol offset is based on the time period. The number of symbols that must be expressed is the difference between the slot start and the fire time. The upper limit to this value is the time period, and thus the time period is used to determine the number of bits needed. The size of the fire message is determined as follows:

1. Calculate the integer number of symbols for the maximum amount of time to be expressed in the fire message: Max # symbols = Integer(Time Period * Symbol Rate)
2. Calculate the number of bits needed to express this number of symbols:

$$\text{Number of bits} = \left\lceil \frac{\log_{10} \text{Max number of symbols}}{\log_{10} 2} \right\rceil$$

3. Add 1 bit to the # bits for the 1-bit flag that starts the fire message: Total # bits = # bits + 1
4. Round the Total # bits up to the next byte boundary: Fire Size Bits = ceil(Total # bits/8)

For example, if the time period is 1.0 second and the fire symbol rate is 62500, then the Max # symbols is 62500, the number of bits needed to express it is 16, adding 1 bit for the flag, the total number of bits is 17, and rounding up to the byte boundary results in a fire message that is 24 bits.

Table 3 shows the maximum time period for a given size in bytes of the fire message. For typical values of time period, the Symbols Offset field will be a total of 3 bytes and would likely not exceed 4 bytes.

TABLE 3

Maximum Time Period for Fire Byte Size, at 62500 Symbol Rate

Fire Message Size (bytes)	**Symbol Offset** (bits)	**Max # of symbols**	**Max Time Period** (sec)
2	15	32,767	0.52
3	23	8,388,607	134
4	31	2,147,483,647	34,359

3.6 DETERMINING FIRE TIME WHEN RECEIVING A MESSAGE

When a node receives a fire message from a neighbor, it must store the fire time of the node that sent that message. But how does the node determine the fire time of the neighbor?

First, the fire time has to be relative to the receiving node's clock. Because nodes do not have to be synchronized in time, a node can't embed its own fire time in the fire message for the receiving nodes to use as fire time because it may have a different time base than the nodes that receive the fire message. Thus, a node must use its own clock to determine the fire time of the sending node.

Next, when a node receives a fire message from a neighbor, does it consider the reception start or reception end as the time basis? Figure 20 shows an example packet reception to consider. The sender begins sending the message, but the receiver does not start reception until the first bit has reached its receiver, which will occur after the propagation delay. Propagation delay is a function of the distance between the nodes and the speed of light (distance/speed of light). Reception end occurs when the final bit is received, which will occur after the transmission delay, which is a function of the size of the message and the media data rate (size/data rate). Finally, there could also be some processing delay associated with receiving the packet, processing it, and then passing it to the MAC.

In general, propagation delay would be fairly small compared with transmission delay. For example, a 9-byte packet sent at a data rate of 250,000 bits per second to a node 3 km away would have a propagation delay of 10 μsec and a transmission delay of 288 μsec. The processing delay will depend on the specific radio system.

In the case of reception start, this would likely require that the radio pass the timestamp as "metadata" with the received packet when it provides the packet to the MAC. In the case of reception end, the MAC could use its own clock to timestamp when the packet was received by the MAC, but this timestamp would include the processing delay that the packet experienced. Ultimately, we have chosen to use the packet reception end as the basis for determining a neighbor's fire time.

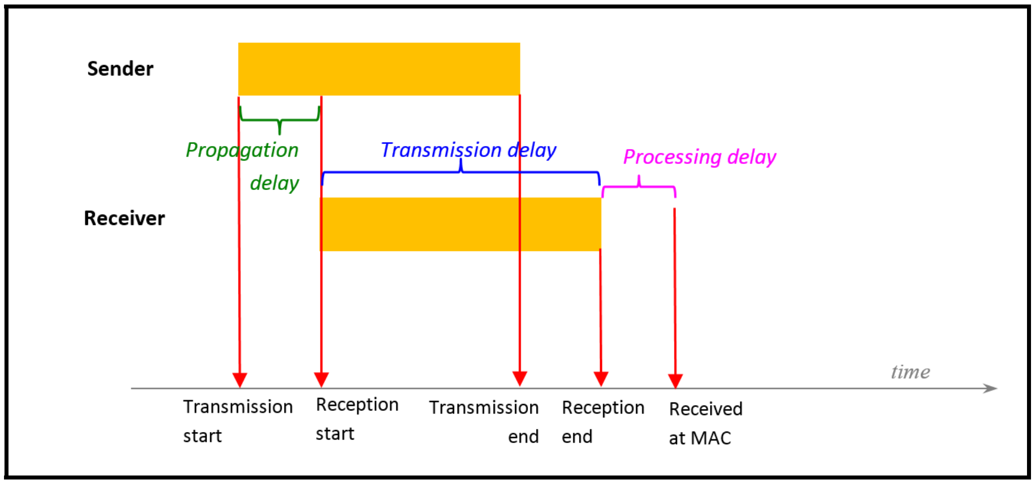


Figure 20. Packet transmission and reception.

Finally, recall that the fire message is not sent at the actual fire time but is sent at the start of the slot and includes the number of symbols of offset. This needs to be taken into account when calculating a node's fire time when a fire message is received.

Using the packet reception end as the basis for calculating fire time and taking into account propagation delay and the symbol offset, the fire time of a neighbor node is calculated as follows:

- When the Desync MAC receives a fire message from a neighbor, get the node's current time at which the message was received at the MAC layer.
- Adjust the time received for transmission delay by subtracting transmission delay from the receipt timestamp, where transmission delay is message size (typically 72 bits) divided by data rate (typically 250,000 bits/sec). In this example, the fire time at a neighbor node equals the reception time at the MAC minus (72/250000) seconds.
- Adjust this neighbor fire time by the number of symbols contained in the last bits of the symbol offset portion of the fire message (typically 23 bits). For instance, the fire time at a neighbor node is adjusted by adding (23/62500) seconds.

3.7 ADVERTISING 1-HOP NEIGHBOR FIRE TIMES

Advertisement of 1-hop neighbor fire times is an optional feature used to support non-mesh topologies. Figure 18 shows the format of the fire message, including the optional portion used for advertising the 1-hop neighbor fire times.

A symbol offset is used to advertise the 1-hop neighbor fire times. As indicated previously, this is because nodes do not have to be time synchronized, so the fire times can't be in absolute time and the symbol offset is used as a relative time. The previous section explained how a neighbor's fire time is determined when the fire message from that neighbor is received. Once this fire time is known, the offset in symbols to that fire time relative to the node's own fire time is calculated. This symbol offset is stored in a list of 1-hop neighbor fire time offsets for advertisement in the node's fire message as its own 1-hop neighbor.

When a node receives a fire time, it calculates a symbol offset to represent the fire time of the neighbor from which it received the fire message. The calculation of the 1-hop fire time symbol offset is as follows:

$$\#Symbols_{neighbor} = Integer(SymbolRate * (neighborFireTime - myFireTime)) \tag{11}$$

Here, neighborFireTime is the fire time of the neighbor calculated as shown in the previous section. Note that this number of symbols can be a positive or a negative number indicating that the fire time is after the node's (positive) or before the node's (negative).

During the frame, a node accumulates this list of symbol offsets to each of its 1-hop neighbors and then includes those offsets in its fire message. The number of bits needed to advertise the 1-hop neighbor fire times is based on the time period. The maximum difference between any two fire times is the time period, and thus this value is used to determine the number of bits. That is, the number of bits needed to express the number of symbols that represents the time period. This calculation is as follows:

- Calculate the integer number of symbols for the maximum amount of time to be expressed in the fire message

$$\text{Max number of symbols} = \text{Integer(Time Period * Symbol Rate)}$$

- Calculate the number of bits needed to express this number of symbols

$$\text{Number of bits} = \left\lceil \frac{\log_{10} \text{Max number of symbols}}{\log_{10} 2} \right\rceil$$

- Add 1 bit to the # bits for sign bit since the symbol offset to the neighbor fire time can be a negative value. 0 is used for a positive number and the bit is set to 1 if the value is negative. Total # bits = # bits + 1

The reader will note that this calculation for the number of bits is the same as shown in the previous section for calculating the number of bits for a node to advertise its own fire time. That is, the number of bits needed to express a node's own fire time is the same as the number of bits needed to express the fire time of 1-hop neighbors. However, the number of bits needed for each 1-hop neighbor is NOT rounded to the byte boundary. Instead the total number of bits needed for all neighbors is rounded to the byte boundary. For example, suppose the time period is 1 second and the symbol rate is 62,500, then the number of bits needed to represent 1.0 second is 16 bits; adding 1 for a sign bit, the total is 17 bits needed to advertise a 1-hop neighbor fire time. Now suppose a node has four 1-hop neighbors, then the total number of bits is 68 bits. It is this number of bits that is rounded up to the next byte boundary. Thus, only 4 bits of pad are needed to get to 72 bits (9 bytes). If each 1-hop neighbor fire time was rounded up, 12 bytes would be needed to advertise the four 1-hop neighbors. Thus, the byte boundary is enforced only on the total count of bits to advertise all neighbors. Figure 21 shows an example fire message for a 1.0 second time period, a 62,500 symbol rate, and four 1-hop neighbors. Note that each node calculates the number of bits needed for advertising each 1-hop neighbor fire time and therefore can properly decode this message.

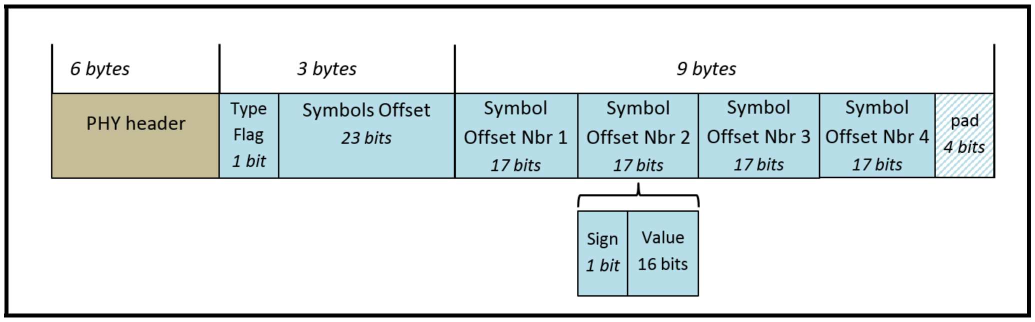

6 bytes
3 bytes
9 bytes
PHY header
Type Flag 1 bit
Symbols Offset 23 bits
Symbol Offset Nbr 1 17 bits
Symbol Offset Nbr 2 17 bits
Symbol Offset Nbr 3 17 bits
Symbol Offset Nbr 4 17 bits
pad 4 bits
Sign 1 bit
Value 16 bits

Figure 21. Example fire message format - 1.

When a node receives a fire message with 1-hop neighbor fire information, it needs to convert the symbol offsets to actual times. These symbols are offsets relative to the fire time advertised in the message, so the fire times are calculated relative to this time and not the node's own time. This is done as follows.

$$FireTime_{1-hopneighbor} = FireTime_{neighbor} + \#Symbols/SymbolRate \tag{12}$$

Note that 1-hop neighbor fire times will have some inaccuracy since they are expressed as symbols. This is because each symbol represents 16 microseconds of time (1/62500).

The operation of handling fire messages when a node receives a fire message is as follows:

- Handle the Symbol Offset for the sending node
  - Use the symbol offset in the fire message to calculate $FireTime_{neighbor}$
  - Store $FireTime_{neighbor}$ on the list of known fire times
  - Calculate the symbol offset of $FireTime_{neighbor}$ relative to the nodes fire time, i.e., $\#Symbols_{neighbor}$
  - Store $\#Symbols_{neighbor}$ on the list of 1-hop neighbor fire symbols to advertise
- Handle the OPTIONAL 1-hop neighbor Symbol Offsets
  - For each 1-hop symbol offset in the fire message, calculate the associated fire time of that 1-hop neighbor, $FireTime_{1-hopneighbor}$
  - Store $FireTime_{1-hopneighbor}$ on the list of known fire times

Thus, after processing a fire message, a node has added 1 entry to the list of 1-hop neighbor symbols that it needs to advertise and added 1 to N entries to the list of known fire times that it will use to make adjustments to its fire time and slot start/end.

When a node receives a fire message from a neighbor with the optional 1-hop neighbor fire times advertised, one important aspect to note is that the neighbor will echo back the node's own fire time. Referring to Fig. 7 as an example, when Node C sends a fire message with its 1-hop neighbor information, it will include the fire times for Nodes B and D. When Node B receives the fire message, it will see Node C advertising two 1-hop neighbors. However, Node B needs to recognize that one of the fire times advertised is its own. As indicated previously, the calculation of 1-hop neighbor fire times will suffer some inaccuracy due to the use of symbols. Though small, this inaccuracy means that a node needs to consider a range of fire times as its own fire time echoed back to it. The inaccuracy range is set to 4/(Symbol Rate) if the symbol rate is 62500 (total of 64 μsec), or 3/(Symbol rate) if the symbol rate is the robust mode rate of 40000 (total of 75 μsec).

The 1-hop neighbor fire time is compared to the node's own fire time from the current frame and the node's own fire time in the last frame. Referring to Fig. 9 frame 4 as an example, when Node 1 advertises Node 2's fire time, it will use the fire time for Node 2 in the previous frame (2.183), but when Node 3 advertises Node 2's fire time, it will use the fire time for Node 2 in the current frame (3.300). Thus, Node 2 needs to recognize both of these as its own echoed back to it.

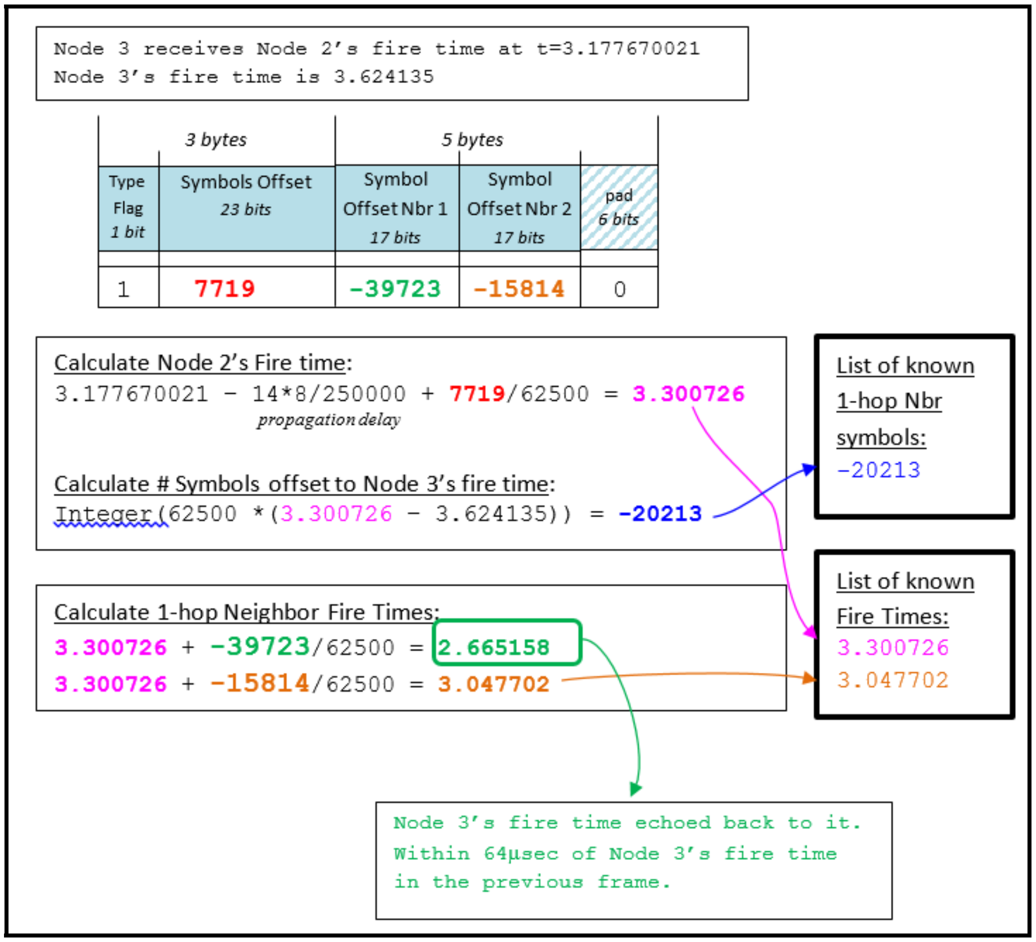
Figure 22 shows an example of processing a fire message using Node 2's fire message for frame 4 in Fig. 9.

3.8 INITIAL FIRE TIMES

When a node is initialized, how does it determine what its initial fire time will be? We have implemented two possible methods.

- Random: a node selects its initial fire time randomly on the interval (0, T], where T is the frame period.
- Based on Network Size: a node selects its initial fire time as $NodeId \times T/N$, where N is the maximum number of nodes in the network.

When the Random method is used, a node does not need any self-identification or information about the network. However, the Based on Network Size method requires that a node has a unique node ID assigned to it and that it has a value for the maximum number of nodes in the network. Both of these can be configured on each node so that a node does not need information about other nodes in the network. As well, the maximum number of nodes in the network can be larger than the actual number of nodes.


```
Node 3 receives Node 2's fire time at t=3.177670021
Node 3's fire time is 3.624135
```

3 bytes		5 bytes		
Type Flag *1 bit*	Symbols Offset *23 bits*	Symbol Offset Nbr 1 *17 bits*	Symbol Offset Nbr 2 *17 bits*	pad *6 bits*
1	7719	-39723	-15814	0

Calculate Node 2's Fire time:

$$3.177670021 - \underbrace{14*8/250000}_{\text{propagation delay}} + 7719/62500 = 3.300726$$

Calculate # Symbols offset to Node 3's fire time:

```
Integer(62500 *(3.300726 - 3.624135)) = -20213
```

List of known 1-hop Nbr symbols:
-20213

Calculate 1-hop Neighbor Fire Times:

$$3.300726 + -39723/62500 = \boxed{2.665158}$$

$$3.300726 + -15814/62500 = 3.047702$$

List of known Fire Times:
3.300726
3.047702

```
Node 3's fire time echoed back to it.
Within 64µsec of Node 3's fire time
in the previous frame.
```

Figure 22. Example fire message format - 2.

3.9 INITIAL FIRE TIME INTERFERENCE

When nodes randomly select their initial fire times, it is possible that nodes can select fire times that cause interference with each other and thus prevent any neighbors from hearing their fire messages. Figure 23 shows an example. In the first frame, Nodes 2 and 3 have initial fire times that cause their fire messages to interfere, and thus no node hears the fire messages of Nodes 2 and 3. When each node makes their adjustment, Nodes 2 and 3 both view the "previous" and "next" fire times as Nodes 1 and 4, and they make nearly identical adjustments and still interfere in the next frame. Meanwhile, Node 4 adjusts to be midway between Nodes 1 and 5. This pattern will continue until eventually Nodes 1, 4, 5 will adjust to consume the entire frame, and Nodes 2 and 3 will adjust to be midway between Nodes 1 and 4 and still interfering with each other.

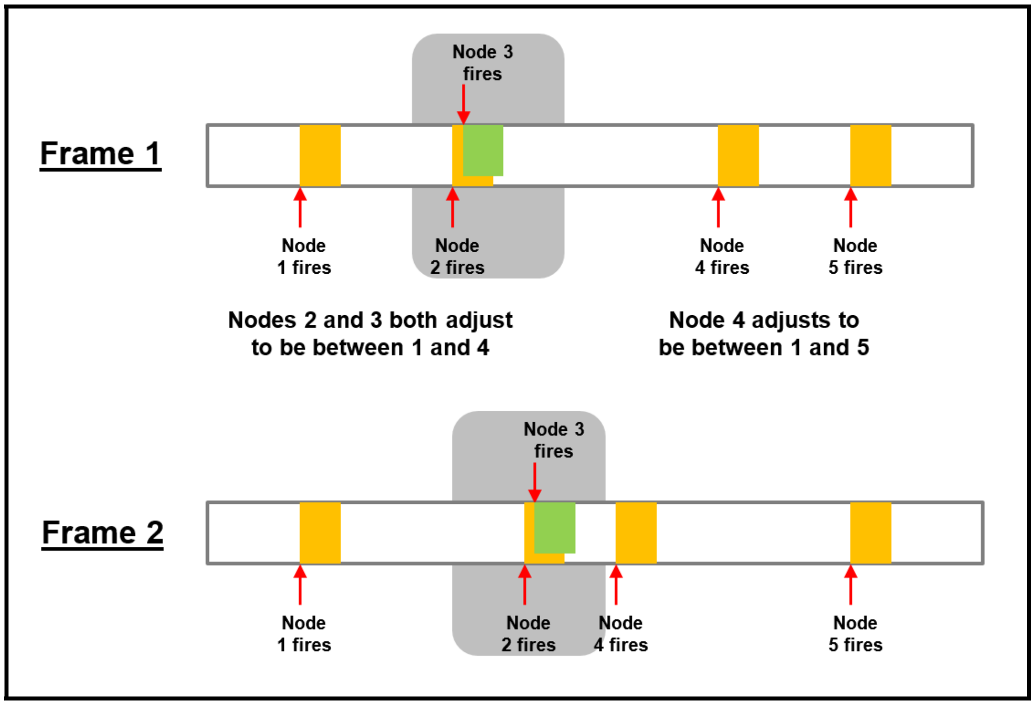


Figure 23. Initial fire time interference.

It will be difficult to assuredly prevent nodes from selecting non-interfering initial fire times, so the best approach to address this issue is to detect this interference. This is done by using carrier sense on the fire message. Before a node transmits its fire message, it first performs carrier sense to determine if there is another transmission currently active. If so, the node does not send its fire message but will still make an adjustment to its fire time.

Figure 24 shows an example when carrier sense is used. As in the previous example, Nodes 2 and 3 have initial fire times that cause their fire messages to interfere. However, because of carrier sense, Node 3 detects that another node is transmitting and does not send its own fire message. Thus, all nodes hear Node 2's fire message (but of course they don't know about Node 3). When each node makes their adjustment, Node 2 adjusts to be between Nodes 1 and 4, Node 3 adjusts to be between Nodes 2 and 4, and Node 4 adjusts to be between Nodes 2 and 5. Because Nodes 2 and 3 made adjustments based on different previous and next neighbors, their fire times do not interfere in the next frame, which will allow the Desync algorithm to work correctly from that next frame.

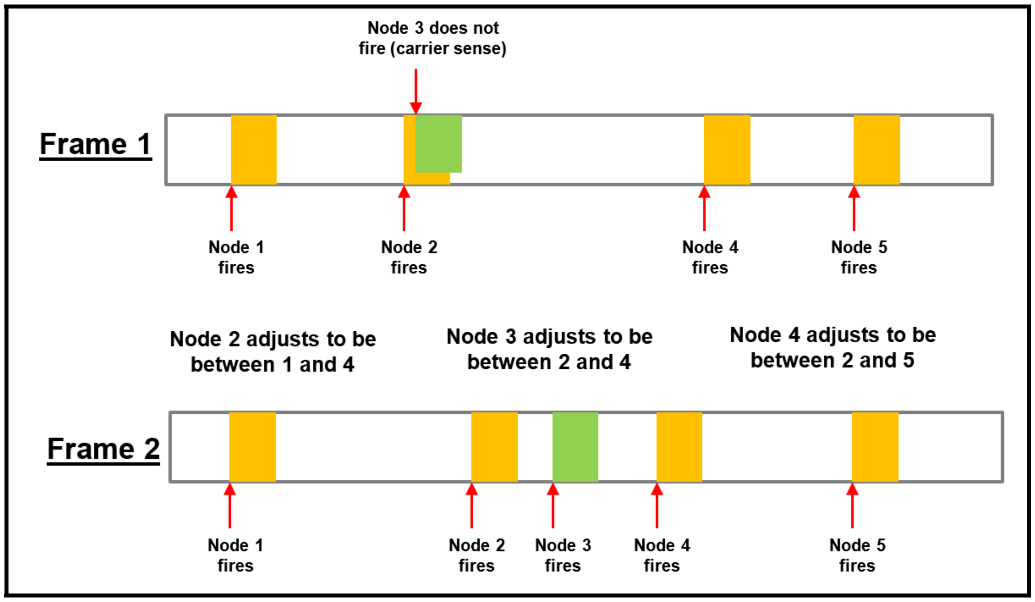

Node 3 does not fire (carrier sense)
Frame 1
Node 1 fires
Node 2 fires
Node 4 fires
Node 5 fires
Node 2 adjusts to be between 1 and 4
Node 3 adjusts to be between 2 and 4
Node 4 adjusts to be between 2 and 5
Frame 2
Node 1 fires
Node 2 fires
Node 3 fires
Node 4 fires
Node 5 fires

Figure 24. Initial fire time interference with carrier sense.

When fire time interference is detected and a node skips its own firing, it still adjusts its slot in the same manner as usual; that is, it uses the list of known fire times to make its adjustment. However, when a node skips its firing because of interference, it must skip its slot in the next frame and wait until its fire time is no longer interfering before using its slot. This prevents the node's data packets from interfering with the fire packet of other nodes.

When a node is carrier sensing on fire messages, it works as follows:

- Before a node transmits its fire message, it checks the receiver for the receiver power level
- If this power level exceeds a user-specified packet reception power threshold (default -100 dBm), then the receiver is considered busy so the node does not send its fire message
- The node does not schedule slot start or end events but does make fire time adjustments

3.10 FRAGMENTATION OF DATA PACKETS

As discussed earlier, fragmentation of data packets is necessary to maximize slot utilization. For fragmentation, we have assumed the following. First, no packet is larger than the slot, so a packet will always fit in one slot. This means that a packet will only ever be broken into two fragments. Second, packets are only fragmented at the end of the slot so that the first fragment is the last packet sent in a slot and the second fragment is the first packet sent in the slot in the next frame. Finally, in order to fragment a packet, at least 1 byte of the payload excluding PHY and MAC headers must be able to fit in the remaining slot time. Thus, the smallest fragment that will be sent consists of the PHY header, the MAC header, and 1 byte of payload. This also means that the most amount of unused time in the frame will be the size of the PHY header plus the size of the MAC header divided by the data rate. With a PHY header of 6 bytes, a MAC header of 9 bytes, and a data rate of 250 kbps, this is just 480 microseconds.

In our implementation, transmission of the second fragment of a packet takes priority over all other packets. Fragment 2 of a packet is stored in a separate queue from other packets. Thus, we will always have fragment 1 transmitted at the end of a slot and then fragment 2 transmitted at the start of the slot in the next frame. As a result, it is not possible to receive fragments out of order. It is possible to drop packets on the channel, but once a packet is fragmented, it is expected that the fragments will be received in order. This assumption results in several "rules" associated with fragment reception.

- If fragment 2 is received but a node does not have fragment 1, fragment 2 is dropped
- If a node receives fragment 1 for a packet but already has a fragment 1 for another packet from the same node, it is assumed that fragment 2 of the first packet was lost, so fragment 1 for that first packet is dropped
- A node only needs to store one fragment 1 from any source node

In order to be able to assemble the packets at the destination, the node needs to know two pieces of information, which can be carried in the existing MAC header without having to add any bytes to the header.

- Fragment number: this is conveyed in bits 12, 13 of the Frame Control field of the MAC header
- Packet identifier: this uses the existing Sequence Number field of the MAC header

3.11 QUEUEING OF PACKETS AND DYNAMIC QUEUE RESIZING

When packets are received from the network layer at the MAC, they are placed in a queue for transmission. The queueing method is first-in-first-out so that new packets are placed at the tail of the queue and packets are removed from the head of the queue for transmission. The queue has a finite size, and thus, if the amount of data sent from the network layer exceeds the bandwidth available, packets will accumulate in the queue until it hits the maximum size and then packets must be dropped. There are two drop methods from which the user can choose:

- Drop Newly Arriving Packet: If the queue is full when a packet arrives at the MAC from the network layer, the newly arriving packet is dropped

- Drop Oldest Packet: If the queue is full when a packet arrives at the MAC from the network layer, the oldest packet at the head of the queue is removed from the queue and dropped so that the newly arriving packet can be enqueued

It should be noted that fragment 2 of a packet that was fragmented is not maintained in this queue and will not be dropped due to any queue overflows. Thus, when it is time to transmit a packet, the queue (which has size 1) of fragments is checked first for a fragment 2 to dequeue before the queue of packets is accessed.

The size of the queue needs to be dependent on the network. For example, if 20 nodes are sharing the network, each node will be able to transmit more packets than if 100 nodes are sharing the network. Thus, the queue needs to be sized accordingly. Queue size is handled as follows.

1. Queue size at initialization is based on a user configured value

2. When the slot size is stable, the node recalculates its queue size so that the size is twice the number of maximum-sized packets that could fit in the slot time that is used for sending data messages. The calculation is as follows:
   2*(slot duration guardband time to send fire)*data rate/Max packet size

3. This calculation is first performed when the slot size is stable, which is based on the exponential moving average of the % change in slot size. When this running average drops below a specific threshold (default 5%), the node performs the above calculation to resize its queue and at this point the node can begin sending data (discussed further in the next section).

4. Once the initial queue resizing is performed, a node will check its queue size every frame when it recalculates its slot size. This may or may not result in a change in the queue size.

5. When the queue size changes, if the new queue size is smaller than the current queue size and the queue is full, packets are removed from the queue to bring its size down to the new value. The packets are removed using the method selected for dropping packets when the queue is full (drop new or drop oldest).

Note that there is no minimum queue size. The queue size is recalculated every frame, and there is no limit to the minimum or maximum size.

3.12 DETERMINING WHEN TO BEGIN SENDING DATA

A node should not start sending data until the network has reached a stable state. This prevents interference between fire packets and data packets, which could cause issues with the network reaching a de-synchronized state. Take, for example, the case of interfering fire messages shown in Figure 24. In this case, no nodes have heard the first fire message of Node 3, but Node 3 adjusts its fire time to be non-interfering with the fire times of its neighbors. However, if one of those neighbors transmitted data in the second frame, Node 3's fire message would interfere with that node's data packets. Thus, data packets are not sent until a node reaches a stable state.

We have chosen to use the percent change in the slot duration as the threshold to decide when to begin sending data. An exponential moving average of the percent change in the slot duration is maintained by each node as follows:

1. Determine the current percent change in slot size
2. Determine the exponential running average of percent slot size change as

$$\alpha(\text{Current percent slot change}) + (1-\alpha)(\text{Average percent slot change})$$

When this running average drops below a specific threshold to stop carrier sensing (default 5%), the node may begin to send data packets. A value of 0.50 is used for α so that equal weight is given to old and new samples, which results in a slower convergence than high values of alpha. This is important because the slot size can undergo dramatic shift at startup, and converging too quickly can result in sending data too soon in the process.

4. EXPERIMENTATION

4.1 SETTING

The Desync protocol, as described in the previous section, has been implemented in stand-alone C++, and this code library interfaces with Riverbed Modeler version 18.0.3 for simulation testing. We have also used a standard Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol, conforming to the 802.15.4 standard and provided by Modeler. We have chosen to test in simulation due to the ease of testing large networks, which are of primary interest.

The model used in experiments is described as follows. At each network node, a generic application generates UDP/IP traffic, where each packet is intended for multicast transmission to all one-hop neighbors of the node. Packets arrive through a regular, deterministic process at the same rate to all network nodes, where the packet arrival rate, or offered traffic load, is a variable in our experiments. Standard UDP/IP encapsulation is applied, and for all of our experimental results, we treat the UDP/IP headers as payload, rather than overhead. The link and physical layers model the 802.15.4 standard, as described in Section 3.3. The link layer does not support any form of packet loss recovery or automatic repeat request (ARQ); there is no reliable delivery mechanism provided at any layer of the protocol stack. The frame size of Desync is set to 1.0 seconds, except where noted otherwise. Finally, the channel is modeled to incur distance-squared pathloss; at 900 MHz this is a reasonable model for air-air communication links where the distance is sufficiently short (e.g., up to a few kilometers) and the altitude is sufficiently large (e.g., 500 m).

For simulations, network nodes are placed in fixed locations to create two different network topologies: a fully connected mesh network and a line network. In the full mesh network, all nodes can communicate directly in a single hop, and all pairs of nodes mutually interfere. The one-hop multicast traffic model in this case is equivalent to network-wide broadcast. For the line topology, each node can communicate with two neighbors directly, except for nodes at the end of the line, which communicate with only one. In this case, each node's transmission interferes with nodes two hops away, so whenever a node initiates transmission while any of its one- or two-hop neighbors are already transmitting, a collision occurs and the transmitted packet is lost.

The performance metric we consider most often is the throughput, which is a measure of the rate that network nodes can communicate. We consider two different variations of throughput. First, the one-hop multicast throughput, denoted S, is defined as the bits/sec *received* by *any* one-hop neighbor. In other words, we count a bit twice when it is received by two neighbors. Moreover, when measuring throughput in simulation, we count only the payload bits in each data packet; in-band control messages and packet headers effectively reduce the throughput, and that is accounted for here. The other throughput metric we consider is the normalized throughput per node, which is computed by dividing S by the number of nodes, number of neighbors, and the physical layer data rate of 250 kbps. This normalized throughput per node reflects the fraction of the physical layer data rate that each network node successfully *sends*. Another performance metric we consider is the delay, which counts from the time a packet arrives at the link layer until it is received at the link layer of a destination node. This delay metric includes the time spent waiting in a link-layer queue before access to the channel, propagation time, and transmission time.

4.2 RESULTS

4.2.1 Impact of Initial Conditions

As discussed in Section 2.2, in certain topologies, the convergence of the Desync algorithm depends on the initial transmit or fire time of nodes in the first frame. For a full mesh topology, the algorithm is known to converge for any initial condition [1]. In Section 2.2, a proof sketch is provided to show that when the initial fire times are ordered, the algorithm converges on a ring topology. While Desync can provide a conflict-free schedule, the convergence of the algorithm on general topologies has not been shown, and the general approach for initial conditions is for each node to randomly and independently pick its first transmit time.

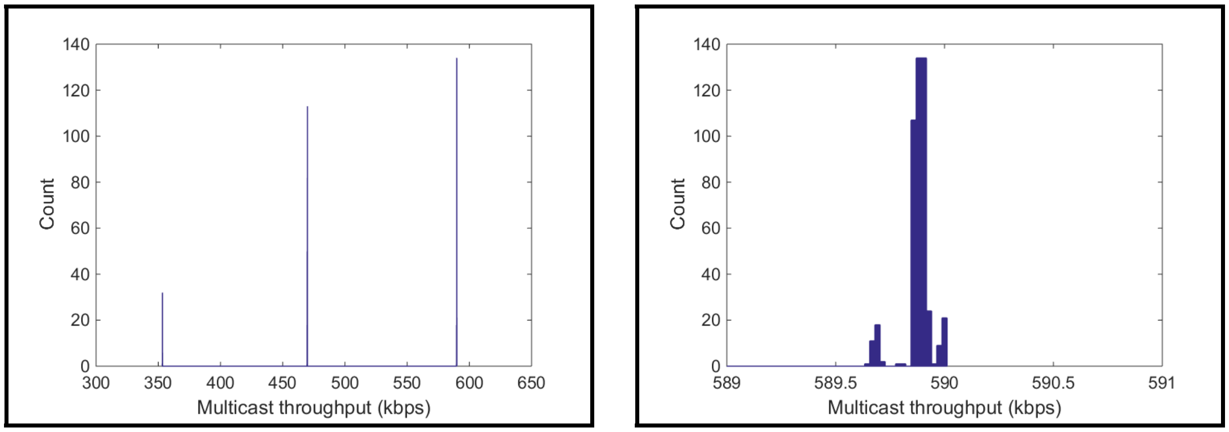


Figure 25. Histogram of throughput achieved on a 5-node line network, over 1000 realizations of initial conditions.

Here we outline experimental results on the convergence of Desync in a line topology assuming random initial conditions. Consider a line network of n nodes. The optimal solution to interval coloring on this line, given that nodes interfere with 2-hop neighbors in our experiments, is a frame with 3 equal-sized timeslots, for any $n > 2$. The Desync algorithm will converge to this optimal solution for a subset of all possible initial conditions. For other initial conditions, it will converge to one in a set of possible sub-optimal solutions; in the worst case, the solution will be a frame with n equal-sized timeslots. Which solution the algorithm will converge to depends on the order of the initial fire times for the nodes on the line. Unfortunately, the exact mapping between an initial condition and the converged solution is not understood, but we can provide approximate bounds on the throughput that Desync achieves as follows. In our experiments the multicast throughput S approximately satisfies the following:

$$\frac{(2(n-2)+2)*250kbps}{n} \leq S \leq \frac{(2(n-2)+2)*250kbps}{3}, \quad n > 2 \tag{13}$$

In the numerator, the term $2(n-2)+2$ reflects multicast throughput in a line topology, where $n-2$ nodes have 2 neighbors, while the 2 nodes on the end have just one neighbor. The $250kbps$ models the physical-layer data rate shared by all nodes. In the denominator, the term n in the lower bound reflects convergence to the worst possible sub-optimal solution where the frame contains n equal-sized time slots, while the upper bound with denominator 3 reflects convergence to the optimal coloring solution. Note that the lower bound is equivalent to the throughput achieved by a time-division scheme with no spatial reuse.

We performed extensive experiments on a line network of $n = 5$ nodes. In this case, there are $5! = 120$ possible orderings of initial fire times among nodes in the network. In this small network, it is simple to enumerate the set of possible solutions that Desync could converge to; in addition to a frame with 3 or $n = 5$ timeslots, a frame of 4 equal-sized timeslots is possible. From Eqn. (13), we may expect that the multicast throughput achieved when the system converges will be either $400, 500$, or 666 kbps. Accounting for roughly 10% overhead due to fire messages, packet headers, fragmentation, and guard times, we should expect the throughput to converge to either $360, 450$, or 600 kbps. We simulated the five-node line using 1000 different seeds for initial fire times; note that 1000 is approximately eight times the number of possible orderings of initial fire times, and we ensured that each ordering was realized at least once. A histogram of throughput over these 1000 realizations is shown in Fig. 25, which demonstrates that Eqn. (13) is a valid approximation in this case. Of these 1000 realizations, the multicast throughput converged to the optimal solution of 600 kbps 464 times (46%), while it converged to 450 kbps 461 times (46 %) and 360 kbps 75 times (8%). In the following sections, the results on throughput achieved in a line network are averaged over 100 realizations of random initial conditions. We argue that this is a valid metric since, as shown in Section 2.2, as n increases, the distribution of slot sizes achieved by Desync concentrates around a mean.

4.2.2 Convergence

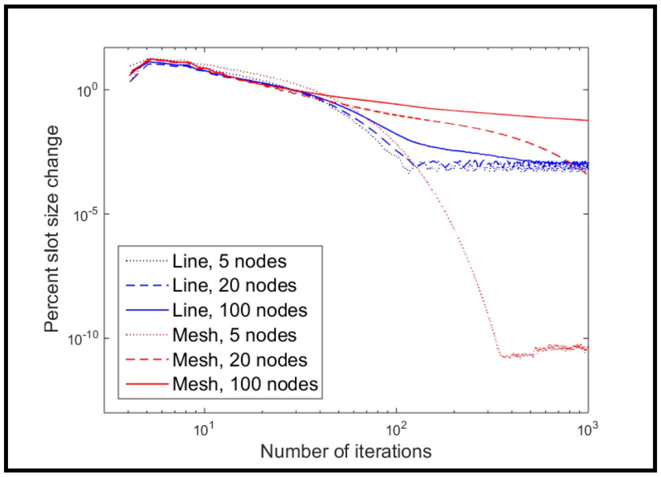


Figure 26. Convergence of Desync: average slot size change for various network sizes and topologies.

The results in Fig. 26 show the evolution over time of the percentage change in slot size averaged over all nodes. For Desync, this is a meaningful metric for convergence. The unit of time shown here is an iteration of the algorithm, where each of the n nodes adjusts their fire time once;

in this simulation, a single iteration lasts 1.0 sec. The results here show basic trends: the algorithm converges more quickly on a line network and on smaller networks. Furthermore, at convergence, the variability in slot sizes is much larger on a line network than in a mesh network; this is an intuitive result reflecting more variability in slots that last roughly 0.3 seconds than in slots that last 0.2 seconds or less. Finally, we note that the line network settles to a level of variability that is constant in network size, which reflects the fact that the optimal schedule for a line is the same for any $n > 2$. By contrast, in a mesh network of n nodes, Desync converges to a slot duration of $1/n$ for each user, and the variability around this slot duration is proportional to $1/n$.

4.2.3 Throughput and Delay Performance

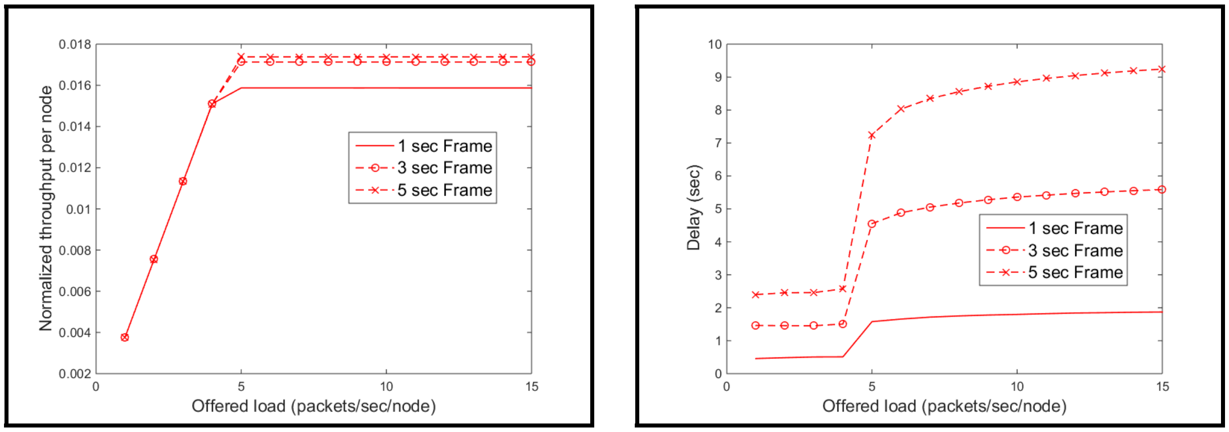


Figure 27. Throughput and delay performance versus offered load, mesh network of 50 nodes.

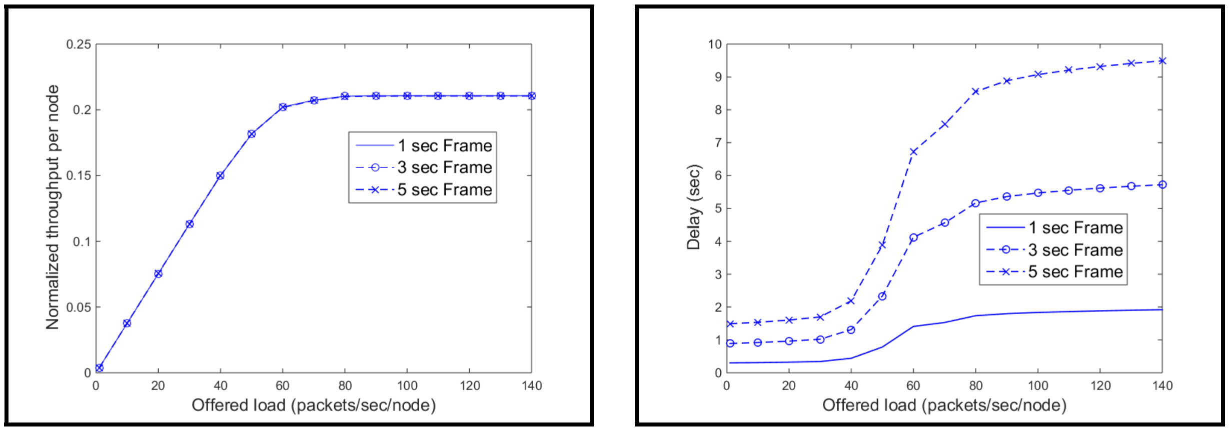


Figure 28. Throughput and delay performance versus offered load, line network of 50 nodes.

The tradeoff between throughput and latency is a phenomenon that occurs in many settings, and we have performed experiments to quantify the behavior of Desync in this respect. Figures 27 and 28 show the normalized throughput per node and delay versus offered traffic load, for various Desync frame sizes, and for networks of 50 nodes. These curves demonstrate expected behavior. The throughput increases linearly with offered load until the network saturates and queues remain full of packets, after which the throughput remains roughly constant in offered traffic rate and quantifies performance when there is always a packet to send. Similarly, the delay initially increases slowly with offered load until the network saturates, which leads to a sharp increase in delay that ultimately settles to a value dominated by the maximum waiting time in a link-layer buffer. An increase in the Desync frame size can sometimes result in higher throughput, but reflecting the tradeoff, it universally results in higher delay.

The throughput and delay performance for a mesh network is shown in Fig. 27, while Fig. 28 provides results for a line network. In the mesh network, increasing the frame size can provide moderate increases in throughput, due to a reduction in the fraction of the frame occupied by overhead from fire messages and guard times. With respect to delay, the value at low offered loads is roughly half of the frame duration, which reflects the fact that when there are infrequent packet arrivals, delay is dominated by time spent waiting to access the medium; this is a standard phenomenon that arises in any time-division system. At high offered loads, the delay is predominantly waiting time in the buffer, which is roughly twice the frame duration and reflects the implementation choice of sizing buffers to be twice the expected slot size. By contrast, in the line network, increasing the frame size provides at best a negligible increase in throughput: due to spatial reuse, the overhead occupies such a small fraction of the frame that increasing its size does not make a measurable difference. With respect to delay, at low offered loads it's approximately one-third of the frame duration, which reflects the fact that in a frame of just three slots, a significant number of packets may be sent as soon as they arrive.

4.2.4 Impact of Control Information Relay

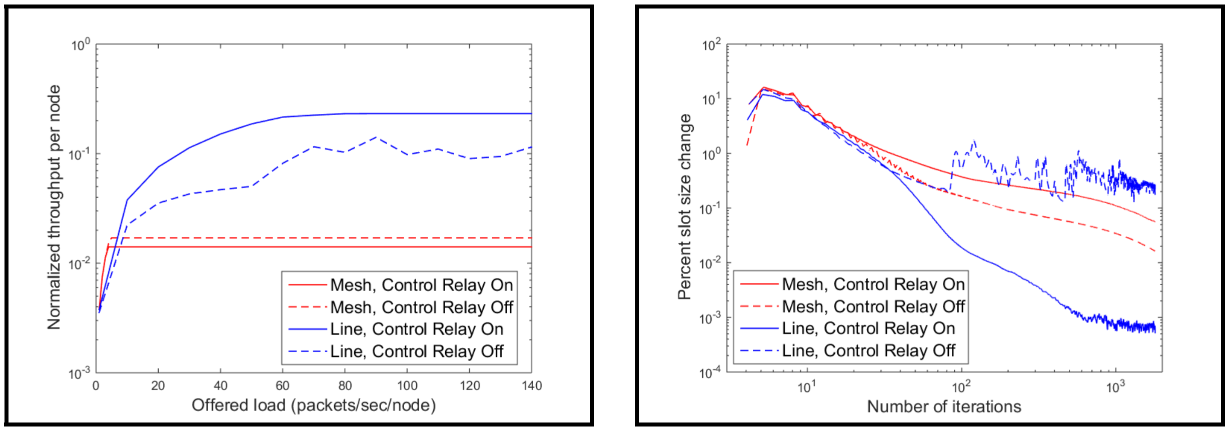


Figure 29. Impact on Desync of control information relay, in networks of 50 nodes.

As described in Section 3, the relaying of control information, specifically of the Desync fire time, is necessary to support operation of the algorithm on any network of diameter greater than two where interference occurs between nodes that cannot directly communicate. We have performed experiments to demonstrate this point, and to more broadly quantify the impact of relaying control information, which has been implemented as a feature that can be enabled or disabled. In Fig. 29, we show the impact on throughput and algorithm convergence of enabling or disabling this feature. In the line network, relaying of control information is necessary for proper performance of Desync, and this is reflected in the results. Compared with the throughput and convergence behavior when relaying of control information is turned on, the performance with this feature disabled shows that the algorithm operates at reduced throughput levels and at a high level of variability in slot size. Further debugging of these experiments showed that the algorithm tends to settle on a schedule where interfering nodes transmit concurrently, resulting in lost data packets and lost fire messages; the algorithm may then recover and eventually settle again on a schedule with conflicts, and the process repeats.

Though relaying of control information is necessary in a line network, it creates unnecessary overhead in a mesh network. As shown in Fig. 29, the Desync algorithm converges to a normalized per-node throughput of approximately $1/n$ when the relaying of control information is disabled. By contrast, when the relaying feature is enabled for a mesh network, there is a penalty in throughput due to additional unneeded fire messages occupying resources that could be dedicated to data transmissions. For large mesh networks, unnecessary fire messages could consume the majority of a slot. Moreover, in the implementation of relaying fire messages, new inaccuracies are introduced by these unnecessary relays. Each node will hear the fire time of its one-hop neighbors directly from that neighbor, and this will be an accurate time; however, it will also hear the fire times of its one-hop neighbors indirectly through relaying, and this fire time will not be as accurate, as it's based on quantizing the fire time according to the symbol duration. These additional, inaccurate fire times recorded at each node will directly impact the Desync algorithm, and provide an explanation for the higher variability in slot duration when relaying of control information is enabled.

In summary, relaying of control information is *necessary* in the line network setting, but leads to inferior performance in a full mesh network. This results in a tension over the appropriate use of the feature in general scenarios. For networks where the topology changes over time between dense and sparse connectivity, a reasonable approach is to enable the relaying of control information to ensure operation, while recognizing that this leads to sub-optimal behavior when the network is densely connected.

4.2.5 Network Scalability

Our next experiment demonstrates performance as a mesh network is scaled up to large numbers of nodes, which may require access to the channel at increasingly infrequent intervals. In addition to examining throughput and access delay, or delay as the offered load approaches zero, we consider the impact of increasing frame sizes up to 1000 sec. These results are shown in Fig. 30, where we have enabled the relaying of control information, which corresponds to providing worst-case throughput results. As the results show, increasing the frame size can result in modest improvements in throughput, at the cost of significant increases in access delay. At $n = 100$ nodes, increasing the frame size from 1 to 10 seconds results in a factor of 2 improvement in throughput, but a factor of 10 increase in access delay. In general, the throughput on a full mesh topology never

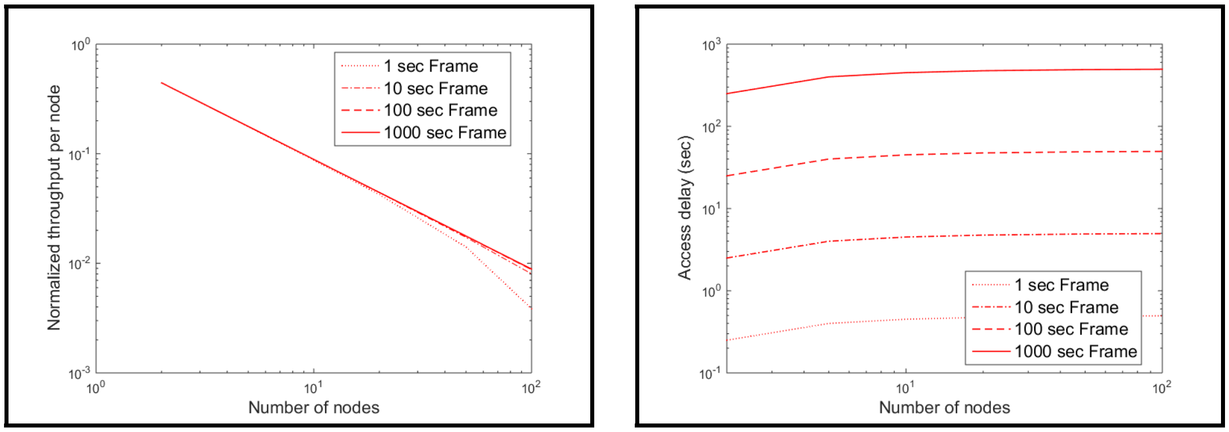


Figure 30. Desync performance as network scales in number of nodes, mesh network.

scales well with network size, due to the absence of spatial reuse; this is true for Desync and for any medium access technique.

4.2.6 Comparison of Medium Access Techniques

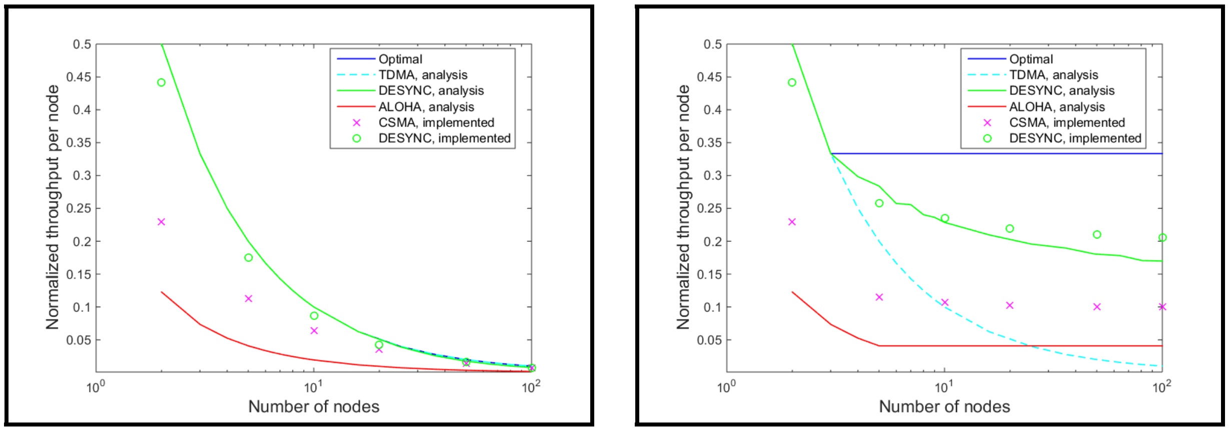


Figure 31. Throughput versus number of nodes for different medium access techniques, for a mesh (left) and line (right) network topologies.

Finally, we conducted experiments comparing the throughput achieved for different multiple access schemes, as predicted by the idealized performance analysis in Section 2 and by the fully implemented protocol described in Section 3. Again, considering both the full mesh and line topologies, we compare throughput versus number of nodes in Fig. 31. The “Optimal” result is the solution to interval graph coloring of vertices, while the scheme labeled “TDMA” is a simple time division scheme without spatial reuse. The “Desync analysis” curve is generated by a simple idealized simulator that lacks proper modeling of a communication channel, while the “ALOHA” curve is generated by results on unslotted ALOHA from Section 2. As expected, the implemented CSMA approach outperforms ALOHA, while both of these random access techniques provide reduced throughput compared with Desync. Moreover, the implemented Desync protocol performs close to the predictions provided by a simple idealized simulator, and this provides evidence that we have developed an effective implementation.

5. CONCLUSION

As evident from the experimental results presented in Section 4, there are a number of avenues for future work in the development of Desync. First, a better understanding of the impact of initial conditions and the convergence of the algorithm in general network topologies is needed. For this question, analysis may prove difficult, and simulation of particular cases of interest or of a more extensive set of scenarios may be needed. Moreover, the impact of node mobility and the arrival and departure of nodes on the algorithm must be understood; preliminary investigations suggest that the algorithm is robust to these types of changes, but further study is needed. The use of Desync on a channel with losses or fading is an important avenue for testing; while data traffic in the applications of interest may not require reliability, control messaging, particularly fire messages, will. Finally, the operation of Desync with other physical layer models, including multifrequency and spread-spectrum systems, must be considered and may drive changes in the operation of the algorithm.

The experimental studies in Section 4 compare Desync to CSMA, the best available contention scheme, and there are outstanding questions about the choice between the two. Results in this report show that Desync provides a factor of two improvement in throughput over CSMA on a line network with 2-hop interference. Is a factor of two improvement worthwhile considering the costs of implementing and testing Desync, relative to the maturity and availability of CSMA implementations? Would the introduction of channel losses or different physical layer waveforms change this factor of two? Are there other performance metrics that change the considerations in the choice between the two? On this matter, the metric of energy per bit seems particularly relevant: assuming that Desync suffers zero lost packets due to collisions, while approximately $1/2$ of packets are dropped in the use of CSMA, then Desync consumes roughly $(1/2) * (1/2)$, or $1/4$, of the energy of CSMA, for the same throughput.

REFERENCES

[1] J. Degesys, I. Rose, A. Patel, and R. Nagpal, “DESYNC: self-organizing desynchronization and TDMA on wireless sensor networks,” in *Proceedings of the 6th international conference on Information processing in sensor networks*, ACM (2007), pp. 11–20.

[2] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs),” *IEEE Std. 802.15.4 - 2006* (2006).

[3] “Bluetooth Core Specification v5.0,” *Bluetooth Special Interest Group* (2016).

[4] “Third Generation Partnership Project (3GPP),” *Evolution of LTE in Release 13* (2015).

[5] “LoRa Alliance,” *A technical overview of LoRa and LoRaWAN* (2015).

[6] M. Freeman, “On-ramp wireless becomes Ingenu, launches nationwide IoT network,” in *San Diego Union Tribune* (2015).

[7] R. Rom and M. Sidi, *Multiple Access Protocols: Performance and Analysis*, Springer-Verlag, New York (1990).

[8] P. Karn, “MACA-a new channel access method for packet radio,” in *ARRL/CRRL Amateur radio 9th computer networking conference* (1990), vol. 140, pp. 134–140.

[9] S. Kaul, R. Yates, and M. Gruteser, “Real-time status: How often should one update?” in *IEEE International Conference on Computer Communications (INFOCOM)* (2012).

[10] X. Zhu, “Circular chromatic number: a survey,” *Discrete mathematics* 229(1-3), 371–410 (2001).

[11] J. Degesys and R. Nagpal, “Towards desynchronization of multi-hop topologies,” in *Self-Adaptive and Self-Organizing Systems, 2008. SASO’08. Second IEEE International Conference on*, IEEE (2008), pp. 129–138.

[12] A. Arenas et al., “Synchronization in complex networks,” *Physics reports* 469(3), 93–153 (2008).

[13] O. Simeone, U. Spagnolini, Y. Bar-Ness, and S.H. Strogatz, “Distributed synchronization in wireless networks,” *IEEE Signal Processing Magazine* 25(5) (2008).

[14] A. Motskin, T. Roughgarden, P. Skraba, and L.J. Guibas, “Lightweight coloring and desynchronization for networks,” in *INFOCOM* (2009), pp. 2383–2391.

[15] N. Deligiannis, J.F. Mota, G. Smart, and Y. Andreopoulos, “Fast desynchronization for decentralized multichannel medium access control,” *IEEE Transactions on Communications* 63(9), 3336–3349 (2015).

[16] H. Gao and Y. Wang, “Analysis and design of phase desynchronization in pulse-coupled oscillators,” *arXiv preprint arXiv:1603.03313* (2016).

[17] T. Anglea and Y. Wang, "Phase desynchronization: A new approach and theory using pulse-based interaction," *IEEE Transactions on Signal Processing* 65(5), 1160–1171 (2016).

[18] R. Pagliari, Y.W.P. Hong, and A. Scaglione, "Bio-inspired algorithms for decentralized round-robin and proportional fair scheduling," *IEEE Journal on Selected Areas in Communications* 28(4) (2010).

[19] D. Buranapanichkit, N. Deligiannis, and Y. Andreopoulos, "Convergence of desynchronization primitives in wireless sensor networks: A stochastic modeling approach," *IEEE Transactions on Signal Processing* 63(1), 221–233 (2015).

[20] R.P. Stanley, *Enumerative Combinatorics*, *Cambridge Studies in Advanced Mathematics*, vol. 49, Cambridge University Press, Cambridge, 2nd ed. (2012).

[21] T.K. Petersen, "Cyclic descents and p-partitions," *Journal of Algebraic Combinatorics* 22(3), 343–375 (2005).