IMPERIAL

MEng Individual Project

Imperial College London

Department of Computing

SyncWave: Rapid and Adaptive Decentralized Time Synchronization for Swarm Robotic Systems

Author:
Luca Seimon Mehl

Supervisors:
Prof. Julie McCann
Dr. Michael Breza

Second Marker:
Prof. William Knottenbelt

June 19, 2024

Abstract

Time synchronization in Robotic Swarm Systems can be an extremely powerful construct, allowing swarms to coordinate behaviour and perform consensus on future actions within a bounded time[1], and is among the most essential areas of research within Swarm Systems. However, these systems have unique requirements that are not common in either of the fields of Distributed Systems, Wireless Sensor Networks, and Pulse-Coupled Oscillators – from which nearly all time synchronization protocols originate. Notably, robustness to arbitrary node and link failures, extremely short synchronization times, and robustness to a variety of challenging multi-hop, dense, and dynamic network topologies are all fundamental requirements that existing time synchronization protocols struggle with. In this report we propose SyncWave, a novel fully-decentralized time synchronization protocol that has beyond state-of-the-art millisecond-level convergence speeds, adaptive radio usage, and extreme robustness to challenging multi-hop, dense, and dynamic topologies. These findings have been experimentally shown through evaluation on a large-scale IoT testbed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.0.1 Motivation

Time synchronization in swarms can be an extremely powerful construct. For drone swarms, time synchronization is essential in order coordinate behaviour and perform consensus on future actions within a bounded time[1]. Consider a search-and-rescue scenario, in which a swarm of drones is exploring an indoor space: it would be common for arbitrary nodes to lose line of sight–and thus communication–with the rest of the swarm. Thus, any swarm time synchronization protocol would need to allow for arbitrary network partitioning and rapid merging when clusters of individual nodes re-establish connection. However, nearly all existing network-layer time synchronization algorithms (including PTP, GPS, and Synchronous Transmissions) are reliant on a centralized time reference, making the swarm vulnerable to a single point of failure.

Distributed Time synchronization is a well-known problem within the fields of Distributed Systems and Wireless Sensor Networks (WSNs); it is a much younger area of exploration within the fields of Swarm Robotic Systems and Flying Ad-Hoc Networks. The majority of existing research in Swarm Time Synchronization is conducted from the perspective of adapting Time Synchronization Algorithms from WSNs. However, as we will show, the existing literature is not suited to or practical for performing time synchronization on robotic swarms.

In WSNs the paradigm generally consists of static network topologies, in which the time to network synchronization is measured in minutes to hours [3]. They have extremely low power requirements, so the slow time to convergence is an accepted tradeoff within the field. However, in swarm applications the timescales are far shorter, often in the milliseconds, and rapid re-sychronization in the face of topology changes is paramount.

At the other end of the spectrum we have an entirely different paradigm of time-alignment protocols, called Pulse-Coupled Oscillators (PCO). These are entirely decentralized and have excellent resiliency to topology changes, but until now have only been capable to providing *synchrony* (aligning the phase of an oscillator for each node), with poor convergence guarantees on multi-hop topologies, and an inflexible communication rate, hindering their widespread adoption.

We approached the problem of decentralised time synchronization from the perspective of PCO, and adapt it to leverage a variety of concepts to achieve the desired properties.

1.0.2 Contributions

The aim of this project was to make decentralized time synchronization possible, and practical, on highly mobile topologies such as drone swarms. To achieve this, our goals included: rapid network synchronization; high convergence rate and guarantees on "worst-case" challenging topologies, including large-scale, dense, and multi-hop network configurations; minimizing radio and power usage during and after synchronization; and resiliency to dynamic topology scenarios including network partitioning, cluster merging, and node churn.

- **SyncWave algorithm for decentralized, rapid, and adaptable time synchronization**: We introduce the SyncWave algorithm, with several key concepts that are novel within the field of decentralized time synchronization algorithms. These include: a total ordering on time within the Pulse-Coupled Oscillator framework allowing for optimal convergence properties; a total decoupling of fire time from information propagation for Pulse Coupled Oscillators in the wireless medium, using exponential backoff to modulate information rate based

on network synchronization progress; and a unique-sender Message Suppression optimization for densely connected networks. These concepts provide SyncWave with performance beyond the state of the art in every one of our stated goals.

- **SyncWave Implementation in RIOT-OS**: We implement SyncWave on an nrf52840 SoC using the RIOT embedded operating system, and detail the process of adapting the theoretical algorithm for real hardware and relaxing any assumptions. We further make the source code for our implementation of SyncWave on RIOT-OS open-source, and immediately integrable within existing applications thanks its structure as a RIOT module, allowing it to be hot-swappable with the built in ZTimer module. This allows compatibility with hundreds of microcontroller hardwares with minimal adjustment, and a wide range of possible use cases.

- **Large-scale testbed evaluation**: We evaluate our protocol experimentally on topologies consisting of up to 161 nodes and up to 12 hops, in both sparse and dense configurations. To our knowledge this represents the largest scale testing of a decentralized time synchronization algorithm, and provides interesting insight into how such algorithms perform outside of simulation. Through this evaluation, we find that SyncWave massively improves upon the state of the art with respect to average and worst-case convergence time in both sparse and dense topologies, the number of broadcasts used post-synchronization, and has highly attractive scaling properties.

Chapter 2

Background

In this chapter, we will introduce the relevant background for understanding SyncWave. We will start by giving an overview of several decentralized synchronization protocols for Pulse-Coupled Oscillators and Wireless Sensor Networks.

2.1 Pulse-Coupled Oscillators

2.1.1 The Reachback Firefly Algorithm (RFA)

RFA presents a fully decentralized approach to achieve synchronicity across a wireless sensor network, inspired by the synchronous flashing behavior of fireflies. The goal is to end up in a network configuration in which all nodes emit a pulse at the exact same time. Each node couples with its neighbors' behavior through observation of "flashing" (packet transmission) times. Nodes adjust their flashing periods (Φ) to their neighbors', by advancing the current "phase" ϕ of their oscillation. This allows the entire network to converge to a synchronous operational state from an initial unsynchronized scenario, with no need for a central timing reference, and strong scaling properties, since only local neighbourhood information is required.

2.1.2 The FireFly-Gossip (FiGo) Protocol

The FiGo algorithm combines clock synchronization (from RFA) and information spreading (Gossip) for WSNs. Among the key concepts it uses are a "refractory period" of half a period ($\Phi/2$) after it fires, during which it will not adapt to any incoming pulses. The idea behind this is so that within each period there will be an ordering on node firing: neighbour pulses that occur in the half-period "before" a node was planning to fire will be treated as "ahead", and should therefore be synced to, and pulses that arrive "after" the node planned to fire (during the refractory period) should be treated as "behind", and not adapted to. FiGo then introduces the concept of "Message Suppression", with the idea that only the most "ahead" node need to fire within each period, and all nodes that were planning to fire within the next half-Φ after it should instead "suppress" their firing and adapt to the first node. It has extremely good single-hop performance, but unfortunately does not perform well in multi-hop, as will be discussed later.

2.1.3 Randomization in Self-Organized Synchronization

This paper investigates pulse-coupled oscillator (PCO) synchronization in wireless networks where synchronization messages can interfere with each other, rather than reinforce like pulses do in traditional PCO theory. The authors propose using randomization techniques to mitigate the negative impacts of interference on the synchronization process. Two main randomization approaches are explored: 1) stochastic power switching where nodes randomly alternate between high and low transmission power levels for synchronization messages, and 2) having each node randomly select its own "fire phase" when it sends synchronization messages, rather than all nodes using a common fire phase. Through analysis and simulations, the authors show that both techniques can significantly improve synchronization time compared to traditional non-randomized approaches, by reducing message collisions and interference. The randomized fire phase approach tends to perform

best. The paper also demonstrates applying the power switching concept to the popular Glossy flooding synchronization protocol as an example of using randomization for non-PCO schemes.

2.2 Wireless Sensor Networks (WSNs)

2.2.1 Trickle

Trickle is considered a seminal paper in the field of wireless sensor networks. It introduced several key concepts that result in properties desirable to WSNs, alongside simple and elegant algorithmic implementations. Among these, the notions of *exponential backoff* and *message suppression* are particularly relevant to this project, and have been adapted in SyncWave.

Wireless sensor networks consist of a large number of resource-constrained nodes that require periodic code updates for bug fixes, security patches, and functionality enhancements. The Trickle algorithm is designed to propagate and maintain code updates in WSNs in a robust, energy-efficient manner. It uses a "polite gossip" policy where nodes periodically broadcast code metadata to their neighbors. If a node detects that one of its neighbors has newer code, it requests that code image. Trickle's key features are its simplicity, low overhead, and inherent suppression of redundant data transmissions. The algorithm scales to any network density and avoids problems like network-wide broadcast storms. It is robust to network loss and disconnections by employing a "polite gossiping" approach with an adaptive, randomized transmission timing mechanism. This ensures that code propagates quickly when there are updates, while keeping bandwidth low when no updates are needed. Trickle can be used for reprogramming, data dissemination, and routing control in WSNs.

Chapter 3

Simulation

In order to facilitate the development of our protocol and demonstrate the drawbacks of various existing time synchronization protocols, we developed a simulation for testing time synchronization and PCO algorithms, and compared them against our own prototypes of SyncWave–which will be formally introduced in the following section. As will be seen in chapter 5, the implementation of time synchronization protocols on real hardware can require major changes from the theoretical algorithm, as assumptions no longer hold and dozens of other confounding variables can cloud the performance of the algorithm. Testing algorithms on a physical testbed can also require several orders of magnitude longer to run. Thus, this simulation proved an absolutely essential tool for rapidly prototyping and validating improvements on the theoretical algorithm.

3.1 Assumptions

Unlike Schmidt et al. [4], who solely evaluated their algorithm in a simulation, we are using the simulation only as a framework for prototyping and comparison. For this reason, we made several assumptions concerning the setup of the topology in order to not over-complicate the simulation design.

1. **Perfect Links**: Any two nodes that were connected by an edge in the provided topology were considered connected, and would receive a message with probability 1. In a more sophisticated simulation, one would employ Rayleigh Fading [5] to model the effect of a propagation environment on a radio signal.

2. **No Interference**: The simulation has no model for message interference, meaning that extremely high message throughput in dense network topologies do not degrade the radio communication channel. In reality, this is a real concern and will necessitate the Message Suppression optimization in section 4.3. One could model interference at the receiver in a manner similar to Schmidt et al., and express the probability of message reception in terms of the Signal to Noise Ratio [4]. An extension to this would be to model the capture effect, whereby at least one of the overlapping messages is likely to be successfully decoded, but not the others.

3. **No processing time**: This was also an assumption made by Schmidt et al., however in real implementations the processing time can become extremely relevant depending on the chosen period length. In the implementation section, a processing time of around ~ 4 ms per message was found, and when a 50 ms period is employed this can have serious repercussions for the algorithm's stability. This assumption is one of the reasons why the algorithm put forward by Schmidt et al. is unimplementable on real hardware without major modifications to account for hardware delays.

4. **No message propagation time**: The simulation assumed that the time from node deciding to transmit to message reception at a neighbouring node took zero time. In reality a variety of delays can increase this timing to the order of several milliseconds.

3.2 Simulation Setup

1. **Discrete time simulation**: The SimPy discrete time simulation package was used as a base, with each node operating as a separate agent. A single tick in the simulation was chosen to represent one microsecond, in order to simulate clock drift in the order of milliseconds, and also in order to have a significant granularity in message sending and arrival times.

2. **Clock Drift**: We accounted for clock drift within our simulation with a drift of 500 ppm, representing a typical value for a Low Frequency Hardware Clock. A value that proved to be slightly worse than was later observed in the physical testbed, but useful in order to simulate the effect of running the algorithm for longer time intervals.

3. **User-defined Topologies**: Unlike Schmidt et al. who generated topologies randomly using a Poisson Point Process on a 2D space, we chose to manually define a variety of worst-case topologies, which would prove far more insightful for worst-case timing guarantees than random graphs. Among these were the Barbell graph [6], Path (Linear) graph [7], Cycle graph [7], and Random Internet Autonomous Systems graph [8]. Of these, only the first two will be discussed for brevity.

4. **Random starting time**: Nodes are assigned a random starting time within the first two seconds of the simulation, allowing for variation in both starting phase and epoch. This randomization is seeded such that when algorithms are compared they have the same node starting times.

5. **Synchronization**: We define Sychronization within our model as the maximum pair-wise phase difference between all nodes in the graph at a specific instant in time being less than 2% of the period. Hence, the Time to Synchronize (TTS) is defined at the first tick at which the network is synchronized, and remains so for the next 1000ms.

3.3 Worst-case Topologies

Among the primary aims of the simulation was to discover and test worst-case topologies for the existing algorithms in literature, in order to concretize their worst-case guarantees on convergence. In this section we present one of these: the barbell graph.

3.3.1 Barbell Graph

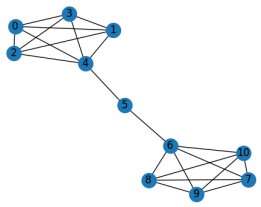

Figure 3.1: Barbell Graph with complete sub-graph size $n = 5$ and a single bridge node

"The n-barbell graph is the simple graph obtained by connecting two copies of a complete graph K_n by a bridge." [6] We additionally add a node to serve as a bridge node. This graph is selected due to the challenge it poses to both vanilla FiGo (Figure 3.2) and the Random Phase algorithm (Figure 3.4). This is shown visually in the following example graphs, performed on the same $n = 5$ Barbell graph with the same random seed for node initial firing times. The graphs are organized into four figures sharing the same time on the x-axis: the first shows the phase of each node in the network at the given time, which will take on a saw-tooth pattern as the phase resets as it surpasses the period. The second shows which nodes fires (broadcasts), suppresses (if another node has already fired in that epoch), and messages receptions. For the sake of comparison, an "epoch" counter for each node has been added to the implementation of all algorithms, and this is shown in figure three. Lastly, the instantaneous maximum pair-wise phase difference is shown in figure four, with a line demarcating the Time to Synchronize (if applicable).

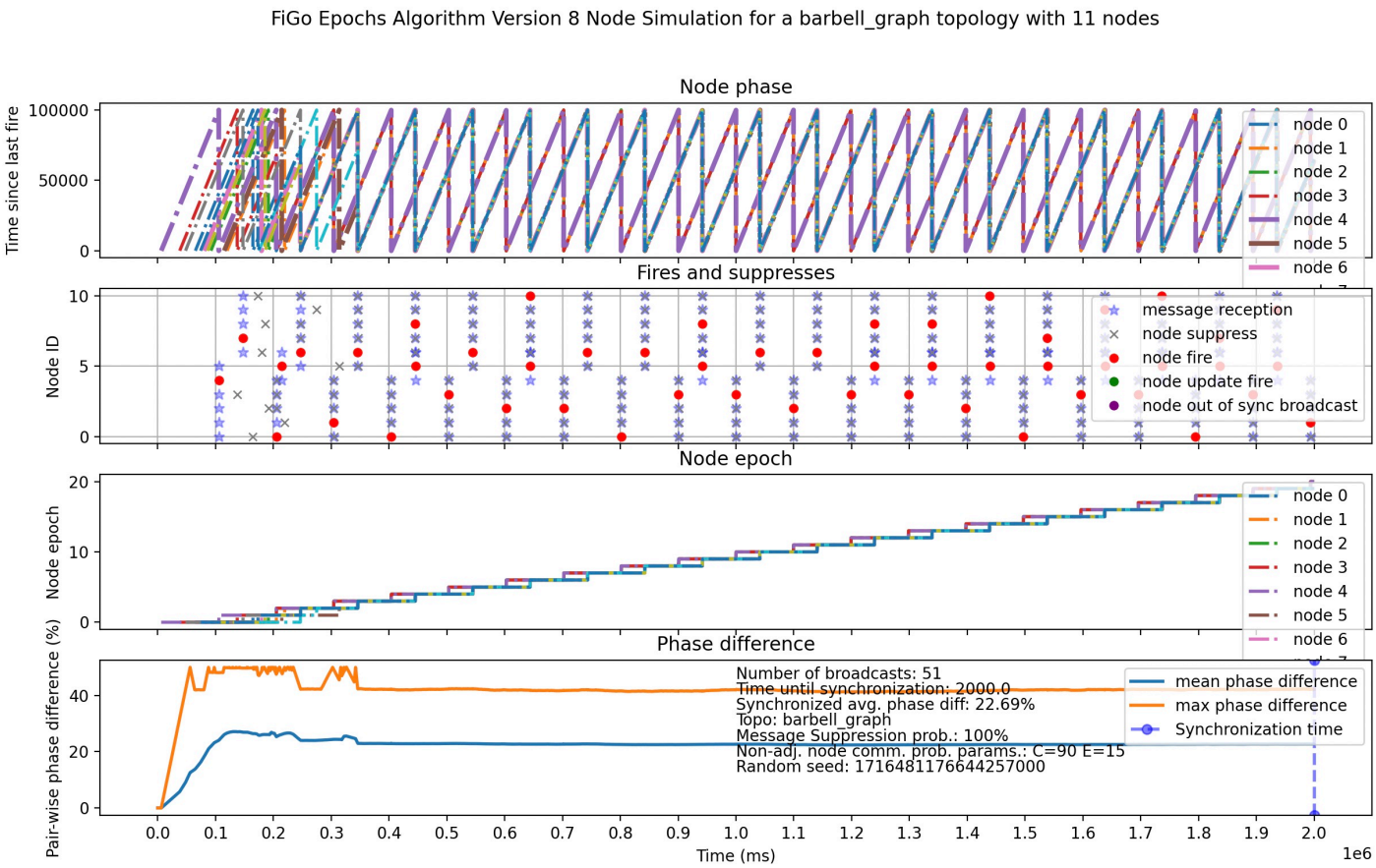

Figure 3.2: FiGo Algorithm with Message Suppression (Fails to Converge)

3.3.2 FiGo

To begin with, we show the performance of the FiGo algorithm on our $5-barbell$ graph. The individual sub-networks rapidly converge to a single local notion of time, however, they are not able to communicate this notion of time between the two sub-networks since communication is contingent upon three conditions: First, that the singular node in K_1 connected to the bridge node fires (with $p = \frac{1}{|K|+1}$), then that this is not ignored by the bridge node, that the bridge node retransmits this, and then again that the node on the opposite side adopts this and can retransmit. Given a time equal to 20 periods, the algorithm fails to converge at all.

Some of the key takeaways from vanilla FiGo are that deterministic message suppression can massively increase the time to fire in multi-hop networks, particularly if the clusters on either side of the hop are dense, and the connectivity low.

3.3.3 Bruteforce FiGo (No Message Suppression)

FiGo with no message suppression (implemented primarily as a proof of concept) serves to illustrate what "brute forcing" a time synchronization algorithm would look like. Every node fires every period, and continues to do so even after the network has synchronized. It converges quickly, but scales extremely poorly with the number of nodes. Furthermore, since the period determines both the firing rate and the speed of convergence, there is a clear tradeoff between setting a short period in order to synchronize quickly but then sending an enormous volume of wasted traffic, or setting a long period with fewer messages but then synchronizing much more slowly.

The key takeaway from this experiment is that any successful algorithm should find a solution that avoids this tradeoff entirely, and decouples the firing rate and time to synchronize from the period.

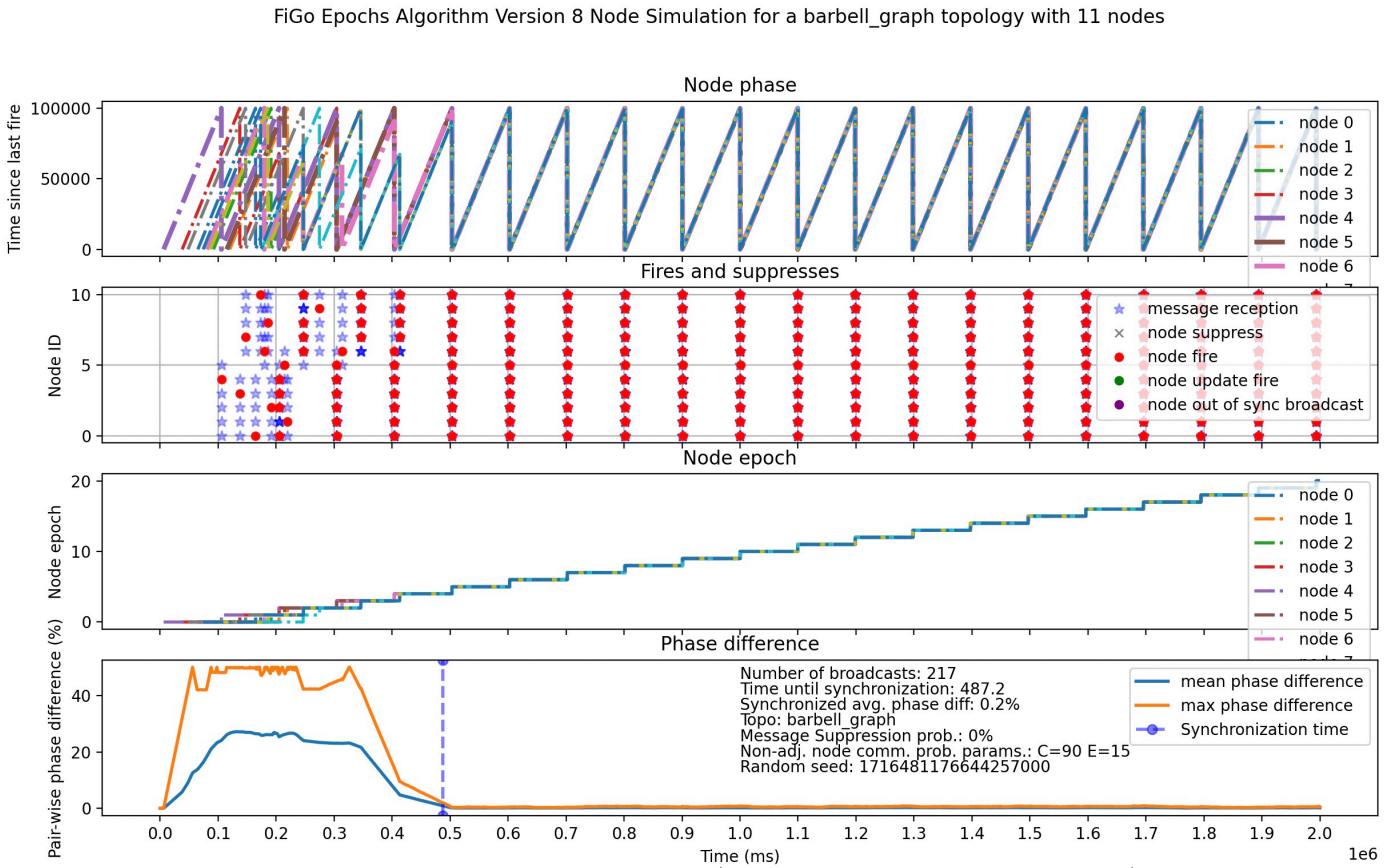

Figure 3.3: Bruteforce FiGo Algorithm (FiGo with no Message Suppression)

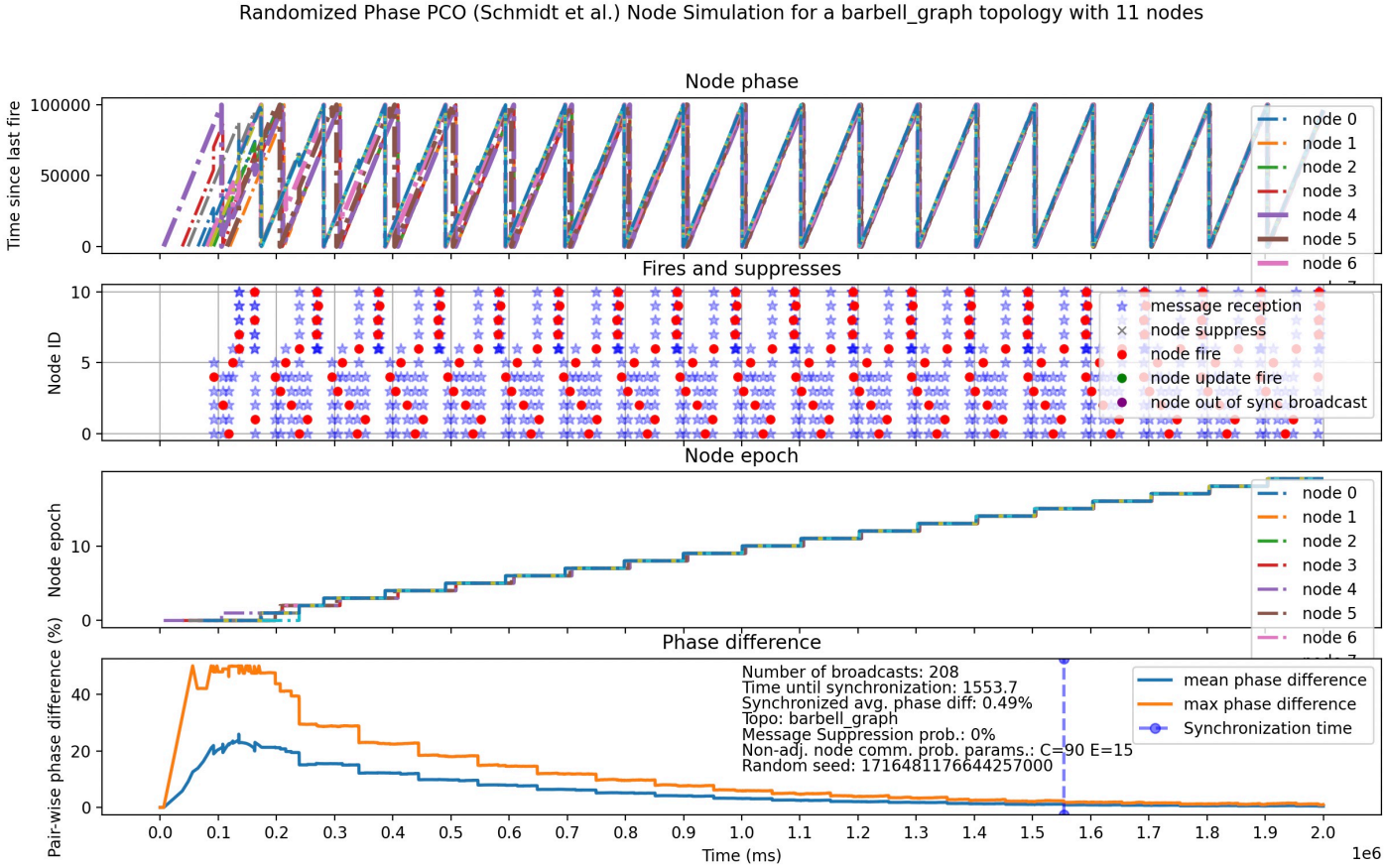

Figure 3.4: Random Phase (Schmidt et al.) with 50% adaption

3.3.4 Random Phase

Lastly, the random phase algorithm from Schmidt et al. is shown in Figure 3.4 (with a single random firing phase selected at wakeup), and suffers from similar issues to Bruteforce FiGo. The coupling between period, time to synchronize, and firing rate mean that the algorithm must choose to either synchronize fast and waste energy, or save energy and synchronize slowly. The one advantage of Random Phase is that it converges very predictably and smoothly, since every node is guaranteed to fire at some point in the period that has been selected from a uniform distribution.

3.4 Conclusion

In designing SyncWave, several graphs were chosen as particularly difficult cases that the existing PCO literature performed poorly on, and that any new algorithm should improve upon. We have shown that even with a plethora of idealized assumptions, these three state-of-the-art time synchronization algorithms perform extremely poorly on a worst-case Barbell topology. In the next section, we will use the takeaways from observing these algorithms to inform our algorithm SyncWave, and explain how we overcome this issues.

Chapter 4

SyncWave

In this chapter we propose the SyncWave decentralized time synchronization algorithm. In section 4.1, we will provide a high-level overview of SyncWave, followed by a more detailed description of the algorithm and its core concepts in section 4.2. In doing so, we will discuss how SyncWave satisfies the objectives outlined in section **??**, as well as a general overview of its theoretical scaling and timing guarantees.

4.1 SyncWave In Brief

SyncWave is a Message-Coupled Oscillator with Maximum Time Synchronization. Instead of operating on pure "pulses" like PCO, it exchanges information with its neighbors by means of packets, similar to other wireless protocols. Like PCO, SyncWave has an internal oscillator that it attempts to synchronize with its neighbors. This is where similarities with PCO end, however.

SyncWave extends PCO by maintaining an internal 'epoch' counter, counting how many times an internal timer with a constant period has expired. The combination of Epoch and Phase (how far along in the timer we are) is sufficient for defining a notion of "time" on the node. This differs significantly from classical Pulse-Coupled Oscillators which, since they have no way of communicating the epoch between nodes, attempt only to synchronize their phase. This means that when nodes are firing, it is difficult for them to tell if another node is late to the current firing, or just very early for the next. Keeping track of the epoch simplifies and enables some very powerful interactions in the protocol.

Every so often, the node will 'fire', broadcasting the current value of its 'time', defined by the tuple (e, ϕ), to its neighbors as a message. Once these neighbors receive the message, they will then compare the received state to their own, and set their local state to whichever of the two is greater. In centralized time synchronization algorithms, this mechanism is referred to as Maximum Time Synchronization (MTS).

Node 'firing' in SyncWave is entirely decoupled from the internal timer and phase, unlike PCO: a separate timer chooses when to fire randomly from a "firing interval" I, which can vary between $[I_{min}, I_{max}]$. This random firing time allows the network to reduce packet collisions, in comparison to message-based PCO algorithms.

In SyncWave, broadcasts are equivalent to disseminating information to help neighbors synchronize. Therefore, the algorithm shortens the firing interval when the network is first starting up or appears desynchronized to the node, and lengthens it using exponential backoff when the node goes without hearing any messages that are out of sync with itself. This period-modulation mechanism makes SyncWave exceedingly fast at converging when the network is out of sync (in the order of 1.1 s for a 27-node, 18-hop network), and extremely resource-efficient once synchronized (less than 1 message per 5 minutes for 163 nodes across 7 hops).

Finally, SyncWave uses Message Suppression to limit the number of messages sent within a given firing interval, so that in extremely dense networks, nodes will avoid sending if they have heard another node propagate a similar time to their own. Depending on the network topology this can reduce broadcasts 10x+, massively reducing collisions, which speeds up convergence, and frees the radio for app-level uses.

List of Symbols

Symbol	Description	Units	Origin
e	Epoch	$\mathbb{N}$	Variable
ϕ	Phase	μs	Variable
ψ	Firing Counter	μs	Variable
ψ_{fire}	Firing Counter Time at which to Transmit	μs	Variable
I	Firing Interval	μs	Variable
c	No. Messages From Unique Senders Heard This Firing Interval	$\mathbb{N}$	Variable
Φ	Epoch Length / Period	μs	User-defined
I_{min}	Minimum Firing Interval	μs	User-defined
I_{min}	Maximum Firing Interval	μs	User-defined
k	Message Suppression Threshold	$\mathbb{N}$	User-defined
$\hat{c}$	Empirical Propagation	μs	Empirical
ϵ	Out of phase threshold	μs	Empirical
β	Exponential Backoff base	$\mathbb{N}$	Empirical

4.2 SyncWave in Detail

In this section we will describe SyncWave more formally, explaining the constituent concepts, the reasoning behind them, and how they fulfill the features required of the protocol.

4.2.1 Phase and Epochs

At the most basic level, SyncWave is similar to standard PCO algorithms: it has a "phase" counter ϕ, based on the node's internal clock or timers, that increases monotonically in the range $\phi \in [0, \Phi]$ where Φ is referred to as the period. Upon reaching Φ, the phase resets back to 0 and increments an "epoch" counter e–the first difference from standard PCO. We use the combination of epoch and phase to define the node's local "time" $t = e \cdot \Phi + \phi$. Therefore, stated in terms of these variables, the aim of the algorithm is to synchronize all nodes' epoch and phase, and thus their local time, to a single value.

It should be noted that the inclusion of an epoch counter does not make sense for traditional PCO algorithms, which originate from the study of dynamical systems and in which the only form of communication are 'pulses' of energy; for these, the only information-sharing construct available is the timing of their pulses, so there is no way to communicate the *total number* of pulses that each has performed. Thus, classical PCO is unable to define any total ordering on the nodes' local times, resulting in firing synchrony, not time synchronization.

The key that enables SyncWave to agree on an epoch, is to note that when PCO is implemented in the wireless medium, any practical implementation will use packets as a primitive, which can easily accommodate other metadata (such as the epoch). This is inspired by Schmidt et al. [4] in their 2023 paper, in which they included the phase in their message, allowing them to send messages at a random "firing phase" in their period, reducing packet collisions. expands on this by including an id, epoch, and phase in all broadcasts.

4.2.2 Maximum Time Synchronization

Having the ability to communicate our epoch, not only our phase, to our neighbors proves to be extremely powerful: allowing us to define a total ordering on time in the network. Thus, upon message reception $(id_{msg}, e_{msg}, \phi_{msg})$ a node will compare their local time $t = e \cdot \Phi + \phi$ to the message's time $t_{msg} = e_{msg} \cdot \Phi + \phi_{msg}$, and choose the maximum of the two as its new value for the local time.

This gives us some desirable properties essentially for free. In SyncWave, there is no need to perform consensus to decide on a reference node to synchronize to, since nodes will converge to the same maximal time organically, thereby providing network-wide agreement.

Choosing maximum time synchronization also preserves liveness properties, unlike a bivalent scheme (adapting to any incoming message, such as Schmidt et al.) which could oscillate forwards

Algorithm 1 SyncWave algorithm

$\phi \leftarrow 0$
$e \leftarrow 0$
$I \leftarrow I_{min}$
$\psi \leftarrow 0$
$\psi_{\text{fire}} \leftarrow randint(0, I)$
$c \leftarrow 0$
heard_ids $\leftarrow \{\}$
while $True$ **do**
    if $receive\_message(id_{msg}, e'_{msg}, \phi'_{msg})$ **then** ▷ Event: Message Reception
        $t_{msg} \leftarrow e'_{msg} \cdot \Phi + \phi'_{msg} + \hat{c}$
        $t \leftarrow e \cdot \Phi + \phi$
        $e_{msg} \leftarrow \lfloor t_{msg} \div \Phi \rfloor$
        $\phi_{msg} \leftarrow t_{msg} \mod \Phi$
        if $t_{msg} > t$ **then** ▷ We're behind
            $e \leftarrow e_{msg}$
            $\phi \leftarrow \phi_{msg}$
            if $t_{msg} > t + \epsilon$ **then** ▷ We're behind by a lot, tell our cluster
                ResetFiringInterval()
            else if $c < k \wedge id_{msg} \notin$ heard_ids **then** ▷ Sender had roughly same time as us
                heard_ids $\leftarrow$ heard_ids $\cup\, id_{msg}$
                $c \leftarrow c + 1$
            end if
        else ▷ We're ahead
            if $t_{msg} < t + \epsilon$ **then** ▷ We're ahead by a lot, get sender up to date
                ResetFiringInterval()
            else if $c < k \wedge id_{msg} \notin$ heard_ids **then** ▷ Sender had roughly same time as us
                heard_ids $\leftarrow$ heard_ids $\cup\, id_{msg}$
                $c \leftarrow c + 1$
            end if
        end if
    end if

    if $\phi > \Phi$ **then** ▷ Event: Epoch Timer Expired
        $\phi \leftarrow 0$
        $e \leftarrow e + 1$
    end if

    if $\psi > \psi_{\text{fire}}$ **then** ▷ Event: Firing Timer Expired
        if $c < k$ **then**
            Tx(id, e, ϕ)
        end if
        $c \leftarrow 0$
        heard_ids $\leftarrow \{\}$
        $I \leftarrow min(\beta \cdot I, I_{min})$
        $\psi_{\text{fire}} \leftarrow randint(0, I)$
        $\psi \leftarrow 0$
    end if
end while

function ResetFiringInterval()
    $c \leftarrow 0$
    heard_ids $\leftarrow \{\}$
    $I \leftarrow I_{min}$
    if $I < \psi_{\text{fire}} - \psi$ **then** ▷ Don't reset firing timer if we're about to fire
        $\psi_{\text{fire}} \leftarrow randint(0, I)$
        $\psi \leftarrow 0$
    end if
end function

and backwards in time, based on its neighbors, effectively making no progress towards synchronization. Or a minimum time scheme, which would be need to re-synchronize for every newly joined node or could potentially make no progress at all in the presence of faulty nodes.

Epoch + Phase and Maximum Time Synchronization alone provide us with guaranteed convergence. In the following sections, we will look at the timing of our broadcasts in order to both converge faster, and to use fewer messages to do so.

4.2.3 Randomized Firing Phase

In SyncWave, broadcasts are equivalent to disseminating information. For this reason, we would like to be able to choose when to share information at arbitrary times and timescales. In order to achieve this, a break is required from both traditional PCO algorithms, which can only fire once per period, as well as Schmidt et al., which can choose the phase at which to fire within the period arbitrarily, but is limited to one fire per period.

In order to solve this problem, SyncWave goes a step further than Schmidt et al., and entirely decouples a "firing timer" ψ from the phase ϕ. This firing timer increases monotonically in the range $\psi \in [0, \psi_{fire}]$, where ψ_{fire} represents the chosen firing time, at which point the node will broadcast and reset ψ to 0. Thus, ψ and ψ_{fire} are entirely independent of the current epoch and phase, and ψ_{fire} can assume values significantly shorter, or longer, than a single period Φ.

Furthermore, SyncWave builds on the concept of Random Phase selection from Schmidt et al., choosing ψ_{fire} randomly from the uniform distribution parameterized by the "firing interval" I: $\psi_{fire} \sim \mathcal{U}(0, I)$. This random sampling helps ensure that node firing times are evenly spaced in time, minimising packet collisions. Additionally, the firing interval I gives us the ability to easily scale the range from which firing times are selected, without affecting the period Φ. Thus, if we want to rapidly synchronize I can be set extremely low, and if we want to save energy, we can set I to be in the order of seconds or minutes.

We now have the groundwork in place for dynamically adjusting our firing rate depending on how out-of-sync the network is. This brings us to our next section: Exponential Backoff.

4.2.4 Exponential Backoff

Our goals with SyncWave are two-fold: first, we want *rapid synchronization*, where convergence time is critical, and resource usage is a secondary consideration; secondly, once the entire network is time synchronized, our goal shifts to maintaining synchronization with *minimal resource consumption*, including the number of broadcasts and radio uptime, ideally allowing the nodes to enter deep sleep or perform app-level activities during this time.

Traditional PCO algorithms, as well as most WSN time synchronization algorithms, chose to optimize for only a single objective:

- If they choose to optimize for time to synchronize, then they must either keep the high-frequency synchronization and waste bandwidth and energy, or they need to manually re-run time synchronization from scratch at regular intervals, which sees significant clock drift between intervals and a massive flood of messages at each re-synchronization, with the same time complexity as the first.

- Or the other, far more common, option is to perpetually synchronize, but at a much slower messaging rate (effectively always in the maintenance regime) meaning that initial synchronization can take in the order of minutes to hours. [3]

To achieve these two conflicting goals, SyncWave borrows the concept of Exponential Backoff from Trickle [9], a seminal paper in the field of code propagation in WSNs.We implement Exponential Backoff for time synchronization in SyncWave as follows:

1. On startup, when the network is out of sync, nodes are initialized with a firing interval I equal to some minimal interval I_{min}. This ensures that rapid information propagation occurs, which is necessary for rapid synchronization.

2. After each firing interval of length I if the node has not heard any incoming messages with a "conflicting" time value, it will update its value of $I \leftarrow \beta \cdot I$. This provides the property that if nodes are sufficiently synchronized (and hence all messages received contain $(e_{msg}, \phi_{msg}) \approx (e, \phi)$), then the rate of information transfer will decrease as it is no longer necessary. This

backoff is limited to a value I_{max}, which represents the lowest communication rate necessary to account for clock drift in the network, and *maintain* synchronization. We can estimate bounds for clock drift, and thus a sensible I_{max} can be estimated for a given topology and hardware, often 5+ minute range.

3. If, on the other hand, a node receives a "conflicting" time value in a message, i.e. a value of (e_{msg}, ϕ_{msg}) either $\epsilon\mu s$ ahead of it, or $\epsilon\mu s$ behind it, the node will reset its firing interval to I_{min}. The reasoning is as follows: In the first case, if a node receives a message with a significantly greater timestamp than its own, it will want to immediately inform its neighbors of this information, so that they can rapidly synchronize to the new information. In the second case, if the node receives a message significantly behind its own, then it will want to alert them of the true greatest time in the network, and thus should reset its firing interval to do so.

To the best of our knowledge, SyncWave is the first decentralized time synchronization algorithm to leverage exponential backoff for adapting the rate of communication to the desynchronization of the network. And while simple, this mechanism handles several challenging dynamic scenarios very well:

- **Scenario: Node joining cluster after synchronization.** In this scenario, the main cluster have already synchronized and are in a maintenance regime, sending very few messages. We exploit the bi-direction of communication, such that: When the new node arrives within reception range of the synchronized cluster, A. if it fires, and 1. is behind the cluster, then a receiving node from the cluster will resets its firing interval and broadcast in within I_{min}, to which the new node will adapt. 2. it is ahead of the cluster, then the cluster will receive the message and reset their I to I_{min}, adapting to the new greatest time. B. The cluster fires, 1. the new node is ahead of the cluster, so it will reset its firing interval and inform the cluster of its own time within I_{min} 2. it is behind: it immediately adapts, and the network is synchronized.

- **Scenario: Temporary Network Partitioning.** SyncWave is resilient to temporary network partitioning, since the two clusters will continue to increment at a rate bounded purely by the node with the maximum time's hardware clock, and any minimal drift inaccuracy it may contain. This means that even if a network is temporarily broken into clusters, they will remain synchronized for a time parameterized by the underlying hardware clock's random drift.

- **Scenario: Cluster merging.** Two disjoint clusters could come into contact after each has synchronized to a separate notion of time. In consensus-based algorithms such as Demos [10] it can be a tricky to decide which cluster should synchronize to the other, or if the entire network needs to re-synchronize entirely (which takes time). In the case of FiGo [11], it is possible that the two could never synchronize at all. SyncWave, however, handles this case with ease thanks to it tracking both e and ϕ for Maximum Time Synchronization. The time until the first message to be exchanged between these two synchronized clusters is bounded by $I_{max} \div 2 \cdot (N_1 + N_2)$ where N_1, N_2 are the number of nodes in each cluster. From the moment a node hears a message from a different cluster, it will reset its firing interval to I_{min} and depending on if $t_{c1} > t_{c2}$ holds, the node will either propagate this new information to its neighbors in the cluster for them to adapt, or send out a message within I_{min} to inform the other cluster that they are behind and should synchronize to it instead.

4.3 Polite Gossip / Message Suppression

Now the key components of the algorithm have been laid out, we can add an optional optimization called Message Suppression (MS) to reduce the number of redundant broadcasts in dense networks. These "dense" networks are characterized by a high average node degree relative to the number of nodes in the graph, meaning that if you hear a message, there is a high probability that it has been heard by your neighbors as well. Therefore, if you have already heard c unique neighbours fire within a given firing interval I and $c > k$ some threshold k, you can "suppress" your next broadcast, since you'll just be repeating the same information that your neighbours have just heard. This is

the principle behind the Firefly Gossip Algorithm (FiGo) [11], which is able to halve the number of messages required for synchronization when compared to a similar protocol without MS [12].

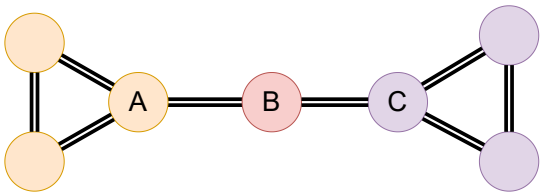

Figure 4.1: Hidden Node Problem

Node B is the only connection between the two clusters, but might go a long time before firing if it suppresses because A fired before it, delaying synchronization

However, there is a drawback to naive/indiscriminate Message Suppression: due to the hidden terminal problem, a node may not be aware that it is the only link to another cluster (e.g. B in Figure 4.1), meaning that if it always suppresses, the information may not propagate until it randomly happens to be the first to fire, massively slowing down convergence in certain low-connectivity topologies. In FiGo, 4% of trials failed to synchronize at all, effectively jeopardizing convergence guarantees in the worst case[11]. For this reason, we make Message Suppression an optional optimization, intended for dense and high-connectivity networks. Moreover, SyncWave's version of MS still vastly outperforms FiGo on lower-connectivity networks thanks to the following improvements:

- We only consider *similar* times (i.e. within distance ϵ) of our own to increment the value of c, and we only consider unique senders by using a temporary $heard\_ids$ buffer of received id_{msg}s. This avoids the issue of us never having the chance to send if another node with a very short firing interval is frequently sending similar messages.
- The chance of firing in SyncWave with MS is stochastic, thanks to the random firing phase, sampled from our firing interval. This is in sharp contrast to FiGo, for which it is possible for a node with a slightly slower than average hardware clock to always suppress and never fire, which compromises worst-case convergence guarantees.
- Unlike FiGo, which performs MS within each period Φ (which has a constant length), we instead use the firing timer ψ as the length of our message suppresion window (after which point we reset c to 0 and clear the $heard\_ids$ buffer. Thus, SyncWave's Message Suppression is adaptive to the timescale and desynchronization of our network.

Chapter 5

Implementation

In this chapter we will explain the process of implementing the SyncWave algorithm on nrf52840dk microcontrollers (MCUs) using the RIOT embedded operating system. As will be described, this process of translating the theoretical algorithm onto actual hardware is far from trivial, and required some major changes to the semantics of the different Procedures in Algorithm 1 and their intercommunication. Furthermore, as embedded programming was a field entirely novel to me at the start of the project and not something that I had come across in my prior courses, a great deal of work was required to get up to speed and understand the subtleties associated with it.

We will start by describing the hardware in section 5.1, followed by the RIOT operating system (5.2). Next, we will discuss our use of the timing primitives in RIOT OS as well as the considerations involved with threading and synchronization when translating SyncWave from theory to real hardware. Then we will provide a detailed look into the implementation on a per-thread basis, discussing the decisions and adaptions made to accomodate the tools available within the embedded OS. Finally, we will state the message encryption scheme employed, and examine how the implementation of SyncWave may be integrated into existing applications by means of a library API.

5.1 nrf52840 MCU and Network Stack

SyncWave was implemented for the nrf52840 System on a Chip (SoC), which is a modern MCU with multiprotocol support, including Bluetooth Low Energy, Bluetooth Mesh, Zigbee, and several other proprietary protocols. It is compatible with a wide range of applications including as a (custom) communication chip for Crazyflie 2.1 drones [13]. The nrf52840 is specifically chosen for its robust support within the RIOT operating system, which is widely utilized in networked, memory-constrained systems.

We chose to implement the protocol at the network layer, instead of the medium access control (MAC) layer in this proof of concept implementation. The downsides of this decision are less control over the execution and timing of the protocol, leading to slower convergence time and tightness. However, the benefits are huge: since we built our algorithm entirely within the RIOT operating system, our implementation makes almost no assumptions as to the underlying hardware upon which it is running, vastly increasing the portability and simplifying the implementation process to be hardware-agnostic. After implementing our algorithm once, we were able to compile it with almost no changes to the nrf52840dk, adafruit-feather-nrf52840, and iotlab-m3 architectures.

5.2 RIOT Operating System

We chose to implement SyncWave in the RIOT embedded operating system. RIOT-OS is intended for networked, memory-constrained systems and is commonly used within the field of wireless sensor networks. RIOT includes several useful features by default, including a slightly more abstracted interface compared to alternatives such as Contiki-NG (hence the slogan "The friendly Operating System for IoT!"), high resolution long-term timers, and support for over 200 different MCUs including nrf52840 and STM32.

Figure 5.1: nrf52840 SoC (dongle variant), image via Digikey [2]

Using RIOT libraries was helpful for providing some level of abstraction from hardware specifics and peripherals, allowing us to compile the same algorithm implementation for multiple target boards. This was particularly helpful when evaluating the algorithm on the FIT-IoT Lab experimental testbed, as discussed in Chapter 7.

5.3 Implementation Challenges

The theoretical algorithm assumes that we are able to run the entire program within a single loop, and have access to counters ϕ and ψ which increment automatically based on the node's internal clock until we reset them. Unfortunately these assumptions do hold in our hardware environment and for this reason significant modifications are required to translate the theoretical algorithm into the language of threads, message passing, and low-level timers. We will start by presenting an overview of the general challenges and our approach to solving them, before delving into the algorithm implementation on a per-thread basis.

5.3.1 Timers

An implementation of the counter variables ϕ and ψ would need to satisfy the following requirements:

1. Real-time incrementing that is unaffected by code execution and interrupts
2. Automatically resetting to 0 once an arbitrary threshold is reached
3. A current value that must be mutable, such that the counter could be moved forwards in the event of a message arriving with a greater phase than the one we currently store.

The first condition makes our options incredibly narrow: any functionality that occurs "real-time" is managed extremely carefully in operating systems, since this limits its ability to make scheduling decisions within that timeframe. For this reason, the only real-time primitive we have access to in RIOT-OS is a singular Real-Time Clock (RTC). On its own, this clock does not meet our requirements, since it is not mutable (given that other parts of the OS rely on it), and does not reset.

RIOT helpfully provides a timing library called ZTimer, which abstracts the underlying hardware's real-time clock and multiplexes accesses to it by means of "virtual timers" on a given clock. These virtual timers are fairly simple, consisting of only a future timestamp at which to trigger the timer and a callback to be executed on timer wakeup, but have the advantage that we are able to create an arbtirary number of them. Thus, we proceed as follows:

In lieu of moving the clock itself, our solution is to operate on an "offset" ϕ_{offset} from the initial clock time, and define our current time to be $\phi = \text{ztimer\_now(ZTIMER\_USEC)} - \phi_{\text{offset}}$, where ZTIMER_USEC is the ztimer-provided pointer to a real-time clock with micro-second (μs) accuracy. This setup allows us to arbitrarily adjust our local notion of time, while preserving the 'real-time' aspect of the clock, meaning that increments will be unaffected by user-space code execution or scheduling decisions from the OS–satisfying (1) and (3).

And in order to reset our clock at a specific threshold (2), we will schedule a virtual timer for the time $\text{ztimer\_now(ZTIMER\_USEC)} - \phi_{\text{offset}}$ is expected to exceed the period Φ, at which point a callback will be triggered to check if $\phi >= \Phi$ holds and if so, reset ϕ by setting $\phi_{\text{offset}} = \text{ztimer\_now(ZTIMER\_USEC)}$.

5.3.2 Threading and Synchronization

While the theoretical algorithm is effectively running in a busy-loop, this is not practical when implemented on real hardware, since some operations such as message reception are blocking and others processing-intensive, meaning that the entire system's turnaround time would be bottlenecked by the slowest operation. Furthermore, for our use cases it desirable to minimize energy consumption where possible, and thus finding places to put the process or system to sleep are paramount.

In order to achieve these goals we factor our algorithm into 5 separate threads. This comes with a host of additional considerations, ranging from shared memory management to thread scheduling priorities to inter-process communication (IPC) to thread sleeping. This will be further detailed on a per-thread basis section 5.4, but we will discuss some of the commonalities as follows:

- **Shared State**: A reference to a shared state object is passed to all threads in order to facilitate immediate retrieval and modification of variables common to multiple threads. If managed improperly, this shared state could be a serious source of bugs, such as variables changing values mid-thread execution due to RIOT's preemptive scheduling.

  There are only two threads which are capable of writing to the epoch e and ϕ_{offset}: the reception thread and the epoch timer thread. The common pattern in these threads is that of reading these variables, and then depending on their current state, potentially writing to them. If not properly synchronized one thread may read a value that is overwritten subsequently, resulting in undefined behaviour. Thus, in order to protect critical sections of the code, we use a mutex to protect the epoch counter and the ϕ_{offset}. The overhead associated with acquiring the mutex was found to be negligible, thanks to the simple implementation of mutexes in RIOT, as well as the minimal computation performed in the critical sections. A significant reduction in synchronization accuracy was observed without such synchronization mechanisms.

- **Scheduling Priority**: The highest priority is assigned to the Reception and Send threads, whose execution is required to be as fast as possible in order to maintain the correctness of the algorithm, for the Reception thread, and to maintain high fidelity to the estimated message propagation time $\hat{c}$ for the Send thread. Next in priority is the Epoch Timer thread, to minimize the delay between ztimer_now(ZTIMER_USEC) $- \phi_{\text{offset}} >= \Phi$ holding and the epoch counter being incremented. And in lowest priority we have the Fire thread, since choosing to fire a couple μs later has essentially no impact on the time to synchronize and does not affect the correctness or accuracy of the algorithm.

- **Inter-Process Communication** (IPC): In addition to the shared state, we employ two other means of communicating information between threads. In cases where we want to perform communication asynchronously between two threads, such as between the Reception thread (which just receives messages) and the Update thread (which acts according to their contents), we employ a message queue pattern using RIOT's built-in IPC.

  In cases where immediate action from another thread is required, we take the approach of interrupting the thread's sleep, and having it automatically perform checks on the shared state. This is how the Update thread resets the Epoch Timer thread and asks the Send thread to fire.

- **Thread Sleeping & Wakeups**: Lastly, we try to operate in an event-driven paradigm, in the sense that each thread will only perform a minimal amount of code before either: A. blocking until a packet is received by radio or an IPC message is received (which is equivalent to the thread sleeping); or B. scheduling a virtual wakeup timer for x μs in the future, and calling thread_sleep directly.

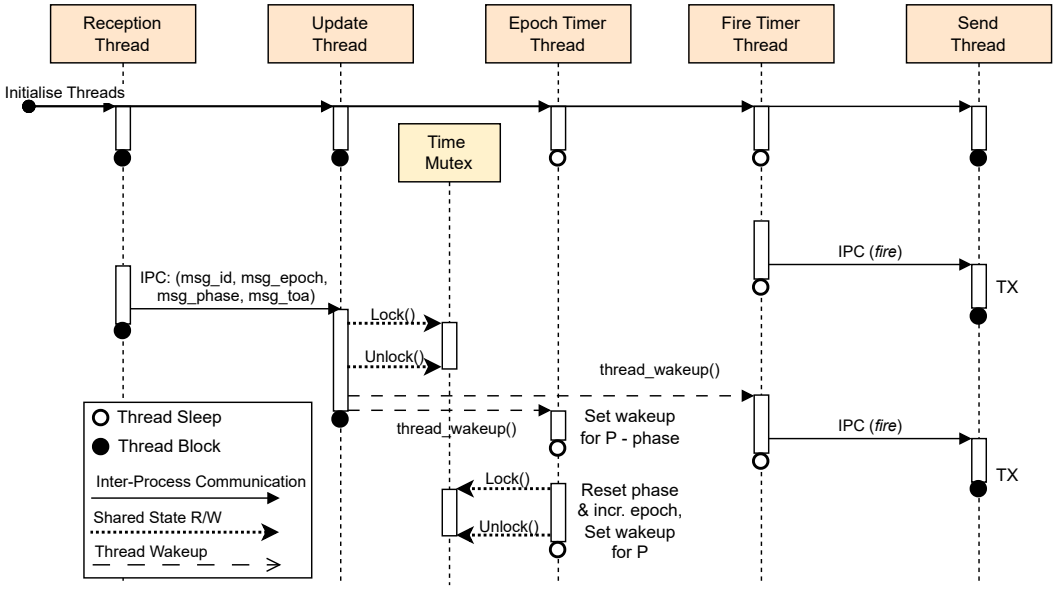

Figure 5.2: Process Diagram of Thread Interleaving and Communication

List of Symbols

Symbol	Description	Units	Origin
e	Epoch	$\mathbb{N}$	Variable
ϕ	Phase	μs	Variable
ϕ_{offset}	Phase Offset From Internal Clock	μs	Variable
ψ_{offset}	Firing Offset From Internal Clock	μs	Variable
I	Firing Interval	μs	Variable
c	No. Messages From Unique Senders Heard This Firing Interval	$\mathbb{N}$	Variable
Φ	Epoch Length / Period	μs	User-defined
$I_{\min}$	Minimum Firing Interval	μs	User-defined
$I_{\max}$	Maximum Firing Interval	μs	User-defined
k	Message Suppression Threshold	$\mathbb{N}$	User-defined
$\hat{c}$	Empirical Propagation	μs	Empirical
ϵ	Out of phase threshold	μs	Empirical
β	Exponential Backoff base	$\mathbb{N}$	Empirical

5.4 Algorithm Implementation in Detail

The following section describes the SyncWave implementation on RIOT-OS in detail. We explain the algorithm from the perspective of each thread in turn, justifying key decisions and adaptions from the original.

5.4.1 Reception Thread

The Reception Thread, detailed in the algorithm snippet below, is responsible for message reception from other nodes. The process of reception (RX) has been purposefully factored in to a separate thread, in order to maximise the thread's availability to receive a new message. Thus, in the interest of keeping this latency-sensitive thread lightweight, the reception thread only timestamps the message arrival time, decrypts the message, then forwards these values using IPC to the Update Thread's queue.

```
1: procedure Reception Thread                                   ▷ Priority = 2
2:     Initialize socket
3:     while True do
4:         Block until message received from socket
5:         msg_toa ← ztimer_now()
6:         id_msg, φ_msg, e_msg ← decrypt_message()
7:         Send (id_msg, e_msg, φ_msg, msg_toa) to Update Thread
8:     end while
9: end procedure
```

5.4.2 Epoch Timer Thread

The Epoch Timer Thread has a simple task, theoretically: to reset the phase ϕ once it exceeds the epoch length Φ, and on doing so, increment the epoch counter e. In practice, this is slightly more complicated. It is desirable for the thread to sleep right up until $\phi >= \Phi$ holds (where ϕ is defined as the current value of the internal real-time clock, accessed via ztimer_now() - ϕ_{offset}). However, the ϕ_{offset} can be changed by the Update Thread as well – meaning that if it changes the phase offset, as a response to hearing a message with an $e + \phi$ greater than ours, then the timer thread should wake up earlier in order to account for this change.

This behaviour is implemented as follows:

- Having the Epoch Thread schedule a virtual timer using ztimer to wake it up in ϕ μs (line 21), then calling thread_sleep() (line 20).
- Allowing the Update Thread to preempt the Epoch Timer's sleep using a call to wakeup_thread() if we have moved ϕ forward, in which case the Epoch Timer Thread will cancel its previously scheduled timer (line 12) and set a new shorter one at the bottom of the loop.
- Allowing the timer thread to gracefully handle cases where it may have overslept, by setting the next epoch's phase to the amount by which the current phase exceeds the period (line 17)
- Encasing reads and writes to e and ϕ_{offset} in a mutex, to prevent the values being changed mid-way through and leading to unexpected results (lines 13 and 19).

```
10: procedure Epoch Timer Thread(e, φ_offset, time_lock, I)      ▷ Priority = 3
11:     while True do
12:         Remove previous scheduled wakeup timer               ▷ (nullop if not scheduled)
13:         mutex_lock(time_lock)
14:         now ← ztimer_now()
15:         if now − φ_offset >= Φ then
16:             e ← e + 1
17:             φ_offset ← now − ((now − φ_offset) mod Φ)
18:         end if
19:         mutex_unlock(time_lock)
20:         Set wakeup timer in ztimer to trigger Φ − (ztimer_now() − φ_offset) from now
21:         thread_sleep()
22:     end while
23: end procedure
```

5.4.3 Fire Timer Thread

The Fire Timer thread schedules future timestamps to broadcast, then waits for these timestamps, using a mechanism similar to the Epoch Timer thread. It has its own offset ψ_{offset} from the ztimer real-time clock, as well as timestamp ψ_{fire} representing the next desired broadcast time, sampled from the current firing interval I. In each loop it uses the same pattern of removing the previously scheduled wakeup timer (if it was preempted by the Update thread, otherwise a

NULLOP), performing its checks to see if it needs to fire or assign a new ψ_{fire} and ψ_{offset}, then setting a wakeup timer depending on the timer remaining until ψ_{fire} and calling thread_sleep() – allowing it to handle cases where it was woken up mid-sleep gracefully.

```
24: procedure FIRE THREAD(I, c, heard_ids_buffer)                    ▷ Priority = 5
25:     while True do
26:         Remove previous scheduled wakeup timer
27:         now = ztimer_now()
28:         if now − ψ_offset >= ψ_fire then
29:             Send fire message to Send Thread
30:             ψ_offset ← now
31:             c ← 0
32:             heard_ids_buffer ← {}
33:             I ← min(β · I, I_max)
34:             ψ_fire ← randint(ϵ, I)
35:         else if I < ψ_fire − (now − ψ_offset) then        ▷ If firing interval was just reset,
36:             ψ_fire ← randint(ϵ, I)                       ▷ we might want to pick a new, shorter, time to fire
37:         end if
38:         Set wakeup timer in ztimer to trigger ψ_fire − (ztimer_now() − ψ_offset) from now
39:         thread_sleep()
40:     end while
41: end procedure
```

5.4.4 Update Thread

The Update Thread is responsible for updating the node's internal state depending on the messages forwarded by the Reception Thread. The way in which it does this is similar to its theoretical definition, however, with some key differences:

- As with the Epoch Timer Thread, the Update Thread is capable of both reading and writing to the ϕ_{offset} and e variables, and so this section (lines 30-43+51) must be wrapped in a mutex in order to avoid unexpected behaviour.
- The epoch and phase received in the message are adjusted to account for A. the processing delay $\phi_{processing}$ that has occurred since they were first received at time of arrival toa_msg, and B. the expected propagation time for a message $\hat{c}$, an estimation of the delay from a neighbour making a call to its radio until the message is received and decrypted locally (lines 34-35).
- As described in the Epoch Timer Thread, if the Update Thread alters the value of ϕ_{offset}, then the current sleep time of the Epoch Timer will be out of date. Thus, the Epoch Timer thread should be woken up immediately to reschedule its sleep (lines 40). (Note: this will only occur if the node perceives receives a message 'ahead' of it in time, so this wakeup process will only happen with a small fraction all received messages.)
- A similar process occurs for the Fire Timer thread if the Update thread handles a message with a significantly different notion of time ($t \pm \epsilon$), in which case the Update thread will reset the firing interval to I_{min}, clear c and the heard_ids_buffer, and preempt the Fire Timer's sleep.

5.4.5 Send thread

The Send thread's job is simply to handle the process of encoding the message, encrypting it, and then sending it to the socket for broadcasting. The separation of Fire Timer Thread and Send Thread is important, since the process of sending a message can have unpredictable delays, which we don't want to affect our Fire Timer; in order to minimize any of these delays that could be caused by thread scheduling, this thread is assigned the highest priority among the five. This also helps keep our propagation constant $\hat{c}$ accurate.

```
42: procedure UPDATE THREAD(e, φ_offset, c, heard_ids_buffer, time_lock, I)        ▷ Priority = 4
43:     initialize msg queue
44:     heard_ids_buffer ← {}
45:     while True do
46:         Block until IPC message
47:         (id_msg, e'_msg, φ'_msg, msg_toa) ← received_message
48:         mutex_lock(time_lock)
49:         now ← ztimer_now()
50:         φ ← now − φ_offset
51:         t ← e ∗ Φ + φ
52:         φ_processing ← now − msg_toa
53:         t_msg ← e_msg ∗ Φ + φ_msg + φ_processing + ĉ
54:         e_msg ← ⌊t_msg/Φ⌋
55:         φ_msg ← t_msg  mod Φ
56:         if t_msg > t then
57:             e ← e_msg
58:             φ_offset ← now − φ_msg
59:             mutex_unlock(time_lock)
60:             thread_wakeup(epoch_timer_thread)
61:             if t_msg > t + ϵ then
62:                 c ← 0
63:                 I ← I_min
64:                 heard_ids_buffer ← {}
65:                 thread_wakeup(fire_timer_thread)
66:             else if c < k ∧ id_msg ∉ heard_ids_buffer then
67:                 heard_ids_buffer ← heard_ids_buffer ∪ id_msg
68:                 c ← c + 1
69:             end if
70:         else
71:             mutex_unlock(time_lock)
72:             if t_msg > t + ϵ then
73:                 c ← 0
74:                 I ← I_min
75:                 heard_ids_buffer ← {}
76:                 thread_wakeup(fire_timer_thread)
77:             else if c < k ∧ id_msg ∉ heard_ids_buffer then
78:                 heard_ids_buffer ← heard_ids_buffer ∪ id_msg
79:                 c ← c + 1
80:             end if
81:         end if
82:     end while
83: end procedure
```

```
84: procedure SEND THREAD(e, φ_offset, c)                                          ▷ Priority = 1
85:     Initialize TX socket
86:     Initialize msg queue
87:     while True do
88:         Block until IPC message
89:         msg ← (id, e, ztimer_now() − φ_offset)
90:         msg_encrypted ← encrypt(msg)
91:         if c < k then
92:             socket_send(msg_encrypted)
93:         end if
94:     end while
95: end procedure
```

5.5 Message Encryption

Messages are encrypted with the ChaCha20Poly1305 [14] authenticated encryption with additional data (AEAD) algorithm, using a built-in RIOT-OS library. ChaCha20-Poly1305 incurs very little processing overhead and only 24 additional bytes per message, while making the SyncWave far more robust to tampering, a quality that purely "pulse"-based PCO algorithms lack, and is one of the obstacles that has hindered their adoption in industry.

While in this implementation we use a built-in library for message encryption, in the Further Work section we discuss the *significant* potential for SyncWave to be used in as a fundamental building block for security in swarms. For example, leveraging the time synchronization provided by SyncWave as well as its notion of epoch to align a channel-hopping scheme without performing consensus on an initial seed or using a central time reference.

5.6 SyncWave Library API and Parameterization

Finally, we describe the set of tools and options made available to users wanting to include SyncWave in their application. The interface is intentionally similar to ZTimer, with the aim of making the two timing libraries easily hot-swappable. This implementation of these methods is not described in this report for brevity, but the general concept is included.

- **initialize_SyncWave(args)**: Starts the threads in the background and allows users to optionally specify parameters for the algorithm, depending on their particular use case. These include the period length Φ, minimum and maximum firing intervals I_{min}, I_{max}, and the message suppression threshold k (0 to disable MS). The empirical variables can also be tuned according to the specific hardware, including the exponential backoff base β, propagation constant $\hat{c}$, and out of phase threshold ϵ.
- **get_time(args)**: Returns the current value of $t = e \cdot \Phi + \phi$ in μs. Can also optionally return only the epoch e or phase ϕ.
- **set_timer(args)**: Adds a given $(time, callback)$ tuple to an ordered list that is checked in the Epoch Timer thread. Callback is executed once the internal time passes the provided timestamp.
- **set_periodic_timer(args)**: Adds a given $(time, callback, \Phi_{arg})$ tuple to an ordered list that is checked in the Epoch Timer thread. Callback is executed once the internal time passes the provided timestamp, and a new tuple is generated for $t + \Phi_{arg}$.

5.7 Conclusion

In this chapter we have presented our implementation of the SyncWave algorithm on the nrf52840dk MCU using the RIOT embedded operating system. We have shown that many of the concepts and assumptions in the theoretical algorithm require major modification to work on the constrained toolkit available in an embedded operating system. In the next section we will evaluate our implementation on the physical FIT-IoTLAB testbed.

Chapter 6

Evaluation

In this report we have described a variety of serious and widespread problems with state-of-the-art decentralized time synchronization algorithms. These include slow synchronization time, excessive radio usage and broadcasts, poor convergence guarantees on multi-hop topologies, and slow responsiveness to dynamic changes in topology. In this chapter we will evaluate SyncWave's performance on a large-scale IoT testbed, and demonstrate experimentally that SyncWave improves upon the state of the art with respect to these issues.

6.0.1 Algorithm Goals

In particular, we wish to solve the following problems:

1. Slow **initial synchronization time**
2. Excessive radio usage and broadcast rate **post-network synchronization**
3. Unreliable convergence and slow synchronization time on highly **multihop** networks
4. Slow adaptation to **dynamic topologies**, including nodes joining post-synchronization, cluster merging, and network partitioning
5. Excessive radio usage, packet interference, and high synchronization time in highly **dense** networks

6.0.2 Metrics

In order to evaluate SyncWave's performance on these issues, we will assess it based on the following metrics:

1. **Time to Synchronize (TTS)**: Where synchronization is defined as the first timestamp at which the maximum pair-wise epoch + phase difference between all nodes in the network attains a value below 5 ms. On IoT-LAB M3 nodes significant measurement inaccuracies (as will be discussed later) meant that the synchronization accuracy could only be measured to within $\pm$ 16 ms, and as such the target synchronization accuracy was defined as 21 ms.
2. **Number of Broadcasts**: The number of broadcasts will be collected over the entire run, as well as within 2-second windows, to analyse the radio usage throughout the experiment.
3. **TTS on Multi-hop Topologies**: We will run SyncWave on multi-hop topologies of varying lengths, observing how it performs in an extremely multi-hop environment with low graph connectivity, as defined by the network graph diameter. These graphs approximate the worst-case scenario for algorithms such as FiGo and Random Phase (which struggles in low-connectivity graphs), so any improvement will be noteworthy.
4. **TTS For Random Initial Starting Times**: We argue that the Time to Synchronize measured on a network with random initial starting times represents the worst-case time to converge for SyncWave and is equivalent to most dynamic topology scenarios including cluster merging and "ahead" nodes joining post-synchronization. As for network partitioning, this is

trivially satisfied as mentioned previously, since the drift between two partitioned sub-graphs will be bounded solely by the accuracy of their hardware clocks. Similarly, our network is entirely unaffected by churn (the arrival and departure of nodes) for the same reason as for network partitioning.

5. **Number of Broadcasts in Dense Topologies**: In order to measure SyncWave's performance on dense topologies, we will run SyncWave on an increasing number of co-located nodes, from 2 up to 163, all within 7 hops of each other and with an average degree of 37. This represents a scenario in which any algorithm that does not have some form of message suppression and/or time-division scheme (e.g. TDMA or our Random Phase) is likely to struggle. And thus low synchronization times and low numbers of broadcasts in such a topology will demonstrate success.

The synchronization accuracy in ms will also be measured; however, due to limitations of the testbed, alongside the implementation taking place at relatively high level in the software stack, this will be used solely to determine the TTS. Were optimizations performed using hardware-specific features, as is common in nearly all industry time synchronization algorithms, we assert that the accuracy could easily be improved by several orders of magnitude.

6.1 Experimental Setup

FIT IoT-LAB [15] was used as a testbed, owing to its active development, wide array of deployment targets, and particularly large-scale deployments, numbering in the hundreds of nodes per site. Normally we would use the Flocklab 2 testbed for testing time synchronization algorithms; unfortunately, it was taken down for maintenance in June of 2023 [16] and thus IoT-LAB acted as its stand-in.

6.1.1 Measurement Inaccuracy in IoT-LAB M3 Nodes

We discovered a bug with the logging infrastructure on the most widely available node, the IoT-LAB M3 (based on a STM32 (ARM Cortex M3) micro-controller with a 802.15.4 PHY standard, 2.4 Ghz radio (AT86RF231))[15]. Unfortunately, this bug resulted in all serial outputs (which we use to track each node's epoch and phase) from m3 nodes to be assigned a timestamp dated either 8 milliseconds too early, or 8 milliseconds too late, according to an unpredictable +8, -8 oscillation. This can be easily seen some of the TTS graphs shown in this chapter, wherever any periodic behaviour is apparent in the synchronization accuracy values over time.

The bug was only discovered after several days of research, and has since been reported to the FIT IoT-LAB administrators. The tangible impact of this bug is that the large-scale deployment results (all based on M3s) are unable to synchronize beyond 16 ms (caused by constructive interference between the oscillations of any two nodes in the network), and we have thus modified our definition of "synchronization" on M3s to accommodate this particular case.

6.1.2 Synchronization Accuracy Laboratory Lower Bound

In order to set a lower bound for the synchronization accuracy possible with our current implementation, we followed up with synchronization testing in the laboratory using a Rigol1074Z Digital Oscilloscope. Four nodes were initialized at random times, and allowed to synchronize for several seconds, with their GPIO pins attached to the Digital Oscilloscope and set to fire on epoch timer increment. Once synchronized, the pair-wise synchronization accuracy was measured over 5 epochs and then averaged.

As can be seen in 6.1, a maximum observed stable synchronization accuracy of 488 μs was achieved. This is an impressively tight bound to achieve, especially considering that we are not specifically leveraging any features of the hardware in our implementation such as shorts or higher-accuracy external crystal oscillators. This synchronization accuracy proves to be on a par with or better than FiGo ($\sim$ 500 μs reported) and within the same realm as SwarmSync (127 μs) despite no specific optimizations [11, 3].

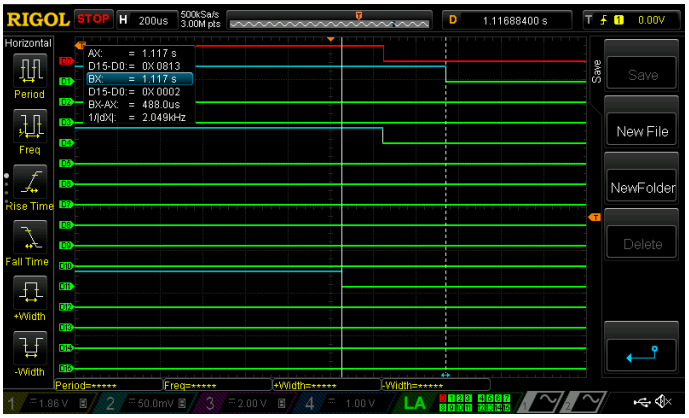

Figure 6.1: Maximum observed stable synchronization accuracy, as measured with a digital oscilloscope

6.2 Testbed Results

The following parameters in Table 6.1 were used for all experiments shown below unless stated otherwise. The values were chosen to produce satisfactory results across a range of scenarios, however they are by no means optimal. The parameters are also closely tied to the chosen implementation and available hardware, and as such these should only serve as a starting point for future implementations.

Symbol	Parameter Name	Value	Unit
Φ	Period	1	s
I_{min}	Minimum Firing Interval	75000	μs
I_{max}	Maximum Firing Interval	300	s
$\hat{c}$	Propagation Constant	9000	μs
k	Message Suppression Threshold	2	$\mathbb{N}$
β	Exponential Backoff Base	2	$\mathbb{N}$

Table 6.1: Parameters used for all topologies in this evaluation

6.2.1 Experimental results in a Large-Scale High-Density Network

We will start by presenting SyncWave's performance on what we believe is one of the largest testbed evaluations of a decentralized time synchronization algorithm in the literature. The topology, as can be seen in Figure 6.2, consists of 161 m3 nodes with a graph diameter of 7 hops, a graph density of 0.23, and an average degree per node of 37.6.

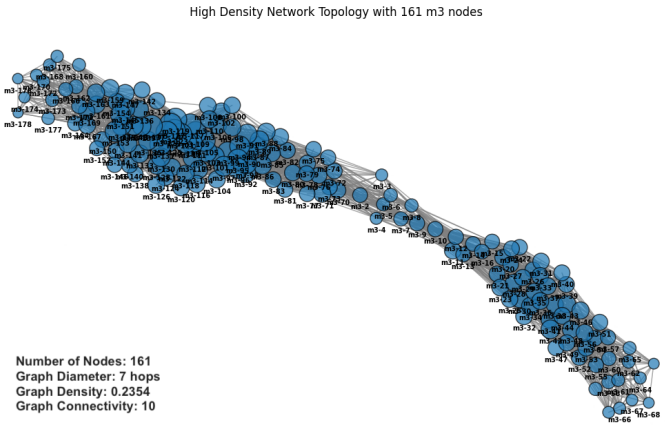

Figure 6.2: High density network topology with 161 m3 nodes.

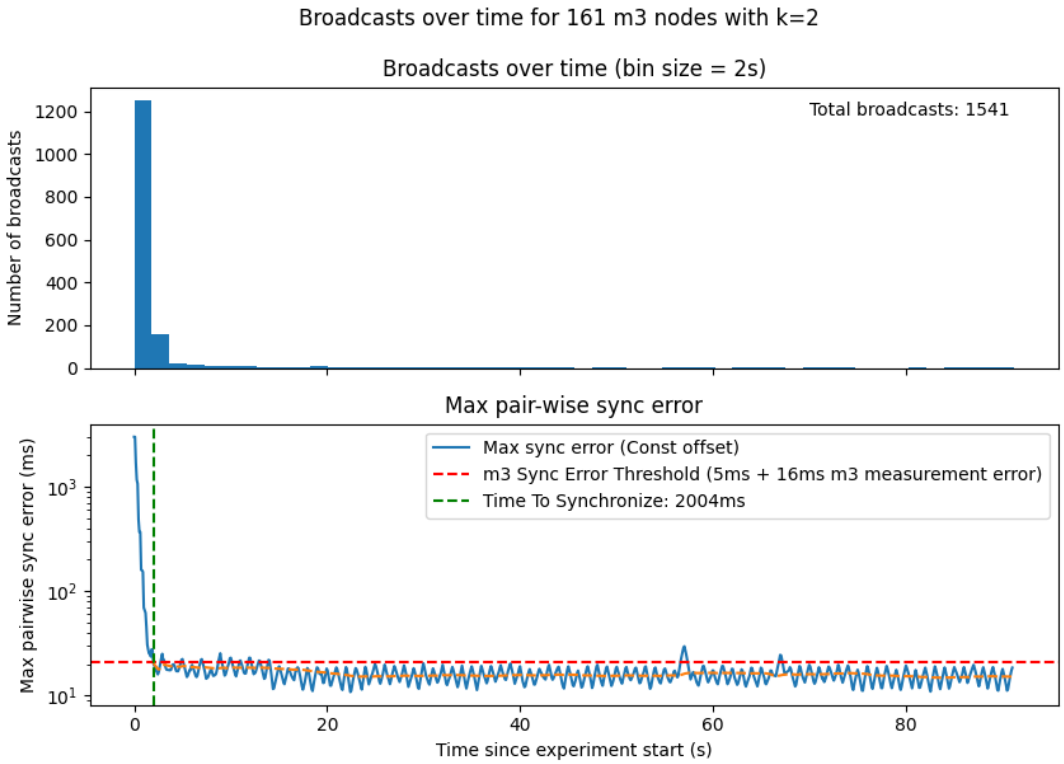

Figure 6.3: Time to Synchronize and Number of Broadcasts in a Dense Topology. (Note: variation in the reported synchronization accuracy after the TTS is an artefact of the testbed measurement error)

To give an example of the behaviour of SyncWave (before showing averaged results in the next section), in Figure 6.3 above we plot the synchronization accuracy and the number of broadcasts against the experiment run time. We can see clear evidence of several desirable properties in this experiment:

- The time to synchronization, 2004 ms for 161 nodes over 7 hops, is the lowest observed in the literature to date for a network of this size. For reference, the current gold standard decentralized time synchronization algorithms that are designed to minimize the convergence rate, CTMS and CCTS, are only able to synchronize a 20-node network within ∼4 seconds. And the next best algorithms ATS and MTS, which had dominated the field for years, only synchronize the 20 nodes in 50+ seconds [17, 18, 19]. As a reminder, this is simply a demonstration experiment run time, and was not selected specifically to minimize its TTS, as will be shown in the scaling section.

- The number of broadcasts, at 1541 for 161 nodes, is also cutting edge; for reference, CCTS issues 700 broadcasts for only 40 nodes when evaluated *in simulation*[17].

- The broadcast rate starts high when the network is unsynchronized on initialization, before exponentially decreasing as the network synchronizes until I_{min} is reached and only a trickle of messages are exchanged, answering the second problem from our Algorithm Goals. This is further evidenced by the logarithmic shape seen in the left hand graph in Figure 6.4, which plots the Cumulative Number of Broadcasts over time – resulting from our exponential backoff working as intended.

- Nodes only adapt to neighbours that are further ahead in time, which can be seen in the right-hand diagram in Figure 6.4 that plots each node's local notion of time (defined by its epoch + phase) over time. The nodes start with random initial times, and within 0.8 s already have a pair-wise synchronization accuracy of ∼60 ms (max-min).

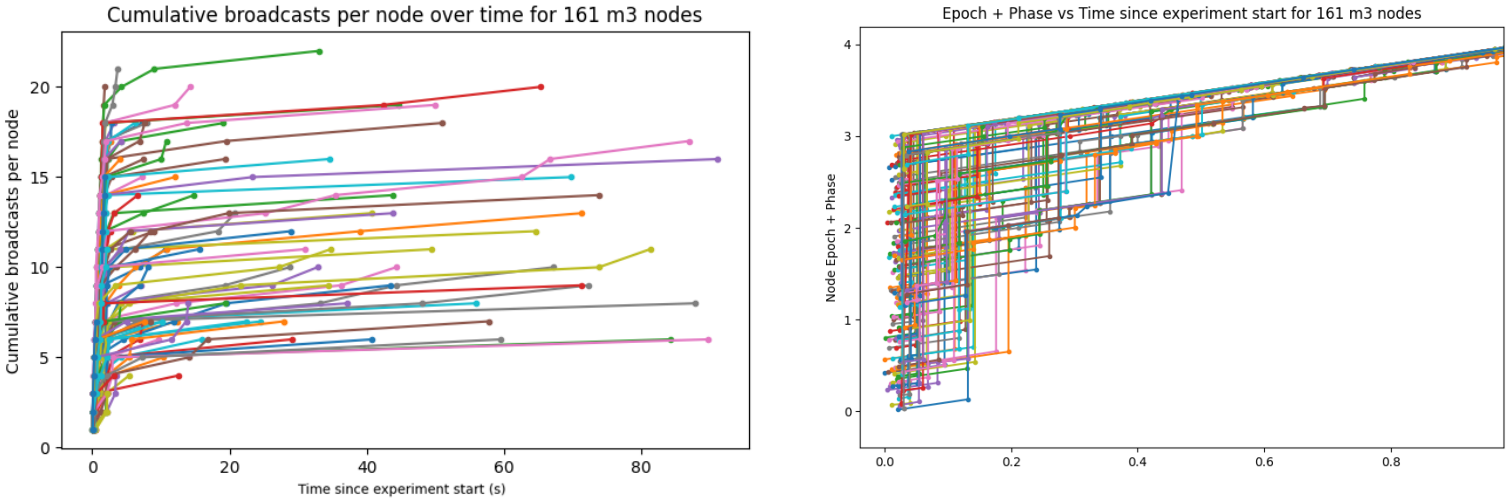

Figure 6.4: Cumulative Broadcasts per node over time, and Local Epoch + Phase per node over time

6.2.2 Scaling in High-Density Networks

Next, to lend credibility to the 161-node example, we tested SyncWave on dense topologies of increasing number of nodes, from $n = 2$ to $n = 128$, with 10 trials apiece. We have plotted graphs for each of the key metrics, including the TTS, Number of Broadcasts, and Synchronization Accuracy.

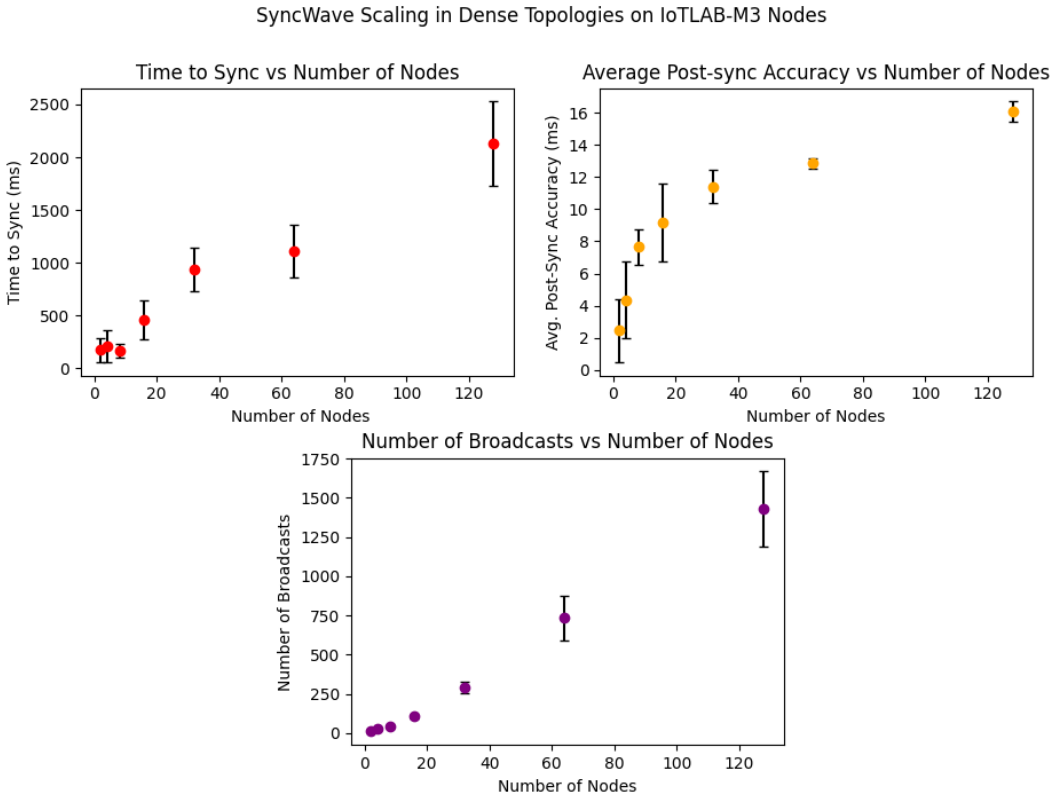

Figure 6.5: SyncWave Scaling in Dense Topologies on IoTLAB-M3 Nodes

Some of the key takeaways here are that both the time to synchronize and the number of broadcasts appear to scale linearly with the the number nodes. The TTS is also fairly stable, with

a standard deviation of only a quarter of the mean in the 128-node case. And in all cases, the TTS and the Number of Broadcasts are extremely low, relative to the state of the art.

6.2.3 Experimental results on a Linear Topology

The second topology tested was a linear (path) topology, with the aim of evaluating SyncWave in a highly multi-hop and low-connectivity network. Conforming exactly to a path topology was difficult to enforce without compromising link integrity, and therefore the network diameter is not exactly equal to the number of nodes, which has been accounted for by plotting against the diameter instead. Figure 6.6 visualizes the topology and overlays it with the nodes' accurate physical location within the Grenoble FIT IoT-LAB testbed.

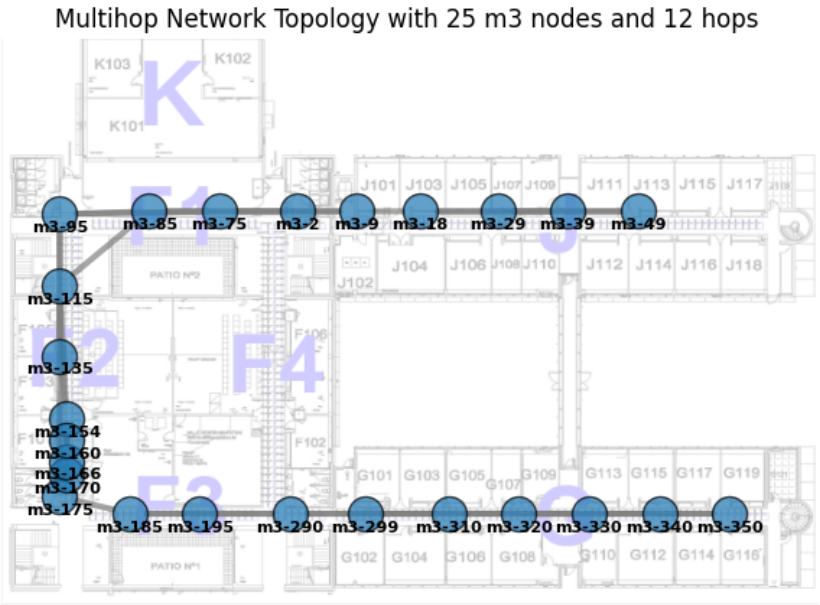

Figure 6.6: Multi-Hop topology with 25 nodes and 12 hops with testbed physical positioning information

An example diagram for the linear topology's convergence is shown in Figure 6.7. It demonstrates almost exactly the same positive behaviours as the Dense network, with the main difference being slightly less aggressive backoff (visible in the number of broadcasts from ∼3 s to ∼30 s). This was intentional, having set the the backoff base $\beta = 1.5$ instead of the default 2, in order to decrease the likelihood of part of the topology prematurely backing off.

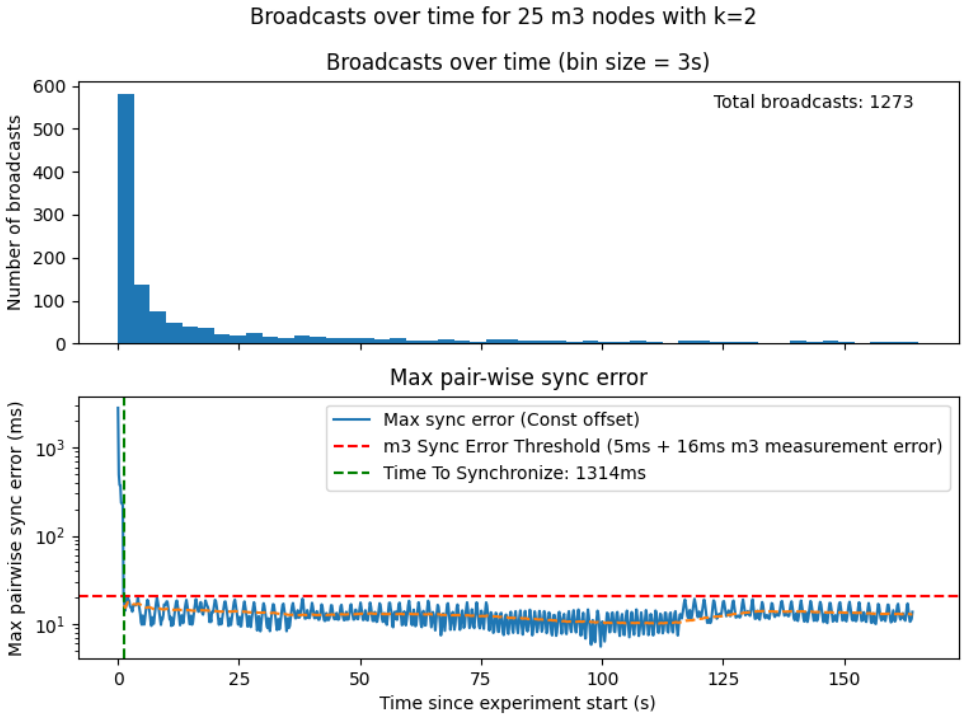

Figure 6.7: Time to Synchronize and Number of Broadcasts for an extreme multihop topology with 25 nodes and 12 hops.

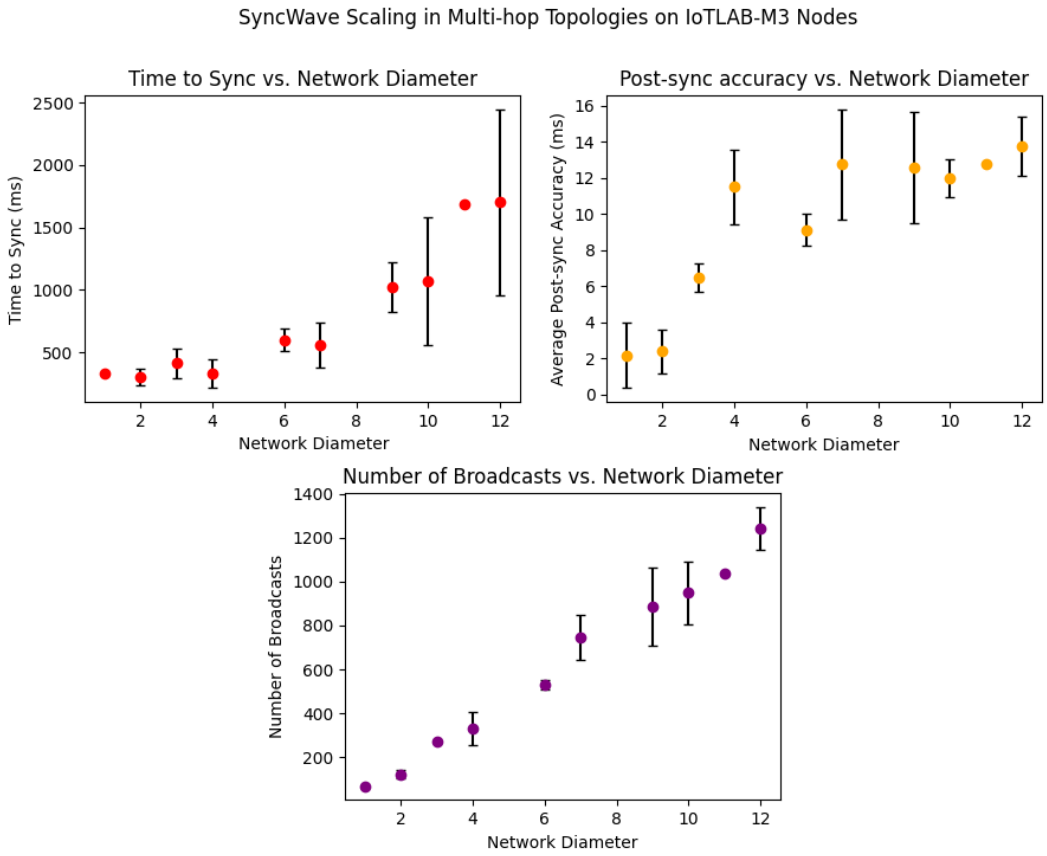

Figure 6.8: SyncWave Scaling in Multi-hop Topologies on IoTLAB-M3 Nodes

6.2.4 Scaling in extremely Multi-Hop Networks

To contextualize our example multi-hop topology convergence diagram, Figure 6.8 shows a scalability analysis on linear topologies ranging in number of nodes from $n = 2$ to $n = 25$, with network diameters ranging from 1 to 12, and 10 trials for each diameter. The evaluation metrics are plotted against the network diameter, and yield some of the following takeaways:

- TTS is likely exponentially related to the network diameter, although this is hard to confirm due to the large standard deviation in the results. This is an expected result, since effectively all time synchronization algorithms (not just decentralized) tend to accumulate error in each hop, which will slow convergence as diameters become large [20]. This can directly be seen in the Post-sync Accuracy vs Network Diameter graph, which shows the accuracy approaching the 12-14 ms $\pm$16 ms range with large network diameters. As with the Dense Topologies, the number of broadcasts appears to scale predictably and linearly with the number of nodes; this also means that not many firing interval resets were required, i.e. that the network was capable of synchronizing in one go from initialization without prematurely backing off.

6.3 Summary

In this section we have evaluated the SyncWave algorithm on a variety of metrics chosen to test the current drawbacks associated with state-of-the-art decentralized time synchronization algorithms. We have shown that our algorithm converges faster than the state of the art on a variety of topologies including large-scale, dense, and highly multi-hop topologies. We provide evidence that the exponential backoff primitive can be an incredibly effective tool for decoupling convergence time from the number of broadcasts, and we show that our algorithm has strong convergence guarantees, with all 140 trials on high-density and high-hop topologies converging within 3 seconds.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this report we have

- Investigated the worst-case scenarios and failures of current state-of-the-art decentralized time synchronization protocols for drone swarms, and identified key problems that must be solved;
- Introduced the SyncWave algorithm, comprising key concepts such as employing a total ordering on time within the Pulse-Coupled Oscillator framework, using exponential backoff to modulate information rate based on network synchronization progress, and a unique-sender Message Suppression optimization for densely connected networks;
- Implemented SyncWave on the nrf52840 SoC using the RIOT-OS embedded operating system, detailing the complicated process of adapting a theoretical time synchronization algorithm into one with relaxed assumptions;
- Evaluated the SyncWave algorithm on a large-scale physical testbed with metrics chosen to test the drawbacks associated with state-of-the-art decentralized time synchronization algorithms, finding that SyncWave massively improves upon the SOTA with respect to convergence time and the number of messages used post-synchronization.

7.2 Future Work

- **Secure Swarms**: We believe that SyncWave has significant potential for use cases in swarm security including encryption, authentication, and resiliency. The rapidly-converging, decentralized nature of SyncWave makes it an excellent building block for a channel hopping scheme, whereby it could replace the initial centralized time synchronization sequence, and then after synchronization, leverage its notion of "epoch" to define when to swap channels without having to perform network-wide consensus or use a central reference.
- **Deep Sleep**: While the current implementation minimizes power usage by entering thread sleep wherever possible, the radio is still kept listening for incoming messages which prevents the microcontroller from entering a deep sleep mode. While this isn't a concern for many robotic swarms, for whom the limiting factor on power consumption will be their motors, this optimization could immediately make SyncWave more attractive within the related field of Wireless Sensor Networks. Deep sleep could be implemented such that once the nodes are synchronized to a sufficient accuracy, they could achieve consensus on a length of time to enter deep sleep before re-synchronizing – using a fraction of the normal operating power required to keep the radio always listening.
- **Optimization for Accuracy**: An implementation of SyncWave at a lower level in the network stack and utilizing more hardware-specific features could massively improve the synchronization accuracy and time to converge. A more sophisticated propagation time estimation scheme could also be helpful for achieving single-digit μs synchronization accuracy.

Bibliography

[1] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 91–100, 1993.

[2] nrf52840-dongle image (digikey). URL https://www.digikey.co.uk/en/products/detail/nordic-semiconductor-asa/NRF52840-DONGLE/9491124.

[3] Meetha V Shenoy and KR Anupama. Swarm-sync: A distributed global time synchronization framework for swarm robotic systems. *Pervasive and Mobile Computing*, 44:1–30, 2018.

[4] Jorge F Schmidt, Udo Schilcher, Arke Vogell, and Christian Bettstetter. Using randomization in self-organized synchronization for wireless networks. *ACM Transactions on Autonomous and Adaptive Systems*, 18(3):1–20, 2023.

[5] Sinan Kurt and Bulent Tavli. Path-loss modeling for wireless sensor networks: A review of models and comparative evaluations. *IEEE Antennas and Propagation Magazine*, 59(1):18–37, 2017.

[6] Arpita Ghosh, Stephen Boyd, and Amin Saberi. Minimizing effective resistance of a graph. *SIAM review*, 50(1):37–66, 2008.

[7] Jonathan L Gross, Jay Yellen, and Mark Anderson. *Graph theory and its applications*. Chapman and Hall/CRC, 2018.

[8] Ahmed Elmokashfi, Amund Kvalbein, and Constantine Dovrolis. On the scalability of bgp: The role of topology growth. *IEEE Journal on Selected Areas in Communications*, 28(8): 1250–1261, 2010.

[9] Philip Levis, Thomas Clausen, Jonathan Hui, Omprakash Gnawali, and JeongGil Ko. The trickle algorithm. Technical report, 2011.

[10] Andreas Biri, Marco Zimmerling, and Lothar Thiele. Demos: Robust orchestration for autonomous networking. In *The International Conference on Embedded Wireless Systems and Networks (EWSN'23)*. Association for Computing Machinery, 2023.

[11] Matt Webster, Michael Breza, Clare Dixon, Michael Fisher, and Julie McCann. Formal verification of synchronisation, gossip and environmental effects for wireless sensor networks. *Electronic Communications of the EASST*, 76, 2019.

[12] Michael Breza and Julie McCann. Polite broadcast gossip for iot configuration management. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6. IEEE, 2017.

[13] Wojciech Giernacki, Mateusz Skwierczyński, Wojciech Witwicki, Paweł Wroński, and Piotr Kozierski. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 37–42. IEEE, 2017.

[14] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. URL https://www.rfc-editor.org/info/rfc8439.

[15] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frédéric Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. Fit iot-lab: A large scale open experimental iot testbed. Milan, Italy, December 2015. URL https://hal.inria.fr/hal-01213938.

[16] Roman Trüb, Reto Da Forno, Lukas Sigrist, Lorin Mühlebach, Andreas Biri, Jan Beutel, and Lothar Thiele. FlockLab 2: Multi-Modal Testing and Validation for Wireless IoT. In *3rd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench 2020)*. OpenReview.net, 2020. doi: 10.3929/ethz-b-000442038. URL https://doi.org/10.3929/ethz-b-000442038.

[17] Zhaowei Wang, Peng Zeng, Mingtuo Zhou, and Dong Li. Cluster-based maximum consensus time synchronization in iwsns. In *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring)*, pages 1–5. IEEE, 2016.

[18] Jianping He, Peng Cheng, Ling Shi, Jiming Chen, and Youxian Sun. Time synchronization in wsns: A maximum-value-based consensus approach. *IEEE Transactions on Automatic Control*, 59(3):660–675, 2013.

[19] Luca Schenato and Federico Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.

[20] Fanrong Shi, Xianguo Tuo, Simon X Yang, Jing Lu, and Huailiang Li. Rapid-flooding time synchronization for large-scale wireless sensor networks. *IEEE Transactions on Industrial Informatics*, 16(3):1581–1590, 2019.