IMPERIAL

SyncWave

Rapid and Adaptive Decentralized Time Synchronization for Swarm Robotic Systems

Luca Seimon Mehl

Supervisors: Prof. Julie McCann
Dr. Michael Breza

Second marker: Prof. William Knottenbelt

Introduction

What is a Time Synchronization Algorithm?

- Given a set of machines, each with internal clock with offset and skew
- That communicate (wirelessly) in some network topology
- Goal: agreement on single time value
  - *All non-faulty processes must agree on the same (single) value*

Introduction
Uses of Time Synchronization

- Provides nodes with a global clock for:
- Coordinating future events, e.g. takeoff for drone swarm
- Correlate sensor data between nodes
- Speeding up consensus

Introduction
The Problem: Scenario

Firefighters are deployed for search-and-rescue in a burning building

To assist them, a swarm of drones is immediately deployed

The inside of the building does not have GPS, and communication between drones can be fleeting as they navigate inside

When they do communicate, they want to rapidly perform consensus on search area allocation

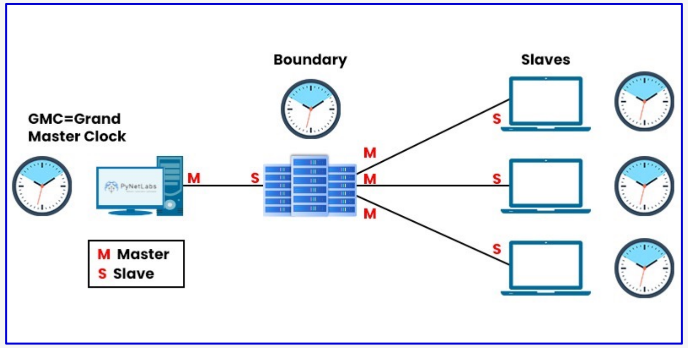
If any drones are lost, this shouldn't jeopardize the whole swarm's mission

Crazyflie 2.0 Micro Drones navigating indoors

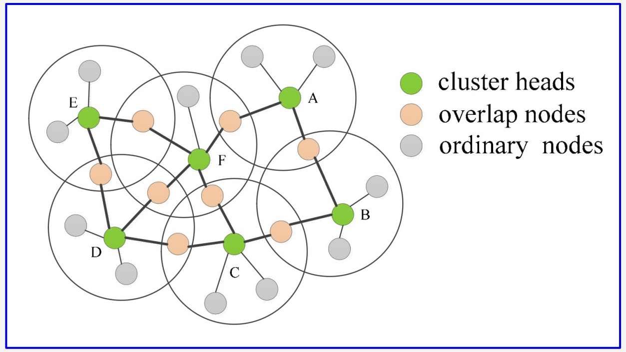
Introduction

Existing Time Synchronization Algorithms

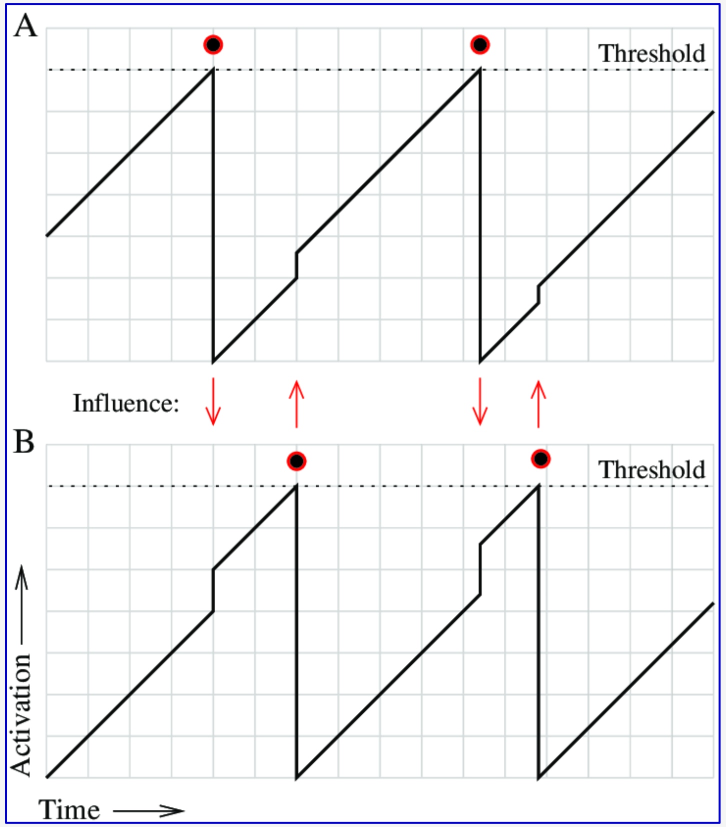
1. Centralized, single-hop (e.g. PTP)
2. Wireless sensor network algorithms (e.g. MTS, CMTS)
3. Pulse-coupled oscillators (e.g. FiGo, Random Phase)
4. Attempts at TS for drone swarms (e.g. Swarm-Sync)



PTP algorithm overview



CMTS overview



PCO example

Introduction
Problems with Existing Time Sync Algorithms

1. **Slow initial synchronization time**
2. **Excessive radio usage post-synchronization**
3. **Multi-hop topologies : unreliable convergence and slow synchronization time**
4. **Dynamic topologies: slow adaptation to arbitrary node failures, cluster merging, network partitioning, and node churn**
5. **Dense topologies : excessive radio usage and packet interference**

Simulation

Simulation
Aims

- Aim: Environment for developing our protocol (2 s turnaround)

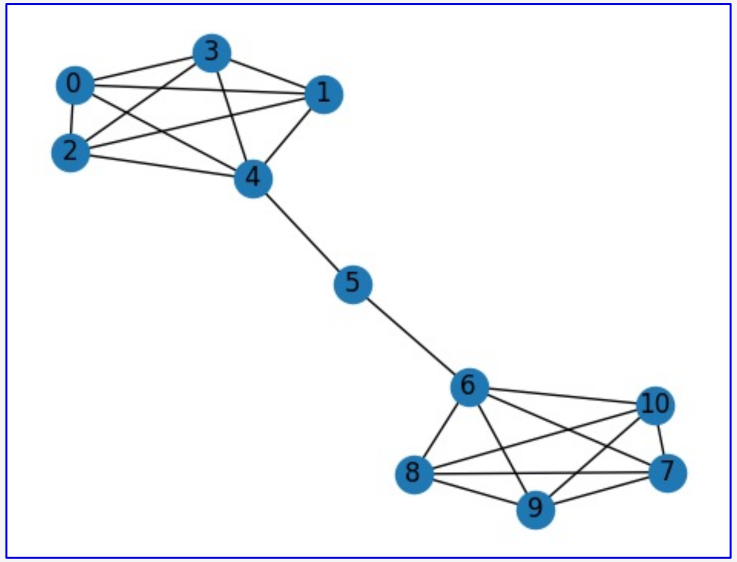
- Assumptions: perfect links, no packet collisions, no processing / propagation time

Simulation

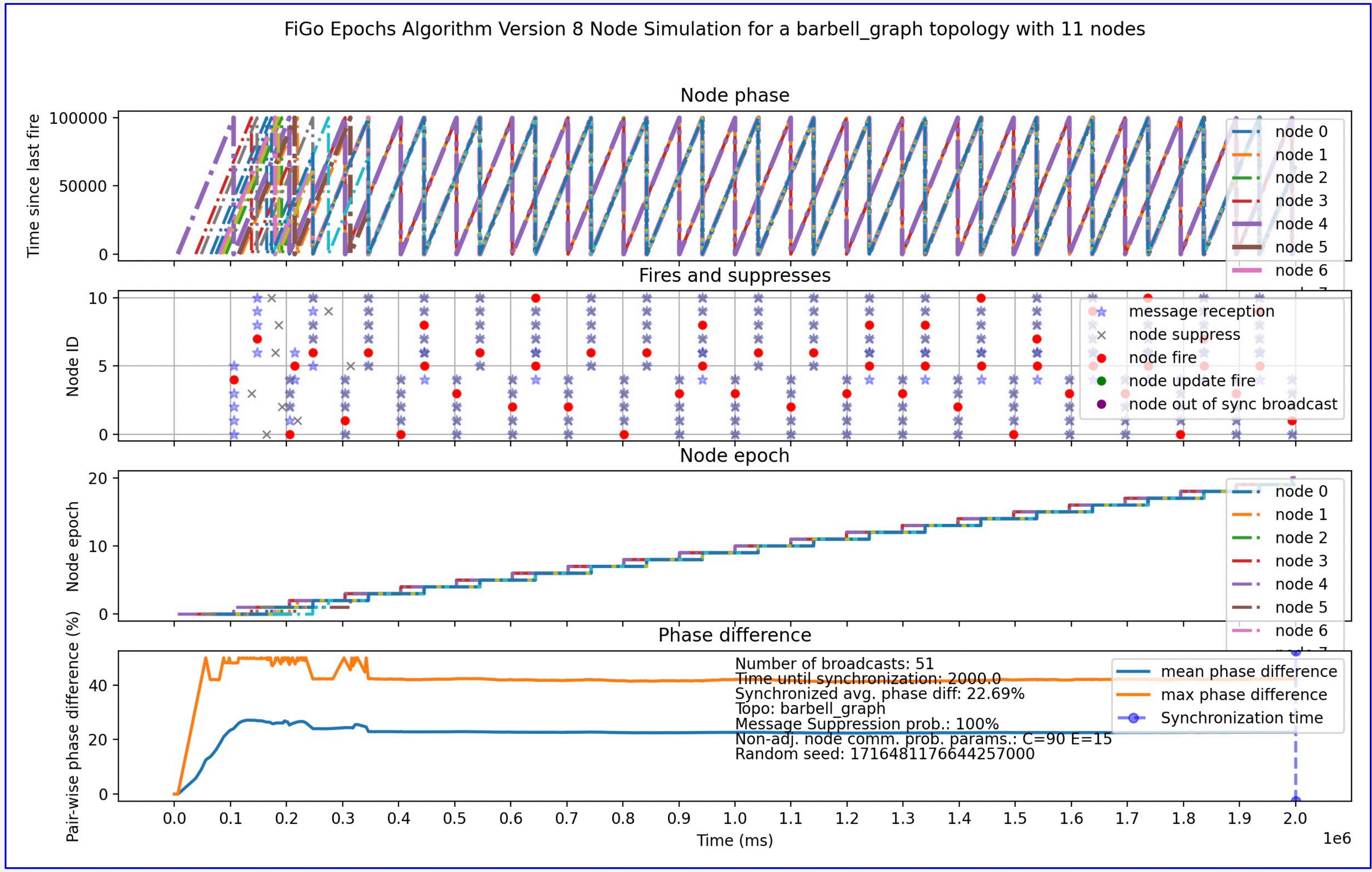
Example: FiGo

Here is an example of what we would get out of our simulation

Barbell Topology (Challenging)



FiGo (normal, with message suppression)



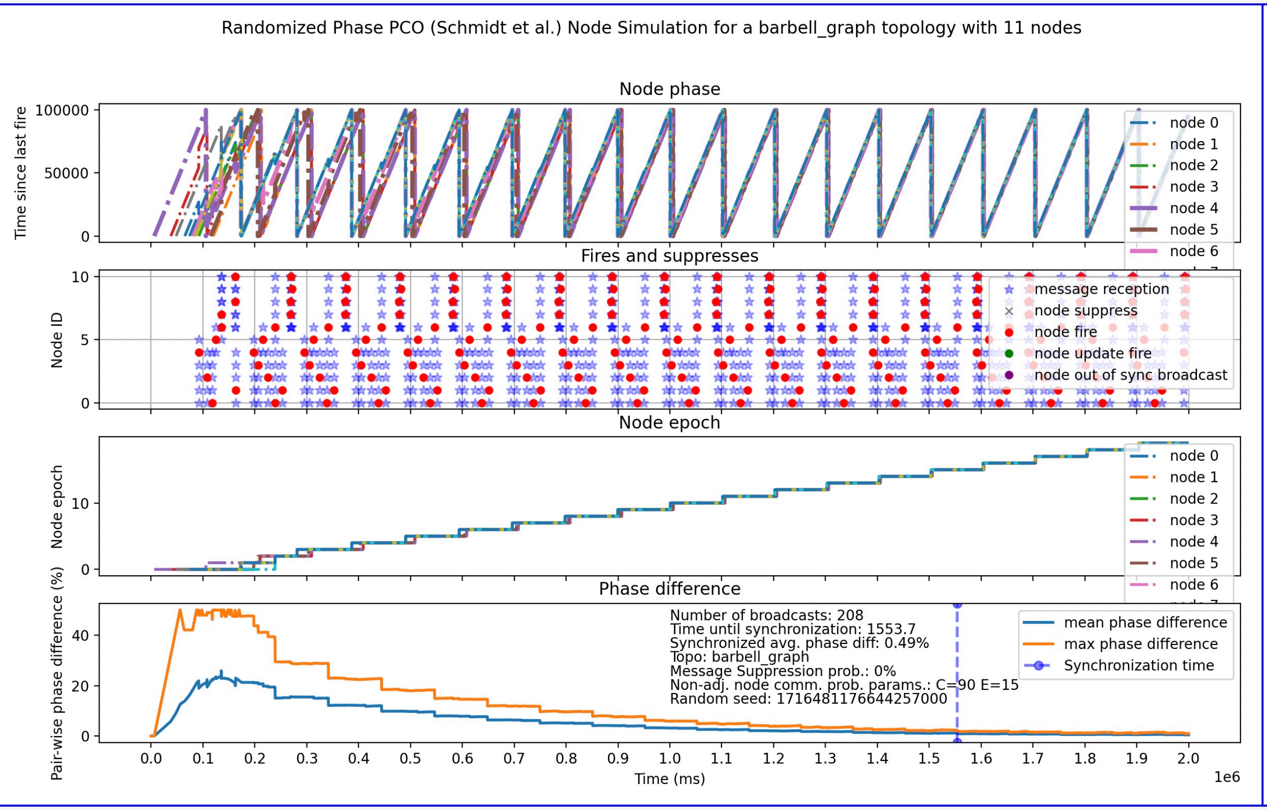
- Does not converge within 20 periods (1s each)

\+ Low number of fires

Simulation

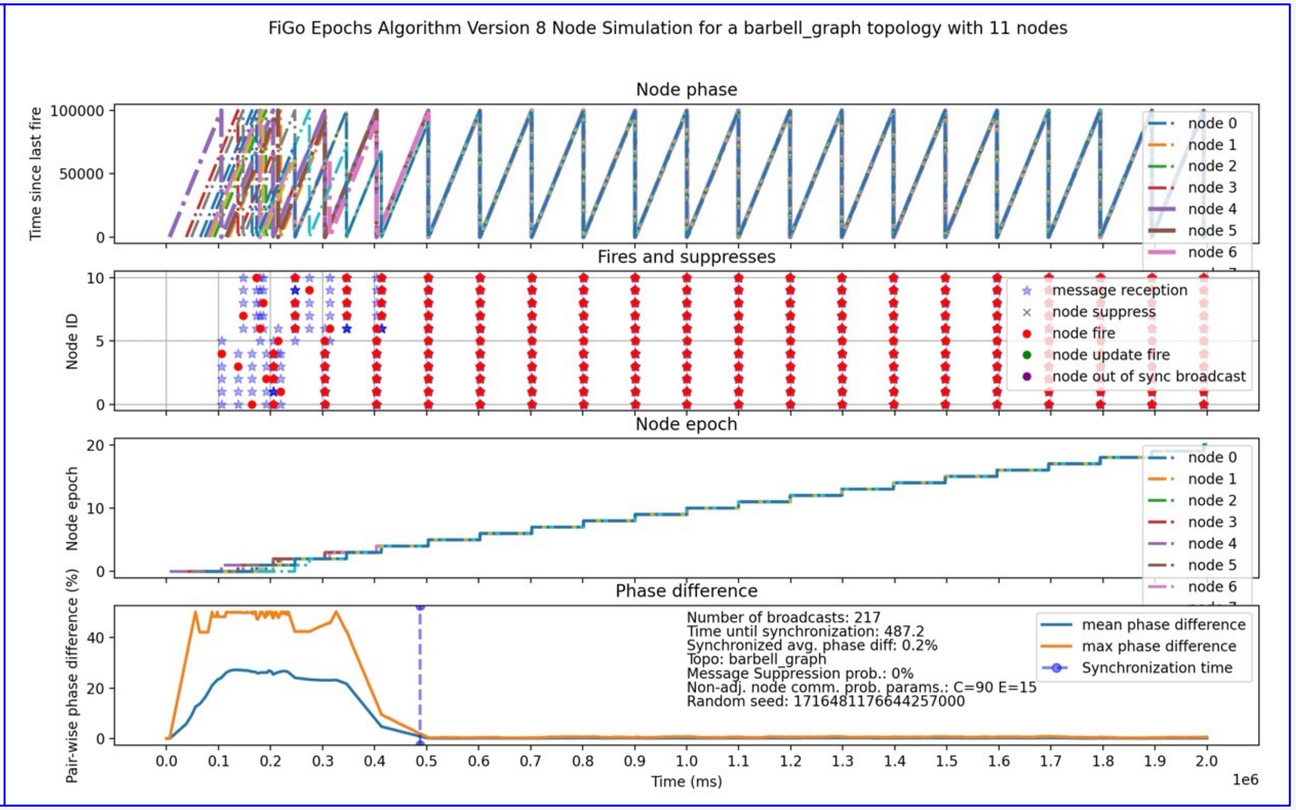
Example: Randomized Phase and FiGo with no message suppression

Randomized Phase algorithm



+ staggered fire times
- High number of broadcasts

"Bruteforce" FiGo (no message suppression)



+ Faster convergence
- High number of broadcasts

Simulation
Conclusion

- FiGo:
  - \- Poor convergence
  - \+ low broadcast

- Randomized Phase:
  - \- High broadcast
  - \+ staggered fire times

- FiGo (no msg supression):
  - \- High broadcast
  - \+ Fast convergence

- Next, incorporate and extend these features in our own algorithm: SyncWave

SyncWave Algorithm

SyncWave Algorithm

Phase and Epochs

- Let's build up SyncWave piece by piece
- We need some way of keeping track of time:
- Theoretical: run algorithm in busy-loop
- Incrementing a "Phase" ϕ
- until period Φ, when reset
- Epoch e is number of times it has been reset

Algorithm 1 SyncWave algorithm

```
ϕ ← 0
e ← 0

while True do

    if ϕ > Φ then
        ϕ ← 0
        e ← e + 1
    end if

    ϕ ← ϕ + 1

end while
```

SyncWave Algorithm
Randomized Firing Phase

- Want to:
  - Send current time to neighbors
  - Easily scale sending frequency
  - Not send at same time as neighbors

- Solution:
- broadcast whenever a separate "fire" timer ψ reaches a firing time ψ_{fire}
- To avoid packet collisions, firing time ψ_{fire} sampled randomly from range $[0, I]$
- Where Firing Interval I can be scaled

Algorithm 1 SyncWave algorithm

$\phi \leftarrow 0$
$e \leftarrow 0$
$I \leftarrow I_{min}$
$\psi \leftarrow 0$
$\psi_{\text{fire}} \leftarrow randint(0, I)$

while $True$ **do**

        end if
    end if
    if $\phi > \Phi$ **then**
        $\phi \leftarrow 0$
        $e \leftarrow e + 1$
    end if

    if $\psi > \psi_{\text{fire}}$ **then**
            $\text{Tx}(id, e, \phi)$
        $\psi_{\text{fire}} \leftarrow randint(0, I)$
        $\psi \leftarrow 0$
    end if
    $\phi \leftarrow \phi + 1$
    $\psi \leftarrow \psi + 1$
end while

SyncWave Algorithm

Maximum Time Synchronization

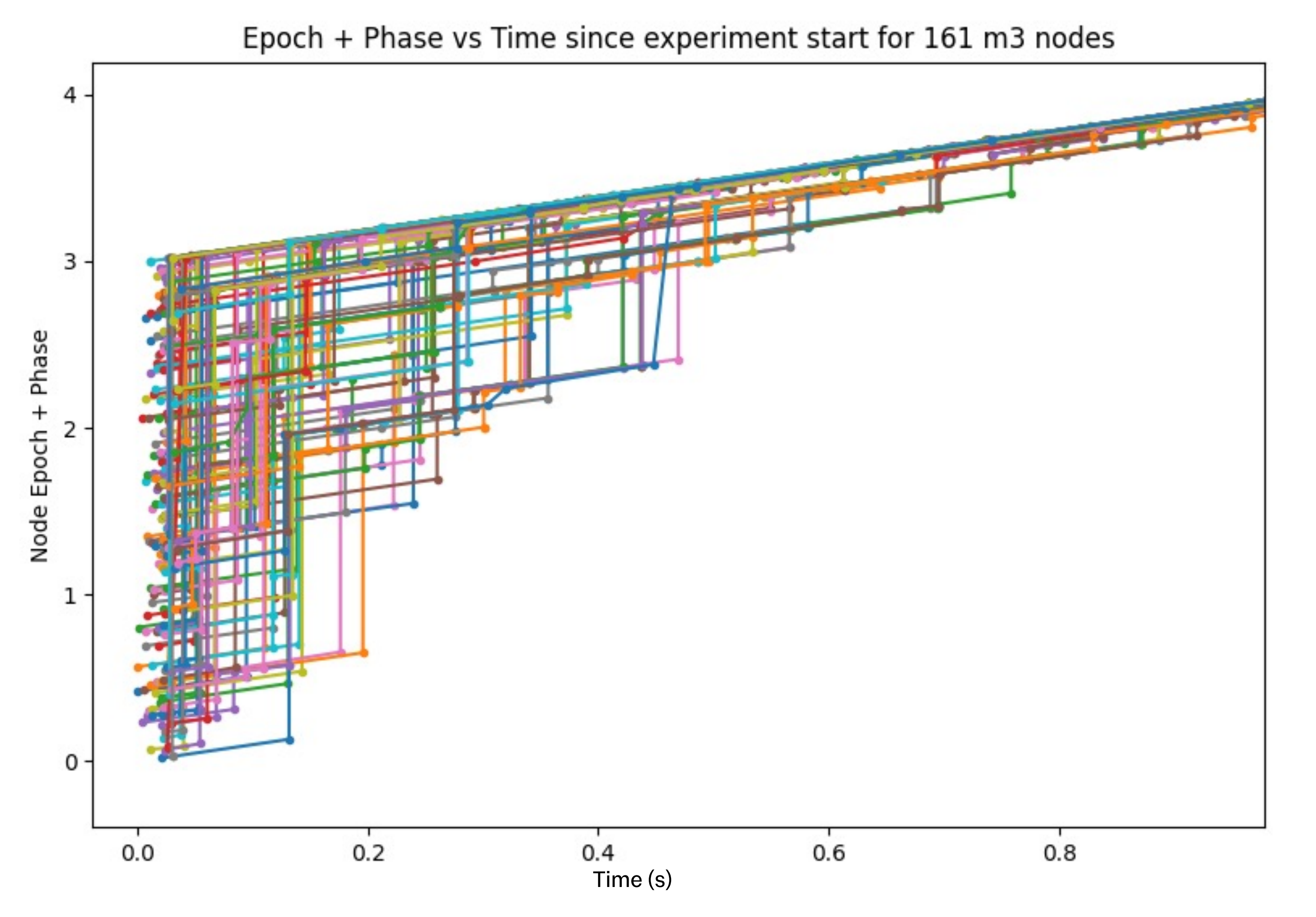
- On message received from neighbour, want to:
  - Use this information to refine our own time estimate
  - Account for time in the air

- Solution:
- Use Max Time Synchronization
- Pick whichever of the msg time and our time is greater
- Account for time in the air $\hat{c}$

Algorithm 1 SyncWave algorithm

$\phi \leftarrow 0$
$e \leftarrow 0$
$I \leftarrow I_{min}$
$\psi \leftarrow 0$
$\psi_{\text{fire}} \leftarrow randint(0, I)$
while *True* **do**
  if $receive\_message(id_{msg}, e'_{msg}, \phi'_{msg})$ **then**
    $t_{msg} \leftarrow e'_{msg} \cdot \Phi + \phi'_{msg} + \hat{c}$
    $t \leftarrow e \cdot \Phi + \phi$
    $e_{msg} \leftarrow \lfloor t_{msg} \div \Phi \rfloor$
    $\phi_{msg} \leftarrow t_{msg} \mod \Phi$
    if $t_{msg} > t$ **then**
      $e \leftarrow e_{msg}$
      $\phi \leftarrow \phi_{msg}$
    end if
  end if
  if $\phi > \Phi$ **then**
    $\phi \leftarrow 0$
    $e \leftarrow e + 1$
  end if
  if $\psi > \psi_{\text{fire}}$ **then**
      $\text{Tx}(id, e, \phi)$
    $\psi_{\text{fire}} \leftarrow randint(0, I)$
    $\psi \leftarrow 0$
  end if
  $\phi \leftarrow \phi + 1$
  $\psi \leftarrow \psi + 1$
end while

SyncWave Algorithm
Maximum Time Synchronization (example)



SyncWave Algorithm
Exponential Backoff

- Want to:
  - Synchronize quickly with short firing interval
  - Once synchronized, free up radio with long firing interval

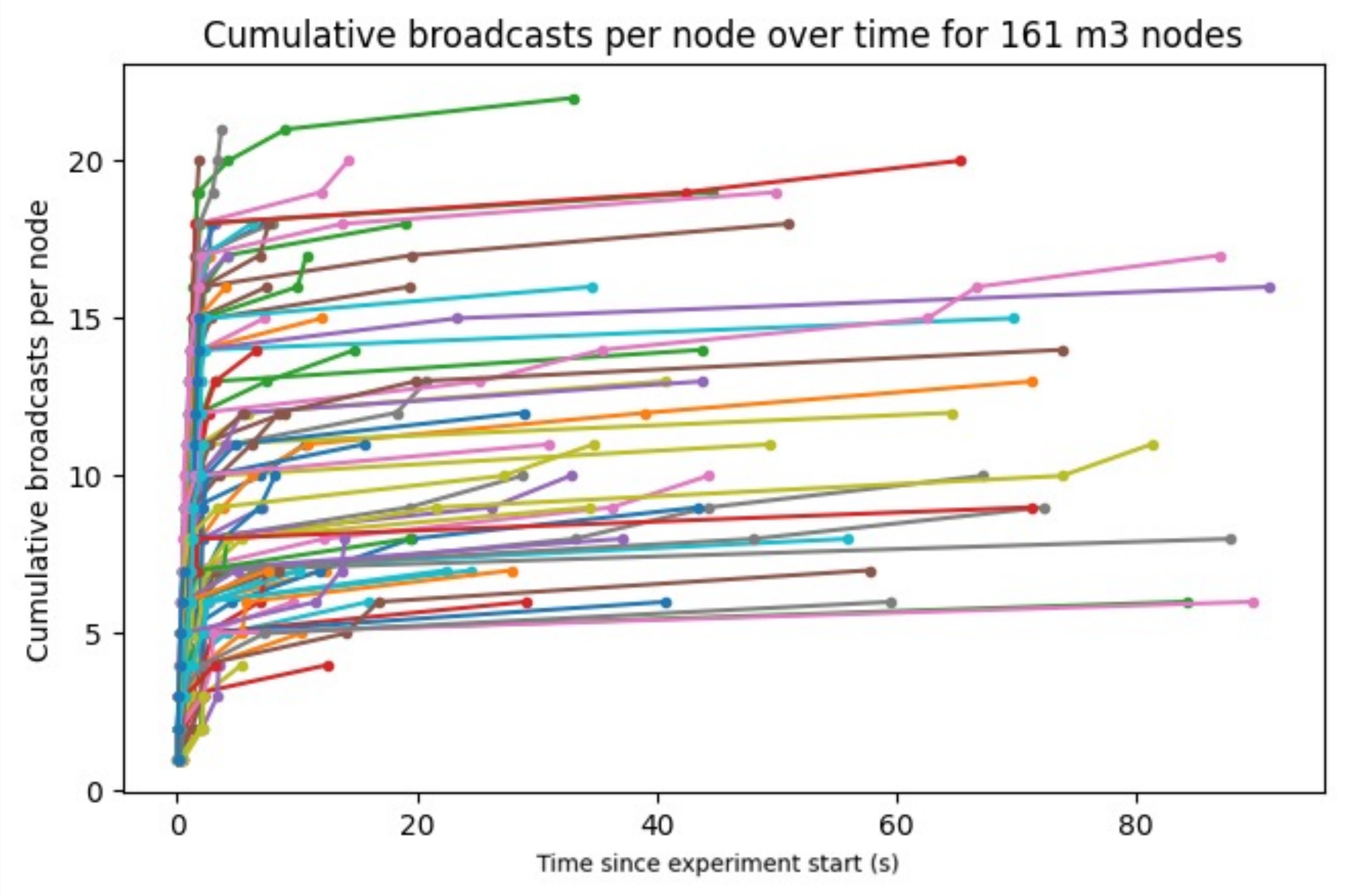
- Exponential Backoff on firing interval for each fire
- Start at I_{min}, double up to I_{max}
- Reset to I_{min} if "unsynchronized" msg heard that's off by $\pm\epsilon$

Algorithm 1 SyncWave algorithm

```
φ ← 0
e ← 0
I ← I_min
ψ ← 0
ψ_fire ← randint(0, I)
while True do
    if receive_message(id_msg, e'_msg, φ'_msg) then
        t_msg ← e'_msg · Φ + φ'_msg + ĉ
        t ← e · Φ + φ
        e_msg ← ⌊t_msg ÷ Φ⌋
        φ_msg ← t_msg mod Φ
        if t_msg > t then
            e ← e_msg
            φ ← φ_msg
            if t_msg > t + ε then
                ResetFiringInterval()
            end if
        else
            if t_msg < t + ε then
                ResetFiringInterval()
            end if
        end if
    end if
    if φ > Φ then
        φ ← 0
        e ← e + 1
    end if
    if ψ > ψ_fire then
            Tx(id, e, φ)
        ψ_fire ← randint(0, I)
        ψ ← 0
    end if
    φ ← φ + 1
    ψ ← ψ + 1
end while

function ResetFiringInterval()
    I ← I_min
    if I < ψ_fire − ψ then
        ψ_fire ← randint(0, I)
        ψ ← 0
    end if
end function
```

SyncWave Algorithm
Exponential Backoff on Firing Interval (example)



SyncWave Algorithm
Message Suppression

- Want:
  - Fewer broadcasts in dense networks (+better scaling)

- *If you receive a message, your neighbours probably did too (so be quiet)*

- Solution:
- Suppress next broadcast if k unique neighbours have broadcast a similar time, since our last fire
- And override message suppression if "unsynchronized" message heard

Algorithm 1 SyncWave algorithm

```
ϕ ← 0
e ← 0
I ← I_min
ψ ← 0
ψ_fire ← randint(0, I)
c ← 0
heard_ids ← {}
while True do
    if receive_message(id_msg, e'_msg, ϕ'_msg) then
        t_msg ← e'_msg · Φ + ϕ'_msg + ĉ
        t ← e · Φ + ϕ
        e_msg ← ⌊t_msg ÷ Φ⌋
        ϕ_msg ← t_msg mod Φ
        if t_msg > t then
            e ← e_msg
            ϕ ← ϕ_msg
            if t_msg > t + ε then
                ResetFiringInterval()
            else if c < k ∧ id_msg ∉ heard_ids then
                heard_ids ← heard_ids ∪ id_msg
                c ← c + 1
            end if
        else
            if t_msg < t + ε then
                ResetFiringInterval()
            else if c < k ∧ id_msg ∉ heard_ids then
                heard_ids ← heard_ids ∪ id_msg
                c ← c + 1
            end if
        end if
    end if
    if ϕ > Φ then
        ϕ ← 0
        e ← e + 1
    end if

    if ψ > ψ_fire then
        if c < k then
            Tx(id, e, ϕ)
        end if
        c ← 0
        heard_ids ← {}
        I ← min(β · I, I_min)
        ψ_fire ← randint(0, I)
        ψ ← 0
    end if
    ϕ ← ϕ + 1
    ψ ← ψ + 1
end while

function ResetFiringInterval()
    c ← 0
    heard_ids ← {}
    I ← I_min
    if I < ψ_fire − ψ then
        ψ_fire ← randint(0, I)
        ψ ← 0
    end if
end function
```

SyncWave Algorithm
Conclusion

To summarise, this is how SyncWave meets our original requirements:

1. **Slow initial synchronization time**
   - Exp. Backoff: Algorithm starts with firing interval I_{min}, so network rapidly converges
2. **Excessive radio usage post-synchronization**
   - Exp. Backoff: Firing interval increases up to I_{max} as network is synchronized
3. **For multi-hop topologies : unreliable convergence and slow synchronization time**
   - Max Time Sync: Guaranteed convergence, impossible to form local time maxima
   - Exp. Backoff: “Bridge” node to next hop will reset firing interval on hearing diff time
4. **For dynamic topologies: slow adaptation to arbitrary node failures, cluster merging, network partitioning, and node churn**
   - Exp. Backoff: Firing interval reset to I_{min} when new cluster detected
   - Max Time Sync: Invariant to arb. node failures, network partitioning, and churn by default
5. **For dense topologies : excessive radio usage and packet interference**
   - Msg. Sup: Number of messages capped at k per hop per I_{max}
   - Random Firing Phase: Broadcasts uniformly distributed in time

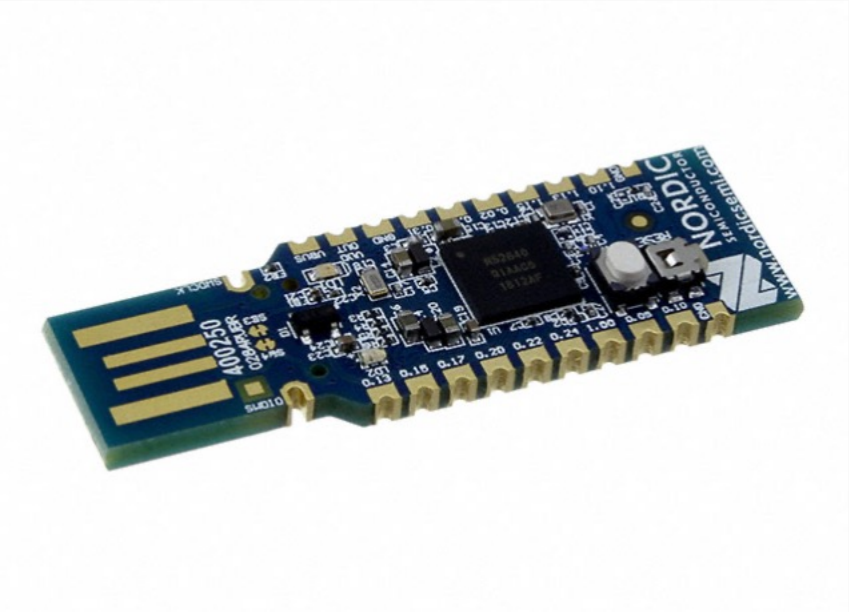
Implementation

Implementation
Hardware & Embedded OS

- Developed for nrf52840 SoC
  - ARM M4 CPU
  - BLE, Bluetooth Mesh, 2.4GHz ESB

- RIOT embedded operating system
  - Level of abstraction and portability
  - Built-in timing tools

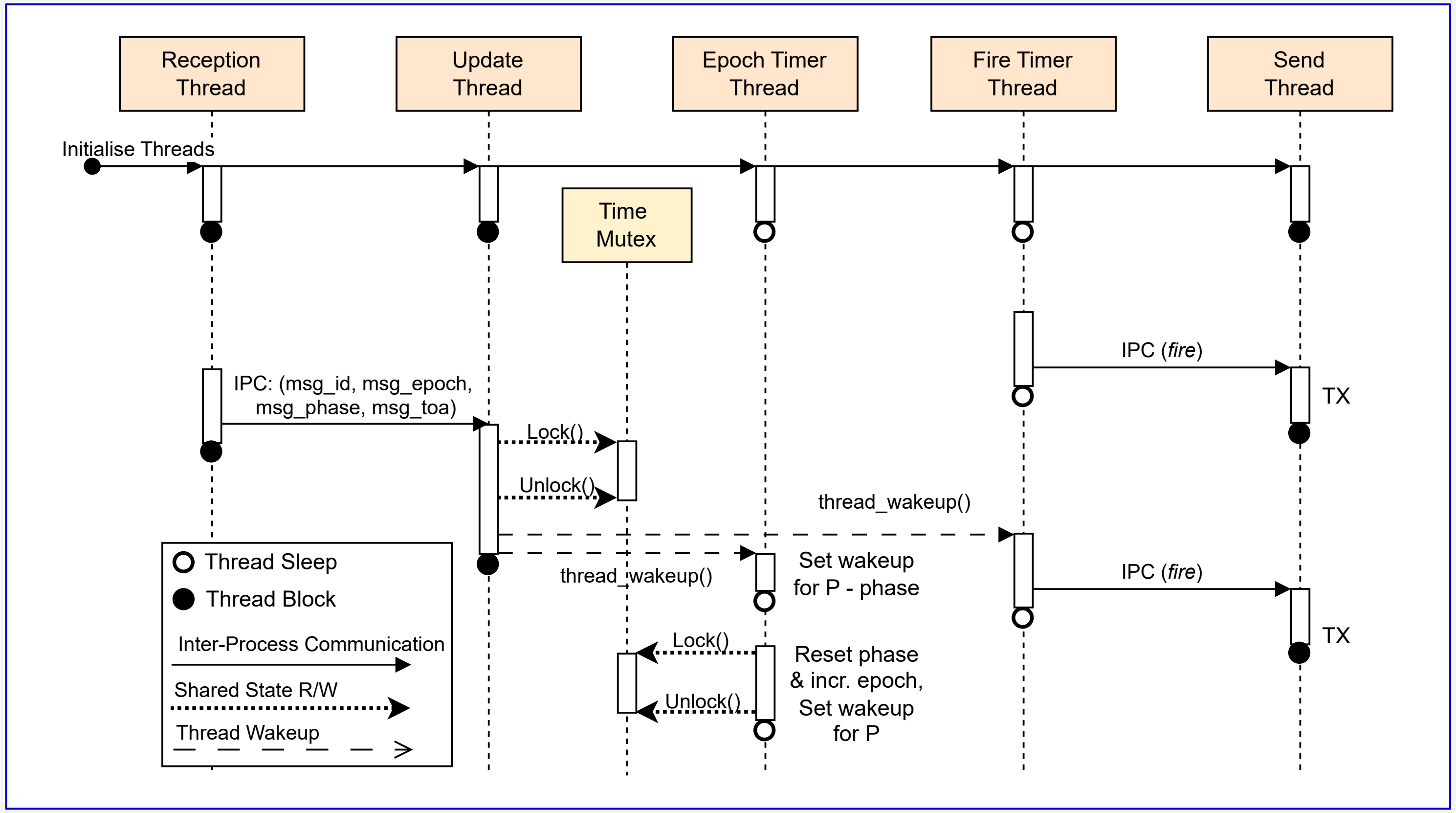
- Implemented at Network layer
  - - Lower possible accuracy and timing
  - + Ease of development
  - Compatibility with both nrf52840dk and iotlab-m3



Implementation

Challenges from Theoretical Algorithm

- Timers
- Division into threads
  - Thread scheduling priority
  - Inter-process communication
  - Thread sleeping and wakeups
  - Shared state



Implementation

Algorithm Implementation

```
procedure RECEPTION THREAD                                          ▷ Priority = 2
    Initialize socket
    while True do
        Block until message received from socket
        msg_toa ← ztimer_now()
        id_msg, φ_msg, e_msg ← decrypt_message()
        Send (id_msg, e_msg, φ_msg, msg_toa) to Update Thread
    end while
end procedure
```

```
procedure EPOCH TIMER THREAD(e, φ_offset, time_lock, I)             ▷ Priority = 3
    while True do
        Remove previous scheduled wakeup timer          ▷ (nullop if not scheduled)
        mutex_lock(time_lock)
        now ← ztimer_now()
        if now − φ_offset >= Φ then
            e ← e + 1
            φ_offset ← now − ((now − φ_offset) mod Φ)
        end if
        mutex_unlock(time_lock)
        Set wakeup timer in ztimer to trigger Φ − (ztimer_now() − φ_offset) from now
        thread_sleep()
    end while
end procedure
```

```
procedure FIRE THREAD(I, c, heard_ids_buffer)                       ▷ Priority = 5
    while True do
        Remove previous scheduled wakeup timer
        now = ztimer_now()
        if now − ψ_offset >= ψ_fire then
            Send fire message to Send Thread
            ψ_offset ← now
            c ← 0
            heard_ids_buffer ← {}
            I ← min(β · I, I_max)
            ψ_fire ← randint(ε, I)
        else if I < ψ_fire − (now − ψ_offset) then   ▷ If firing interval was just reset,
            ψ_fire ← randint(ε, I)     ▷ we might want to pick a new, shorter, time to fire
        end if
        Set wakeup timer in ztimer to trigger ψ_fire − (ztimer_now() − ψ_offset) from now
        thread_sleep()
    end while
end procedure
```

```
procedure UPDATE THREAD(e, φ_offset, c, heard_ids_buffer, time_lock, I)   ▷ Priority = 4
    initialize msg queue
    heard_ids_buffer ← {}
    while True do
        Block until IPC message
        (id_msg, e′_msg, φ′_msg, msg_toa) ← received_message
        mutex_lock(time_lock)
        now ← ztimer_now()
        φ ← now − φ_offset
        t ← e ∗ Φ + φ
        φ_processing ← now − msg_toa
        t_msg ← e_msg ∗ Φ + φ_msg + φ_processing + ĉ
        e_msg ← ⌊t_msg/Φ⌋
        φ_msg ← t_msg mod Φ
        if t_msg > t then
            e ← e_msg
            φ_offset ← now − φ_msg
            mutex_unlock(time_lock)
            thread_wakeup(epoch_timer_thread)
            if t_msg > t + ε then
                c ← 0
                I ← I_min
                heard_ids_buffer ← {}
                thread_wakeup(fire_timer_thread)
            else if c < k ∧ id_msg ∉ heard_ids_buffer then
                heard_ids_buffer ← heard_ids_buffer ∪ id_msg
                c ← c + 1
            end if
        else
            mutex_unlock(time_lock)
            if t_msg > t + ε then
                c ← 0
                I ← I_min
                heard_ids_buffer ← {}
                thread_wakeup(fire_timer_thread)
            else if c < k ∧ id_msg ∉ heard_ids_buffer then
                heard_ids_buffer ← heard_ids_buffer ∪ id_msg
                c ← c + 1
            end if
        end if
    end while
end procedure
```

```
procedure SEND THREAD(e, φ_offset, c)                               ▷ Priority = 1
    Initialize TX socket
    Initialize msg queue
    while True do
        Block until IPC message
        msg ← (id, e, ztimer_now() − φ_offset)
        msg_encrypted ← encrypt(msg)
        if c < k then
            socket_send(msg_encrypted)
        end if
    end while
end procedure
```

Evaluation

Evaluation
Testbed Setup

- FIT IoT-LAB used as a testbed

- Used most widely available deployment target:
  - Iotlab-M3 (STM32 MCU, 802.15.4 (LR-WPAN) links, 2.4GHz radio)

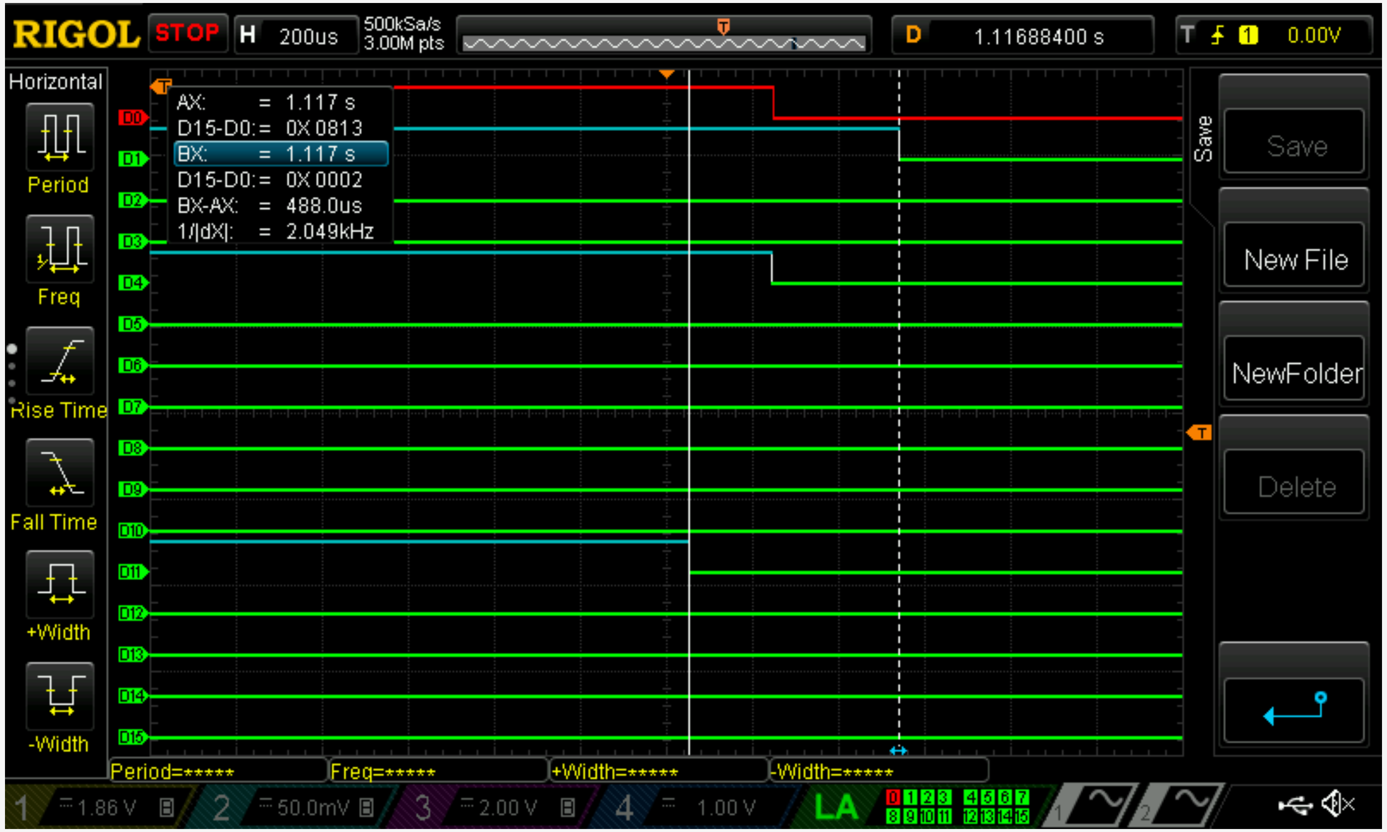
- Large scale deployment size (300+)

Evaluation
Testbed Measurement Error Bug

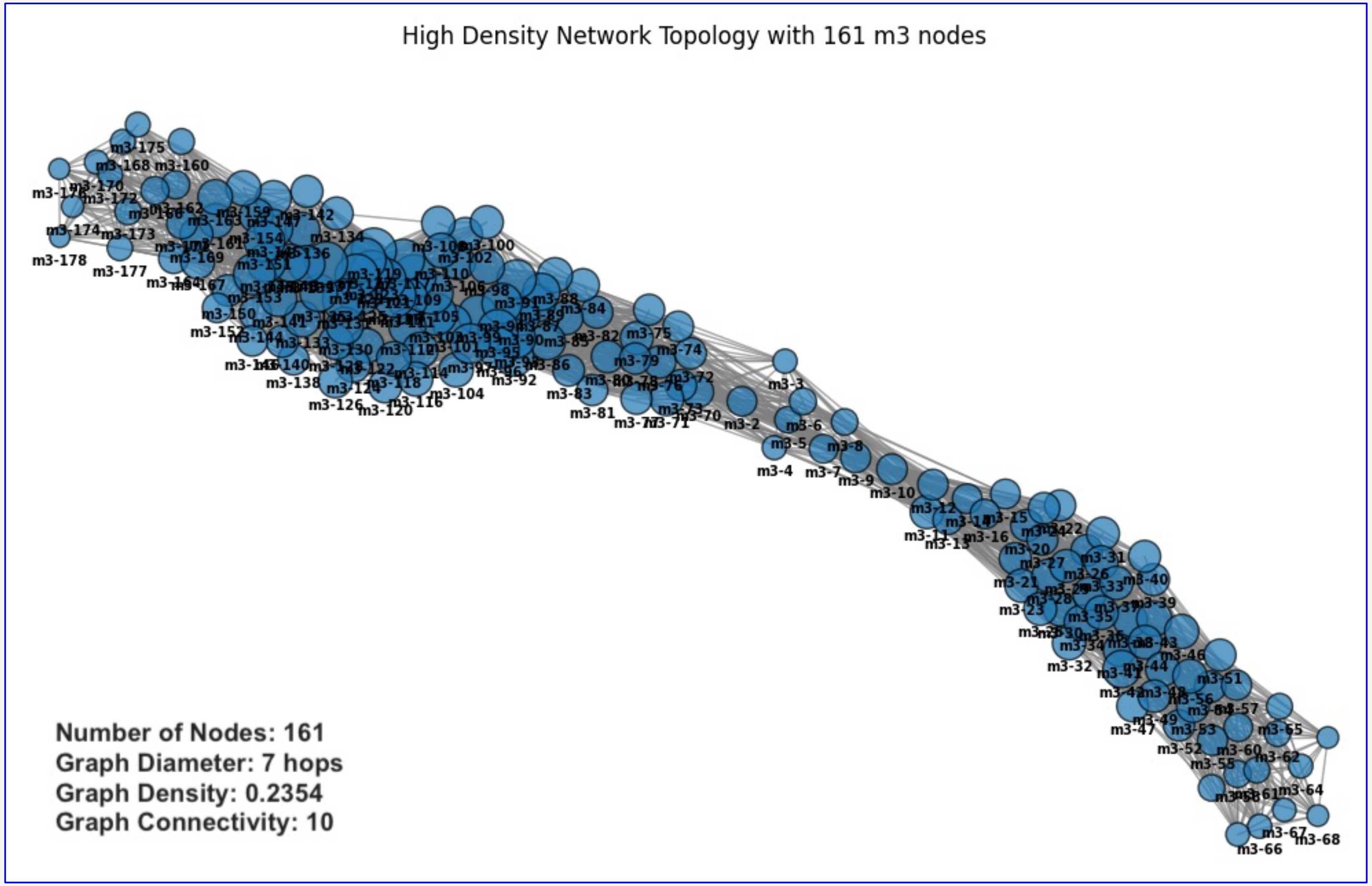
- Bug discovered in Iotlab-M3 nodes
- Causes measurement error of 8-16 ms
- Unpredictable oscillation in error for each node
- So, we measured maximum accuracy in lab

Evaluation
Synchronization Accuracy

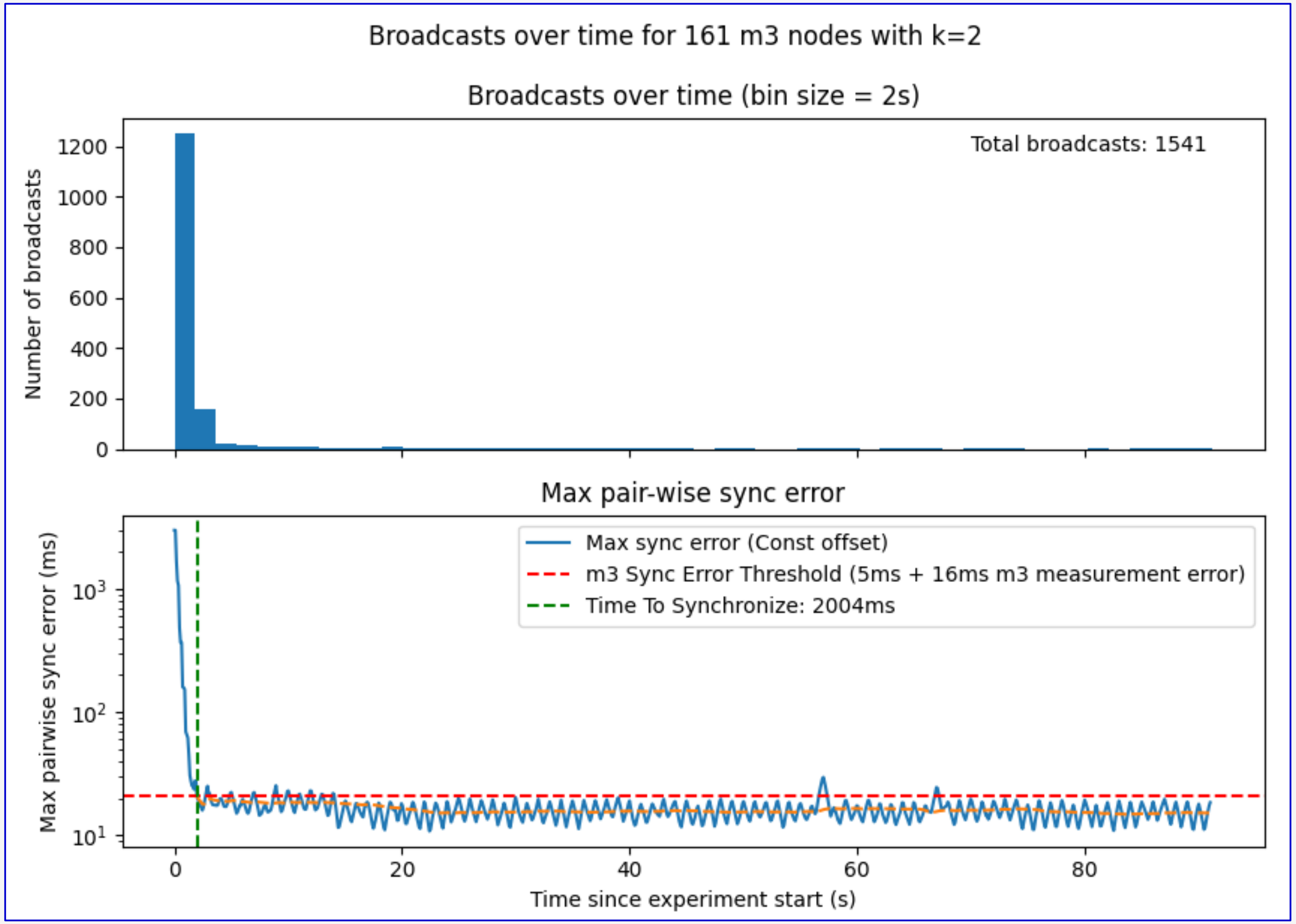
Using a digital oscilloscope, avg. synchronization accuracy of 488 μs (0.4 ms) for 4 nodes



Evaluation
Results: Dense Topologies



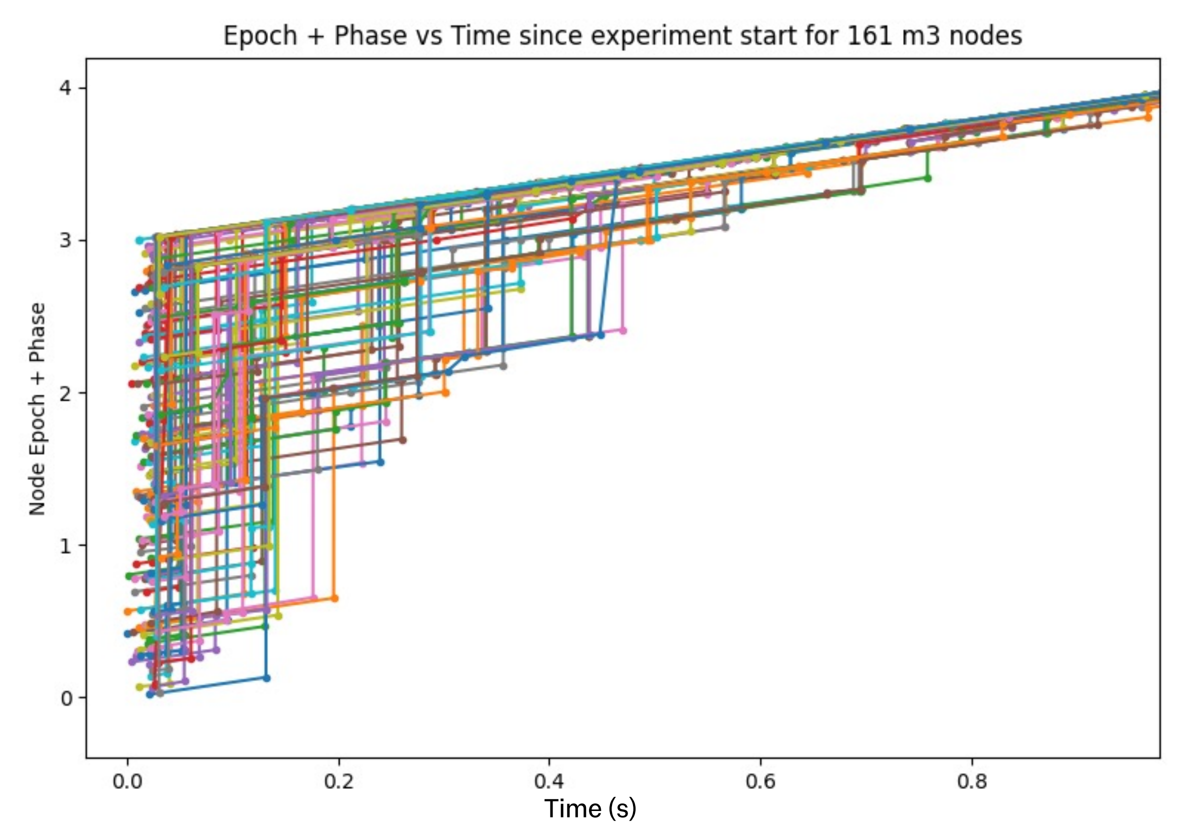
Evaluation
Results: Dense Topologies



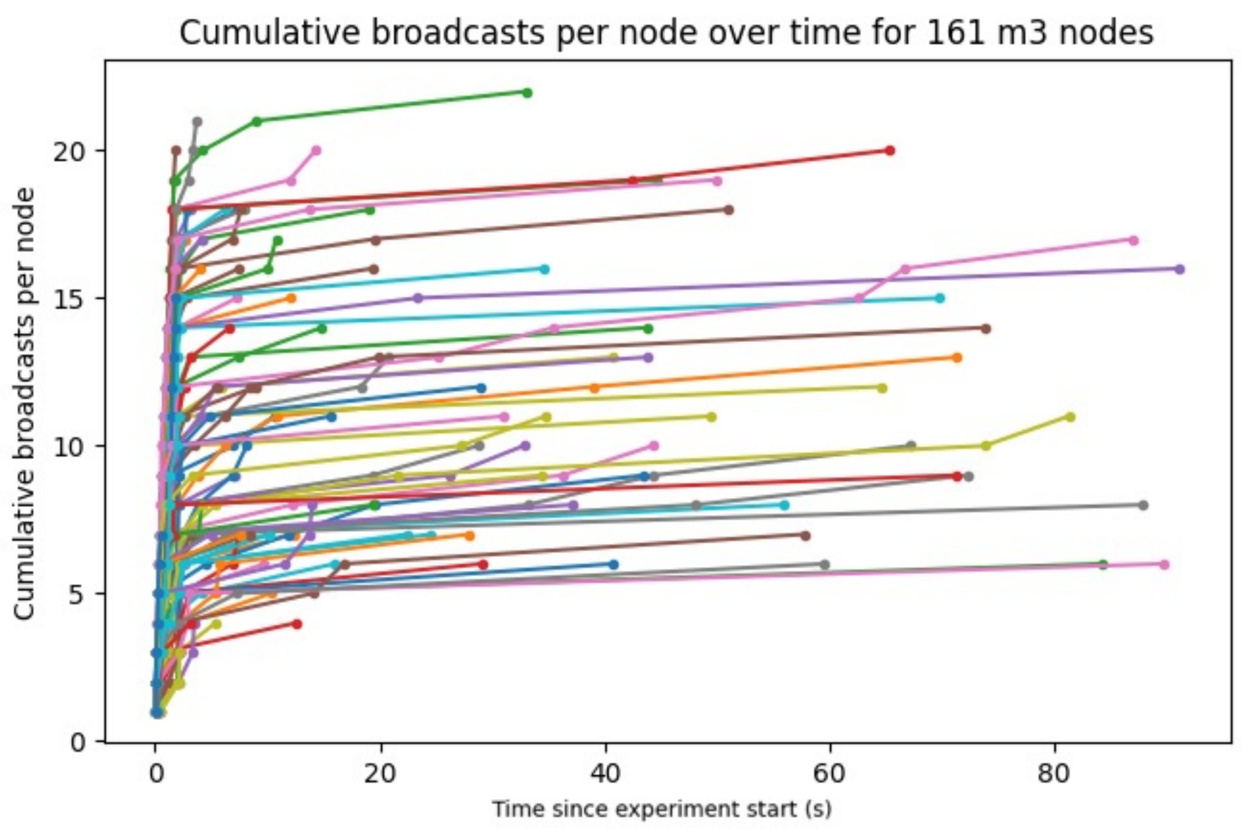
- Time to Synchronization is lowest found in literature
  - 2004 ms for 161 nodes over 7 hops
- Prev. best on equiv. topo: 48s (CMTS)
- Num. broadcasts in same range
  - 700 for 40 nodes to sync (CCTS)

Evaluation
Results: Dense Topologies

Converging to a global maximum time:

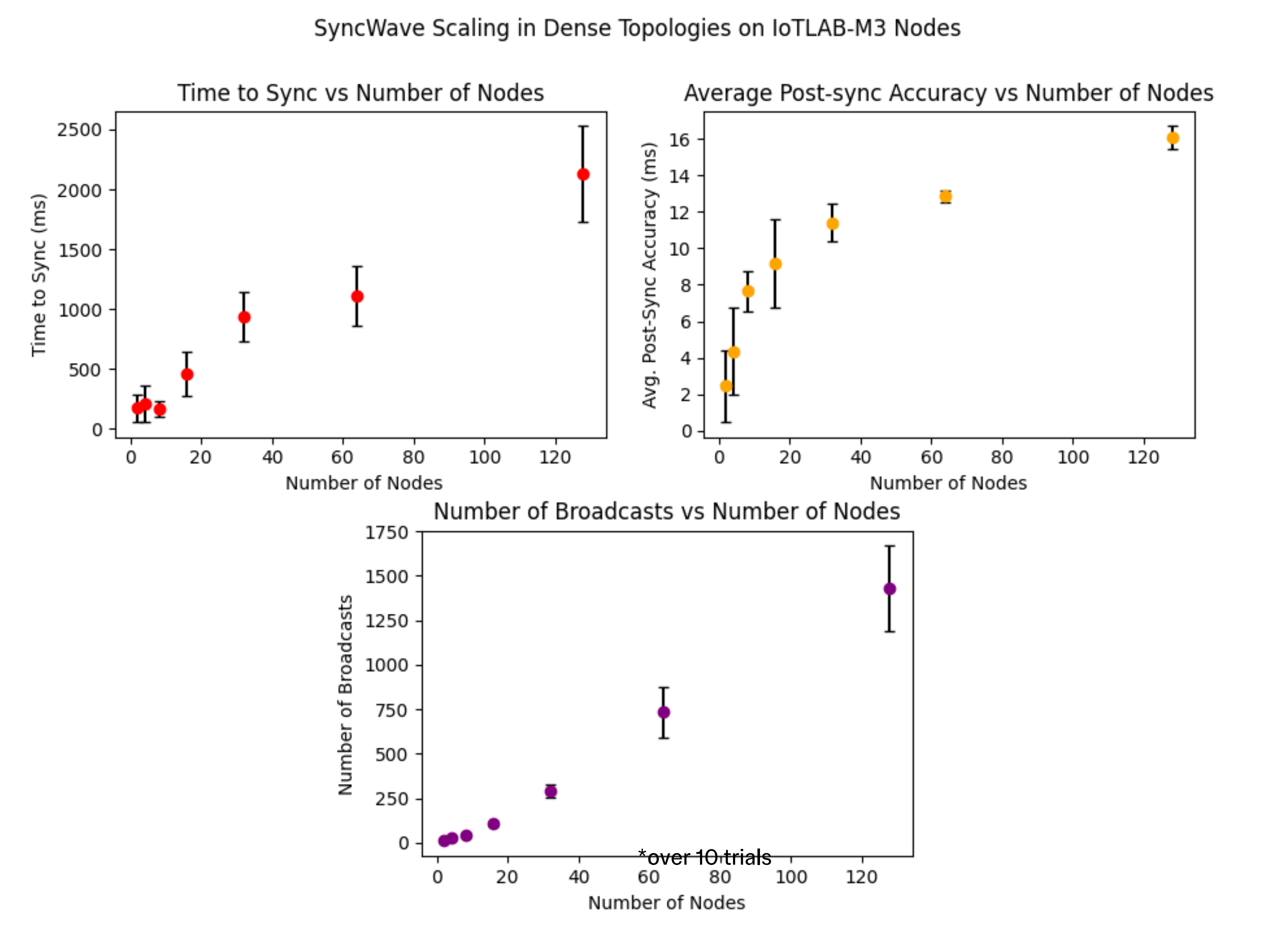


Each node's num. broadcasts over time are logarithmic:



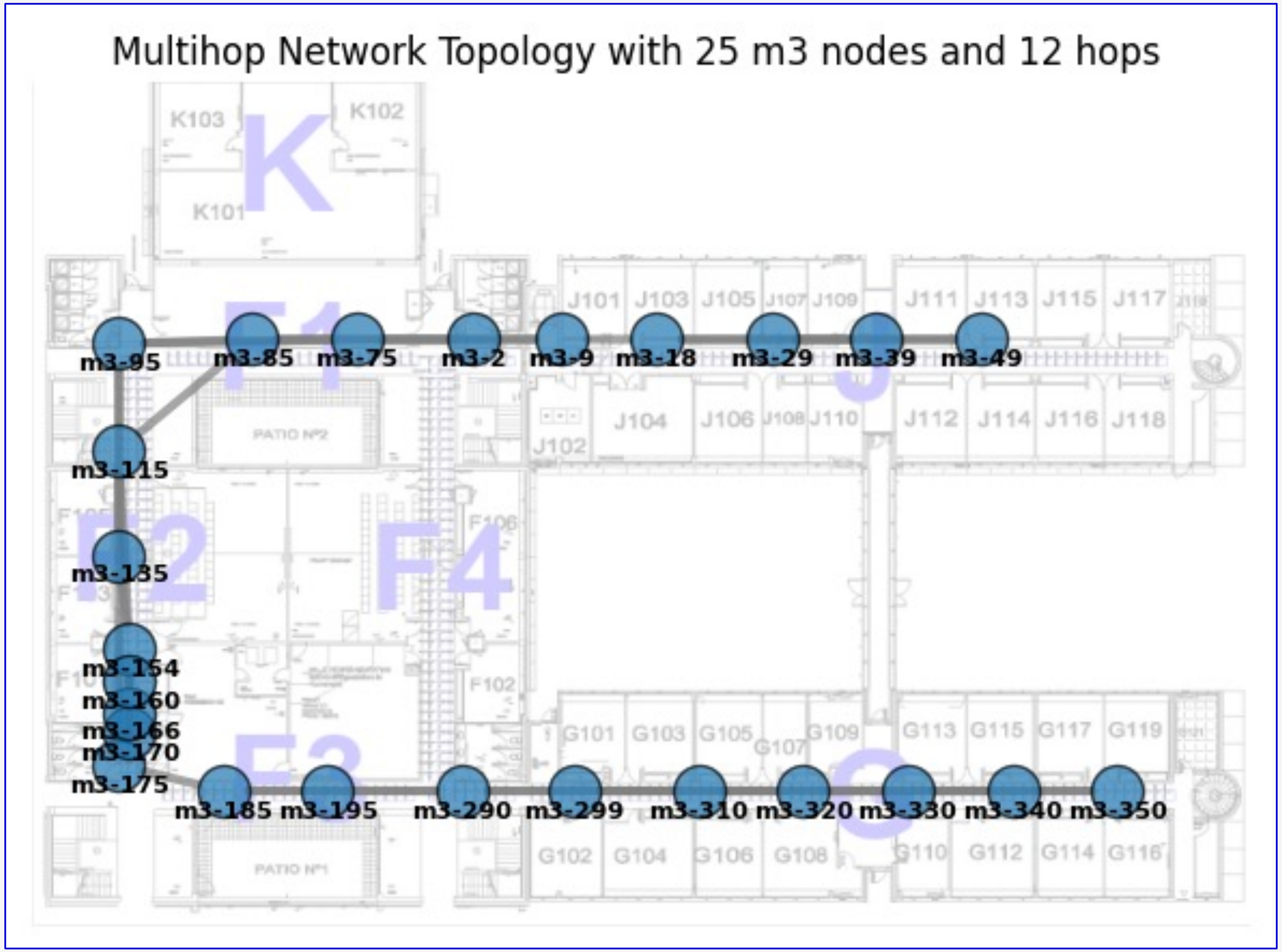
Evaluation
Results: Dense Topologies

- Time to sync linear w.r.t. num. nodes
- Num. broadcasts linear w.r.t num. nodes



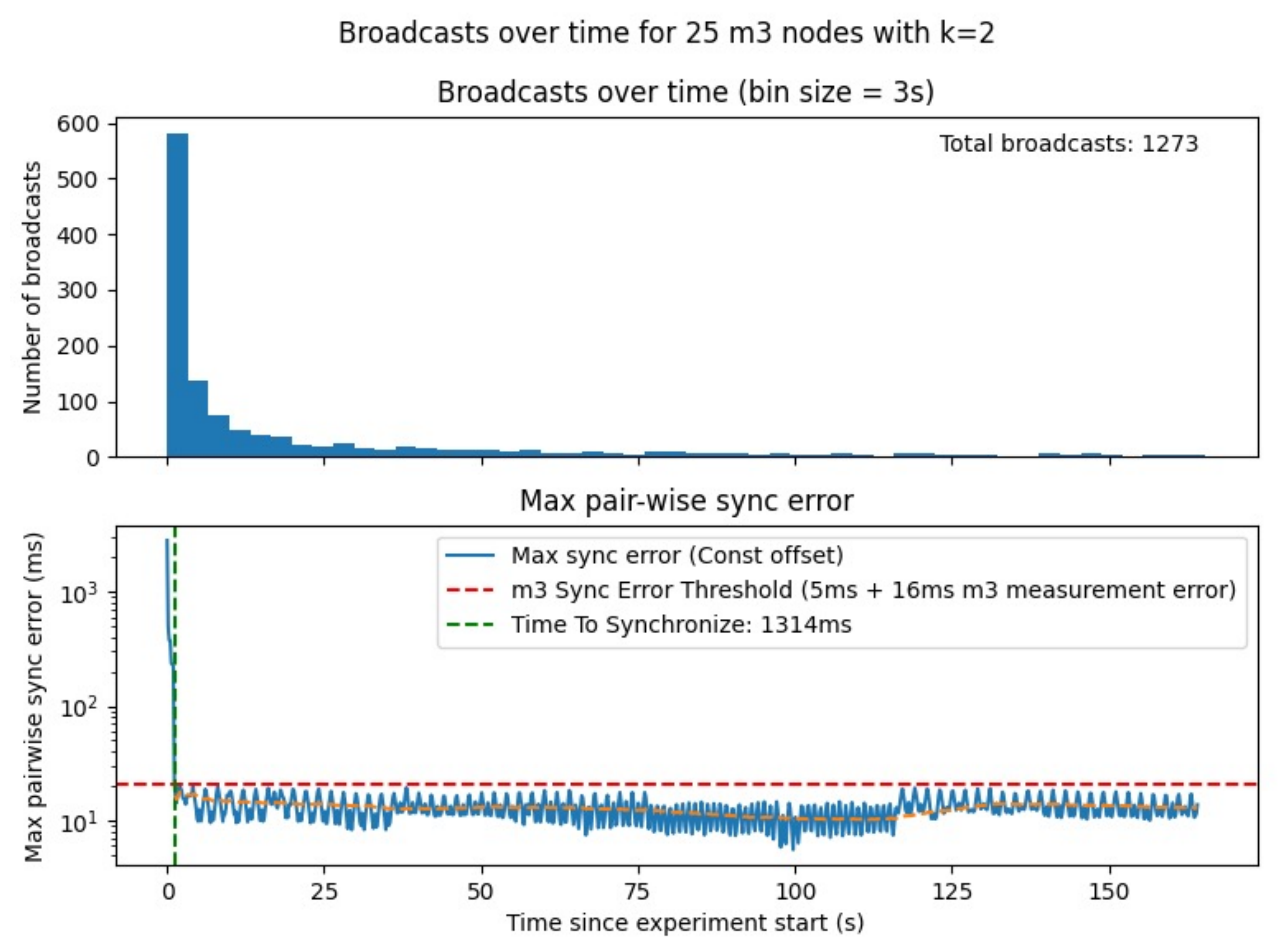
Evaluation

Results: Highly Multi-Hop Topologies



Evaluation

Results: Highly Multi-Hop Topologies

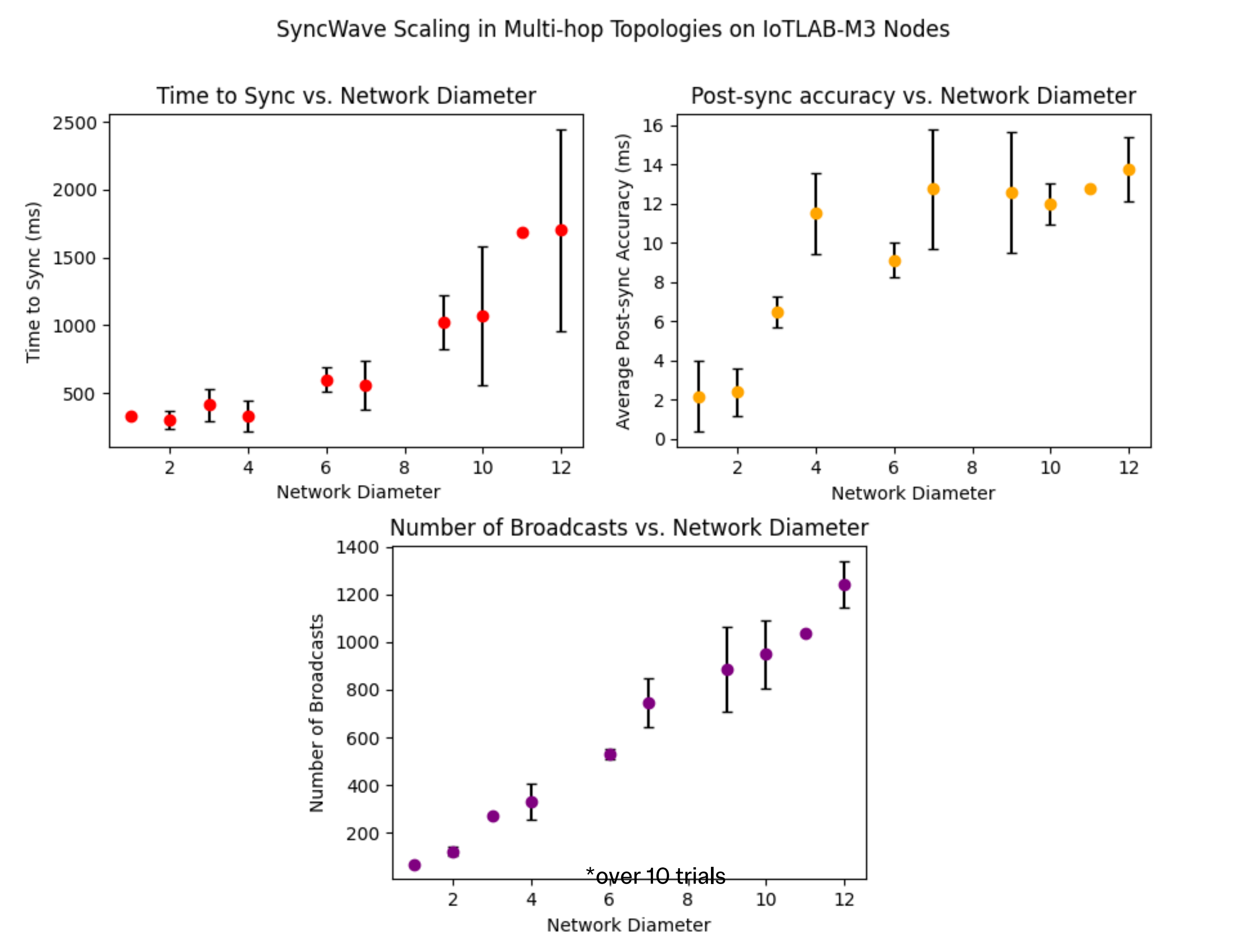


- Performs well on low connectivity, highly multi-hop
- Fewer broadcasts
- Better final accuracy

Evaluation

Results: Highly Multi-Hop Topologies

- Time to sync potentially exponential w.r.t. num. hops
  - Common in time sync. algorithms, since propagation error accumulated with each hop
- Num. broadcasts linear w.r.t num. hops



Evaluation

Results: Comparison with State of the Art Multi-Hop Time Sync. Algorithms

Algorithm	Convergence Time	Tested Topology	Sync. Error (μs)	Re-sync interval (s)
Swarm Sync	5+ mins	4 nodes, 3 hops	128	600
FTSP	6-7 mins	25 nodes, 8 hops	15	30
PulseSync	4 mins	25 nodes, 8 hops	19	10
RMTS	2 mins	25 nodes, 8 hops	6	30
CCTS	1 min	100 nodes, 4 hops	30.2	1
MTS	50 s	20 nodes, 4 hops	100	1
CMTS	48s	100 nodes, 4 hops	30.2	1
SyncWave	**2 s**	**161 nodes, 7 hops**	**440 (unoptimized)**	**None**

*Note: The re-sync interval is analogous to the period in PCO algorithms and is chosen based on the convergence time vs. radio usage trade-off.
We remove this coupling, enabling faster convergence with a low synchronized broadcast rate.

Conclusion

Conclusion

- Discussed and simulated drawbacks of existing algorithms
- Designed the SyncWave algorithm
- Implemented and adapted algorithm for real hardware
- Tested SyncWave implementation on large-scale testbed
  - Finding state-of-the-art results for our requirements
- Should help accelerate development of more intelligent and responsive swarm robotic systems

Future Work

Mac-layer implementation

A lower-level implementation of SyncWave (e.g. at the MAC layer) could massively improve accuracy and convergence time

And a more sophisticated estimation of propogation time

Deep Sleep for WSNs

Our protocol is intended for drone swarms, which have different power requirements from WSNs

Radio kept listening even once synchronized

For use on WSNs would want to enter deep sleep for some percentage of the firing interval or agree to all sleep at same time

Secure Swarms

Potential as building block for encryption, authentication, and resiliency, thanks to "epoch"

E.g. Channel hopping:

- Synchronized for free
- Hop according to epochs
- Completely de-centralized

IMPERIAL

Thanks for coming!

IMPERIAL

Questions?