Microsecond-Accuracy Time Synchronization Using the IEEE 802.15.4 TSCH Protocol

Atis Elsts*, Simon Duquennoy†‡, Xenofon Fafoutis*, George Oikonomou*, Robert Piechocki*, and Ian Craddock*
*Faculty of Engineering, University of Bristol
†Inria, Lille
‡SICS Swedish ICT

***Abstract*—Time-Slotted Channel Hopping from the IEEE 802.15.4-2015 standard requires that network nodes are tightly time-synchronized. Existing implementations of TSCH on embedded hardware are characterized by tens-of-microseconds large synchronization errors; higher synchronization accuracy would enable reduction of idle listening time on receivers, in this way decreasing the energy required to run TSCH. For some applications, it would also allow to replace dedicated time synchronization mechanisms with TSCH.**

We show that time synchronization errors in the existing TSCH implementations on embedded hardware are caused primarily by imprecise clock drift estimations, rather than by real unpredictable drift variance. By estimating clock drift more precisely and by applying adaptive time compensation on each node in the network, we achieve microsecond accuracy time synchronization on point-to-point links and a $< 2\,\mu s$ end-to-end error in a 7-node line topology. Our solution is implemented in the Contiki operating system and tested on Texas Instruments CC2650-based nodes, equipped with common off-the-shelf hardware clock sources ($\pm$20 ppm drift). Our implementation uses only standard TSCH control messages and is able to keep radio duty cycle below 1 %.

I. Introduction

There is a growing need to make low-power wireless networks more reliable and more predictable in order to open them up to a wider range of applications (*e.g.*, industrial, automotive, e-health applications). Time-Slotted Channel Hopping, specified in the IEEE 802.15.4-2015 standard [1], offers these features and has recently attracted attention from both industry and academia.

As a consequence, there is a need for interoperable, high-quality TSCH implementations for major Internet-of-Things (IoT) hardware platforms and software ecosystems. The two main qualities required by such implementations are, firstly, high accuracy timing, required to keep timeslots closely aligned on all devices in the network; secondly, high energy efficiency in order to support devices with limited energy budgets. These two qualities are closely interrelated; for example, optimizing the energy efficiency by using a lower duty cycle bounds the maximal frequency of network resynchronization packet transmission; in turn, improving the timing accuracy allows the system to have a lower duty cycle. Besides this, high-accuracy network-wide time synchronization is beneficial for many of the aforementioned applications.

A popular solution to improve the accuracy in TSCH networks is to use adaptive time synchronization [2], [3], which consists of learning the local clock drift and compensating for it on each node in the network. Adaptive synchronization for TSCH was first implemented in the OpenWSN [4] network stack with the main goal to reduce resynchronization frequency. Our work builds on the same idea, but aims to reduce the absolute synchronization error, and our implementation is able to achieve an order-of-magnitude higher accuracy.

We start by identifying the properties required from a hardware platform to achieve high-quality time synchronization, and argue that even high amplitude clock drift can be accurately compensated in software as long as the platform satisfies two properties: firstly, high timing precision (in practice, high clock resolution), secondly, low variance of the drift. We then overview how a selection of popular IoT hardware platforms lives up to these properties (Section III-B), and we show that achieving simultaneously high-accuracy timing and low Micro-Controller Unit (MCU) and radio duty cycle is challenging, as it requires the combination of timing measurements from two different hardware clock sources. Subsequently, in Section IV we mathematically model time synchronization in TSCH networks and analyze the causes of synchronization errors. Section V describes our implementation of platform-independent adaptive time synchronization for TSCH in the Contiki operating system and our port of TSCH to Texas Instruments CC2650 System-on-Chip (SoC) hardware platform, where we implement high-precision low-duty cycle timing for TSCH.

By combining the adaptive synchronization with high-precision timing, we create a high-accuracy version of TSCH. The experimental results in Section VI show that using 4 second resynchronization period for neighbors we measure $< 2\,\mu s$ end-to-end maximum error in a 7-node line topology. The improved accuracy allows to reduce idle listening during TSCH guard time by an order of magnitude, and therefore the total radio duty cycle by a factor of two: from 1.4 % to 0.73 %.

To the best of our knowledge, this work contributes the first open-source port of TSCH to the CC2650 hardware platform[1], and we are the first to show how a standards-compliant TSCH implementation can achieve microsecond-accuracy time synchronization on common IoT hardware, while keeping the MCU active <2 % and the radio active <1 % of time.

[1]Source code available at http://github.com/atiselsts/contiki/tree/ptsch

II. Related work

A. TSCH on IoT hardware

At the time of writing this paper, OpenWSN [4] is the *de facto* TSCH reference implementation, while the Contiki port for the JN516x hardware platform [5] is the *de facto* reference implementation within the Contiki world [6]. OpenWSN and Contiki versions of TSCH were demonstrated to be interoperable in Internet Engineering Task Force (IETF) plug-tests in Prague (2015) and in Paris (2016).

However, the Contiki TSCH implementation was not evaluated in an energy-efficient mode since deep-sleep was not supported on JN516x when the plugtests took place. We show why adding support for deep sleep raises new challenges for time-synchronized TSCH networks (Section V-C).

B. Time synchronization in TSCH

Adaptive synchronization for TSCH is described by Stanislowski *et al.* [2] and was initially implemented in the OpenWSN networking stack. The main idea behind adaptive synchronization is that each network node learns its drift compared to the time source and adaptively compensates for it. The authors report 60 μs maximal synchronization error on point-to-point links when using 60 sec resynchronization period.

Chang *et al.* investigate how adaptive synchronization can be improved by adapting control message exchange frequency depending on synchronization quality, and tackles the additional challenges multihop networking brings [3]. Similarly to Stanislowski *et al.*, they report synchronization errors of up to tens of microseconds.

Our work builds on these existing ideas, but substantially reduces synchronization errors by using higher-resolution clocks readily available on existing IoT platforms (Table I). We do that while continuing to use low-resolution timing during low-power modes, therefore not decreasing energy-efficiency.

There exist a number of dedicated time synchronization protocols for low-power and lossy networks [7], including some that report sub-microsecond maximum errors [8]. However, they (1) are typically not optimized for energy efficiency and (2) require additional message exchange, complicating the implementation and making it standards-non-compliant. Other options for accurate time synchronization include the use of wired systems or having Global Navigation Satellite System (GNSS) receivers on sensor nodes. However, these options may be expensive, impractical, or outright impossible to implement, as could be the case for indoor GNSS.

III. Timing on IoT hardware

A. Desirable properties

The IEEE 802.15.4 standard specifies TSCH as a time-sensitive Time Division Multiple Access (TDMA) protocol. As a consequence, it sets the following timing-related requirements for devices implementing TSCH:

- The *accuracy* requirement: According to the IEEE 802.15.4-2011 standard, radio symbol rate should be within a ± 40 ppm deviation when using Quadrature Phase Shift Keying (QPSK) modulation [1].
- The *clock resolution* requirement: Standard TSCH timings are defined in microseconds [14] (in practice, all timings are multiples of either 10 or 8 μs), so the device should be able to keep track of time with at least close-to-microsecond granularity.
- The *performance* requirement: An implementation must be sufficiently fast to receive and process (*e.g.*, decrypt) a packet, as well as construct, encrypt, and send an acknowledgment within a 10-millisecond timeslot. The maximum processing time between the end of frame and the start of acknowledgment's preamble is just 840 μs according to the standard, therefore hardware-accelerated decryption is mandatory.

If adaptive synchronization [2] is used to improve the accuracy of time synchronization, predictability of clock drift becomes another important requirement. On IoT hardware complex prediction algorithms are not common; this restricts to a more specific notion of being predictable by simple extrapolation from past values by using moving average or linear regression algorithms. In particular, if the environment of the network remains stable, the drift also should remain *stable* (*i.e.*, has low variance).

B. Hardware overview

In this paper we focus on Class-0 and Class-1 [15] IoT hardware platforms; for examples see Table I. A typical IoT system has several independent clock sources:

- Low-frequency (LF) crystal oscillator. Useful for accurate, but low-resolution timing; all of the platforms in the Table I use crystal oscillators with ≤ 20 ppm drift in room temperature [16].
- High-frequency (HF) crystal oscillator used by the radio. Accurate (≤ 40 ppm [11] [12] as required by the standard or better) and high-resolution.
- High-frequency (HF) MCU-internal RC oscillator. Useful for high-resolution timing, but typically not meeting the 40 ppm accuracy requirement; further in this paper, we focus only on crystal oscillators.

With the exception of platforms equipped with the oldest-generation CC2420 radio, all aforementioned systems offer some access to the HF crystal oscillator used by the radio, therefore enabling accurate, high-resolution packet timestamps based on the internal radio clock. The CC2520 radio exports this internal clock signal over a General-Purpose Input/Output (GPIO) pin, while other systems offer an Application Programming Interface (API) that can be used to read timestamps. Resolutions vary between 62 500 Hz and 32 MHz.

In terms of *accuracy*, TSCH does not introduce any new prerequisites. All systems with IEEE 802.15.4-2006 capable radio chips meet this requirement: failure to do so would break their ability to communicate with other nodes.

Existing TSCH implementations (OpenWSN [4], TinyOS [17], Contiki before December 2015) pick either an

TABLE I: Comparison of timing sources on IoT hardware platforms

Radio or SoC	Platform examples	LF clock frequency (on example platforms)	HF clock frequency, max	Radio-clock sourced timestamp resolution, max	Synchronization between HF and LF crystals
CC2420 [9]	Tmote Sky and Zolertia Z1	32 768 Hz	4 MHz and 8 MHz	n/a	−
CC2520 [10]	Wismote	32 768 Hz	16 MHz	16 MHz	−
CC2538 [11]	OpenMote and Zolertia RE-Mote	32 768 Hz	32 MHz	32 MHz	+
CC2650 [12]	SensorTag and SPES-2	32 768 Hz	48 MHz	4 MHz	+
JN516x [5]	JN5169	32 768 Hz	16 MHz	62 500 Hz	−
Atmega256RFR2 [13]	Radio-Sensors	32 768 Hz	16 MHz	4 MHz	+/−

HF or an LF clock as the source for TSCH timing, but not both. This is relevant to the *stability* requirement. In a node that uses one hardware clock source during its wakeup periods and a different clock source during its deep sleep periods, the cumulative drift for this node is dependent on its sleep schedule, which in turn is dependent on tasks running on it and on network traffic levels. For example, for a node with HF clock drifting at -20 ppm relative to its time source and LF clock drifting at $+20$ ppm, the cumulative drift is a linear combination of these values, with coefficients depending on how much time is spent in each of the modes. Essentially, this situation makes drift hard to predict.

Furthermore, if two or more unsynchronized clocks are used, an error is introduced every time the node switches from the low-power mode to the active mode; the magnitude of the error depends on the granularity of these clocks; the maximal error is at least as big as the tick duration of the highest-frequency clock. Using Contiki's IPv6 implementation with its default parameters, we measured 10 to 20 MCU wakeups per second. These errors are statistically independent and, while asymptotically zero on the average, cumulative in the worst case. To avoid this problem, if two or more clock sources are used as the basis of TSCH timing, they must be mutually synchronized. Few of the listed platforms offer an API for such a synchronization (Table I).

Using a low-frequency clock source in general leads to a large error in drift estimation (Fig. 5). Our results (Section VI) show that this measurement error by itself can be significant enough to severely limit the accuracy of network-wide time synchronization achievable by TSCH.

The *performance* requirement, on the other hand, is satisfied by all except the older generation Class-0 platforms (Tmote Sky and Zolertia Z1), but even those can still run TSCH, albeit using extended, nonstandard timeslot duration.

IV. Motivation: Analytic Model

According to the IEEE 802.15.4 standard, a node must start transmitting MAC-layer frame exactly τ_o (transmission offset) microseconds after the start of a timeslot. This transmission is preceded by the transmission of the PHY-layer preamble and the start-of-frame descriptor, which cumulatively takes τ_p microseconds.

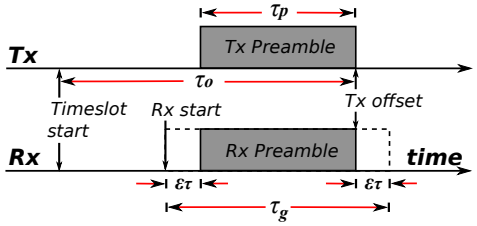


Fig. 1: The guard time τ_g is asymmetric with respect to the transmission offset τ_0 in order to leave equally large error margins ϵ_τ both before the expected start of the preamble transmission and after the expected end of it.

TSCH incorporates a guard time to deal with loss of synchronization. To account for both positive and negative clock drift, the receiver wakes up before the expected start of frame transmission offset and keeps the radio on for at least τ_g microseconds, waiting for the start-of-frame descriptor to be received. In the standard, the guard time is equally spaced around the transmission offset τ_o, *i.e.*, the node starts listening at $\tau_o - \frac{\tau_g}{2}$ and ends listening at $\tau_o + \frac{\tau_g}{2}$. However, in this paper we consider a more efficient implementation, in which the guard time τ_g is equally spaced around $\tau_o - \frac{\tau_p}{2}$, as shown in Fig. 1. Thus, for a certain maximum synchronization error ϵ_τ the minimum guard time is given by:

$$\tau_g = \tau_p + 2\epsilon_\tau \tag{1}$$

Increasing the guard time makes the network robust to larger synchronization errors, while decreasing it reduces idle listening, in this way allowing to save energy.

Fig. 2 plots the maximum synchronization error for a given guard time. The value of τ_p is hardware-dependent; in this paper, we assume $\tau_p = 160\,\mu s$ as defined by the standard for radios operating in the 2.4 GHz frequency band. The figure shows that the standard guard time ($\tau_g = 2200\ \mu s$) is appropriate for TSCH networks with millisecond-accuracy time synchronization. Decreasing the maximum synchronization error to 10 μs (for example) allows to reduce the guard time by an order of magnitude to $\tau_g = 180\ \mu s$, leading to a

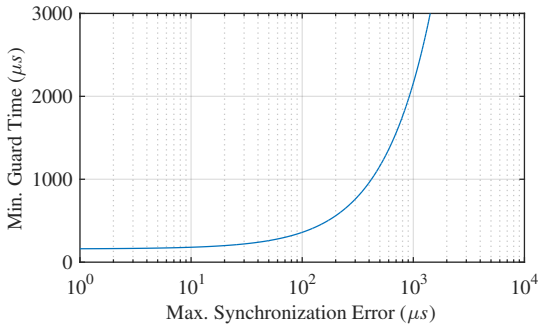


Fig. 2: Minimum TSCH guard time depending on the maximum synchronization error.

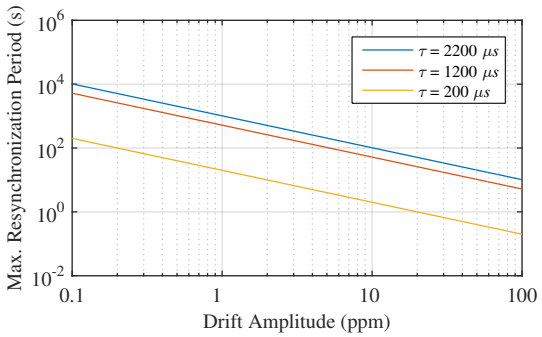


Fig. 3: Maximum resynchronization period depending on the drift amplitude δ.

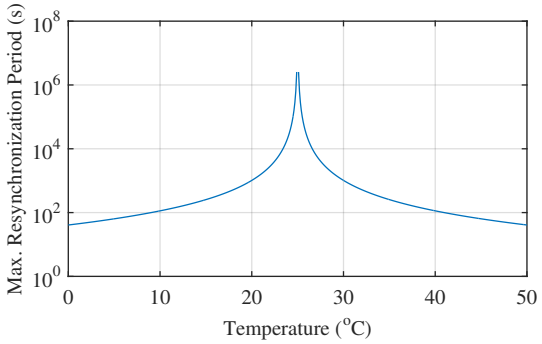


Fig. 4: Dependency of the maximum resynchronization period Δt on the operating temperature T, assuming constant temperature T_0 on the remote node; $T_0 = 25\ ^o$C, $\tau_g = 2200\,\mu s$.

proportional reduction in idle listening duration.

A. Timing errors due to production spread

One of the main reasons for clock drift is caused by oscillator crystals deviating from their nominal frequency due to production spread. Let us assume that the timings of sender and receiver nodes are scheduled using crystals with $\pm e_f$ ppm maximal frequency error, and that the environment remains stable at the nominal operating temperature of crystals ($T_0\,^o$C). The worst case scenario is when one of the crystals operates with $+e_f$ error, whilst the other operates with $-e_f$ error, leading to $2e_f$ *drift amplitude* (δ) between them. The synchronization error ϵ after time interval Δt is:

$$\epsilon = \Delta t\Big(\frac{1}{1-e_f} - \frac{1}{1+e_f}\Big). \tag{2}$$

For realistic (*i.e.*, $< 1\,\%$) frequency errors it is approximately equal to:

$$\epsilon \approx \Delta t \times 2e_f\ . \tag{3}$$

Further in this section, we assume constant sender-receiver resynchronization period Δt. In these settings, ϵ denotes the maximum synchronization error for a given drift e_f. For TSCH to operate without packet loss due to synchronization errors the following inequality needs to hold: $\epsilon_\tau \geq \epsilon$, therefore from Eq. 1 and 3 one can calculate the minimum guard time (τ_g) for a given retransmission period Δt and clock drift e_f:

$$\tau_g \geq \tau_p + 4\Big(\Delta t \times e_f\Big)\ , \tag{4}$$

as well as the minimal Δt for a given guard time:

$$\Delta t \leq \frac{\tau_g - \tau_p}{4e_f}\ . \tag{5}$$

Fig. 3 plots the maximum transmission period for various drift amplitudes for various guard times and $\tau_p = 160\,\mu s$. As the drift amplitude increases, more frequent packet transmissions are required to maintain the nodes synchronized.

B. Timing errors due to differences in operating temperature

Crystal oscillators are also characterized by a parabolic, temperature-dependent error. Typically, a crystal resonates close to its nominal frequency at $T_0 = 25^o$C, but slows down at temperature T at a rate of $B(T-T_0)^2$, where B is the parabolic coefficient. Let us assume again the worst case scenario for a TSCH link: one of the crystals operates at a temperature T_0, whilst the other operates at a temperature T. The temperature-specific synchronization error ϵ after time interval Δt is:

$$\epsilon = \Delta t\Big(\frac{1}{1-B(T-T_0)^2} - 1\Big) \approx \Delta t \times B(T-T_0)^2\ . \tag{6}$$

For crystal model FC-135 ($T_0 = 25\ ^o$C, $B = -0.04$ ppm [16]) in room temperatures (20 to $30\,^o$C), the temperature-dependent clock drift is ≤ 1 ppm, whereas at $15\ ^o$C the drift for this crystal rises up to 4 ppm. Fig. 4 shows how the maximum resynchronization period Δt depends on temperature.

C. Timing errors due to measurement errors

If implemented and enabled, adaptive synchronization allows nodes running TSCH to learn the drift relative to their time sources. However, the accuracy of the learning and compensation is limited by the clock resolution of timers used for received packet timestamps e_{Rx}, packet transmissions e_{Tx}, and scheduling of the TSCH state machine $e_{schedule}$.

The existing TSCH implementations typically schedule packet transmissions and other TSCH operations on the edge of a clock tick, therefore both $e_{Tx} \approx 0$ and $e_{schedule} \approx 0$; let us use η to denote the sum of these small errors:

$$\eta = e_{Tx} + e_{schedule}\ . \tag{7}$$

The expected value of e_{Rx}, in contrast, is quantization-dependent. The worst-case value of e_{Rx} is equal to t_{tick}. One way to decrease these quantization-specific errors is to average

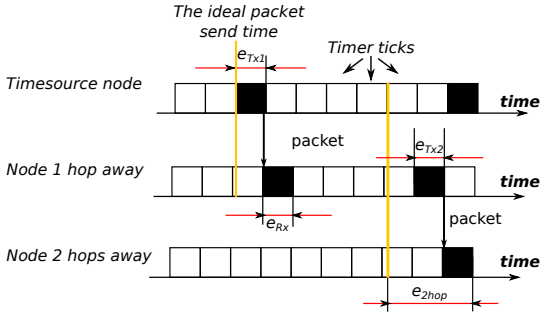


Fig. 5: Measurement error accumulation across multiple hops

them out by taking multiple measurements. As TSCH control message transmission times are randomized, the errors can be treated as statistically independent. The expected value of the measurement error e_m, obtained by averaging N samples, is:

$$e_m = \frac{t_{tick} + \eta}{N} \ . \tag{8}$$

Assuming resynchronization period Δt, the per-link drift estimation error e_{link} is given by:

$$e_{link} = \frac{e_m}{\Delta t} \ . \tag{9}$$

In contrast to the synchronization error due to a real drift (Eq. 3), this error is *inversely* proportional to Δt.

Finally, accumulation of per-link errors can happen across multiple hops (Fig. 5) and in the worst case is bounded only by the diameter d of the network:

$$e_{network} \leq d \times e_{link} \ . \tag{10}$$

For example, if low frequency crystals are used (typical frequency $f_0 = 32\,768\,\text{Hz}$), the maximum quantization error is equal to $t_{tick} = \frac{1000000}{32768} \approx 30.5 \mu s$. Assuming $\Delta t = 10\,\text{s}$, the drift estimation error using one sample ($N = 1$) is approximately $3.05\,\text{ppm}$ per-link. Assuming a 10-hop network, the drift estimation error becomes $30.5\,\text{ppm}$ in the worst-case scenario. In contrast, with a $4\,\text{MHz}$ clock source the maximum measurement error is just $t_{tick} = 0.25 \mu s$ and the drift estimation error $0.025\,\text{ppm}$ and $0.25\,\text{ppm}$ respectively. These values are two orders of magnitude smaller than the real drift of off-the-shelf crystals, therefore measuring the local drift and compensating for it allows to significantly improve synchronization accuracy in TSCH networks.

D. An example

Let us assume that: (i) temperature is learned and compensated on all nodes in the network for changes larger than $e_t\,^o\text{C}$; (ii) local drift is measured and compensated on all nodes with measurement error e_m; and (iii) the resynchronization interval is Δt. Under these conditions, the unlearned remainder ϵ of the drift on a link is given by:

$$\epsilon = \Delta t \times B e_t^2 + \frac{e_m}{\Delta t} \ . \tag{11}$$

For example, if the constant clock drift amplitude drift is estimated with error $e_m = 0.25\,\mu s$, there is a $e_t = 1\,^o\text{C}$ temperature change, and drift-temperature dependence is $B = -0.04\,\text{ppm per}\,^o\text{C}^2$ [16], then for $\Delta t = 10\,\text{s}$ resynchronization period the maximum absolute error between a pair of directly connected nodes is $abs(\epsilon) = 0.425\,\mu s$. These results suggest that sub-microsecond time synchronization in a standards-compliant TSCH network is possible.

V. Design and Implementation

We select the TI CC2650 for an implementation case study, as this SoC offers access to the high-resolution internal radio timer and also provides an API for synchronizing the radio timer with the system's LF clock.

A. Adaptive time synchronization (AS)

The time on each node in a TSCH network is periodically resynchronized with the global time of the network; it happens upon reception of either a data packet, a TSCH control packet, or an acknowledgment from a timesource node. In 6tisch [18] networks, the RPL parent of a node is used as its timesource.

In all known IoT implementations, the timing of TSCH state machine is scheduled on top of *clock tick*-granularity timing. The duration of a tick (t_{tick}) is hardware-dependent and bounds the achievable synchronization accuracy.

AS consists of two steps: *learning* (Fig. 6) and *compensation* (Fig. 7).

1) The learning step: Upon each resynchronization event, the node learns the timing error of the local clock, therefore is able to estimate the amplitude of the local clock drift by dividing the error in clock ticks with the number of ticks passed since the previous synchronization.

Our implementation of AS in Contiki uses these drift estimates as the input. It stores the last n drift estimates in a buffer and averages them to get an estimate to use in the compensation step. We follow the assumption that the largest part of the error in these drift estimates is due to imprecise measurements, rather than due to real variability of the drift, therefore averaging them gives a more accurate cumulative estimate (Fig. 6). The implementation stores the drift estimates as fixed-point numbers, expressed in units of $\frac{1}{1024}$ of *ppm*.

2) The compensation step: Once the drift of the local clock is known, it is used to adjust the timing of the TSCH state machine. Upon finishing the actions in a given timeslot, the Contiki TSCH implementation normally schedules next wakeup at the time t, the start of the next active timeslot. AS enhances this process by calculating the number c: the expected drift in ticks until the start of that timeslot, and scheduling the wakeup at time $t + c$ instead (Fig. 7).

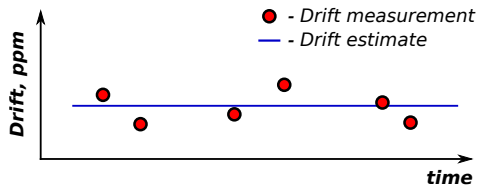


Fig. 6: Learning the clock drift by averaging measurements in the average case increases accuracy in contrast to using a single measurement.

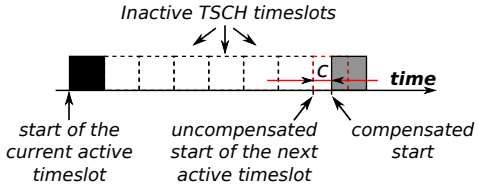


Fig. 7: Compensating for clock drift. Compensation amount c is equal to the time until the next active slot times the estimated drift.

In order to avoid error accumulation in subsequent clock compensations, our implementation uses $t_{unit} = \frac{\mu s}{1024}$ as the timing unit irrespective of t_{tick} granularity. On each compensation step i it calculates the compensation c_{units_i} and splits it in parts $c_{ticks_i} = \lceil \frac{c_{units_i} t_{unit}}{t_{tick}} \rceil$ and $c_{remainder_i} = c_{units_i} - c_{ticks_i} \frac{t_{tick}}{t_{unit}}$. In the subsequent compensation, the previously uncompensated value $c_{remainder_i}$ is included in the new value $c_{units_{i+1}}$.

3) Differences from related work: Our AS approach follows the main ideas from OpenWSN [2], [3], but with notable differences:

- OpenWSN uses just the last measurement of the local drift in the compensation step, therefore requires a large resynchronization interval Δt to accurately estimate drifts, while our implementation uses the average of multiple (n) measurements, and is able to achieve the same or better accuracy by using n times shorter resynchronization interval. More frequent resynchronization makes the system more robust and quicker to adapt to changes in drift, and in typical settings does not have large impact on energy consumption (Section VI, Fig. 13b).
- In order to adaptively increase the resynchronization interval Δt, OpenWSN integrates the estimation of the drift with a mechanism for adaptive scheduling of resynchronization messages. We instead provide just the basic learning & compensation functionality, on top of which, in principle, application-specific adaptive resynchronization strategies can be implemented as part of the future work.

It is also worth noting that in contrast to OpenWSN, the existing Contiki TSCH implementation [6] does not require the MCU to wake up at each slot, but only at the active slots. Therefore our AS mechanism is designed to be able to compensate drift across several inactive slots.

B. Porting TSCH to CC2650

Compared to asynchronous MAC layers popular in the current IoT ecosystems, TSCH places significantly different requirements on the OS and hardware.

- TSCH uses enhanced ACK packets, which current radio chips do not support in hardware. Therefore, the radio hardware's automatic ACK transmission capability has to be disabled by the driver and ACKs must be constructed and sent by software;
- CCA in TSCH is optional, therefore the radio driver should be able to turn it off;
- Packets are only expected at specific points of time, therefore polling is preferable to interrupt-based reception;
- Because of the time-sensitive nature of TSCH, accurate HW-based packet Rx timestamps are highly desirable.

We started our implementation by adapting Contiki's CC2650 radio driver based on these guidelines. Additionally, it turned out that the initialization of the CC2650 is too slow (up to $1000\,\mu s$) to function well with the existing TSCH implementation. We were forced to make two changes in Contiki's TSCH code to be able to keep the standard 10 ms slot duration on CC2650. Firstly, we made the system wake up $1000\,\mu s$ before the start of the next active TSCH timeslot; it preemptively initializes the HF crystal oscillator and then immediately allows the MCU go back to the sleep mode. Secondly, we disabled TSCH code that turned off the radio within a TSCH timeslot, between the Tx/Rx of a packet and its acknowledgment.

C. Enabling high-accuracy synchronization

The CC2650 radio core features an internal, high-frequency (4 MHz) radio timer called "RAT". Among other functions, the radio uses the RAT to timestamp incoming frames. Using the RAT is the most attractive option for TSCH timing. However, it is only accessible when the radio core is powered-up and running, therefore cannot be used *e.g.* to schedule TSCH wakeup from a low-power mode. The other option is to follow existing implementations on other systems and use the LF low-resolution always-on real-time clock (RTC) for TSCH; this achieves only $60\,\mu s$ accuracy even when AS is enabled [2], however this RTC consumes extremely low-power and is very suitable in order to minimize current draw during deep sleep.

These two options seemingly create a tradeoff between high accuracy and low energy usage. However, the CC2650 exposes an API that allows developers to synchronize the RAT with the RTC. We exploit this option to achieve better results than using each clock separately would give: every time the CC2650 radio is turned on, our Contiki implementation performs resynchronization of those two timing sources, therefore creating a unified stable high-precision low-power clock source.

This high-precision implementation is able to transmit packets and timestamp received packets with sub-microsecond error ($t_{tick} = 0.25\,\mu s$) while keeping a low duty cycle:

- In order to *transmit* packets and acknowledgments in specific radio-timer time, we use the "triggered send" option of CC2650 radio.
- We configure the CC2650 radio to include RAT-based timestamps as part of the information provided for each *received* packet.
- To schedule TSCH *wakeups*, we rely on the LF RTC.

The HF timing is used only when the radio is already on; therefore this approach does not introduce energy consumption overhead as compared to relying solely on the LF clock. In order to avoid rounding errors in time conversions, the implementation keeps track of TSCH timing in units of $\frac{\mu s}{1024}$, selected because the least common multiple of $65\,536$ and $4\,000\,000$ is $1\,024\,000\,000$.

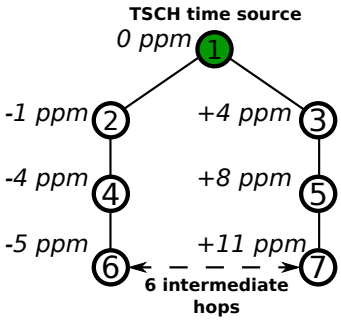


Fig. 8: Testbed network topology and approximate clock drifts. To estimate the network-wide error, we use the desynchronization detected between nodes 6 and 7.

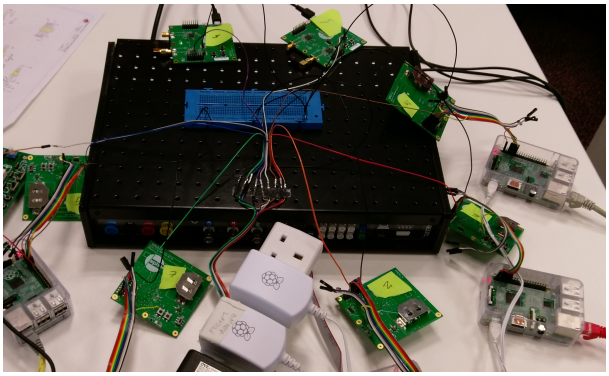


Fig. 9: Logic analyzer setup with seven nodes under test.

VI. Experimental results

We use a network of seven CC2650 nodes for the experiments (Fig. 8). TSCH schedule on the nodes is configured so that a circle topology is created by TSCH even though the nodes are physically co-located. RPL routing tree in this topology has two branches: one branch has nodes with positive drift compared to the designated router, the other branch with negative drift, creating the worst-case drift amplitude between nodes 6 and 7. Unless stated otherwise, we enable AS with measurement buffer size $n = 8$ and high-precision timing (Section V-C) and run each experiment for 10 min. We rely on TSCH beacons as the main form of synchronization packet and use the Contiki-default beacon period of 4 sec; as for the schedule, TSCH slotframe size is 47 and each node has one active timeslots for Tx, and two active timeslots for Rx: one for each of its two neighbors.

It is convenient to use the TSCH implementation itself to estimate the desynchronization: each time a TSCH node receives a packet from another node, it prints the estimated synchronization error. To validate this technique, we attach all seven nodes to a logic analyzer (Fig. 9) sampling with 16 MHz frequency and compare its measurements with the errors simultaneously logged by our software running on the nodes. The results (Fig. 10) show that this software-based error estimation is accurate, but slightly overestimates the actual desynchronization because of a larger clock granularity ($0.25\,\mu s$ *vs.* $0.062\,\mu s$). Further in this paper, software-estimated values are reported.

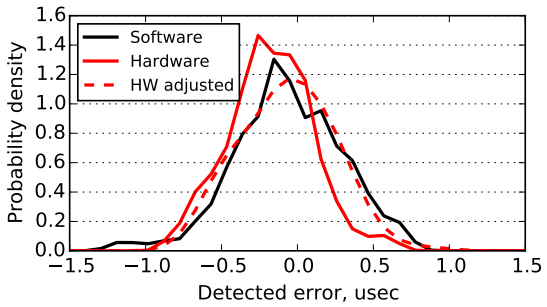


Fig. 10: Distribution of synchronization errors estimated in software *vs* errors measured with a logic analyzer. Software-estimated errors are $0.250\mu s - 0.062\mu s = 0.188\mu s$ larger on average because of larger timing granularity (4 MHz *vs* 16 MHz) and are systematically offset by around $0.1\,\mu s$ (detection overhead). Compensating for these two factors gives good a match between the methods (the dashed line).

Fig. 11 displays the drift corrections on individual nodes, while Table II includes the errors detected between nodes 6 and 7. Although both are just 3 hops away from the network timesource, in the RPL topology they are separated by 6 intermediate hops. Even though they are not logical neighbors, they may be physically close and capable of communicating. If that is the case, then broadcast messages (e.g., RPL DIO) coming from, for example, node 7 should be received not only by its current routing-tree parent node 5, but also by node 6. Therefore it is practically important to keep nodes 6 and 7 tightly synchronized to avoid losing packets.

We compare four configuration options of TSCH: LF-only timing without AS ("default", Fig. 11a) and with AS ("adaptive", Fig. 11b), and high-precision (Section V-C) timing without AS ("precise", Fig. 11c) and with AS ("adaptive precise", Fig. 11d) while using LF during sleep mode.

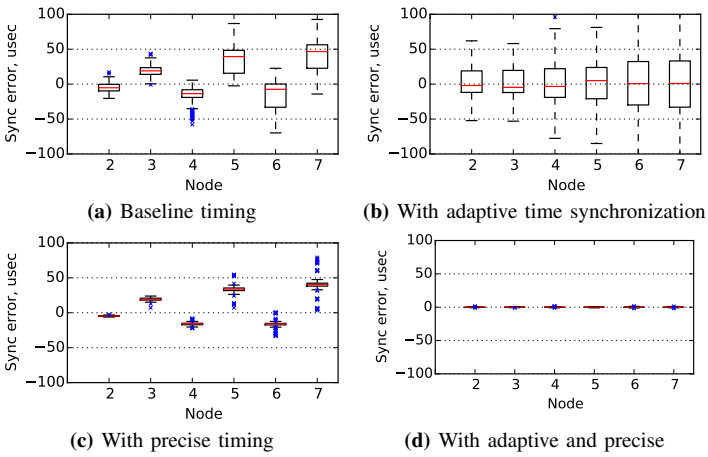
The adaptive method with LF-timing gives larger maximum error than the other methods due to imprecise estimation of the drift-to-be compensated. High-precision timing is able to better estimate drift values and therefore gives much better results (Fig. 11d): $< 2\,\mu s$ error between nodes 6 and 7.

Further experiments (Fig. 12) show that our implementation requires neither high frequency of resynchronization nor large buffers for keeping history of drift estimates in order to get accurate results. The figure shows that reducing the beacon period to less than 10 sec does not improve timing accuracy, validating our assumption that most of the errors in drift estimations are caused by measurement errors.

A longer-term evaluation (15 hour experiment) of high-accuracy synchronization in star topology further confirms that the errors have the magnitude expected when using a 4 MHz clock (Section IV-D): on point-to-point links, the average error

TABLE II: Synchronization errors in the 7-node network

Method	Maximum error, μs	Average error, μs
Default	101.7	29.8
Adaptive	165.3	45.5
Precise	110.0	58.0
Adaptive & precise	1.8	0.4



(a) Baseline timing **(b)** With adaptive time synchronization

(c) With precise timing **(d)** With adaptive and precise

Fig. 11: TSCH time correction amplitude depending on configuration options.

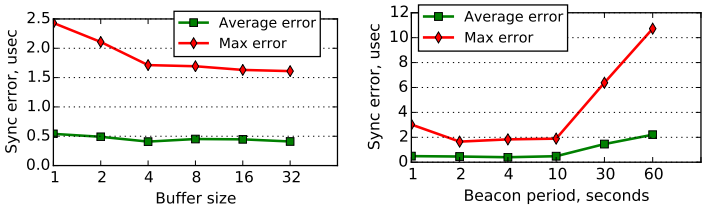


Fig. 12: Synchronization errors depending on TSCH settings in the 7-node network.

is 0.24 μs; 99.8 % of errors are below 1.0 μs and 90.4 % of errors are below 0.5 μs. The maximum error is between 0.97 μs and 1.5 μs depending on the node.

This single-hop multi-node experiment also gathered sufficient number of *longer* packets to show the effect of the packet size. Figure 14 demonstrates that the start of transmission of 65-byte packets takes place on average 0.4 μs later

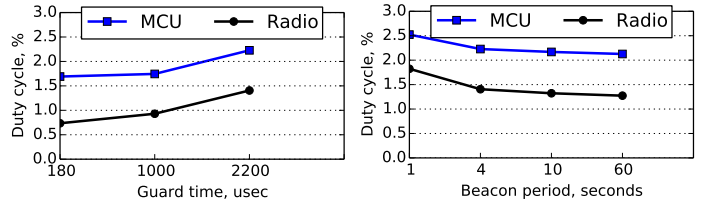


Fig. 13: MCU and radio duty cycles depending on TSCH guard time duration and beacon period. In our setup, decreasing the guard time leads to larger gains than increasing beacon period.

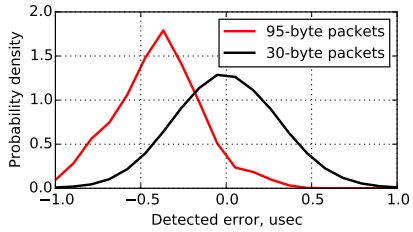


Fig. 14: Distribution of synchronization errors for 30-byte ($n = 112052$) *vs* 95-byte ($n = 383$) packets.

than start of transmission of 30-byte packets. The result is negative synchronization "errors" being detected on receivers for the longer packets. This suggests that the synchronization accuracy on CC2650 can be further increased by taking into account packet sizes.

Thanks to the high-accuracy time synchronization, we are able to reduce the TSCH guard time from its standard value of 2200 μs to 180 μs while observing no decrease in PRR. With our two-Rx-slots-per-47 TSCH slotframe, the radio duty cycle is reduced by approximately 50 %, *i.e.* from 1.40 % to 0.73 % (Fig. 13a), an effect that would be even stronger if the Contiki-default 1-in-7 slotframe were used. In contrast, increasing the resynchronization period Δt while keeping guard time and schedule constant has smaller impact on energy efficiency (Fig. 13b): in our setup using $\Delta t = 60$ sec results in 1.27 % radio duty cycle. Further increase would give negligible effects: in the experiment with 60-second period, > 99.9 % of radio-on time is spent in idle listening. This result is at odds with the direction pursued in existing work [2], [3], which aims specifically to increase the resynchronization period.

VII. Conclusions and Future Work

We have shown that there is no need to make a trade-off between high-accuracy time synchronization and low duty cycles. By synchronizing the LF and HF timing sources on CC2650 System-on-Chip nodes, we are able to accurately learn and compensate for the drift of the local LF clock while keeping MCU duty cycle below 2 % and radio below 1 %. The 4 MHz HF clock, active during the "on" mode, allows to precisely measure the local drift enabling sub-microsecond synchronization on point-to-point links. For multihop networks, the error increases no more than the number of hops between nodes, leading to $< 2\,\mu s$ empirically observed maximal error in a 7-node line topology. The method presented in this paper does not require any protocol extensions on top of standard IEEE 802.15.4 TSCH and is robust to changes in configuration settings.

In the future work, we plan to verify that the same idea is applicable to other modern IoT hardware platforms using similar architectures, such as the CC2538 System-on-Chip. Due to a higher timestamp resultion (Table I), the CC2538 is likely to produce even better results. We also plan to study the implications on the networking layer, for example, on TSCH scheduling.

Acknowledgments

This work was performed under the SPHERE (a Sensor Platform for Healthcare in a Residential Environment) Interdisciplinary Research Collaboration (IRC) funded by the UK Engineering and Physical Sciences Research Council (EPSRC), Grant EP/K031910/1. It was also partly supported by a grant from CPER Nord-Pas-de-Calais/FEDER DATA and by the distributed environment Ecare@Home funded by the Swedish Knowledge Foundation.

The authors are thankful to Antonis Vafeas for his help with logic analyzer measurements.

REFERENCES

[1] "IEEE Standard for Local and metropolitan area networks—Part 15.4," IEEE Std 802.15.42015, 2015.

[2] D. Stanislowski, X. Vilajosana, Q. Wang, T. Watteyne, and K. S. Pister, "Adaptive synchronization in IEEE802.15.4e networks," *Industrial Informatics, IEEE Transactions on*, vol. 10, no. 1, pp. 795–802, 2014.

[3] T. Chang, T. Watteyne, K. Pister, and Q. Wang, "Adaptive synchronization in multi-hop TSCH networks," *Computer Networks*, vol. 76, pp. 165–176, 2015.

[4] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "OpenWSN: a standards-based low-power wireless development environment," *Transactions on Emerging Telecommunications Technologies*, vol. 23, no. 5, pp. 480–493, 2012.

[5] "JN516x IEEE802.15.4 Wireless Microcontroller," http://www.nxp.com/documents/data_sheet/JN516X.pdf.

[6] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne, "Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2015, pp. 337–350.

[7] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 39–49.

[8] R. Lim, B. Maag, and L. Thiele, "Time-of-flight aware time synchronization for wireless embedded systems," in *EWSN 2016*, pp. 149–158.

[9] "CC2420: 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver," http://www.ti.com/lit/gpn/cc2420.

[10] "CC2520: 2.4 GHZ IEEE 802.15.4 / ZIGBEE RF TRANSCEIVER," http://www.ti.com/lit/gpn/cc2520.

[11] "CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee Applications," http://www.ti.com/lit/ds/symlink/cc2538.pdf.

[12] "CC2650 SimpleLink Multistandard Wireless MCU," http://www.ti.com/lit/ds/symlink/cc2650.pdf.

[13] "8-bit AVR Microcontroller with Low Power 2.4GHz Transceiver for ZigBee and IEEE 802.15.4: ATmega256RFR2," http://www.atmel.com/images/atmel-8393-mcu_wireless-atmega256rfr2-atmega128rfr2-atmega64rfr2_datasheet.pdf.

[14] "IEEE Standard for Local and metropolitan area networks—Part 15.4, Amendment 1: MAC sublayer," IEEE Std 802.15.4e-2012, 2012.

[15] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," IETF, RFC 7228, 2014.

[16] "FC-135R / FC-135 kHz range crystal unit," https://support.epson.biz/td/api/doc_check.php?dl=brief_FC-135R_en.pdf.

[17] "TinyOS implementation of TSCH," https://github.com/EIT-ICT-RICH/tinyos-main/tree/tkn-tsch.

[18] "IPv6 over the TSCH mode of IEEE 802.15.4e IETF working group," https://tools.ietf.org/wg/6tisch/.