Computer Networks 76 (2015) 165–176

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Adaptive synchronization in multi-hop TSCH networks

Tengfei Chang [a,c,*], Thomas Watteyne [b], Kris Pister [a,b], Qin Wang [c]

[a] BSAC, University of California, Berkeley, CA, USA
[b] Linear Technology/Dust Networks, Hayward, CA, USA
[c] University of Science and Technology, Beijing, China

ARTICLE INFO

Article history:
Received 23 July 2014
Accepted 5 November 2014
Available online 15 November 2014

Keywords:
Time Slotted Channel Hopping
IEEE802.15.4e
Synchronization

ABSTRACT

Time Slotted Channel Hopping (TSCH) enables highly reliable and ultra-low power wireless networking, and is at the heart of multiple industrial standards. It has become the de facto standard for industrial low-power wireless solutions, and a true enabler for the Industrial Internet of Things. In a TSCH network, all nodes remain tightly synchronized by periodically communicating with one another to compensate for clock drift. The synchronization algorithm used in a network determines how often the nodes need to re-synchronize, which greatly influences their energy consumption.

This article presents an adaptive synchronization technique which allows a node to learn and predict how its clock is drifting relative to its neighbors', and coordinates the instants at which the nodes re-synchronize. This technique increases synchronization accuracy, while reducing synchronization communication overhead, thereby extending the battery lifetime of the network.

Through simulation, we show how adaptive synchronization allows the nodes in a 3-hop deep network to maintain synchronization within 76 μs of one another, while sending an average of only 18.9 re-synchronization packets per hour, a 83% reduction compared to a network not using adaptive synchronization. Through experimentation on a range of hardware platforms, we show how adaptive synchronization is needed for interoperability.

1. Introduction

Reliability and ultra-low power operation are critical requirements for industrial applications. Multi-path fading, path obstruction and external inference challenge the reliability of a low-power wireless network. The energy efficiency of a low-power wireless node largely depends on how the communication protocol duty-cycles the use of its radio.

Time Slotted Channel Hopping (TSCH) is a technique which achieves both low-power operation and high reliability. All nodes in a TSCH network are synchronized; communication is scheduled so radios are only switched on when required, leading to low-power operation. Subsequent (re) transmissions happen at different frequencies, resulting in "channel hopping". The frequency diversity exploited by channel hopping combats the effects of interference and multi-path fading, leading to high reliability.

TSCH is at the core of established industrial standards such as WirelessHART [1], ISA100.11a [2] and IEEE802.15.4e-2012 [3] (an amendment to the IEEE802.15.4-2011 [4] standard). The recently formed IETF 6TiSCH working group [5] standardizes how to combine the efficiency of IEEE802.15.4e TSCH with the ease-of-use of IPv6. Commercial wireless mesh networking solutions exploiting TSCH are available today in which the network exhibits over 99.999% end-to-end reliability and in which an individual device draws an

* Corresponding author at: BSAC, University of California, Berkeley, CA, USA.
E-mail addresses: tengfei.chang@eecs.berkeley.edu (T. Chang), watteyne@eecs.berkeley.edu (T. Watteyne), pister@eecs.berkeley.edu (K. Pister), wangqin@ies.ustb.edu.cn (Q. Wang).

http://dx.doi.org/10.1016/j.comnet.2014.11.003
1389-1286/

average current below 50 uA at 3.6 V. With tens of thousands of TSCH networks deployed today, TSCH is a proven technology [6].

In a TSCH network, time is cut into timeslots. Timeslots are grouped into a slotframe which continuously repeats over time. Each timeslot is associated an index – called the Absolute Slot Number (ASN) – which increments at each timeslot. The ASN indicates how many slots have elapsed since the formation of the network. The duration of a timeslot (typically 10 ms) is large enough to fit a data packet, followed by an acknowledgment indicating successful reception. Within a slot, the data packet is transmitted precisely `TsTxOffset` (a duration) after the beginning of the slot, as depicted in Fig. 1. To allow for slight de-synchronization between neighbor nodes, the receiver starts listening `GuardTime` *before* `TsTxOffset`. If the receiver has not started receiving a packet `GuardTime` *after* `TsTxOffset`, it turns off its radio to conserve energy. This requires two neighbors nodes to never be de-synchronized by more than `GuardTime`.

Wireless nodes use a clock source (e.g. a crystal) to keep track of time. Differences in manufacturing, temperature and supply voltage cause clocks from two nodes to beat at a slightly different frequency, resulting in clock "drift". If the drift between two neighbor is measured to be 30 ppm (a typical value), the clocks of these two neighbor nodes drift apart by 30 μs every second. Neighbor nodes need to periodically re-synchronize to never be de-synchronized by more than `GuardTime`. Re-synchronizing is done by exchanging packets and using the timestamp of the data packet arriving to re-align the clocks. How often re-synchronization is needed depends on the importance of the clock drift, and the value of `GuardTime`. For example, with a `GuardTime` of 1 ms (a typical value) and a 30 ppm drift, re-synchronization needs to happen at least every $\frac{1\text{ ms}}{30\text{ ppm}} = 33$ s. If a node does not re-synchronize to a neighbor for more than that duration, it will have drifted by more than the `GuardTime` from that neighbor. The lower the relative drift, the less often they need to communicate to re-synchronize, and hence the lower the power consumption, as the radio is kept off a larger portion of the time.

This article proposes an adaptive synchronization technique in which neighbor nodes learn and predict their relative drift, resulting in less frequent re-synchronization. By keeping a history of time offsets when resynchronizing, neighbor nodes measure their relative drift. They use that information to periodically adjust their clocks to compensate for this drift. On top of this, this article proposes to coordinate the re-synchronization instants of the different nodes in a multi-hop network. This causes a node to re-synchronize right after its time parent has, i.e. when its parent is most tightly synchronized to the network. The result is that the network is more tightly synchronized as a whole and does not suffer from "synchronization swing".

The remainder of this article is organized as follows. Section 2 presents related work on adaptive synchronization in low-power wireless networks. Section 3 describes how synchronization is achieved in a TSCH network. Section 4 presents the adaptive synchronization technique proposed in this article. Section 5 extracts the performance of adaptive synchronization through simulation and experimentation, and discusses the importance of adaptive synchronization for the development of an interoperable Internet of Things (IoT). Finally, Section 6 concludes this paper.

2. Related work

Synchronization among nodes in a network is a common requirement. This section presents the previous work on synchronization in low-power wireless networks most closely related to the technique developed in this article.

Wireless mesh networks are used in many different applications, and different classes of applications put different constraints on network synchronization. [7] provides a taxonomy for synchronization protocols, organized around the different synchronization techniques available, and the nature of the applications.

In particular, [7] differentiates master–slave from peer-to-peer architectures. In a master–slave architecture, a clear hierarchy between nodes exists; it is for example typical to have a "time master" node, to which all the other nodes synchronize, possibly over a multi-hop network. The article also classifies techniques based on whether they are deterministic (the adaptive synchronization can guarantee an upper bound on the clock offset with certainty). According to this taxonomy, the techniques presented in the current article follow a master–slave architecture, and use clock correction to time sources internal to the network, with a high degree of determinism.

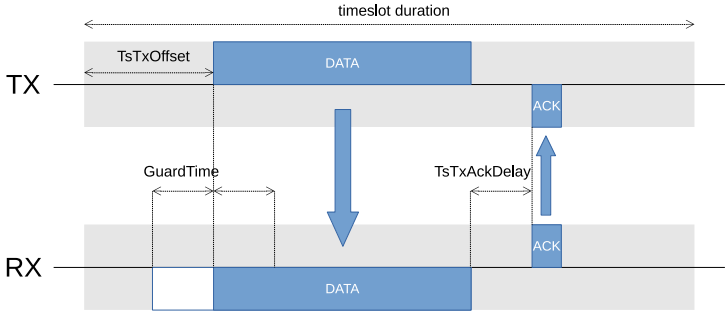


Fig. 1. The timeslot template as defined in the IEEE802.15.4e standard [3].

In [8], the author indicate that, if one wants to increase the sleep periods between resynchronization instants to minimize energy consumption, clock drift must be taken into account. Manufacturing inaccuracies and temperature variation are the two major contributors to drift. The article analyzes the impact of the two factors on the resynchronization rate, and presents a mathematical model to calculate the resynchronization period as a function of the required accuracy. Similar to [8], the current article provides a boundary for the resynchronization period which depends on the maximum acceptable time offset between neighbors.

In [9], the authors provide a method for estimating the long-term drift which minimizes the energy consumption of the nodes. The authors use long-term empirical measurements to analyze the relationship between resynchronization rate, history of synchronization packets and estimation algorithm. They use a linear model to represent relative clock drift between two indoor neighbors nodes. The authors discuss how to predict the probability of different time errors.

The Flooding Time Synchronization Protocol [10] (FTSP) is a time synchronization method to deal with clock drift in a low-power wireless network. When using FTSP, neighbor nodes estimate their relative drift by using a linear regression on the past 8 time offset measurements. This estimation is then used to predict and reduce the apparent drift. [10] presents experimental results in which FTSP runs on two Mica2 nodes. They show that the nodes can keep synchronized within 10 μs while re-synchronizing only every 10 min. The single-hop synchronization method proposed in this article (see Section 4.1) is similar to FTSP, although it uses only the last time-offset measurements. Unlike FTSP, the current article applies adaptive synchronization to TSCH networks, which adjusts synchronizing period dynamically through learning the drift of nodes, and introduces a technique to synchronize a complete multi-hop network (Section 4.1).

The technique presented in [11] aims at keeping a TSCH network synchronized while reducing the frequency of re-synchronizations. In [11], a node measures the time it takes for it to drift by 30 μs (a 32 kHz crystal clock tick) relative to its neighbor. It then periodically applies a one-tick time correction, using the previously calculated period.

Ref. [12] presents a detailed analysis of adaptive synchronization. It formalizes the relationship between calculated drift accuracy and re-synchronization interval, and shows how it is possible to extend the re-synchronization interval to 5 min outdoors, and 1 h in a temperature-controlled environment [12] formalizes how the main error of calculation of clock drift can be attributed to time correction accuracy, which is itself determined by clock frequency. For a 32 kHz clock, the time correction accuracy would be 30 μs (one clock tick). It shows how extending the re-synchronization interval increases the accuracy of the calculated drift. Through mathematical modeling and analysis, the paper shows that it is possible to gradually extend the re-synchronization interval while keeping the time offset below a given threshold.

To the best of our knowledge, and unlike the related work described above, the current article is the first to apply adaptive synchronization to a multi-hop network as a whole. The related work presents techniques to reduce the apparent drift between neighbor nodes, i.e. they focus on single-hop synchronization. Regardless of the technique used, a certain amount of clock de-synchronization will always remain. In a multi-hop setting, these inaccuracies add up at each hop, possibly leading to "synchronization swing". This article proposes a technique to combat this.

This main contributions of this article are:

- We present a single-hop synchronization algorithm, similar to FTSP, for a node to learn and predict the clock drift to its neighbors. This reduces the apparent drift between neighbor nodes, extending the network's lifetime.
- We organize the nodes in a network as a Directly Acyclic Graph (DAG) built around a single time master. The DAG structure, which is loop-free by nature, assigns a time source neighbor to each node. Following the recommendations of the IETF 6TiSCH working group, we propose to reuse the DAG structure built by the RPL routing protocol [13].
- Each node needs to periodically send a packet to its parent to re-synchronize. Rather than having all nodes re-synchronize at un-coordinated times, we propose to coordinate this activity so that a node re-synchronizes right after its parent has. This results in a periodic "synchronization wave" moving from the root of the DAG outwards, resulting in tighter network-wide synchronization.
- The proposed adaptive synchronization technique is implemented in OpenWSN, and evaluated through simulation and real-world experimentation on multiple hardware platforms. Besides showing the robustness of the approach, it also shows that its complexity is low, and can be implemented in a inter-operable way on off-the-shelf platforms.

3. Synchronization in IEEE802.15.4e TSCH

All nodes in a TSCH network are synchronized. When a new node joins a network, it synchronizes to it and must keep synchronized at all times after that. If it looses synchronization, it must re-join the network, which takes both time and energy; this is particularly problematic for nodes forwarding traffic for other nodes. This section details how synchronization is achieved in IEEE802.15.4e TSCH. It first discusses "single-hop synchronization" (i.e. how a node synchronizes to a neighbor) before highlighting the challenges of keeping a multi-hop network synchronized.

3.1. Single-hop synchronization

For a node to be synchronized to a network, it must learn the value of the ASN of the current slot used by the other nodes in the network, and align the edges of its slot to that of the node it synchronizes to.

A node which joins the network keeps its radio on, listening for an Enhanced Beacon (EB), a type of packet. EBs are periodically sent by nodes already in the network, and contain a field indicating the current ASN number.

As soon as the joining node hears an EB, it reads and stores the ASN field. It then increments its internal ASN value at each new slot.

To align its slot boundaries to that of the node sending the EB, the joining node timestamps the instant it started receiving the EB. This instant is agreed upon by all the nodes in the network. It is defined in the IEEE802.15.4e standard as `TsTxOffset`, a duration after the beginning of the slot. The joining node knows the value of `TsTxOffset` a priori, and therefore retroactively aligns its internal timers so its slot starts exactly `TsTxOffset` before the reception of the EB.

Once a node is part of a network, it must keep synchronized to it. Since its clock drifts relative to its neighbors', it needs to periodically re-synchronize. Each TSCH node is assigned a time source neighbor (how it is chosen is detailed below) to which it must keep synchronized at all times.

There are two ways for a node to re-synchronize to its time source neighbor, both defined in IEEE802.15.4e-2012:

- Every time the node receives a data packet from its time source neighbor, it timestamps the instant it starts receiving it. Knowing `TsTxOffset`, it shifts it slot boundaries to match that of its time source neighbor. This is the same procedure as a joining node uses to initially synchronize to the neighbor after hearing an EB. This is called "frame-based synchronization".
- Every time the node sends a data packet to its time source neighbor, the latter timestamps the instant it started receiving the packet, and indicates that value in a field of the link-layer acknowledgment it sends back. The sending node uses this value to realign its clock to that of its time source neighbor. This is called "ACK-based synchronization".

Fig. 2a and b show the two ways of re-synchronizing. At each re-synchronization, either frame-based or ACK-based, the receiver measures the relative de-synchronization, called `offset`. It is defined as the difference between the measure reception time and `TsTxOffset`, its theoretical value. In frame-based synchronization, the `offset` is applied to the receiver's timing; in ACK-based synchronization, it is applied to the sender's timing.

3.2. Challenges of multi-hop synchronization

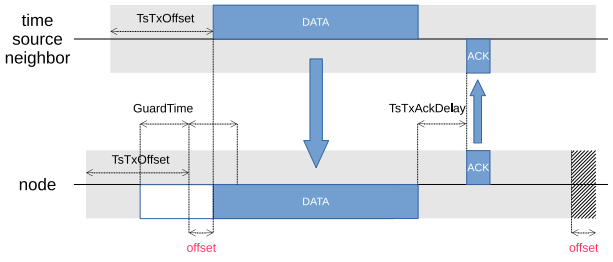
The IEEE802.15.4e standard defines the mechanisms for a node to synchronize to its time source neighbor. It does *not* detail how this time source neighbor is selected, nor how to synchronize a complete multi-hop network. IEEE802.15.4e keeps these elements out of scope, and indicates a "higher layer" is responsible for them.

The IETF 6TiSCH working group [5] was created to define this higher layer. In the architecture of a 6TiSCH network, a single node plays the role of time master; all other nodes synchronize to it, possibly over multiple hops. A 6TiSCH network uses RPL as its routing protocol. RPL organizes the network as a Directed Acyclic Graph (DAG), in which each node has a routing parent. 6TiSCH recommends to reuse the same DAG structure for synchronization: a node's routing parent is also its time source neighbor. A DAG has the advantage of being loop-free (thereby prevented synchronization loops), and since it is already maintained by the RPL protocol, no extra signaling is needed to construct the synchronization structure.

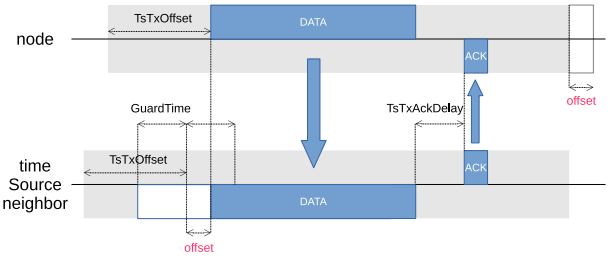
There are subtle challenges to multi-hop synchronization, even with a DAG structure in place. A node needs to periodically synchronize to its time source neighbor to cancel clock drift. Right before synchronizing, a node and its time source neighbor are slightly desynchronized (e.g. by 400 µs); right after synchronizing, they are tightly synchronized (e.g. within 30 µs). In a multi-hop case, the de-synchronization adds up with the number of hops: if the re-synchronization instants of the different nodes at different hop depths are not coordinated, "synchronization swing" may happen, possibly leading to the de-synchronization of nodes deeper in the network.

Fig. 3 illustrates synchronization swing in the canonical case of a linear network of 4 nodes, in which node *A* is the time source neighbor of node *B*, node *B* is the time source neighbor of node *C*, etc. The `GuardTime` in this network is 1 ms, i.e. nodes can never de-synchronize by more than this duration. Let's assume that, at some point in the life of the network, each node synchronizes right *before* its time source neighbor does. It is then possible, as illustrated in the top half of Fig. 3, that each node is desynchronized by 400 us with respect to its time source neighbor. Let's further assume that, following this configuration, nodes synchronize right *after* its time source neighbor, from left to right in Fig. 3. That is, *B* synchronizes to *A*, then *C* synchronizes to *B*. At this point (represented in the lower half of Fig. 3), node *D* and its time source neighbor are de-synchronized by 1200 us. Since this is more than `GuardTime`, node *D* loses synchronization.

One effective way of combating synchronization swing is for a node to learn when its time source neighbor



(a) Frame-based Synchronization.



(b) ACK-based Synchronization.

Fig. 2. The two ways to synchronize defined in IEEE802.15.4e-2012 TSCH.

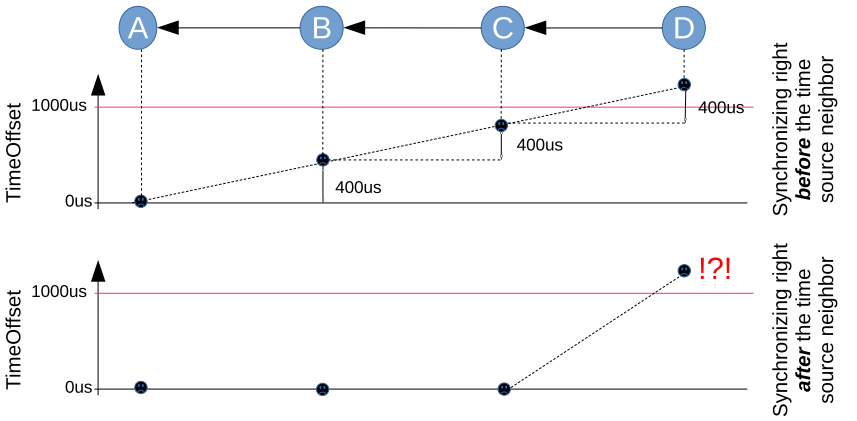


Fig. 3. Multi-hop "Synchronization Swing" can cause nodes to de-synchronize.

synchronized, and synchronize right after it. Thus each nodes' time is related to the time source of whole network (root), which avoids the desynchronization between node and its time source neighbor swings along the time tree and results large desynchronization on deep nodes of network. In this article, we do so by adding a field in EB packet periodically sent by all nodes. Nodes use this information to schedule their re-synchronization instants.

4. Adaptive synchronization

The basic idea of single-hop adaptive synchronization is for a node to measure the drift to its time source neighbor, and compensate for it through software. A node starts by storing the ASN of the slot during which it synchronizes to its neighbor. When it synchronizes again, it measures how much it has de-synchronized to its neighbor, and knows how much time has elapsed since the last synchronization point. It calculates the drift as the ratio between those two values. It expresses this drift as the number of slots it takes for it to drift by a clock tick (e.g. 30 µs when using a 32 kHz clock source). It sets a counter to that value, decrements it every slot, and adjusts its clocks by one tick when the counter reaches zero.

To achieve coordinated multi-hop synchronization, nodes coordinate the instants at which they synchronize, so that a node synchronizes right after its time source neighbor has. They do so by adding a field to all EB and ACK packets, indicating their synchronization period. A node uses that information received from its time source neighbor to schedule when it resynchronizes.

The remainder of this section further details single-hop adaptive synchronization and coordinated multi-hop synchronization.

4.1. Single-hop adaptive synchronization

The adaptive synchronization model was first introduced in [12], and is presented here for completeness.

In (1), t represents time, D_r is the real clock drift and D_c the calculated drift. E_{sync} represent the synchronization error after two neighbor nodes have drifted apart for t without re-synchronizing. Eq. (1) shows that, the better the drift is calculated (i.e. the closer D_c gets to D_r), the slower the synchronization error increases with time.

Eq. (2) indicates how D_c is calculated. When a node resynchronizes to its time source neighbor, it measures their time offset, `offset`. Δt is the amount of time since the last synchronization point. The drift D_c is calculated as the ratio between those two values.

In practice, `offset` is measured by reading the value of a timer. Since this timer is clocked by a clock with a finite frequency (e.g. a 32 kHz crystal), `offset` is necessarily inaccurate. Eq. (3) indicates that the effective drift is bounded by the duration of a clock tick (*tick_duration*, 30 µs when using a 32 kHz crystal), and Δt. That is, to have a smaller effective drift, nodes either need to use a faster clock (which costs extra energy), or increase Δt. The latter is employed by adaptive synchronization, as detailed below.

At each synchronization point, a node uses (2) to calculate D_c, and has to decide when to synchronize next. Its goal is to remain synchronized within `GuardTime` (e.g. 1 ms) to its time source neighbors. It will decide on a required accuracy `RequiredAccuracy` (e.g. 300 µs), smaller than the `GuardTime` by some security factor, and calculate Δt_{next}, the duration until the next synchronization instant. As long as the inequality in (4) is met, under the assumption that the drift rate does not change, the node will never desynchronize by more than `RequiredAccuracy`.

$$E_{sync} = (D_r - D_c) \cdot t \tag{1}$$

$$D_c = \frac{offset}{\Delta t} \tag{2}$$

$$|D_r - D_c| \leqslant \frac{tick\_duration}{\Delta t} \tag{3}$$

$$\begin{aligned} \Delta t_{next} &\leqslant \frac{RequiredAccuracy}{D_c} \\ &\leqslant \frac{RequiredAccuracy}{offset} \cdot \Delta t \end{aligned} \tag{4}$$

This adaptive synchronization model is further detailed in [12].

Fig. 4 shows the behavior of adaptive synchronization, implemented on two GINA [14] nodes. The nodes toggle

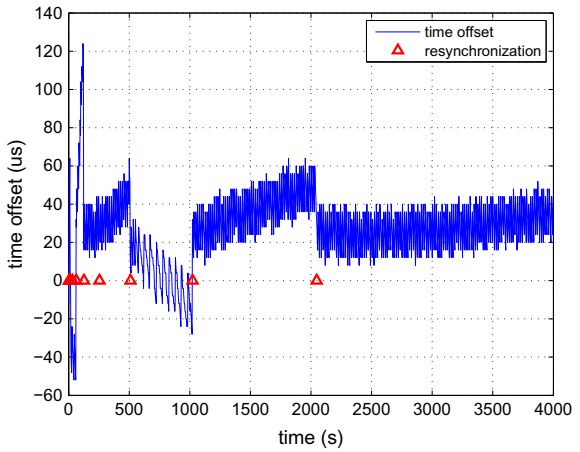


Fig. 4. Adaptive synchronization allows nodes to learn their clock drift, and slow down the rate at which they need to re-synchronize.

a pin on each timeslot edge, a logic analyzer logs this activity. Fig. 4 plots the desynchronization between the two nodes over time, based on those logs. Each triangle represents a synchronization point.

The high-frequency variation of the de-synchronization is due to the periodic adjustment of the timers; the "width" of the resulting signal is 30 µs, corresponding to one 32 kHz clock tick. After each synchronization point, the nodes slowly drift apart (indicated by the slope of the average signal). At the beginning, the measured synchronization is large, requiring for the nodes to re-synchronized often. As the synchronization period increases, the drift calculation gets more accurate, and the slope reduces. In the second half of the plot, although the nodes do not resynchronize for over 30 min, they stay synchronized within 50 µs.

Fig. 4 is obtained by running the nodes in a temperature-controlled environment, where the drift can be considered constant [15]. If temperature is not constant, once can use an on-board temperature sensor to detect temperature swings, and trigger early resynchronization. In our experiments (detailed in Section 5), we cap the resynchronization period at 5 min.

4.2. Coordinated multi-hop synchronization

Section 3.2 discusses the subtle challenges related to multi-hop synchronization, how "synchronization swing" can cause nodes to desynchronize, and how coordinating the instants at which nodes synchronize combats this phenomenon.

The key to "coordinated" multi-hop synchronization is to let a node learn approximately when its time source neighbor synchronizes, and synchronize to it right after. We therefore add a 2-byte data field to all EB and ACK packets, indicating the resynchronization period, in seconds, of the transmitter.

Fig. 5 illustrates coordinated multi-hop synchronization a canonical 3-node linear network. Here, node *B* synchronizes to node *A*, and per (4) decides to resynchronize after

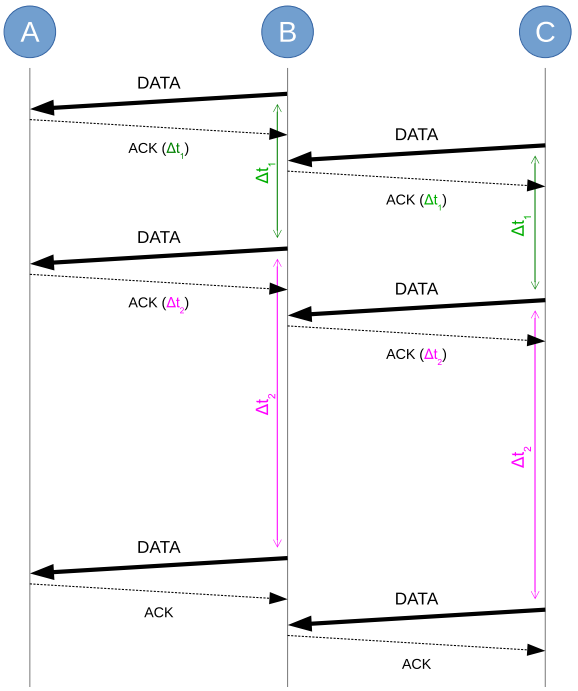


Fig. 5. Coordinated multi-hop synchronization allows a node to learn its time source neighbor's synchronization period and synchronize right after its time source neighbor.

Δt_1. When node *C* synchronizes to *B*, node *B* indicates Δt_1 as part of its ACK message. Node *C* schedules the time it will resynchronize accordingly. The next time *B* synchronizes to *A*, it increments its resynchronization period to Δt_2. Node *C* follows after resynchronizing, modifying its resynchronization instants lockstep.

When a new node joins the network, it needs to learn the synchronization period of its time source neighbor, but also *when* this time source neighbor has just synchronized. We therefore add a `isAccurate` flag to all ACK and EB packets, which is set to true up to 10 s after a node has synchronized to its time source neighbors.

Fig. 6 illustrates the use of the `isAccurate` flag. The newly joined node *C* sends frequent synchronization messages to node *B*. Node *B*, after having synchronize to *A*, sets the `isAccurate` flag in all ACK packets for 10 s. After receiving an ACK with the `isAccurate` flag set, *C* initiates adaptive synchronization, increasing it synchronization period until it is lockstep with node *B*.

The `isAccurate` flag is also used to deal with the situation where a node synchronizes late. This is illustrated in Fig. 7. Node *C* is lockstep with node *B*. Because of a low packet delivery ratio, it takes *B* multiple tries to synchronize to *A*. As a result, node *C*, on the scheduled resynchronization instant, does not receive an ACK with the `isAccurate` flag set. Node *C* therefore keeps resynchronizing frequently to *B*, until *B* synchronizes to *A* and *C* receives an ACK with the `isAccurate` flag set. At that point, nodes *B* and *C* are lockstep again, and the system has recovered.

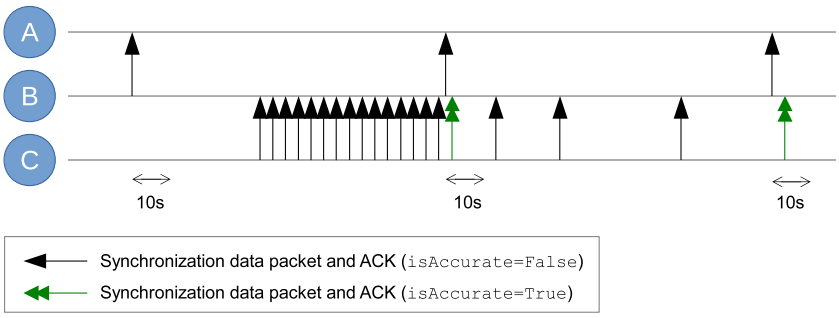


Fig. 6. Newly joined *C* keeps frequently synchronizing to *B* until it hears an ACK with the `isAccurate` flag set.

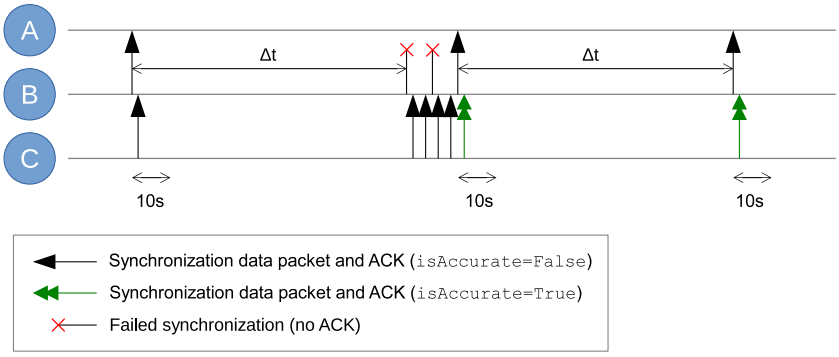


Fig. 7. The `isAccurate` flags is used to recover from a situation where a node takes longer to synchronize than anticipated.

5. Implementation and results

The performance of the adaptive synchronization techniques presented in Section 4 are extracted through simulation and experimentation. Section 5.1 described OpenWSN, the environment used to conduct these experiments, as well as the parameters used. Sections 5.2 and 5.3 present the simulation and experimental results, respectively, before discussing the importance of adaptive synchronization for the development of an inter-operable IoT.

5.1. OpenWSN parameters

UC Berkeley's OpenWSN project[1] is an implementation of an entirely standards-based protocol stack for the IoT, and the *de facto* open-source implementation of IEEE802.15.4e TSCH. It has been ported to 10 different 16-bit and 32-bit platforms. In addition to running on real hardware, the source code can run in a simulation environment called OpenSim. Having the same code running in the simulator augments the confidence that a real deployment behaves like its simulation. The architecture and design choices of OpenWSN are detailed in [16].

OpenSim is an discrete-event simulator which accurately models the timing of the nodes, a characteristic needed to evaluate adaptive synchronization. The simulated node contains a model of the 32 kHz crystal driving its timers. Each node can independently be given a different drift. That is, the frequency of one node's crystal can be 32768.3 Hz while its neighbor's can be 32767.7 Hz, resulting in a 20 ppm relative drift between them.

The IEEE802.15.4e-2012 TSCH implementation of OpenWSN uses the hard-coded communication schedule standardized in [17], with a slotframe of 11 timeslots long, including a timeslot for sending EBs and 5 shared slots for data. Each node is configured to send an EB every 10 s. Synchronization to EBs is disabled, so all synchronization is ACK-based, and relies on a node sending a data packet to its time source neighbor and reading its offset from the ACK. Initially, a node is configured to synchronize to its time source neighbor every 1 s; adaptive synchronization is used to increase the effective synchronization period, but no more than the maximum period of 5 min. Adaptive synchronization is set up to ensure that nodes always stay synchronized within the required accuracy of 120 µs (`RequiredAccuracy` in (4)). To measure only the performance of adaptive synchronization, no application-level data is produced in the network. Per the recommendation of the IETF 6TiSCH Working Group [5], the RPL routing parent of a node is also its time source neighbor.

Since the same code runs in OpenSim and on the real hardware, the parameters listed above apply to both the simulation (Section 5.2) and experimental results (Section 5.3).

5.2. Simulation results

Simulation results were obtained on the 3-hop deep 13-node network depicted in Fig. 8, with 4 nodes at each hop

[1] http://www.openwsn.org/.

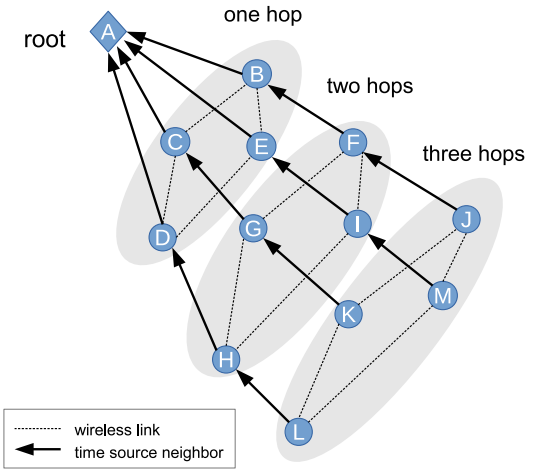


Fig. 8. Topology used for simulation.

depth. The drift of each node is randomly selected in the $[-30ppm \ldots +30ppm]$ range.

Throughout the lifetime of the network, nodes keep synchronizing to their time source neighbors. At each synchronization, a node measures the `offset` to its time source neighbor, which quantifies their de-synchronization. OpenSim is instrumented to log those values at each resynchronization. Fig. 9 shows the `offset` of all the nodes, averaged over a 5 min sliding window. It gives a good idea of the overall performance of adaptive synchronization, and indicates that adaptive synchronization maintains the de-synchronization between neighbor nodes below 2.5 32 kHz ticks, or 76 µs.

As nodes drift and the timestamps are only as accurate as the clock, synchronization errors remain. Fig. 10 shows the average, minimal, and maximal values of the `offset`, as a function of the number of hops to the root. Per Section 3.2, the maximum time offset should be lower than the required accuracy multiplied by the number of hops. This is verified in Fig. 10; at 3 hops for example, all measured `offset` as within $[-244\,\mu s \ldots +305\,\mu s]$, which is within the theoretical $[-366\,\mu s \ldots +366\,\mu s]$.

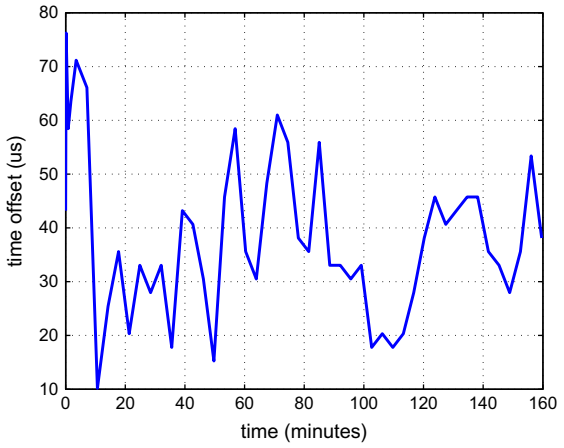


Fig. 9. `offset` measured by all nodes in the network, averaged over a 5 min sliding window, in 32 kHz clock ticks. Results obtained by simulation.

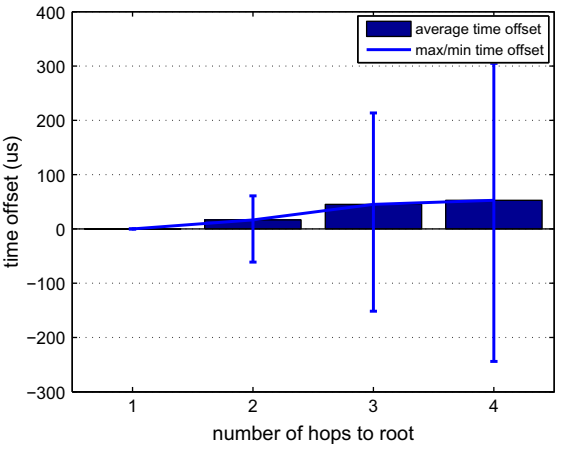


Fig. 10. The `offset` as a function of hop count. Results obtained by simulation.

The goal of adaptive synchronization is for a node to resynchronize less often to conserve energy. A key measurement is hence the number of resynchronization messages. If no adaptive synchronization were used, a node with a 30 ppm drift would require 109 packets per hour to stay synchronized. During the 160 min simulated time, the 12 nodes in the network (not counting the root, which does not resynchronize) performed 604 resynchronizations, which translated in an average of 18.9 packets per hour, for each node. This is a 83% decrease compared to a network which does not use adaptive synchronization.

5.3. Experimental results and interoperability

The purpose of the experimentation is twofold: verify the correct behavior of the implementation on real hardware, and demonstrate how adaptive synchronization enables inter-operation.

The behavior of the adaptive synchronization implementation is verified on the setup depicted in Fig. 11. This setup is composed of four OpenMoteSTM motes connected to a logic analyzer. The boards are programmed to toggle a pin on each slot edge, and when they resynchronize; the logic analyzer logs that activity, and an Matlab analysis script tracks the offset between neighbor devices. The neighbor table is hard-coded on each board to force the linear topology. The drift of node B, C and D refer to node A are 4 ppm, 13 ppm and 18.5 ppm respectively.

Fig. 12 shows the offset of nodes *B*, *C* and *D* relative to their time source neighbor. No graph is shown for node *A* since it is the root and therefore does not resynchronize.

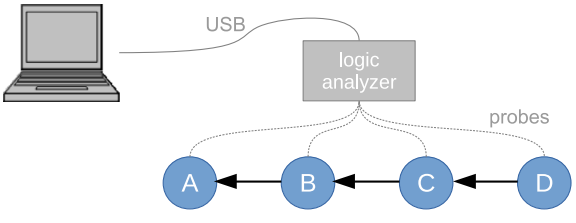


Fig. 11. Experimental setup.

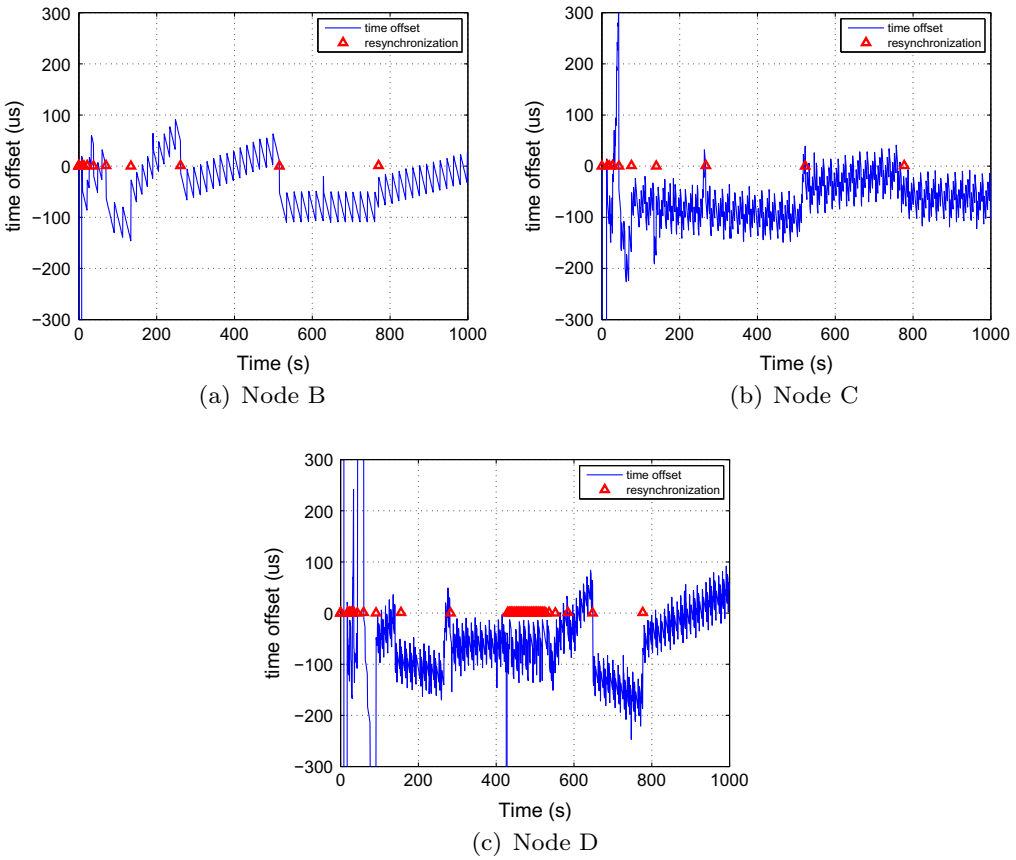


Fig. 12. Experimental time error for nodes relative to time source neighbor in the experimental setup depicted in Fig. 11. At $t = 410$ s, node D is reset. Results obtained experimentally.

Fig. 12a and b shows how the resynchronization period increases to the maximum value of 5 min. Besides, it also shows that the time offset is controlled around 100 µs. To show the behavior when a node (re-) joins the network (see Section 4.2), node D is reset at $t = 410$ s. Fig. 12c shows how it synchronizes frequently until C re-synchronizes at $t = 510$ s, after which it starts adaptive synchronization. Node D get little larger time offset during 510 to 790 since its time source neighbor, node C's time is not accurate during that period.

Fig. 13 shows the offset of nodes B, C and D relative to the root of network, node A. Since node A is also the time source neighbor of node B, Fig. 13a is same with Fig. 12a. The time offset of node C and D is controlled around 200 µs and 300 µs respectively. The time offset of node will increase 100 µs each hop deeper. So a 9 hops network can be deployed with `GuardTime` of 1 ms based on the experiment setting.

With the emergence of the IoT, networks will be more and more often composed of multi-vendor devices. Users will favor standards-based solutions. Interoperability is at the heart of standardization activities at the IEEE and IETF. The recently formed IETF 6TiSCH Working Group [5] works for example on standardizing the use of IPv6 over IEEE802.15.4e TSCH for IoT application requiring ultra high reliability and low-power operation. Such standards allow the devices to inter-operate by agreeing on packets formats and behaviors.

Even so, differences between micro-controllers and crystals cause devices from different vendors to exhibit slightly different timing characteristics. Different devices might use different crystals, each having different drift and beating a slightly different frequency. Adaptive synchronization is a key enabler for this type of interoperability. Without it, nodes would not learn the drift to their time source neighbor, and would have to resynchronize every handful of seconds, thereby increasing their power consumption beyond what is acceptable by most applications.

To demonstrate interoperability between different platforms, we use the basic setup from Fig. 11, but with the four hardware devices shown in Fig. 14. These are representative of the variety of devices available: they are of different generations (TelosB dates back to 2004 while the OpenMoteCC2538 was designed in 2013), with different micro-controller architecture (16-bit for TelosB and GINA, 32-bit for OpenMoteSTM and OpenMoteCC2538), different board requirements (OpenMoteCC2538 use a single-chip solution, while OpenMoteSTM, TelosB and GINA use a separate radio and micro-controller), and different crystals.

One immediate effect of these differences is that, even after carefully tuning, the duration of the TSCH timeslots are slightly different, as shown in Table 1. For example,

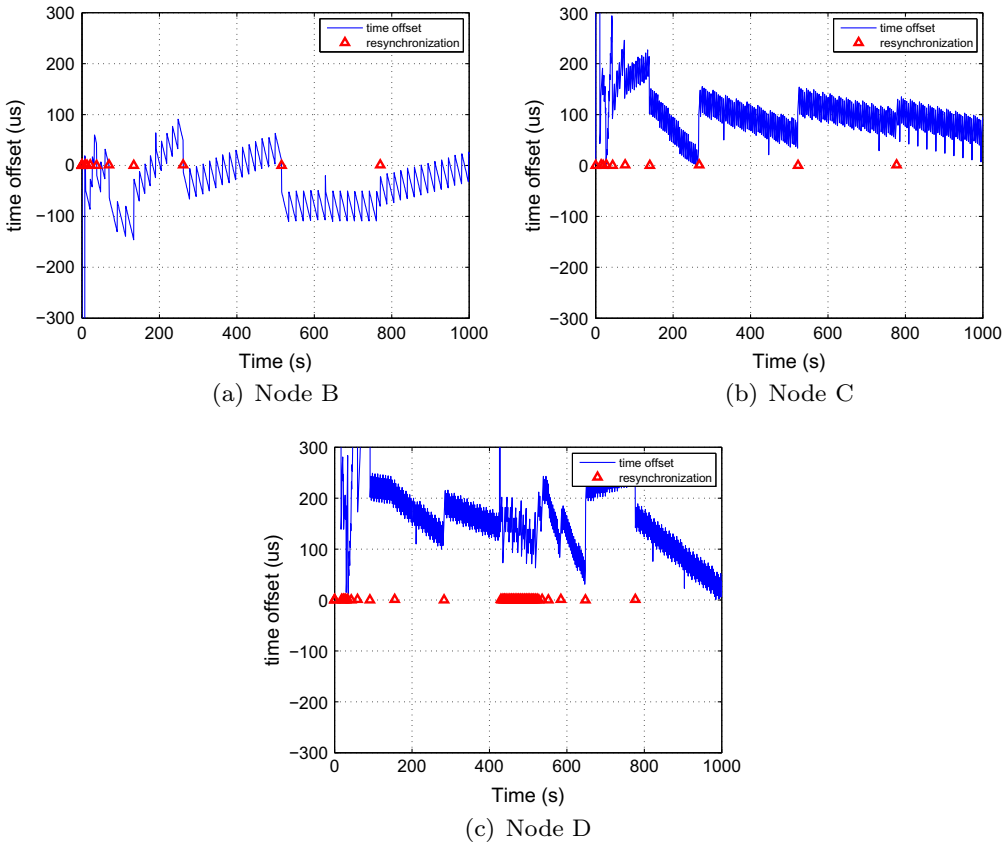


Fig. 13. Experimental time error for nodes relative to root in the experimental setup depicted in Fig. 11. Results obtained experimentally.

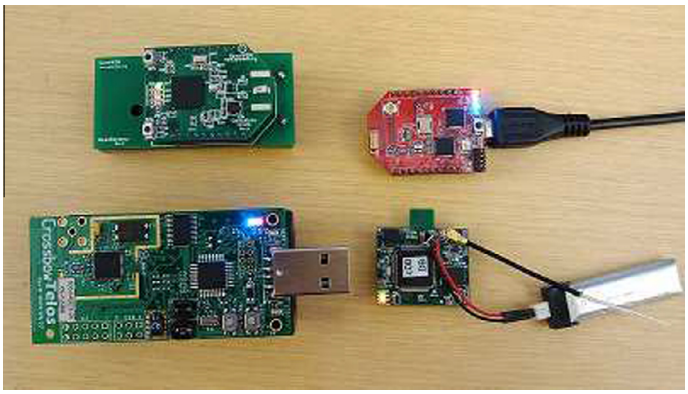


Fig. 14. Adaptive synchronization enables interoperability by building networks with different platforms. Here the OpenMoteCC2538 (top-left), OpenMoteSTM (top-right), TelosB (bottom-left) and GINA (bottom-right) participate in the same TSCH network. All four platforms run OpenWSN.

the difference in slot length between the GINA and OpenMoteCC2538 devices is equivalent to a 667 ppm drift. In this case, a TSCH network could not function without adaptive synchronization. Instead, with adaptive synchronization, nodes measure and learn their relative drift. In this experiment the devices are running the minimal 6tisch draft[2] implementation plus adaptive synchronization. Though they may form a one-hop network because of all devices can hear the packet sent from root, the experiment we shown above has already proved the correctness of adaptive synchronization in multi-hop network.

The IETF 6TiSCH working group organized a plugfest at the IETF89 meeting.[3] During this event, the adaptive synchronization OpenWSN implementation presented in this

[2] 6TISCH minimal draft, https://ietf.org/doc/draft-ietf-6tisch-minimal/.

[3] 6TiSCH plugfest, IETF89, March 6 2014, London, UK.

Table 1
Four hardware with different micro-controller have different feature on timing.

Device	Micro-controller	Slot duration (μs)
GINA	MSP430F2618	15,000
TelosB	MSP430F1611	15,000
OpenMoteSTM	STM32F103	15,004
OpenMoteCC2538	CC2538	15,010

paper was used by 5 different research teams, running on 7 different hardware platforms, all able to inter-operate despite their slight timing differences. Adaptive synchronization makes interoperability between TSCH devices a reality. We believe that it essential to the development of 6TiSCH networks, and its widespread use in the IoT.

6. Conclusion

A TSCH network is fully synchronized, and nodes need to periodically resynchronize to one another to account for clock drift. This is done by exchanging packets, so the less often nodes need to resynchronize, the less energy nodes consume. This article presents an adaptive synchronization technique composed of two part. Through "single-hop" adaptive synchronization, nodes learn and cancel the drift to their neighbor. Through "coordinated" multi-hop adaptive synchronization, a node synchronize to its time source neighbors when that node is most tightly synchronized to the network, thereby combating "synchronization swing".

Adaptive synchronization was implemented in the OpenWSN project. Simulation results show how the nodes in the network are on average synchronized within 76 μs, while requiring only 18.9 synchronization packets per hour, a 83% decrease compared to a network with no adaptive synchronization.

Apart from lowering the energy budget dedicated to synchronization, adaptive synchronization enables TSCH networks to be composed of different hardware platforms. This level of interoperability will be a cornerstone to the development of the IoT for applications which require ultra-high reliability and low-power, as targeted by the IETF 6TiSCH working Group.

From a protocol point of view, adaptive synchronization only requires a 2-byte field to be added to particular IEEE802l.15.4e packets. The results of this work will be proposed to the IETF 6TiSCH working Group for possible standardization.

Acknowledgments

Tengfei Chang's stay at the University of California, Berkeley is funded by the "Short-Term Visiting Project" of the Univ. of Sci. and Tech. in Beijing, China.

References

[1] WirelessHART Specification 75: TDMA Data-Link Layer, HART Communication Foundation Std., Rev. 1.1, 2008, hCF_SPEC-75.
[2] ISA-100.11a-2011: Wireless Systems for Industrial Automation: Process Control and Related Applications, International Society of Automation (ISA) Std., May 2011.
[3] 802.15.4e-2012: IEEE Standard for Local and Metropolitan Area Networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC Sublayer, IEEE Std., 16 April 2012.
[4] 802.15.4-2011: IEEE Standard for Local and Metropolitan Area Networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs), IEEE Std., 5 September 2011.
[5] P. Thubert, T. Watteyne, M.R. Palattella, X. Vilajosana, Q. Wang, IETF 6TSCH: combining IPv6 connectivity with industrial performance, in: International Workshop on Extending Seamlessly to the Internet of Things (esIoT), Taiwan, 3–5 July 2013.
[6] T. Watteyne, L. Doherty, J. Simon, K. Pister, Technical overview of SmartMesh IP, in: International Workshop on Extending Seamlessly to the Internet of Things (esIoT), Taiwan, 3–5 July 2013.
[7] B. Sundararaman, U. Buy, A. Kshemkalyani, Clock synchronization for wireless sensor networks: a survey, Elsevier Ad Hoc Network. 3 (3) (2005) 281–323.
[8] T. Schmid, R. Shea, Z. Charbiwala, J. Friedman, M.B. Srivastava, On the interaction of clocks, power, and synchronization in duty-cycled embedded sensor nodes, ACM Trans. Sensor Networks (TOSN) 7 (3) (2010).
[9] S. Ganeriwal, I. Tsigkogiannis, H. Shim, V. Tsiatsis, M.B. Srivastava, D. Ganesan, Estimating clock uncertainty for efficient duty-cycling in sensor networks, IEEE Trans. Network. 17 (3) (2009) 843–856.
[10] M. Maroti, B. Kusy, G. Simon, A. Ledeczi, The flooding time synchronization protocol, in: international Conference on Embedded Networked Sensor Systems (SenSys), 2004.
[11] D. Stanislowski, X. Vilajosana, Q. Wang, T. Watteyne, K. Pister, Adaptive synchronization in IEEE802.15.4e networks, IEEE Trans. Indust. Inform. 9 (2) (2013) 600–608.
[12] T. Chang, Q. Wang, Compensation for time-slotted synchronization in wireless sensor network, Int. J. Distribut. Sensor Networks 7 (2014).
[13] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, R. Alexander, RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, IETF Std. RFC6550, March 2012.
[14] A. Mehta, K. Pister, WARPWING: a complete open source control platform for miniature robots, in: Intelligent Robots and Systems (IROS), Taipei, Taiwan, IEEE, 18–22 October 2010.
[15] J.R. Vig, Introduction to Quartz Frequency Standards, Army Research Laboratory, Electronics and Power Sources Directorate, Tech. Rep. SLCET-TR-92-1 (Rev. 1), October 1992.
[16] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, K. Pister, OpenWSN: a standards-based low-power wireless development environment, Trans. Emerging Telecommun. Technol. 23 (5) (2012) 480–493.
[17] X. Vilajosana, K. Pister, Minimal 6TiSCH Configuration, 6TiSCH Working Group Std. Draft-ietf-6tisch-minimal-03, October 26, 2014.

Tengfei Chang is a candidate Ph.D student of School of Computer and Communication, University of Science & Technology Beijing (USTB), China. He received BS degree in Computer Science and Technology from Central South University of Forestry and Technology (CSUFT) in 2010. He focuses on the research of wireless sensor networks and embedded systems. Since January 2012, he has been collaborating with the OpenWSN project. During January 2014 to May 2014, he had been working on OpenWSN project in UC Berkeley as a visiting student researcher.

Thomas Watteyne is a Senior Networking Design Engineer at Linear Technology, in the Dust Networks Product Group, which specializes in ultralow power and highly reliable Wireless Sensor Networking. He designs networking solutions based on a variety of IoT standards and promotes the use of highly reliable standards such as IEEE802.15.4e. He is co-chairing the new IETF 6TiSCH group, which aims at standardizing how to use IEEE802.15.4e TSCH in IPv6-enabled mesh networks. Prior to Dust Network, he was a Postdoctoral Researcher at the University of California, Berkeley, working with Prof. K. Pister. He started Berkeley's OpenWSN project, an open-source initiative to promote the use of fully standards-based protocol

stacks in M2M applications. He received the Ph.D. degree in computer science (2008) and the M.Sc. degree in telecommunications (2005) from INSA Lyon, France.

Kristofer S.J. Pister received his B.A. in Applied Physics from UCSD in 1986, and his M.S. and Ph.D. in Electrical Engineering from UC Berkeley in 1989 and 1992. From 1992 to 1997 he was an Assistant Professor of Electrical Engineering at UCLA where he developed the graduate MEMS curriculum, and coined the phrase Smart Dust. Since 1996 he has been a professor of Electrical Engineering and Computer Sciences at UC Berkeley. In 2003 and 2004 he was on leave from UCB as CEO and then CTO of Dust Networks, a company he founded to commercialize wireless sensor networks. He participated in the creation of several wireless sensor networking standards, including Wireless HART (IEC62591), IEEE 802.15.4e, ISA100.11A, and IETF RPL. He has participated in many government science and technology programs, including DARPA ISAT and the Defense Science Study Group, and he is currently a member of the Jasons. His research interests include MEMS, micro robotics, and low power circuits.

Qin Wang is a Professor with the School Computer and Communication, University of Science and Technology Beijing (USTB), China. She received the B.S., M.S., and Ph.D. degrees in computer science and engineering from USTB in 1982, Peking University in 1985, and USTB in 1998, respectively. She joined USTB in 1985, became Full Professor in 2000. As a Visiting Scientist (2005–2006) in the Electrical Engineering and Computer Science Department, Cornell University, NY, and Visiting Researcher (2006–2007) in the Electrical Engineering and Computer Science Department, Harvard University, Cambridge, MA, her research and contributions were on wireless sensor network technology and related power consumption modeling from both device and network system perspective. Recent years, she has focused on low power wireless sensor networks and MPSoC (multiprocessor System-on-Chip) technology in communications and networking systems. Since January 2012, she has been working on OpenWSN project in UC Berkeley as a visiting professor. She has been involved in international wireless network standard development since 2007, including ISA100.11a, IEEE 802.15.4e, and industrial wireless standard WIA-PA proposed to IEC by China.