Industrial Wireless IP-based Cyber Physical Systems
Thomas Watteyne, Vlado Handziski, Xavier Vilajosana, Simon Duquennoy, Oliver Hahm, Emmanuel Baccelli, Adam Wolisz

To cite this version:
Thomas Watteyne, Vlado Handziski, Xavier Vilajosana, Simon Duquennoy, Oliver Hahm, et al.. Industrial Wireless IP-based Cyber Physical Systems. Proceedings of the IEEE, Institute of Electrical and Electronics Engineers, 2016, 104 (5), pp.1025-1038. <10.1109/JPROC.2015.2509186>. <hal-01282597>

HAL Id: hal-01282597
https://hal.inria.fr/hal-01282597
Submitted on 20 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Industrial Wireless IP-based Cyber Physical Systems

Thomas Watteyne, Member, IEEE, Vlado Handziski, Member, IEEE, Xavier Vilajosana, Member, IEEE, Simon Duquennoy, Oliver Hahm, Emmanuel Baccelli, Adam Wolisz, Senior Member, IEEE

Abstract—Industrial control systems have traditionally been built around dedicated wired solutions. The requirements of flexibility, mobility and cost have created a strong push towards wireless solutions, preferably solutions requiring low power. Simultaneously, the increased need for interoperability and integration with the wider Internet made a transition to IP-based communication unavoidable. Following these trends, we survey 6TISCH, the emerging family of standards for IP-based industrial communication over low-power and lossy networks. We describe the state of the standardization work, the major issues being discussed, and open questions recently identified. Based on extensive first-hand experience, we discuss challenges in implementation of this new wave of standards. Lessons learned are highlighted from four popular open-source implementations of these standards: OpenWSN, Contiki, RIOT and TinyOS. We outline major requirements, present insights from early interoperability testing and performance evaluations, and provide guidelines for chip manufacturers and implementers.

Index Terms—6LoWPAN, 6TiSCH, Communication standards, IEEE802.15.4e, Industrial, Protocols, Wireless.

I. INTRODUCTION

Industrial networks have developed alongside the traditional Internet. This is because an industrial network—“Operational Technology (OT)”—has requirements very different from the open Internet—an “Information Technology (IT)”. While the Internet is built to interconnect billions of heterogeneous devices communicating globally large amounts of data, an industrial network is typically deployed within a factory floor, typically connecting 100’s or 1,000’s of devices. And although the amount of data in typical industrial process applications may not be large, what is critical is reliability (all data is received by its final destination), and latency (using guaranteed time bounds, as opposed to best-effort). These requirements have been traditionally met by specifically chosen wired solutions [1].

The cost of wiring is high, to the point of being prohibitive in many cases. Industrial settings such as steel mills, oil refineries, chemical industries, power plants, infrastructures implement complex monitoring and management processes. Hundreds or thousands of devices report sensed values such as temperature, pressure, traffic flows or tank fill level used to both control actuators and coordinate production stages. Planning and installation of the cables are challenging and thus expensive: explosive environments and hot surfaces have to be avoided (e.g. in a refinery). And mobile objects can hardly be connected at all.

Wireless technology is tremendously appealing for industrial applications, as it reduces installation cost dramatically. The desire for such solutions has been expressed very early [2], but meeting the strict requirements of wire-like reliability and maintenance-free operation over several years has proven challenging for battery-operated wireless devices interconnected by unreliable links. Also, the popular wireless link layers based on Carrier Sense Multiple Access (CSMA) did not offer the required hard-time guarantees.

Sensor Networks Change the Game. An initial break-through towards sensing application came in 1997 with Pister et al., from the UC Berkeley Smart Dust project, who brought the vision of miniaturized (down to 1 mm$^3$) battery-powered motes able to sense, compute and wirelessly communicate in a mesh topology. This idea inspired a wave of research in sensor networks: hardware nodes (widely called motes), operating systems (e.g. TinyOS, Contiki), and protocols, also following up on seminal work on packet radio networks in early 70’s [3] (which had first introduced multi-hop mesh network that uses dynamically the diversity of paths to increase the reliability of wireless connectivity). Most importantly, a wide range of practical experiments were carried out, and the industry followed, with the IEEE802.15.4 communication standard and ZigBee protocols. While these developments have addressed the needs of home automation, agriculture, etc., industrial applications using these and other wireless technologies still remained a challenge [4].

A second break-through then came with the invention of the Time Synchronized Mesh Protocol (TSMP) [5] in 2006. At the
core of a TSMP network lies the Time Synchronized Channel Hopping (TSCH) technology: nodes tightly synchronize to reduce energy consumption, and exploit frequency diversity (in addition to spatial diversity) to obtain near wire-like 99.999\% reliability (as reported in [6], after running a 44-node network for 26 days and observing 2 millions of packets out of which only 7 have been lost).

The proof of concept and early commercialization by the spin-off from the inventor's lab—Dust Networks—made it possible to turn the TSCH approach to an industry standard, first as part of WirelessHART [7] (supported by the leading industrial communication consortia: HART, Fieldbus Foundation and Proflbus Organization [8]) and later as part of IEEE802.15.4e [9]. In parallel, numerous technical and commercial success stories highlight the performance of TSCH-based networks.

The need for a low-power wireless network to offer reliable communication has been highlighted for example in [10] which reports on a Smart Factory application where pressure, water flow, and level sensors are networked together using a TSCH network to monitor a pharmaceutical plant in Ireland. A large-scale smart city deployment of wireless sensors which monitor the occupancy of individual parking spots [11] demonstrated the need for scalability in TSCH networks. A deployment of a complex structural monitoring system, providing real-time information of the health of a landmark indoor arena in Barcelona, pushed the throughput limits of the TSCH technology [12]. TSCH-based networks of temperature sensors has allowed building a model of the temperature of a datacenter used for dynamic control of the air conditioning [13]. For more information on the usage of this and other sensor networks technologies in the industrial environment see [14], while the security aspects are specifically discussed in [15].

**IT & OT Converge.** Over a long time, sensor networks—in all areas of applications—have been following proprietary protocol stacks. This has been, among others, the case of ZigBee and WirelessHART. Gradually, however, it becomes clear that there is a need for natural, seamless interconnection to the worldwide IT infrastructure (the vision of the Internet of Things—IoT), see [16]. The IETF has recognized this need, and has introduced a set of standards: 6LoWPAN [17], [18] (an adaptation layer which compacts long IPv6 headers so they fit in short frames typical for sensor networks—like IEEE802.15.4 frames), RPL [19] (a routing protocol) and CoAP [20] (an application-layer protocol allowing low-power devices to appear as web servers).

The specific features of TSCH, notably the necessity of harmonization of the time schedules with the routing into a useful protocol stack for industrial applications called, however, for more attention. The first commercial solution SmartMesh IP [21] has additionally motivated the full-scope standardization being currently carried out in the IETF 6TiSCH [22] working group.

Following IETF best practice, standardization is paralleled with implementations by several independent groups, based on various operating systems for low-end IoT devices [23]. In this paper, we will look in particular at implementations in OpenWSN [24], Contiki [25], RIOT [26] and TinyOS [27]. The implementations, as well as interoperability tests among those independent developments, aim to verify the feasibility of efficient implementation and completeness of the protocols’ specifications. In this paper, we closely analyze those projects, and extract the hardware requirements, implementation painpoints and lessons learned from implementing a full combined TSCH/IPv6 based protocol stack for industrial applications.

The remainder of this paper is organized as follows. The details of this protocol stack (Section II) are followed by a discussion detailing identified implementation challenges (Section III). Section IV is devoted to presenting interoperability testing approaches, as well as test and performance measurement results. Finally, Section V concludes this article.

## II. An IP-Based Protocol Stack for Industrial Low-Power Wireless

### A. Overview of TSCH

Time Synchronized Channel Hopping (TSCH) is a MAC protocol which divides time into slices of fixed length that are grouped in a slotframe. Nodes are synchronized and share the notion of a slotframe which repeats over time. Frequency diversity is used to mitigate effects of multipath fading and to improve robustness against external interference. Channel hopping is achieved by sending successive packets on different frequencies. The channel hopping sequence is fixed and known by all nodes. In a particular cell (a timeslot at a particular frequency), a node may transmit, listen or sleep.

The scheduler builds the communication schedule (i.e. allocates communication cells in the slotframe to different pairs of communicating nodes) in order to satisfy the bandwidth, latency and reliability requirements of the applications. The scheduler must keep the number of schedule cells (in which a mote either transmits or listens) to a minimum in order not to waste energy.

IEEE802.15.4-2011 [28]—the version of the IEEE802.15.4 standard before TSCH was introduced—required the radios of relay nodes to be always on. This is because, without time synchronization, a mote cannot know when its neighbor is going to send a frame. This means that motes equipped with a typical IEEE802.15.4-compliant radio (drawing 10mA when receiving) have a battery lifetime of about a week (assuming typical 2200mAh AA batteries). TSCH, through synchronization, allows a mote to keep its radio off over 99% of time, extending its lifetime to years [29]. This is done without requiring any hardware changes, only through smart management of the radio chip.

Similarly, a network using IEEE802.15.4-2011 (not TSCH) operates on a single frequency. Single channel operation significantly impacts the reliability of communication. Channel hopping (used in TSCH) provides reliability by exploiting channel diversity to combat external interference and multipath fading [30], [31].

Yet, a missing piece in IEEE802.15.4e TSCH is the network scheduling component, part of the network management plane. This scheduler can be a centralized computer (called Path Computation Element—PCE) or a distributed protocol. A central scheduler gathers information about the network (including topology and application requirements) and computes a near-optimal schedule.
a multi-hop routing structure into the network, so a network can be deployed over an extended area without requiring each node to be within radio range of the gateway device. Finally, CoAP [20] turns every low-power wireless device into a (tiny) web server with which computers on the Internet can interact, much like they interact with traditional web servers.

Because it is IPv6-ready, a network running the 6TiSCH stack easily connects to the Internet. This is done through a generic (application-independent) gateway device which is responsible for 6LoWPAN compaction/inflation. That is, when an IPv6 packet goes from the Internet into the low-power wireless mesh, the gateway compacts the IPv6 header into its equivalent (shorter) 6LoWPAN header. Similarly, the gateway inflates an 6LoWPAN header back into an IPv6 header when a packets leaves the low-power mesh.

The 6top sublayer (Fig. 2) is the standardization gap the IETF 6TiSCH working group is filling. IEEE802.15.4e standardizes how to execute a TSCH communication schedule, it does not define how to build and maintain it. And neither does 6LoWPAN, which assumes a connected topology. 6TiSCH defines the architecture and components needed to build the schedule, optimize the operation of the routing protocol and secure communication in the network. 6TiSCH is the final gap to connect operation technologies enabled by TSCH to the information technologies in the Internet.

The 6TiSCH working group was created in October 2013 and has grown to over 300 subscribers. With over 100 face-to-face and virtual meetings since its creation, it benefits from a very active community.

### III. Design and Implementation Challenges

This sections presents an overview of existing open source implementations of the 6TiSCH stack, and discusses related design and implementation challenges. Our focus on open-source software enables us to compare various technical aspects and implementation choices on the stack’s internals. We select the implementations from the following four operating systems: OpenWSN, Contiki, RIOT, and TinyOS, presented in Table I. After a brief overview of the OS’ specifics, we review a number of design challenges, and discuss how existing implementations address them.
A. Open-Source CPS Implementations

OpenWSN. The OpenWSN project [24] is an open-source implementation of a fully standards-based protocol stack rooted in the IEEE802.15.4e TSCH link layer, also featuring the IPv6-enabled IETF upper stack (6LoWPAN, RPL, CoAP). OpenWSN is the de-facto reference implementation of IEEE802.15.4e Time Synchronized Channel Hopping and other standards related to the 6TiSCH working group. The implementation has been ported to 11 popular hardware platforms, from low-end 8-bit micro-controllers to state-of-the-art 32-bit single-chip solutions. OpenMote (http://www.openmote.com/), a spin-off company of the OpenWSN project, is commercializing the hugely popular OpenMote, an open-hardware modular device which is quickly becoming the de-facto low-power wireless hardware platform. The OpenWSN project includes OpenSim, an emulation platform which allows developers to run their firmware on a regular computer, for easier debugging and quicker development. OpenWSN closely tracks the standardization activities at IETF working groups such as 6TiSCH, and has been a catalyst for research around TSCH-related topics on synchronization, power consumption, and security. While the project was started in Prof. Pister’s lab at UC Berkeley in 2010, it has grown to an extremely active open-source community, and has received input from over 35 academic and industrial contributors. The OpenWSN source code is released under a BSD license and available on GitHub (https://github.com/openwsn-berkeley).

Contiki. The Contiki OS was created in 2001, with a focus on Internet connectivity for constrained devices, i.e. $\sim kB$ of RAM, tens of $kB$ or $ROM$. The project, available on GitHub (https://github.com/contiki-os/contiki) has a large community in both academia and industry; it has contributors from top universities such as ETH Zurich and Oxford University, and from leading IT companies such as Cisco, Atmel, or ST Microelectronics. Its main article [25] was cited over 1600 times (Google Scholar, May 2015). Contiki has a lightweight event-driven kernel, and uses protothreads, a programming abstraction that compiles sequential source code into events. It is currently supported by tens of different hardware platforms, typically sensor/actuator and IoT nodes. Since 2008, Contiki features a fully certified IPv6 stack. It has been regularly updated ever since with early implementation of IETF protocols for low-power IPv6 connectivity in the IoT, including 6LoWPAN, RPL, or CoAP. In 2014, a first implementation of TSCH able to run in 6LoWPAN+RPL networks was released (https://github.com/EIT-ICT-RICH/contiki), and is still under development, with partial support for IETF 6TiSCH specifications.

RIOT. RIOT is a free, open-source OS developed for the IoT based on a micro-kernel architecture inherited from FireKernel [33], providing real-time capability, multi-threading and IPC support, as well as a tickless scheduler that works without any periodic events. RIOT aims for a developer-friendly programing model and API [26], aiming at similarity to what is experienced on Linux. While the OS is written in C (ANSI99), applications and libraries can be implemented either in C or in C++. RIOT is developed as such since 2012, by a growing, world-wide open source community including both academic and industrial contributors. The source code is available on GitHub (https://github.com/RIOT-OS/RIOT) under LGPLv2.1. To fulfill strong real-time requirements, RIOT enforces constant periods for kernel tasks (e.g. scheduler run, inter-process communication, timer operations). An important prerequisite for guaranteed runtimes of $O(1)$ is the exclusive use of static memory allocation in the kernel. Constant runtime of the scheduler is achieved by using a fixed-sized circular linked list of threads. RIOT features a variety of network stacks, including (i) gnrc, an implementation of the full IPv6/6LoWPAN stack, and (ii) CCN-lite, an information-centric networking stack [34], [35], as well as (iii) OpenWSN implementations of TSCH and 6TiSCH protocols. Similar to ports on BSD, RIOT features a package system that allows for easy integration of libraries from third-party providers (this feature is used e.g. to make OpenWSN implementations available in RIOT).

TinyOS. As one of the first custom operating systems for resource-constrained networked embedded systems, TinyOS [27] has been a dominant software platform for sensor network research, enabling significant academic research and commercial products for the last fifteen years. The main design goals of minimal resource usage and flexibility are supported by a component-based architecture codified by the nesC language [36], a variant of C. Maximal concurrency under limited resources is supported by an event-driven execution model with two process categories: interrupt service routines and tasks—basic deferred processing primitives that are scheduled by a non-preemptive FIFO scheduler. TinyOS uses a three-layer hardware abstraction model that offers a flexible hardware/software interface with combined access arbitration and energy management [37]. Facilitating protocol stack research is another major design goal for TinyOS, and a reason for its popularity. Many academic precursors of today’s standardized protocols for low power and lossy networks have been prototyped using TinyOS. Together with Contiki, it was one of the first sensor network operating systems that offer an IPv6 stack [38], with support for 6LoWPAN, RPL and CoAP. Since 2009, it also provides a platform-independent implementation of the IEEE802.15.4-2006 MAC. The TinyOS source is a mature code base released under a BSD license and available on GitHub (https://github.com/tinyos/tinyos-main). Support for the TSCH extensions of IEEE802.15.4e, and partial support for the rest of the IETF 6TiSCH stack have recently been developed as part of the EIT Digital RICH Activity. The source code for these extensions is also made available on GitHub (https://github.com/EIT-ICT-RICH/tinyos-os-main).

B. Hardware Requirements

The four implementations of IEEE802.15.4e TSCH listed in Section III-A run on a combined set of 11 platforms (see Table II). These platforms range from decade-old two-chip solutions (with one micro-controller and one transceiver chip) to state-of-the-art system-on-chip solutions (a single chip offering both micro-controller and transceiver capabilities). This highlights that TSCH, despite its strict timing requirements, can be implemented on most low-power wireless hardware platforms.

To be able to implement TSCH, the hardware must provide (1) low-power timers and (2) packet timestamping capabilities.
It is also necessary (3) that the timers can wake up the microcontroller from a deep sleep mode in a bounded short time. In addition, it might be beneficial if it offers acceleration for the security operations.

Because it is the most common configuration, we use Fig. 3 to illustrate the discussion in this section using a 2-chip platform, where the micro-controller and transceiver chips are connected through a digital interface, typically a Serial Peripheral Interface (SPI) bus. Through that interface, the micro-controller loads the frames to be sent into the radio, switches the transceiver chip between sleep/transmit/receive modes, and reads the received frames. When it receives a packet, a typical transceiver asserts a digital pin connected to the micro-controller, which triggers an interrupt indicating that the received frame can be retrieved over the digital interface.

In TSCH, within a timeslot, several actions occur at precise moments in time. Sender and receiver know exactly when a packet has to be received or sent within the timeslot. This requires precise synchronization between nodes (typically < 100µs), achieved by periodic clock re-alignment using control or data packets. The synchronization mechanism is based on the ability for the device to record the exact time within the slot at which a packet is sent or received. Usually, a hardware timer with capture capability is used, and the capture functionality is triggered by the transceiver interrupt. The recorded timestamp is used to determine the deviation of the current clock counter from the ideal time. The correction is then applied at the end of the slot by enlarging or reducing its size.

The low-power timers should be able to operate at a frequency not less than 32kHz, and provide at least one capture/compare register. A maximum drift below 20ppm is desirable, but higher drift rates can be compensated with advanced clock synchronization protocols such as Gradient Time Synchronization Protocol (GTSP) [39], adaptive synchronization techniques [40], or external clock synchronization via e.g. GPS. It is also necessary for timers to be able to wake up the micro-controller from deep sleep modes with bounded delay. The timer frequency introduces a basic trade-off between energy-efficiency and precision, which may, in turn, affect the overall network synchronization and efficiency.

IEEE802.15.4e defines link-layer security mechanisms in which all frames are authenticated and/or encrypted. Encrypting and authenticating a frame takes time, especially if no hardware acceleration is available. This has a direct impact on the minimal duration of a timeslot. Ideally, a timeslot is as short as possible to increase the throughput and efficiency of the network, and lower the communication latency. Yet, because a node needs to decrypt a data packet then encrypt the link-layer acknowledgment, security often impacts the achievable duration of a timeslot, which can vary from 8ms to 80ms depending on the availability of hardware acceleration for security.

Besides energy, memory is typically the most scarce resource on any IoT platform. Some of these constrained nodes—categorized as Class 0 devices in RFC7228—provide only a couple of kB of RAM and some tens of kB of ROM. While the implementation of the TSCH state machine itself has a tiny memory footprint (less than 4kB of ROM and 1kB of RAM in OpenWSN), some memory has to be reserved for neighbor information and packet queues. The amount of data for storing neighbor information mainly depends on the density of the network. The size of the packet queue depends mostly on the traffic load and the density of the schedule. As upper layers are typically agnostic to the slotted behavior of the link layer, TSCH has to queue data from these upper layers until a matching timeslot occurs. An entry in the packet queue in OpenWSN, for instance, consumes about 240 bytes for payload and metadata.

### C. Hardware/Software Interface

Due to the relative novelty of the standard, at the moment, there is scarcity of transceiver chips that provide native support for the IEEE802.15.4e TSCH extensions. As a result, most current solutions rely on software implementation of the missing functionality on top of the generic services provided by the hardware platform. For portability and abstractions, the OS’s define their own hardware/software interface on top of which the 6TISCH stack is implemented. This interface is critical when it comes to achieving the strict timing requirements of TSCH.

Beyond achieving the pure functional requirements, the hardware/software interface has competing goals. It must (1) raise the level of abstraction and make the upper layers portable while (2) being able to exploit platform-specific features required for high performance. The above goals have to be achieved within the constraints of the given operating system, in terms of the supported execution models and code organization principles. In the following, we discuss the major roadblocks to attaining these goals and the strategies applied by the surveyed TSCH stack implementations.
1) Timers: Timers are fundamental hardware peripherals to support a TSCH implementation. A 6TiSCH protocol stack requires several actions to take place at certain moments in time, such as waking up for listening or transmitting. For those tasks, a general purpose timer, usually virtualized (for multiplexing), is implemented using a hardware timer with at least one counter and one compare register. As illustrated in Fig. 4, virtualization is achieved by keeping a sorted list of pending timers, and using the hardware timer to trigger an event for the earliest moment at which the next one is scheduled.

Internal timeslot timing is usually controlled by another hardware timer (the bottom timer in Fig. 4). It is desirable to be able to program its period so an event is triggered when the period overflow occurs. At this time, the counter is reset and starts counting again. The compare register is used to trigger the different actions within the timeslot.

The ability to put the micro-controller in a deep sleep mode, and be able to wake it up with a timer interrupt is another major requirement for energy-efficient TSCH implementations. This allows the system to minimize the energy consumption during the inactive phases within the slot. A major design consideration is then the trade-off between the energy savings offered by a given deep sleep state, the maintained clock sources, and the time needed by the micro-controller to wake up from it.

While the surveyed operating systems have slight differences in the timer implementations and how they are virtualized, the general concept is replicated in all of them: leveraging a dedicated timer for the TSCH implementation and a separate set of virtualized timers for the rest of the system with less time-sensitive operations. The system timers can be based on an OS tick counter as in Contiki, or mapped to another HW

The survey concluded with a final approach for timeslot timing: use a dedicated HW timer and a virtualization layer.

At the same time, features like source and address filtering, PANID matching and automatic CRC calculation etc, might simplify developers task, but tighten the coupling of the low-level driver to the hardware platform.

New single-chip SoCs offer improved performance and reduced HW and firmware complexity by eliminating the serial bus communication between the MCU and the transceiver. Communication is done through direct memory access, or dedicated memory registers devoted to transceiver operation. In this case, it is desirable that HW vendors provide hardware timestamping capabilities at the register level.

From the four OS’s studied, we identified the following minimal set of primitives for transceiver control:

- radioOn(): enables the radio and sets it in receiver mode.
- radioOff(): turns off the radio.
- radioLoadPacket(): loads a packet into the radio. Might use SPI bus or direct memory access if SoC.
- radioGetReceivedPacket(): retrieves the packet buffer from the radio once received.
- radioTxNow(): tells the radio to start sending the SFD and the subsequent byte stream immediately.
- radioStartOfFrameTimestamp(): used to detect when the SFD is received or sent by the radio, enabling precise timestamping.
- radioEndOfFrameTimestamp(): indicates the end of the reception or transmission and is used to trigger next steps in the TSCH state machine.

Analyzing the current implementations and how they combine the timer and the transceiver services, shows that more optimal transceiver services can significantly simplify the implementation of TSCH and TSCH-like protocols. For example, having hardware-supported timer-triggered primitives in the form transmit at time t or transmit in ∆t, as well as the possibility to automatically cancel such commands when they cannot be completed within the pre-allocated time, can be very convenient and reduce the need for precise timing control in the software implementation. A more complete baseband interface for TSCH should also offer automatic generation of

Fig. 4. Example of timers structure in OpenWSN. The concept is similar in Contiki, RIOT and TinyOS implementations. General purpose timers are virtualized while the MAC layer uses a dedicated HW timer.
ACKs with time correction information elements, as well as support for retransmissions, Clear Channel Assessment (CCA) and backoff mechanisms for shared links.

3) Board-Support Packages: An emerging trend from chip producers is to promote the use of Board-Support Package (BSP) libraries that hide the raw register-level HW access. This provides more freedom for the chip producers to innovate on the hardware level while not breaking portability with existing software solutions. Some typical examples are the Jennic SDK Libraries from NXP or the MSP Driver Library from Texas Instruments.

Such low-level libraries can have both positive and negative impacts. On the one hand, they raise the level of abstraction and simplify the use of the hardware. On the other hand, if the abstractions are not aligned with the specific requirements of IEEE802.15.4e, they can have detrimental impact on the performance. Moreover, the duration of the different operations handled by these libraries might be unknown and/or unpredictable (particularly if the source code is not available).

Closed BSP libraries also prevent independent research on new standard extensions, more efficient implementations, or new usages of the underlying technology. As we target new and more challenging CPS scenarios, or explore impact of phenomena like capturing effect and constructive interference [41], it is important to maintain the possibility for very low-level access to the hardware, even on commercial transceiver chips [42]. In absence of such low-level access, evaluation of even small changes in the built-in baseband processing requires experimentation using separate software-defined radio platforms.

D. Upper and Lower Stack

Hard real-time support for a general purpose OS is difficult to achieve and for none of the four surveyed OS’s real-time behavior has been formally proven. The event-driven systems support execution models with two process categories: interrupt service routines (ISRs) for the asynchronous and latency-sensitive code, and tasks or threads as deferred execution constructs for the serial and less latency-sensitive code. On a preemptive, multi-threading system, ISRs are usually kept as short as possible, and latency-sensitive code is executed with a high priority task.

Many modern micro-controllers allow to specify priorities for certain interrupt sources. This can be used to prioritize the critical timer and radio interrupts for TSCH. For older architectures, low-priority interrupt sources (such as UART or GPIOs connected to sensors) have to be either disabled completely or after crossing a certain threshold during time-critical periods.

As a result, for both types of OS’s, the implementation of the full stack is typically split in two parts: (i) the time-sensitive TSCH functions are implemented in interrupt mode and (ii) the upper protocol layers are implemented in task mode.

A significant source of complexity in the lowest levels of the stack is the implementation of the state machine of the TSCH protocol. As mentioned above, this state machine is typically realized as low-level event-driven code that directly reacts to timer or transceiver interrupts. In contrast to the synchronous upper layers, that can leverage process categories like tasks or threads, this code is typically implemented rather monolithically, without structural and scheduling support from the operating system. Recently, dedicated software frameworks for this low-level and latency-sensitive MAC code have been proposed, that can improve the developer productivity and the robustness [43].

Another key concept is the decoupling of the underlying TSCH slot operation, usually asynchronous and guided by hardware interrupts (i.e. timers and radio activity) from upper-stack operation. This decoupling of the upper layers from the MAC protocol represents a challenge and usually can be seen as an asynchronous communication between two independently working subsystems. The interactions are asynchronous and use mechanisms such as queues, semaphores and deferred interrupts to delegate the execution between each other. Certain mechanisms on the upper layers require direct information about the status of the underlying MAC layer, this includes the routing protocol, the network scheduler and the control and management planes.

Packets are handled by queues which may have different priorities. Elements in the queues are extracted according to the required actions in particular slots. For example, a transmission slot from node A to node B will consume the first packet (if any) whose destination address is B, otherwise the slot will not be used despite the possibility that there are other packets waiting to be sent to other nodes. Although this might seem inefficient, the scheduled operation of the network ensures optimal behavior and avoids contention despite some times having idle listening slots if traffic is bursty. For the latter, efficient dynamic scheduling and over-provisioning techniques are widely used by vendors of this technology.

As an example of integration between the OS and the stack, we look at RIOT which handles OpenWSN as a single thread with maximum priority to ensure that all deadlines are met. However, ongoing work is splitting this thread up into one thread for packet reception, one thread for sending a packet down the stack, and one thread for handling timer events and informing the upper stack about the success of packet transmission (after the corresponding ACK has been received). RIOT shows that it is possible to combine the TSCH requirements with an RTOS, which brings a key advantage for demanding applications where high sensor sampling rates are combined with TSCH networks.

To better understand the flow of a packet in a typical 6TiSCH stack, and how interactions with upper and lower layers happen, we go through a simple example. Assume a CoAP packet is created by an application sampling a temperature sensor. The payload of the packet is filled with the sensor reading and the CoAP header appended to the packet. This packet is kept in a memory buffer or queue that is used across layers in order to avoid copying bytes in memory. The UDP headers are also added and the packet is then moved to the 6LoWPAN layer where routing extension headers, IP-in-IP encapsulation and the 6LoWPAN header are appended. Finally, at the 6top layer, the packet is appended with the IEEE802.15.4 header and inserted in the corresponding TSCH queue. Asynchronously to this, the TSCH MAC layer keeps executing actions slot by slot. At the beginning of the slot (or at the end of the previous)
it determines if the next/current slot is active. If so, it looks at the schedule to see what action should be taken, and polls the queue for a packet which is addressed at the target of the current slot. If the packet exists, it is retrieved from the queue and sent to the radio either by serialization through an SPI bus or by direct memory access on a single-chip platform. When the ACK for this packet is received, or the number of retransmissions has reached a maximum value, 6top is notified and the packet is removed from the queue.

IV. INTEROPERABILITY AND EARLY RESULTS

In this section, we review how TSCH implementations interoperate and connect to the Internet, and present performance measurements providing insightful reality checks.

A. Internet Integration

A gateway node, known as IPv6 Low Power Border Router (6LBR)—see Section II—is in charge of adapting and routing IPv6 packets to and from the 6TiSCH network. The 6LBR is a fundamental network component which manages the routing control plane, including source routes. It acts as sub-network routing root and time source neighbor of first hop nodes. There are two general approaches: the 6LBR can be running either on a computer or a wireless node.

6LBR on a Computer. In OpenWSN and Contiki, the 6LBR runs on a computer. The mote connected to the 6LBR acts as a link-layer bridge, relaying packets from the 6LBR over a serial interface to/from the 6TiSCH network. OpenWSN implements the 6LBR in a Python application called OpenVisualizer. The application is connected to the link-layer bridge via a serial port, and to the computer’s IP stack and in turn the Internet through a tun virtual interface. OpenVisualizer acts both as a router and 6LoWPAN compactor: it replicates all traffic intended to the PAN over the link-layer bridge, and directs other traffic to the tun virtual interface. Contiki operates in a similar way, with a SLIP serial interface on one side and a tun virtual interface on the other. A notable difference is that this 6LBR is obtained by compiling and running Contiki as a Unix process (ARM7 or x86). Therefore, the 6LBR runs the same implementation as other nodes in the network, but compiled for Unix and with a dual interface, rather than for a IEEE802.15.4 node.

6LBR on a Wireless Node. A different approach consists in running the 6LBR on a wireless node, connected if needed to a host computer with external network access. This is the approach followed by TinyOS, RIOT, and optionally Contiki. In RIOT, the 6LBR is implemented on the mote by providing full IPv6 and multiple interface support. Packets are transmitted over the serial port using SLIP encapsulation and are received by a tun virtual interface on the host side. The TinyOS implementation is similar. The PppRouterC application is installed on the gateway node and advertises itself as a RPL DODAG root, tunneling packets with the host over a PPP bridge. Contiki can also run as a 6LBR on a node, tunneling packets via SLIP. A native program on the host, tunslip6, is in charge of routing IPv6 traffic to/from the PAN.

B. Memory Footprint

To demonstrate the practicality of the 6TiSCH stack and its usability on constrained devices, Table III presents full-image footprints for the four open source implementations. The images include a basic 6TiSCH stack, a 6TiSCH minimal schedule, 6LoWPAN and RPL. Note that we are interested in orders of magnitude rather than exact numbers here, as different platforms and OS’s have their own properties and feature sets.

Our measurements, as depicted in Table III, show that the 6TiSCH stack is able to run on systems with a few kB of RAM and a few tens of kB of Flash. This leaves significant space for various applications in Class 1 or Class 2 devices, and may even be practical, in simple scenarios, for Class 0 devices.

In Table IV, the memory consumption per layer is shown (using the OpenWSN implementation on a TelosB). The table also presents the memory overhead that is introduced for multi-threading features if the stack is built on RIOT.

C. Multi-threading

We run micro-benchmarks to evaluate RIOT’s multi-threading feature, in particular in terms of overhead due to Inter-Process Communication (IPC). On an IoT-lab_M3 node (equivalent to an Arduino Due with an IEEE802.15.4 radio, see [44]) running RIOT, we measure that IPC on average requires around 550 CPU cycles, which is only one order of magnitude more than a function call. Running at typical frequencies (40-80MHz) we observe that context switch requires less than 14 µs on off-the-shelf IoT hardware (based on ARM Cortex-M3). This fits the requirements of most applications, as well as the stringent timeslot requirements of TSCH (a typical timeslot being 10ms long).
Latency

1.6 %

99.910 %

514 ms

126 ms

Duty Cycle

349 ms

100.0 %

99.870 %

99.996 %

3.1 %

Radio

waking-up just in time for an active slot. For example, the
Tx is scheduled and the actual transmission of the packet.
TinyOS implementation wakes the system one slot before an
active slot too, e.g. between the time the
CPU is active, when the radio is active, and the SPI activity.

We run a full system experiment with Contiki’s implemen-
tation of the 6TiSCH stack in the Indriya testbed [45], with
98 TelosB nodes spanning 3 floors in an office building. Our
experiment consists in a periodic data collection with 1-min
packet interval, i.e. the gateway receives an end-to-end packet
every 612ms, on average. The network runs RPL in storing
mode with ETX as routing metric. The experiments last 1h, and
we exclude the first 10min to account for RPL convergence
time. We run every experiment at least 5 times, and show mean
performance across all runs.

We consider three different scenarios: (1) always-on, where
TSCH is disabled, and nodes use the traditional always-on
CSMA MAC of IEEE802.15.4-2011 [28]; (2) 6TiSCH Minimal [46], where TSCH is enabled with a simple 6TiSCH
minimal schedule with a slotframe of length 3. In this setup,
all nodes wake up simultaneously every third slot, to either emit or listen. This results in contention-based medium
access. The 6TiSCH minimal schedule is designed for network bootstrap rather than to run actual applications. We include
this setup nonetheless as a baseline for TSCH. Finally (3), we
run Contiki’s autonomous TSCH scheduler, Orchestra [47].
In this case, TSCH slotframes and slots are allocated in a
distributed manner, autonomously at every node, following
RPL topology information. Nodes maintain a transmit slot to
their preferred parent, at a time offset calculated as a hash of
the parent’s MAC address. Similarly to the 6TiSCH minimal
schedule, this scheduler also uses a globally shared slot for
broadcast, repeating every 31 slots.

Our results are summarized in Table V. Note that the
topology built by RPL results in an average distance to the root
of 4.2 hops, and a density of 16 (neighbors per node). Overall,
all setups achieve high reliability, in particular 6TiSCH with
the Orchestra scheduler, with 99.996% end-to-end delivery
ratio. The always-on case achieves the lowest latency, as nodes
transmit as soon as they can. This, however, leads to a radio duty
cycle of 100%. With 6TiSCH, nodes can be put to sleep most of
the time to save energy. In our runs, the radio chip was turned on
only 1.6% to 3.1% of the time, yielding an end-to-end latency
in the order of half a second. All experiments are with simple,
contention-based schedules, and with an early prototype of the
6TiSCH stack. We expect more mature implementations with
full-featured distributed or centralized schedulers to achieve
even lower latency and lower duty cycles.

The presented open source implementations are currently
under evaluation in different deployment contexts. For example,
a testbed-based evaluation of a decentralized scheduler
implementation, based on OpenWSN is presented in [48]. In
addition, the Contiki TSCH implementation on the JN5168x
platform, is currently being evaluated as a component of a
system for remote monitoring of a 200m long railway bridge
over the river Llobregat near Barcelona, and for monitoring
the health of machine bearings in a factory automation setting.
Both of these deployments are part of the EIT Digital RICH
Activity.

F. Interoperability

Even though the implementations are in early stages, they
successfully demonstrate that TSCH can be implemented on
a wide range of off-the-shelf hardware, using a variety of software
approaches, and nicely complement the already well-adopted
commercial solutions such as SmartMesh IP [21]. We expect

---

**Fig. 5.** Energy consumption with OpenWSN for a 15ms active slot (left Rx, right Tx), measured on 2-chip platform featuring a TI MSP430 micro-controller and Atmel AT86RF231 radio connected through an SPI bus. At the bottom, a pin indicates when the radio is active. From the energy trace, we can see when the micro-controller is active, when the radio is active, and the SPI activity.

**TABLE V**

PERFORMANCE OF CONTIKI’S 6TISCH IMPLEMENTATION ON A 98-NODE TESTBED. RPL + TSCH DATA COLLECTION AT 1-MIN PACKET INTERVAL.

<table>
<thead>
<tr>
<th>Schedule Type</th>
<th>Delivery Ratio</th>
<th>Latency (ms)</th>
<th>Radio Duty Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always-on</td>
<td>99.910 %</td>
<td>126</td>
<td>100.0 %</td>
</tr>
<tr>
<td>6TiSCH Minimal (3-slot slotframe)</td>
<td>99.870 %</td>
<td>349</td>
<td>3.1 %</td>
</tr>
<tr>
<td>6TiSCH with Orchestra Scheduler</td>
<td>99.996 %</td>
<td>514</td>
<td>1.6 %</td>
</tr>
</tbody>
</table>
An important aspect is their compliance with the Minimal 6TiSCH Configuration [46] which proposes the simplest configuration so different implementations/products can interoperate, mostly motivated by the wide set of configurations supported by the IEEE802.15.4e TSCH standard. The Minimal 6TiSCH configuration ensures that compliant applications are able to form a network, parse Enhanced Beacons and agree on a timeslot size and structure. The IETF 6TiSCH working group periodically organizes interoperability events such as the plugfests at IETF89 and IETF90 (2014) and the first ETSI 6TiSCH plugtests at IETF93 (2015).

We summarize the result of the 6TiSCH plugtests [49], with a focus on interoperability between TinyOS, Contiki and OpenWSN. Table VI presents some of the interoperability results considering simple TSCH synchronization and joining, and the ability of the implementations to fully parse each other’s MAC layer frames and commands. In addition, the early 6TiSCH specification such as the minimal configuration has also been evaluated, showing a very good interoperability between Contiki and OpenWSN except for the fact that one uses RPL Storing Mode of operation while the other uses Non-Storing. In contrast, TinyOS and OpenWSN have been able to interoperate at the RPL level and at the MAC level (being able to synchronize and join one and another), but due to the early stages of development, they still require some more effort to fulfill complete 6TiSCH interoperability. TinyOS and Contiki are currently able to interoperate both at MAC and RPL level, with exclusion of the security features and with minor incompatibility issues due to the default header compression options in the respective 6LoWPAN implementations.

### V. Conclusion

This article has introduced emerging standards for low-power wireless IP-based industrial communications, and focused on open standards enabling IP-based communication on top of low-power and lossy networks such as those developed by 6TiSCH. Through the article, and from a first-hand experience—the current state of the standardization process—major issues under discussion and future aspects to be addressed have been outlined. Based on our experience while implementing those standards, and from four different angles (OpenWSN, Contiki, RIOT, TinyOS), we outline major requirements and lessons learned. We show the common aspects within the four open-source implementations which we believe will facilitate massive industrial adoption and sound architectural design. We believe that they propose a clear approach to implementing TSCH technologies on off-the-shelves hardware platforms. Finally, we present insights from early performance evaluations, provide guidelines for chip manufacturer and implementers of those new trend standards.

### VI. Acknowledgment

This work was partly supported by the distributed environment Ecare@Home funded by the Swedish Knowledge Foundation 2015-2019, by the EIT Digital RICH Activity, by ANR within SAFEST project (ANR-11-SECU-004), and by VINNOVA (Sweden’s Innovation Agency). The authors thank Jasper Büsch and Moksha Birk for their work on the TinyOS implementation and their valuable comments.

### References


Xavier Vilajosana is a computer science engineer, co-founder of Worldsensing and OpenMote Technologies. He is currently Associate Professor at the Universitat Oberta de Catalunya. From January 2012 to January 2014, Xavier was visiting Professor at the University of California Berkeley holding a prestigious Fulbright fellowship. In 2008, he was visiting researcher of France Telecom R&D Labs, Paris. Xavier has been one of the main promoters of low power wireless technologies, co-leading the OpenWSN.org initiative at UC Berkeley, and promoting the use of low power wireless standards for the emerging Industrial Internet paradigm. He also contributed to the industrialization and introduction of Low Power Wide Area Networks to urban scenarios through Worldsensing. Xavier is author of different Internet Drafts, as part of his standardization activities for low power industrial networks. Xavier is contributing actively at the IETF 6TiSCH WG. He holds an MSc degree on Computer Science from the Universitat Politècnica de Catalunya (UPC) and a PhD on Computer Science from the Universitat Oberta de Catalunya. At the moment, Xavier is author of 11 patents, more than 25 high impact journal publications and more than 40 International conference contributions. Technically, Xavier has extensive experience in Distributed Systems, Wireless Networks, Delay Tolerant Networks and Software Defined Networks.

Simon Duquennoy is a senior researcher at the Swedish Institute of Computer Science (SICS Swedish ICT). He obtained his PhD from Université de Lille 1 (INRIA, CNRS, France) in 2010, and did his postdoc at SICS as the holder of an ERCIM Alain Bensoussan fellowship. His research interests are at the intersection between operating systems and networking for constrained embedded devices, with a focus on IP-based sensor networks and the Internet of Things. He is a core developer and maintainer of the Contiki Operating System. He has been TPC member of internationally recognized conferences such as ICDCS, MASS, DCOSS, LCN and EWSN, and publishes regularly in the sensor networks community flagship conferences ACM/IEEE IPSN and ACM SenSys.

Oliver Hahm is a PhD Student at Ecole Polytechnique and member of the Infine research team at Inria in Saclay. He obtained his Diploma degree in Computer Science from Freie Universität Berlin in 2007. Between 2007 and 2009 he was working as a software engineer for ScatterWeb GmbH, a startup for Wireless Sensor Network applications. From 2009 to 2012 he was working as a research assistant at the faculty of Mathematics and Computer Science at the Freie Universität Berlin and responsible for the G-LAB project. His research is focused on operating systems for the Internet of Things, embedded network stacks, and standardization efforts in this area. He is a co-founder and one of the core developers and maintainers of the RIOT operating system.

Emmanuel Baccelli is a researcher at Inria, championing decentralized, self-organized, cooperative and community-driven concepts for Internet connectivity. After working at AT&T Labs in Florham Park, New Jersey, USA, and for Metro Optix Inc. in the Silicon Valley, as software engineer until 2002, then with Hitachi Europe as research engineer, Emmanuel Baccelli received his Ph.D. from Ecole Polytechnique, Paris, France in 2006. He received his habilitation from Université Pierre et Marie Curie, Paris, France in 2012. In 2013-2014, Emmanuel Baccelli held a DAAD Visiting Professorship at Freie Universität Berlin. Emmanuel Baccelli’s main research interests involve spontaneous wireless networks, Internet of Things, mobility, design and analysis of network protocols and algorithms. Emmanuel Baccelli contributes continuously in standardization efforts within the Internet Engineering Task Force (IETF). He has authored or co-authored dozens of research papers in the field’s top conferences and journals, several RFCs, co-chaired several IEEE and ACM workshops and conferences, and coordinated international research projects. He is a co-founder and coordinator of the open source community developing the RIOT operating system.

Adam Wolisz received his degrees (Diploma 1972, Ph.D. 1976, Habil. 1983) from Silesian University of Technology, Gliwice, Poland. He joined TU Berlin in 1993, where he is a chaired professor in telecommunication networks and executive director of the Institute for Telecommunication Systems. He is also an adjunct professor at the Department of Electrical Engineering and Computer Science, University of California, Berkeley. His research interests are in architectures and protocols of communication networks. Recently he has been focusing mainly on wireless/mobile networking and sensor networks.