RIOT and OpenWSN 6TiSCH: Happy Together

Timothy Claeys, Francois-Xavier Molina, Malisa Vucinic, Thomas Watteyne, Emmanuel Baccelli

RIOT and OpenWSN 6TiSCH: Happy Together

Timothy Claeys, François-Xavier Molina, Mališa Vučinić, Thomas Watteyne, Emmanuel Baccelli
Inria, France
e-mail: first.last@inria.fr

***Abstract*—Short development cycles, application-field diversity, and requirements on network size or reliability put an ever increasing strain on Internet of Things (IoT) application developers. Real-time embedded operating systems (RTOS) aim to provide a key set of features, abstractions and services that enable faster development. To fulfill the promise of wire-like communication reliability, wireless standards such as WirelessHART, ISA100.11a and 6TiSCH have been developed and are used in the industry. Keeping these networks synchronized requires precise timing information from the underlying hardware. However, the hardware abstractions of an RTOS do come with an overhead, and the question arises on how these abstractions impact the performance of a complex network stack. To study this, we integrated OpenWSN, a standards-compliant open-source implementation of the 6TiSCH network stack, with RIOT, a prominent open-source RTOS. We compare the minimalistic "bare metal" approach of OpenWSN with RIOT's full-fledged RTOS environment. We study the impact on network performance, power consumption and real-time application properties. On the one hand, we show that using RIOT to execute a 6TiSCH stack does not degrade power consumption or network performance. On the other hand, we demonstrate how RIOT brings improvements on the time it takes to execute application tasks.**

***Keywords*—** Internet of Things, 6TiSCH, OpenWSN, RTOS, RIOT.

I. Introduction

The evolution of Wireless Sensor Networks (WSNs) into the Internet of Things (IoT) has triggered the proliferation of available hardware meeting the low-cost and low-power requirements. As the applications of this new technology have evolved, so have the operating systems (OSs). From TinyOS [1] and Contiki OS [2] in the early 2000s, to more recent real-time approaches like RIOT [3], we have seen a constant drive towards open-source code and community-driven development. In parallel with the development efforts on OSs, standardization bodies have defined open communication specifications like the Internet Engineering Task Force (IETF) 6TiSCH [4] to bridge the performance of industrial low-power wireless networking with the ease-of-use of the Internet Protocol (IP).

Keeping 6TiSCH networks synchronized requires precise timing information from the underlying hardware. The abstractions provided by the OS do come with an overhead. The complexity introduced by a full-fledged OS in order to execute what is often a single application task is not always deemed necessary. The alternative approach is to operate closer to the hardware and directly make use of the available resources. With less code to run, one may expect a better performance at the cost of code extensibility. With this in mind, we answer the question: *How does an OS affect the performance of a 6TiSCH communication stack?*

We do so by making use of the reference implementation of the 6TiSCH protocol stack, OpenWSN [5]. In its vanilla mode, OpenWSN employs a minimalistic approach and runs over a thin hardware abstraction layer with a basic task scheduler. We integrate OpenWSN stack into RIOT, a real-time, full-fledged OS. We abstract OpenWSN as a RIOT thread, while allowing time critical operations to be executed in interrupt mode.

The contribution of our work is twofold. First, we run OpenWSN in its vanilla mode alongside its integration with RIOT, creating heterogeneous networks. We execute over 70 experiments totaling 140 hours on a testbed deployed in Inria-Paris, France. We show that the overhead introduced by the OS does not influence the performance of the network in terms of reliability and duty cycle. When differences in base system configurations are accounted for, we also observe similar power consumption. We study the real-time properties of both approaches and demonstrate improvements when using RIOT to schedule application tasks running on top of the networking stack. Second, we release the developments related to this paper in open-source form for the benefit of OpenWSN and RIOT communities.

The remainder of this paper is organized as follows. Section II gives a primer on OpenWSN and RIOT, necessary for the comprehension of the results. Section III describes the technical details of the OpenWSN and RIOT integration. Section IV summarizes the technical results. Section V discusses the technical challenges encountered. Finally, Section VI concludes this paper.

II. Background

A. OpenWSN

OpenWSN is a free and open-source implementation of the IETF 6TiSCH specification. It supports a wide range of commercial off-the-shelf (COTS) low-power networking hardware, including CC2538, SAMR21-XPRO, IoTLAB-M3 and nRF52840. The project consists of six statically-linked software libraries, see Fig. 1: board support package (BSP), kernel, drivers, stack, web, app.

1) Board Support Package: The BSP library provides a Hardware Abstraction Layer (HAL). It groups the different functions that directly interact with the underlying hardware, i.e. writing into registers. Each supported board has its distinct implementation details, but they all expose the same Application Programming Interface (API). This approach allows the hardware platforms to share the rest of the libraries.

2) Kernel: OpenWSN's kernel (`OpenOS`) consists of a simple non-preemptive task scheduler. It queues different task contexts created by the various layers of the networking stack and the applications. All tasks have an associated priority. In general, tasks scheduled by the lower layers of the stack have a higher priority. When a task gets executed, the scheduler calls the callback in the context. The run-to-completion scheduler regains control when the callback finishes its execution.

3) Drivers: The driver library implements a generic interface to UART (`OpenSerial`), the timer infrastructure (`OpenTimers`), and sensors (`OpenSensors`), e.g., temperature, humidity, light, infrared, etc. The nodes in the OpenWSN 6TiSCH network primarily use serial communication for logging. Additionally, the border router uses the serial interface to transfer packets from the mesh network to the gateway. Different hardware timers are abstracted by the `OpenTimers` code. The API can create many virtual timers that run concurrently.

4) Stack: The OpenWSN stack (`OpenStack`) is built around the Time-Slotted Channel Hopping (TSCH) protocol, introduced in the 2015 IEEE802.15.4e amendment. The TSCH implementation requires micro-second accuracy by the underlying hardware and is entirely interrupt-driven to minimize radio duty-cycle (below 1%) and CPU activity. The 6TiSCH specification extended TSCH with the 6TiSCH Operation Sublayer (6top) and Minimal Scheduling Function (MSF) to provide IP-compliance. To optimize the limited payload space and to carry large IPv6 datagrams, OpenWSN implements 6LoWPAN fragmentation and header compression. Applications can interact with the stack through the UDP socket API.

5) Web & Applications: The web library (`OpenWeb`) implements CoAP and security protocols such as OSCORE. Applications (`OpenApps`) can either directly interact with the stack using sockets or use the CoAP code to expose resources or make RESTful requests to other CoAP servers.

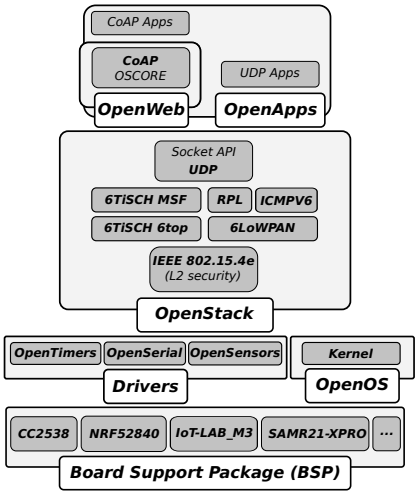

Fig. 1: An overview of the different parts of the OpenWSN.

B. RIOT

RIOT is a general-purpose OS designed for small IoT devices based on micro-controllers.

1) Kernel: RIOT mimicks a micro-kernel architecture [3]. Various software modules are aggregated around a minimal kernel providing:

- Multi-threading: thread synchronization with mutexes, semaphores, small and fast inter-process communication.
- A tickless *O(1)* scheduler with priorities and preemption, which can be used to enforce real-time properties.

The kernel employs separate thread context and memory stack, with a small Thread Control Block (TCB) enabling minimized stack usage globally.

2) Cross-Hardware Support & Unifed APIs: Cross-hardware support is ingrained using a multi-level hardware abstraction including:

- `cpu` which implements and exposes basic microcontroller functionalities.
- `periph` which implements and exposes a unified API to access to microcontroller peripherals, leveraged by drivers.
- `board` which bundles, configures, and maps the selection of CPU and drivers required on a specific IoT device.

3) Network Stack APIs: RIOT provides a number of higher-level APIs including North- and South-bound interfaces to facilitate drop-in-replacement of the network stack.

A generic South-bound interface consisting of a radio Hardware Abstraction Layer (HAL) or radios compliant with IEEE802.15.4, called `ieee802154_dev`. It provides a well defined hardware access that allows implementing agnostic PHY and MAC layers on top. Additionally, it allows for direct radio access and configuration when a network stack needs it.

A generic North-bound interface called `sock` provides a socket-like API for network stacks to offer connections establishment for sending/receiving datagrams on behalf of higher-level logic (i.e. application logic, or another library). Several types are defined, depending on the considered network layer: `sock_ip` for raw IP layer services, `sock_udp` or `sock_tcp` for transport layer services, and `sock_dtls` for DTLS services. Both synchronous and asynchronous network access are possible. Using these abstractions, RIOT applications can run unchanged on a large variety of network stacks supported in RIOT, including but not limited to the following 6LoWPAN stacks: GNRC, OpenThread, lwIP.

4) Power Management: RIOT's power management system is based on two modules: `periph_pm` and `pm_layered`. `periph_pm` defines for each CPU a series of power modes which define different run or sleep modes. Every power mode entails different power savings at the expense of running peripherals, clock speed, wake up sources, RAM retention. However, different applications, drivers, activities have different requirements. When `pm_layered` is used, modules do not explicitly set a power mode but may block and unblock specific power modes according to their functioning requirements. If a power mode is blocked, the blocker requires

that power mode or lower to run. Then, every time a context switch occurs, the lowest possible power mode is selected in "cascade".

5) RIOT Ecosystem & External Library Support: RIOT aggregates a large ecosystem of external libraries, facilitated by the `pkg` system that streamlines the seamless integration at compile-time of a variety of third-party software modules.

For instance, the OpenThread [6] and lwIP [7] network stacks are integrated in RIOT using the `pkg` system. The diversity of libraries integrated with the `pkg` system includes drivers, cryptographic libraries, graphics libraries, machine learning libraries, scripting support [8] etc.

III. Integrating OpenWSN into RIOT

We integrated the OpenWSN project into RIOT using RIOT's `pkg` system, see Fig. 2. Our implementation is open-source, published in [9]. From the point of view of RIOT, OpenWSN operates as any other networking stack supported by the OS. The North-bound interface of the OpenStack library exposes the UDP socket API. The South-bound interface interacts with RIOT's HAL. From the point of view of OpenWSN, RIOT acts as a board support package. It implements all the `BSP` functions required for the correct operation of the OpenStack: `sctimer` and `radio` modules.

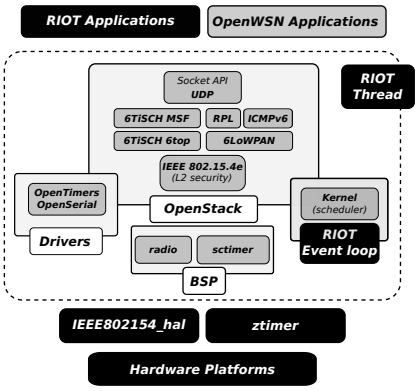

Fig. 2: The OpenWSN stack runs inside a RIOT thread. `OpenOS` is replaced by a RIOT event loop. `sock` provides the Northbound interface and the `radio` and `sctimer` module interface with RIOT's `ieee802154_hal` and `ztimer`.

A. Hardware Abstractions

1) Timer: In a 6TiSCH mesh, nodes periodically resynchronize, which is a critical operation for the network to function. Improved time synchronization between individual nodes translates directly into lower energy consumption. In practice, nodes use stable ($\pm$30 ppm) low-power crystals (32 kHz) as a time reference to minimize synchronization traffic in the network. In OpenWSN, the `sctimer` BSP module abstracts the low-power timers for use by the OpenTimers API.

When porting to RIOT, we replaces the `sctimer` with RIOT's `ztimer` module. The `ztimer` module wraps hardware timers clocked from stable low-power oscillators. In addition, it provides a continuous time reference, even when the device enters a low-power mode (LPM), shutting down most of its other peripherals.

2) Radio: The OpenWSN's TSCH MAC-layer requires direct access to the `radio` BSP module. The TSCH protocol instructs with great time precision the radio to turn on, turn off, transmit, and receive. Any delay in the execution of these state changes would lead to energy waste and the danger of desynchronizing the node from the network. When integrating OpenWSN into RIOT, we mapped the `radio` BSP module to RIOT's `ieee802154_hal` module.

An initial attempt resulted in a conflict between the two finite state machine (FSM) implementations: TSCH state machine of OpenWSN and the RIOT's radio state machine (Fig. 3). RIOT's `ieee802154_hal` enforces a standard radio behavior independent from the capabilities of the underlying radio hardware. For example, the `ieee802154_hal` explicitly requires that the radio resides in the OFF state before changing the radio frequency. In some scenarios, the `ieee802154_hal` FSM conflicted with OpenWSN's implementation of the TSCH FSM. In OpenWSN's TSCH, when the radio receives a frame, an Interrupt Service Routine (ISR) executes. The ISR turns off the radio and reads the contents from the frame buffer. However, when mapped to an `ieee802154_hal`, the `getReceivedFrame` function returns garbage because turning off the radio invalidated the contents of the frame buffer. Both FSMs were later updated to resolve the conflicts.

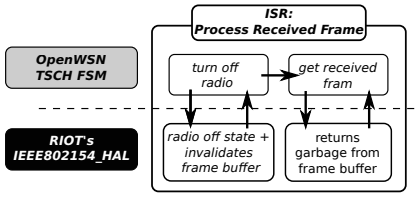

Fig. 3: Conflicting state machine implementations.

B. Network Socket

During the integration process, the OpenWSN socket API was updated to closely resemble the generic RIOT socket interface, i.e., `sock`. By unifying the API, future applications and libraries can be easily shared.

Due to its non-preemptive scheduler, and thus the single-thread nature of OpenWSN, only asynchronous sockets are supported. With OpenWSN integrated into RIOT, the stack and application code is no longer restricted by the non-blocking requirement. The port extends the OpenWSN socket API with blocking sockets and inter-process communication (IPC) between different threads.

C. Scheduler

`OpenOS`, OpenWSN's task scheduler, is replaced by an event loop, using RIOT's Event Queue library. The loop iterates over several queues, each with a specific priority. The OpenWSN tasks are mapped to RIOT events and pushed in a queue with corresponding priority.

IV. Evaluation

To evaluate the effect of RIOT on the performance of the 6TiSCH/OpenWSN implementation, we set up four types of experiments to study: network performance, power consumption, the effect of the kernel on task queue occupancy, and the real-time properties of the scheduling mechanisms involved. In the experiments, we compare a vanilla OpenWSN (OpenWSN_V) implementation with our OpenWSN-RIOT port (OpenWSN_R). Table I lists the different configuration parameters of the OpenWSN stack.

TABLE I: OpenWSN configuration parameters.

Parameter	Value
App. traffic load	Avg. 1 pkt/min
RPL DIO period	10 s
RPL DAO period	60 s
TSCH slotframe length	101
TSCH slot duration	20 ms
TSCH max retransmission	15
Frequency channels	16

A. Network Performance Evaluation

To study the impact of the RIOT's kernel on the 6TiSCH network performance, we perform experiments using the OpenTestbed [10], a testbed consisting of OpenMote-B devices deployed throughout the Inria-Paris office buildings. We form 30-node networks. Using the methodology of Ko *et al.* [11], we start with a fully homogeneous network of OpenWSN_V nodes. In subsequent runs we mix in, in steps of five, OpenWSN_R nodes until we reach a fully homogeneous OpenWSN_R network. Each node transmits 60 UDP datagrams per hour, on average. For each network configuration, we run the experiment 10 times. Each experiment run lasts 2 h. All results are presented with a 95% confidence interval. The average of each run is denoted by a circle, as well as the average over all runs denoted by an asterisk.

1) Reliability: We assess the end-to-end network reliability in terms of the Packet Delivery Ratio (PDR). PDR is computed as the ratio between the number of datagrams received and the number of datagrams generated by the application. Fig. 4 plots the obtained PDR. We can see that in most cases PDR is above 99%, often reaching 100%. Observed PDR drops are explained by network topologies that force the traffic in the network to pass only through a couple of nodes, as it was being routed towards the border router. For instance, in one iteration in the full OpenWSN_V configuration, roughly $3/4$ of the nodes in the network routed traffic through a single node. The latter results in drops at that node due to an overflown internal packet queue. Other PDR drops are caused by routing parent changes and the time it takes for the network to stabilize. We do not observe degradation in terms of the end-to-end PDR when introducing OpenWSN_R nodes in the network.

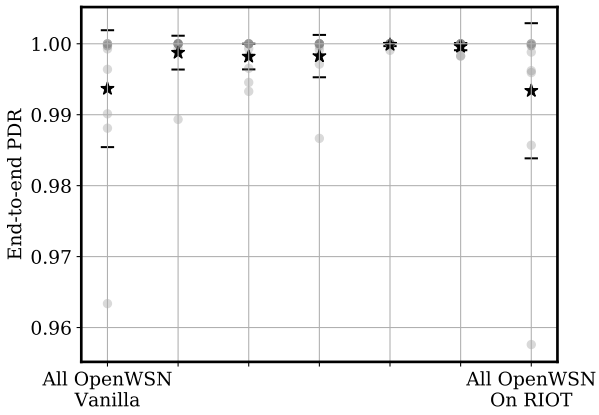

Fig. 4: End-to-end reliability shows no degradation when introducing OpenWSN_R nodes, in steps of five, in the network.

2) Radio Duty Cycle: The radio duty cycle refers to the ratio between the cumulative time that the radio chip is powered on, and the measurement period, as seen by the networking stack. Fig. 5 plots the obtained radio duty cycle. We observe consistency across experiments and configurations. Nodes in the network stay consistently synchronized across experiments, independent of the OS heterogeneity. Synchronization losses in the network would have increased the duty cycle, because desynchronized nodes keep the radio constantly on, listening, while attempting to resynchronize. Minimal differences are due to the randomness in the topology formed for each experiment run. This result is expected since duty cycle values depend on the networking stack operation and are handled from the ISR context.

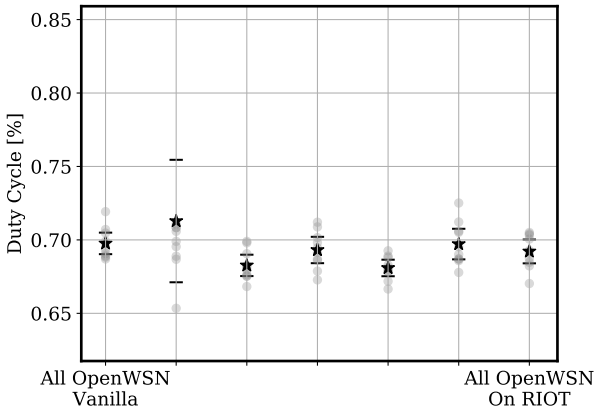

Fig. 5: Radio Duty Cycle stays consistently low when we add OpenWSN_R nodes in steps of five. The Radio Duty Cycle is independent from the OS heterogeneity.

B. Power Consumption

We measure the power consumption by connecting an STM32 Power shield[1], to the nodes in the network. We create a simple star network with one border router, one OpenWSN$_{\text{V}}$ and one OpenWSN$_{\text{R}}$ node. There is no application running on top of the stack. We attach the power shield in turn to the OpenWSN$_{\text{V}}$ and to the OpenWSN$_{\text{R}}$ node. We configure the power shield as specified in Table II. We measure the power consumption once the network is stable. We use the lowest CPU LPM in both cases.

TABLE II: STM32 Power shield parameters.

Parameter	Value
Sampling frequency	100 kHz
Acquisition time	100 s
Supply voltage	3.3 V

We are interested in the following metrics:

1) **Minimum Current** (I_{min}): the consumption of the node when the board operates in LPM.
2) **Peak Current** (I_{max}): the maximal observed current, typically during radio Tx/Rx.
3) **Average Current** (I_{avg}): average current consumption.
4) **Energy** (E): the energy consumed is measured as $V * I_{avg} * t$, where V is the supply voltage and t is the acquisition time.
5) **Normalized Energy** (E_N): $E_N = E - V * I_{min} * t$. We remove from E the component due to I_{min}. I_{min} depends on the base system configuration and is independent of the OS or the power management system.

TABLE III: Power consumption measurements.

Config	I_{min}	I_{max}	I_{avg}	E	E_N
OpenWSN$_{\text{V}}$	0.26 mA	38.3 mA	1.10 mA	0.36 J	0.28 J
OpenWSN$_{\text{R}}$	0.44 mA	37.1 mA	1.20 mA	0.40 J	0.25 J

Table III summarizes the power consumption measurements. We can observe a difference in I_{avg}, where OpenWSN$_{\text{V}}$ outperforms OpenWSN$_{\text{R}}$. The difference is due to the significantly higher I_{min} in the OpenWSN$_{\text{R}}$ case, which comes from a different base configuration of the system when the board is in LPM. If we consider the normalized energy, E_N, we observe that OpenWSN$_{\text{R}}$ consumes 0.08% less than OpenWSN$_{\text{R}}$ when only running the networking stack. The result is surprising, considering that OpenWSN$_{\text{R}}$ introduces overhead due to the mapping of the BSP radio and timer modules, as described in Section III. OpenWSN$_{\text{R}}$ also combines two different radio state machines, as depicted in Fig. 3.

C. Scheduler Performance

We study the impact of the kernel on the performance of the task scheduler. Recall that in case of OpenWSN$_{\text{V}}$, the task scheduler is `OpenOS`, while in the OpenWSN$_{\text{R}}$ configuration, `OpenOS` was replaced with RIOT's Event Library. Although the RIOT documentation uses the term *event* to denote elements of the event queue, in this section we use the term *task* for both OpenWSN *tasks* and RIOT *events*.

1) Impact of the kernel: The first experiment simultaneously monitors the task queue of an OpenWSN$_{\text{V}}$ and OpenWSN$_{\text{R}}$ node over a period of 30 min. Both nodes are synchronized to a border router and have no children. There is no application running on top of the network stack. Once the network is stable, we generate a timestamp for every task being pushed to the queue and a second timestamp when that task is removed from the queue just before being executed. We are interested in the following metrics:

- **Task Wait Time** (T_W): Calculated as $T_W = T_D - T_A$. T_A is the moment when an task arrives in the queue. T_D is the instant when it is removed from the queue.
- **Arrival Rate** (λ): the average number of tasks generated by the task over a given period of time.
- **Long-Term Average Queue Occupancy** (L): the long-term average number of tasks in the queue. Calculated using Little's Law.
- **Maximum Queue Occupancy** T_{max}: maximum number of tasks in the queue at any given moment.

TABLE IV: Scheduler performance measurements.

Config	T_W [µs]	λ $[\frac{\text{task}}{s}]$	L	T_{max}
OpenWSN$_{\text{V}}$	111.92	13.99	1.56e-3	5
OpenWSN$_{\text{R}}$	143.65	16.45	2.36e-3	4

Interestingly, we notice that the task arrival rate (λ) is higher for the OpenWSN$_{\text{R}}$ configuration. At the time of writing, we don't have a clear explanation for this phenomenon. Since there are more tasks being generated per second, the value for T_W is also greater. Another factor that could influence the T_W value is the context switching behavior of RIOT's kernel in the OpenWSN$_{\text{R}}$ node. When the task queue is empty, the RIOT kernel performs a context switch, looking for other threads that need attending. If no other active thread is found, the CPU executes the WFI() instruction[2]. On OpenWSN$_{\text{V}}$ nodes, `OpenOS` runs directly on top of the hardware. `OpenOS` is mono-threaded so no context switches need to occur. The long-term average queue occupancy is derived from the T_W and λ, which explains the discrepancy.

2) Real-time Properties: To study the real-time properties of the scheduling mechanisms involved, we set up an experiment running two simple applications with different priorities on top of the stack. On the OpenWSN$_{\text{V}}$ node, applications are executed by `OpenOS`, while on OpenWSN$_{\text{R}}$, applications live in separate RIOT threads and are thus executed by the RIOT kernel. The applications have real-time requirements, meaning that the schedulers should execute the tasks of the

[1] https://www.st.com/en/evaluation-tools/x-nucleo-lpm01a.html

[2] WFI() is part of the Arm Cortex instruction set, other CPU platforms have similar instructions.

application as soon as possible when they are scheduled. The first application has a task (T_A) which takes $100\,\mathrm{ms}$ to execute. The task is scheduled every $60\,\mathrm{s}$. The second application has a task (T_B) that requires $10\,\mathrm{ms}$ to execute and it is scheduled every $100\,\mathrm{ms}$. Using `OpenOS`, we measure the average wait time T_W of $8.1\,\mathrm{ms}$ for (T_A) and $8.6\,\mathrm{ms}$ for (T_A). Using the RIOT kernel, we measure the average wait time T_W of $68\,\mu\mathrm{s}$ for (T_A) and $69\,\mu\mathrm{s}$ for (T_A). The results clearly show the advantage of RIOT's RTOS kernel over the simplistic `OpenOS` approach when executing several, more advanced apps with specific real-time requirements.

V. Technical Challenges

Issues related to hardware and the BSP. During the initial integration work, we noticed that the networks containing OpenWSN_R nodes, performed significantly worse than the homogeneous OpenWSN_V networks. Further investigation uncovered that the problem was caused by different calculation methods for radio signal indicators RSSI and LQI. This difference affected the formation of the network and its performance in general.

Layering abstractions. The integration work provided an incentive for both communities to verify and improve the different APIs and layering abstractions. OpenWSN developed a new socket API which is closely aligned with RIOT's `sock` interface. The new API provides a better separation between the applications and the CoAP library, and the network stack. For RIOT, the integration of the OpenWSN socket layer, was a valuable test for the design of their `sock` interface. At the lower layers, the mapping of OpenWSN TSCH module to RIOT's `ieee802154_hal` exposed bugs and undefined behavior in the state machines of both implementations, which were fixed.

Role of the operating system. RIOT's multi-threaded kernel allows the applications to execute CPU-intensive functions without the risk of blocking other critical tasks. A relevant example are the applications using application-layer security protocols (e.g. EDHOC and OSCORE). Not all devices have hardware acceleration for all the mandatory cryptographic primitives. This is an issue because software implementation of elliptic curve cryptography operations on an Arm Cortex-M3 typically takes between $1\,\mathrm{s}$ and $2\,\mathrm{s}$ to complete [12]. The OS plays an important role in balancing the execution time and real-time requirements of the different tasks.

VI. Conclusions & Perspectives

In this paper, we study the effect of the OS on the performance of a time-sensitive 6TiSCH communication stack. We integrate the OpenWSN project, the reference open-source implementation of 6TiSCH, into RIOT, a full-fledged real-time OS. When executing the 6TiSCH networking stack in the RIOT context, we do not observe degradation in terms of the end-to-end reliability and radio duty cycle. When measuring power consumption, we observe that RIOT's power management benefits the stack and results in a slight decrease in consumption, once the base system configuration in LPM mode is accounted for. We also demonstrate benefits in scheduling delays from the point of view of applications with real-time requirements. All the code developed in this paper have been published in the open-source form, for the benefit of OpenWSN and RIOT communities.

Using the unified APIs we propose in this paper, our current work includes supporting hardware and libraries (or applications) that can be compatible as-is, with both RIOT and OpenWSN. On-going efforts towards shared OpenWSN/RIOT libraries include implementations of the IETF's Software Updates for Internet of Things (SUIT) and the IETF's Ephemeral Diffie-Hellman Over COSE (EDHOC) key exchange protocol.

Acknowledgements: This work is partly funded by the European Commission through H2020 SPARTA, www.sparta.eu.

References

[1] P. Levis *et al.*, "TinyOS: An Operating System for Sensor Networks," in *Ambient intelligence*, Springer, 2005.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *29th annual IEEE international conference on local computer networks*, IEEE, 2004.

[3] E. Baccelli *et al.*, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, 2018.

[4] X. Vilajosana *et al.*, "IETF 6TiSCH: A Tutorial," *IEEE Communications Surveys & Tutorials*, 2019.

[5] T. Watteyne *et al.*, "OpenWSN: a standards-based low-power wireless development environment," *Transactions on Emerging Telecommunications Technologies*, vol. 23, no. 5, pp. 480–493, 2012.

[6] H.-S. Kim *et al.*, "Thread/OpenThread: A Compromise in Low-Power Wireless Multihop Network Architecture for the Internet of Things," *IEEE Communications Magazine*, 2019.

[7] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," Swedish Institute of Computer Science, Tech. Rep. 77, 2001.

[8] E. Baccelli *et al.*, "Scripting over-the-air: Towards Containers on Low-End Devices in the Internet of Things," in *IEEE PerCom Workshops*, 2018, pp. 504–507.

[9] F.-X. Molina and T. Claeys, "OpenWSN port to RIOT," in *GitHub*, https://github.com/RIOT-OS/RIOT/tree/master/pkg/openwsn, 2020.

[10] J. Muñoz *et al.*, "OpenTestBed: Poor Man's IoT Testbed," in *IEEE INFOCOM, CNERT workshop*, 2019.

[11] J. Ko *et al.*, "ContikiRPL and TinyRPL: Happy together," in *Workshop on Extending the Internet to Low Power and Lossy Networks (IPSN)*, vol. 570, 2011.

[12] K. Zandberg *et al.*, "Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check," *IEEE Internet of Things Journal*, 2019.