# Portability, compatibility and reuse of MAC protocols across different IoT radio platforms

Jan Bauwens, Bart Jooris, Spilios Giannoulis, Irfan Jabandžić, Ingrid Moerman, Eli De Poorter

*Technologiepark-Zwijnaarde 15, 9052 Gent*

*IDLab, Department of Information Technology at Ghent University - imec*

**Abstract**

To cope with the diversity of Internet of Things (IoT) requirements, a large number of Medium Access Control (MAC) protocols have been proposed in scientific literature, many of which are designed for specific application domains. However, for most of these MAC protocols, no multi-platform software implementation is available. In fact, the path from conceptual MAC protocol proposed in theoretical papers, towards an actual working implementation is rife with pitfalls. (i) A first problem is the timing bugs, frequently encountered in MAC implementations. (ii) Furthermore, once implemented, many MAC protocols are strongly optimized for specific hardware, thereby limiting the potential of software reuse or modifications. (iii) Finally, in real-life conditions, the performance of the MAC protocol varies strongly depending on the actual underlying radio chip. As a result, the same MAC protocol implementation acts differently per platform, resulting in unpredictable/asymmetrical behavior when multiple platforms are combined in the same network. This paper describes in detail the challenges related to multi-platform MAC development, and experimentally quantifies how the above issues impact the MAC protocol performance when running MAC protocols on multiple radio chips. Finally, an overall methodol-

*Email addresses:* jan.bauwens2@ugent.be (Jan Bauwens), bart.jooris@ugent.be (Bart Jooris), spilios.giannoulis@ugent.be (Spilios Giannoulis), irfan.jabandzi@ugent.be (Irfan Jabandžić), ingrid.moerman@ugent.be (Ingrid Moerman), eli.depoorter@ugent.be (Eli De Poorter)

ogy is proposed to avoid the previously mentioned cross-platform compatibility issues.

*Keywords:* MAC design architectures, Portability, Compatibility, Cross-platform design, ContikiMAC, TSCH

## 1. Introduction

In the fast growing world of Internet of Things (IoT) devices, wireless sensor networks are deployed in increasingly diverse application domains, ranging from smart homes to factories of the future [1]. Each of these domains have inherently different application requirements such as throughput, battery lifetime, reliability, etc. Wireless network designers need to take into account the trade-offs between these performance metrics. As an example, when using wireless IoT devices to replace wired control loops in industry processes, the network protocols should focus on providing low latency and high reliability, whereas temperature monitoring applications often emphasize long network lifetime requirements.

To this end, an important design decision is the choice of the Medium Access Control (MAC) protocol, which manages how and when the wireless medium is accessed. Countless protocols have been designed with advantages and disadvantages regarding different performance metrics, making it challenging to make an informed decision about the optimal protocol [2]. Some architectures even load several protocols on the device, in order to select the optimal at runtime [3]. For example the TSCH (time synchronized channel hopping) MAC protocol is designed for reliability and low battery usage but has high jitter (deviation of the inter-arrival time between packets) [4], whereas using CSMA/CA (carrier-sense multiple access with collision avoidance) results in low jitter and high throughput but a higher battery consumption [5]. Many open-source implementations of these MAC protocols contain performance limitations. (i) They are often designed for one specific hardware platform (for example the CC2420-radio) and thus can not be reused for other radio chips. (ii) Conversely, many IoT operating

systems only support a single MAC implementation (e.g. RIOT OS only supports CSMA/CA [6]). (iii) Finally, in case multi-platform support is available, the MAC performance typically degrades due to the use of a feature-poor hardware abstraction layer (e.g. Contiki [7] and openWSN [8]). This limited pool of implemented MAC protocols per radio/operating system combination hinders MAC innovations since researchers can not directly use the desired MAC protocol on their development systems. The alternative, implementing the MAC logic from scratch or porting an existing MAC implementation to a new platform, requires extensive knowledge of the target OS and radio chip. Even then it takes a significant amount of time to port a protocol to new platforms, since MAC protocols interact with low level hardware components and as a result have a large and difficult to understand code base. An improvement would consist of implementing MAC protocols once in a hardware independent language, and reuse the code on all desired platforms. These multi-platform protocol implementations could be made available on a MAC protocol cloud storage [9].

Unfortunately, even if devices are able to run the same MAC protocol code, hardware differences might still subtly alter the behavior of a MAC protocol compared to the one running on other radio platforms or legacy devices. For example, the stability and speed of the clock might differ resulting in different timings. If the timing differences become too large, this can result in asymmetrical link behavior. In more extreme cases, the MAC protocol performance can degrade or even become incompatible between the different hardware platforms [10].

The existence of asymmetric links can have an effect on the performance of higher layers, as is shown in [11, 12]. Furthermore, [13] demonstrates that the presence of an asymmetrical link performance severely hurts the performance of the RPL routing protocol.

In contrast to most scientific papers which focus on the design of novel MAC protocol algorithms, this paper proposes a methodology which allows efficient implementation of MAC algorithms on real hardware while simultaneously supporting portability and interoperability between multiple platforms. To this

end, this paper discusses three challenges that influence the performance of MAC implementations running on different hardware platforms.

**Challenge 1: The lack of feature-rich API's which still support portability.** Existing MAC API's or MAC design frameworks do not offer the means to create a protocol which is multi-platform, while still allowing full access to the platform capabilities. In this paper, a hierarchical layered implementation approach is proposed which makes it possible to trade in portability for performance. Our approach gives the developer an informed choice of either building a protocol which is fully portable, or a protocol which uses low-level network/radio capabilities.

**Challenge 2: MAC protocol instruction timings have a strong hardware dependency.** MAC protocols contain numerous timers that determine exactly when radio features such as sleep, transmit, clear channel assessment (CCA), etc. should be executed. However, transitions between radio states are often not documented. Many implementations cope with this by either using timings which are too strict due to a lack of understanding of the low-level timing dependencies of the radio chip instructions, thereby introducing unpredictable MAC behavior. Others choose timings which are too large in order to ensure cross-platform compatibility, thereby reducing throughput and reducing energy efficiency. We demonstrate that the burden of tweaking timers should not fall upon the MAC implementer, but rather the MAC compiler should automatically adjust timings based on the platform in use.

**Challenge 3: Multi-platform networks exhibit unpredictable behavior.** Even when multiple platforms with different radio chips run the same MAC software, issues remain when deploying these different devices in the same network. Unfairness between platforms can occur due to inherent hardware and timing differences. We demonstrate that it is possible to equalize the behavior of a MAC protocol across different platforms to avoid these unpredictable unfairness effects.

The rest of the paper is structured as follows. First, Section 2 gives an overview of related work describing previous approaches aiming to support

MAC protocols running on multiple embedded devices and/or operating systems. Next, Sections 3, 4 and 5 experimentally quantify[1] the performance impact for each of the three above challenges and subsequently propose and evaluate a solution using state-of-the-art MAC design approaches. Finally, Section 6 concludes the paper by giving an overview of all previously mentioned solutions.

## 2. Related work

This section discusses state-of-the art approaches for dynamic MAC frameworks which show promise for multi-platform design.

### *2.1. Multi-platform MAC architectures*

Although a multitude of IoT platforms currently exist, several problems related to running a MAC protocol on different platforms remain unsolved. The authors of [15] highlight several performance issues with (radio duty cycling) MAC protocols, most of which are caused by incorrect timing implementations. It is worth noting that the correct implementation of timers is identified as problematic, even when implementing a MAC protocol for a single platform. This paper puts forward recommendations which could prevent most of the identified issues from happening.

In terms of MAC protocol reuse, a number of existing research papers provide a cross-platform MAC implementation by creating an abstraction layer between the MAC and network layer per operating system [16, 17]. They allow the use of a platform specific MAC protocol on multiple operating systems. One can argue that, although minimizing the effort of a MAC protocol running on

[1] All experimental evaluations used the w-iLab.t wireless testbed facilities, which is a state of the art IoT testbed in Europe [14]. The testbed contains large-scale deployments of sensor nodes of different hardware types, which have been used to conduct cross-platform experiments. All conducted experiments were performed in a single-hop star topology.

multiple operating systems, this is not really cross-platform MAC design but rather cross-operating system MAC design.

Several operating systems, e.g. Contiki [7] and Mantis OS [18], provide an interface for physical layer calls, a so called hardware abstraction layer (HAL). Due to the need for simplicity, these API interfaces omit hardware specific information such as data rates, timings, power modes. As a result MAC protocol developers can not utilize more advanced hardware capabilities. Even worse, MAC protocol implementers lack information regarding the exact implementation and timing constraints of the HAL interfaces. For example, the off-function for the TMote Sky platform in Contiki does not inform the user about the required time to execute this command. Moreover, the actual implementation puts the radio in a low power mode (LPM) rather than completely off, resulting in energy loss. An interface needs to be clear and informative, as the developer should not be limited in his possibilities while creating the MAC protocol code.

A more rich interface is provided by the Wireless MAC Processor (WMP), which is a programmable MAC architecture devised to run a MAC protocol defined in terms of a state machine [19]. It offers flexibility to easily create and adapt novel protocol ideas. However, WMP is only available for the AirForce54G chipset from Broadcom and thus does not support cross-platform MAC portability and reusability.

Another MAC design architecture is the Time Annotated Instruction Set Computer (TAISC) framework, which has a number of advantages compared to traditional MAC protocol development architectures [20]. The main innovation of TAISC was the introduction of time-annotated radio instructions. The compiler adds exact timing information which can be used by an execution engine for instruction scheduling. Although the authors hint that this approach could help to support multi-platform compilation, this claim is not further explored or experimentally validated.

In conclusion, several multi-platform MAC protocol architectures already exist. Unfortunately, most of them only offer a hardware abstraction layer, and do not solve other possible multi-platform issues. In this paper, the TAISC

concept of radio instruction time annotation is utilized and extended to solve the multi-platform problems stated by this paper.

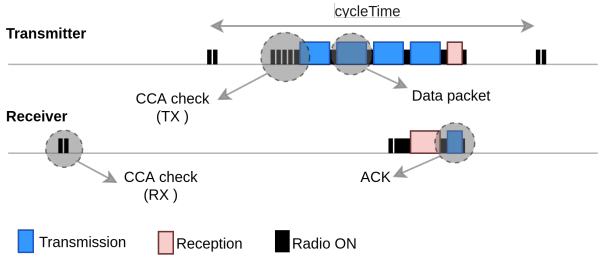
### *2.2. State-of-the-art low power MAC protocols*

To quantify the impact of multi-platform design on the MAC performance, this paper will use two different types of low-energy MAC protocols: the contention-based ContikiMAC and the TDMA-based TSCH MAC protocol.

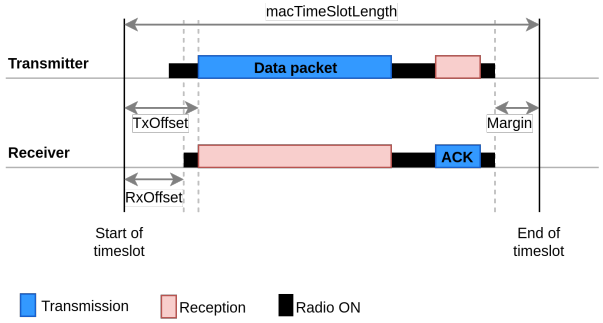
ContikiMAC is a frequently used CSMA-based protocol. From [21]: *"ContikiMAC is a radio duty cycling protocol that uses periodical wake-ups to listen for packet transmissions from neighbors. If a packet transmission is detected during a wake-up, the receiver is kept on to be able to receive the packet. To transmit a packet, a sender repeatedly sends its packet until it receives a link layer acknowledgment from the receiver."* Due to its complex time dependent nature and the fact that it is available for multiple platforms, ContikiMAC represents an ideal protocol for evaluation.

In addition, we also evaluate an implementation of IEEE802.15.4e TSCH, a standardized Multi Frequency Time Divison Multiple Access (MF-TDMA) variant that is dominant in TDMA MAC protocols for wireless sensor networks. From [4]: *"Through time synchronization and channel hopping, TSCH enables high reliability while maintaining very low duty cycles."*

These two widely used MAC protocols serve as a showcase of typical MAC development challenges. The workings of these MAC protocols are visualized in Figure 1.




(a) Conceptual figure of ContikiMAC. The cycletime is the time between consecutive CCA check cycles. CCA RX is a CCA check performed to listen for incoming frames. CCA TX is a CCA check performed to listen if medium is available to transmit a data packet.




(b) Conceptual figure of IEEE802.15.4e TSCH. The most important metric for this paper is macTimeSlotLength, which is the time duration necessary to safely transmit a data packet and to receive the consecutive acknowledgement (ACK).

Figure 1: Illustration of the protocol logic from two widely used MAC protocols: (a) ContikiMAC and (b) IEEE802.15.4e TSCH.

## 3. Challenge 1: The lack of feature-rich API's that support portability.

**Context.** When setting up a wireless sensor network, the MAC protocol is one of the most important decisions as it impacts how efficiently the medium is accessed. If the desired MAC protocol is not available on a platform of choice it needs to be implemented or ported, which is a time consuming process due to the code complexity and hardware dependencies.

**Problem.** In most operating systems and/or MAC development architectures (e.g. Contiki and openWSN) MAC implementations use a hardware abstraction layer (HAL) or radio driver which provides an interface towards the instruction set of the radio chip. A number of limitations are caused due to the inherent design of common hardware abstraction layers.

- A first common pitfall is the creation of a generic instruction set which limits the possibilities of the designer. Most radio chips have advanced features which should be used to their fullest extent in order to create a MAC protocol which is as efficient as possible. However, these are often not included in the hardware abstraction layer due to many interdependencies between the radio system states. For example, putting a device to sleep mode requires a wide range of checks on multiple subsystems of the hardware platform before you can safely decide which LPM mode is possible without loss of data or control of your device. To avoid this complexity, e.g. in Contiki and openWSN the on/off function call is instead implemented as an idle function on some platforms which will not save energy and is different in terms of timing. A developer who only uses the abstraction layer and is not aware of the underlying hardware implementation, will not realize why his protocol is suboptimal.

- Alternatively, some MAC architectures provide a more elaborate interface which is more expressive and intelligent in terms of behavior dependencies. For example RIOT OS and TAISC allow full access to the radio state. Unfortunately, the extended abstraction layer has a drawback. Since not all radio chips support these advanced capabilities they might not be able to run the protocol code, again hindering portability of the MAC implementation.

- TinyOS builds on the concept of abstraction layers. At the lowest layer, the hardware presentation layer (HPL) provides chip specific hardware features (memory access, timers, ADC/DAC, etc.) without providing ex-

plicit interfaces for MAC design. Specifically for MAC design, two abstraction levels are offered: (i) the hardware abstraction layer (HAL) provides platform dependent radio level instructions (e.g. chip specific), and (ii) the hardware interface layer (HIL) provides generic, platform independent MAC level instructions.

As such, current MAC protocol architectures either support multiple platforms by offering only a generic radio/platform-independent instruction set, or they offer full radio control but allow no portability options.

**Solution.** In contrast to existing hierarchies, the major differences between our approaches are as follows. (i) TinyOS and RIOT MAC interfaces are inherently time-unaware, leaving the burden of calculating when to execute a command to the developer, thereby reducing protocol efficiency and hindering portability. (ii) In contrast to the 2 API layers of TinyOS which are either chip specific or very generic, we introduce 3 time-annotated layers. As such, the developer can choose to write a MAC protocol which is not portable (similar to the TinyOS HAL API, but more efficient due to time annotations), portable between devices of the same technology (not supported by TinyOS), or fully portable between all devices (similar to the TinyOS HIL API, but more efficient due to time annotations). (iii) Finally, we have extended the tier-architecture with fallback compilation functions. These functions allow MAC designers to implement a protocol for optimal performance while still allowing compilation towards higher platform independent layers, which is to the best of our knowledge a novel addition.

More specifically, three levels of compatibility were identified.

1. **Radio functionality level.** Using this abstraction layer, the MAC protocol implementation will only be compatible with platforms using the exact same radio chip. This allows to use all radio features.
2. **Technology level.** Using this abstraction layer, the MAC protocol implementation will be compatible with platforms using the same technology type (e.g. BLE, Lora, IEEE802.15.4, etc.). For example the set_bandwidth

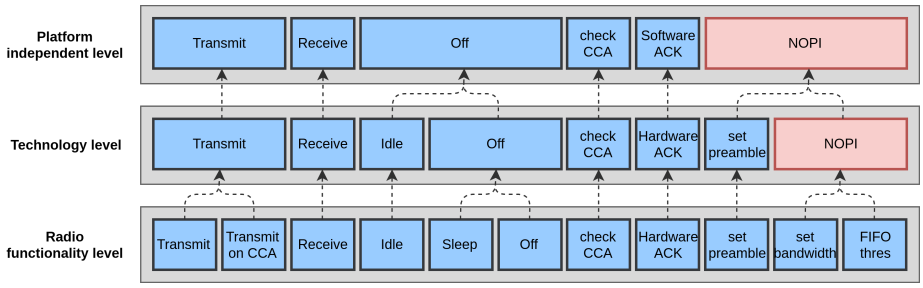



Figure 2: An example hierarchical MAC API is defined allowing explicit portability trade-offs. The radio functionality level provides all features of the radio and only offers compatibility between platforms using the same radio chip. The technology level offers all features of a specific network technology, and are thus compatible with radios supporting the same network type. The last level offers a fully platform independent instruction set. If a function does not exist on a hardware platform, a fall back function or dummy implementation is automatically selected in the compilation phase.

function is available for all LoRa radio chips (but not for e.g. IEEE802.15.4 radios).

3. **Full platform independence.** Using this abstraction layer, the MAC protocol implementation will be compatible with most wireless radio chips. This functionality layer offers basic radio functions (on/off, send, etc.), which are available on most wireless hardware platforms.

In Figure 2 the three levels are visually represented. A developer can choose which interfaces to use, and thus by extension with which platforms he wants to be compatible with. The higher hierarchies contain more generic, often less efficient, implementations of the radio chip specific function calls.

For each function, fall-back functions are explicitly provided. This way, it is possible to implement MAC protocols using a rich-feature HAL, but to still be able to compile MAC protocols to other, less-feature rich platforms[2]. For example, in case fine-grained low power modes are unavailable, the MAC compiler replaces these function calls with the more general "off"-instruction. Some un-

[2]These fall-back mechanisms provide timing information which can also be used to allow interoperability between different radio chips (see Section 5)

usable configuration functions (e.g. set_preamble for 2.4GHz networks) can be replaced by a placeholder instruction ("NOPI") while keeping the MAC protocol logic intact[3]. Since all fallback functions are time-annotated, the timing impact of using these functions can automatically be compensated by the compiler (see Section 4).

**Evaluation.** The hierarchical MAC API functionality has been implemented in the TAISC framework and its benefits have been evaluated using an example scenario. We consider a simple MAC protocol that has been designed for a radio that supports the creation of hardware acknowledgements (lowest API level). When the same MAC protocol is later compiled for a radio without hardware acknowledgement support, the compiler provides a warning and automatically replaces this functionality with a fallback function (in this case: a software based acknowledgement generation). To validate the correctness, the impact of using this specific fall-back function was experimentally quantified. For the experiment, a Zolertia Remote device aims to achieve maximum unidirectional throughput to a second device using the ALOHA MAC protocol. Using hardware acknowledgements, the maximum throughput was measured to be 153.8kbps. Next, we artificially disabled hardware support for acknowledgement generation. Rather than breaking the MAC protocol, the compiler automatically used the software acknowledgement fall-back option. The resulting throughput was 142.5kbps (a decrease of only 7%), showing that fallback functions are a viable approach when porting MAC protocols to hardware with different capabilities, allowing the MAC protocol to run on less-feature rich platforms, albeit at slightly decreased performance.

## 4. Challenge 2: Hardware dependent instruction execution times

**Context.** MAC protocol implementations mainly consist of time critical code, e.g. when to start listening to the wireless medium or when to start

[3]In case no fallback is available on the requested compatibility level a compiler warning is shown.

transmitting. These timers impact the efficiency of the protocol (e.g. reducing the time spent awake or impact the number of transmissions per unit of time) and thus by extension the overall energy consumption of the system.

**Problem.** Unfortunately, due to hardware differences, there exist large timing differences between platforms. To illustrate the impact of these timing differences between platforms, Figure 3 shows the time required to turn a radio on, switch to a specific radio channel, load and transmit a short packet of 32 bytes and turn the radio off again for different IEEE802.15.4-compliant radios[4]. This sequence represents for example the minimum duration of a TDMA slot. As shown in Figure 3 the performance differs dramatically between devices even for a basic sequence of radio instructions. Failure to take into account these specific platform timings into the MAC implementation results in suboptimal timings, which can cause the protocol logic to break. Because the MAC implementation is typically created for one specific radio chip, instances running on other platforms often contain hard to identify bugs [15].

Ideally, each separate radio function call should happen as soon as possible. However, slower platforms might not be able to perform all needed functionality within the defined limited amount of time. This poses a dilemma for the MAC designer: should he include additional safeguard time between each function call, thereby making the protocol less efficient but more portable?

- Consider for example the ContikiMAC time between two consecutive Clear Channel Assessments (CCA) checks. According to the official documentation, this time is assumed to be 500µs. Figure 4 represents the average CCA timings directly measured on several hardware platforms. None of them come even close to reaching the target time, making it likely that a short packet cannot be detected by the CCA checks. To make matters worse there is a time difference between CCA checks in transmit- and receive-mode, due to the logic in between the checks being different.

[4]Respectively the CC2420 radio on the TMote Sky and the Zolertia Z1, the CC2520 radio on the RM090 and the CC2538 radio on the Zolertia Re-mote

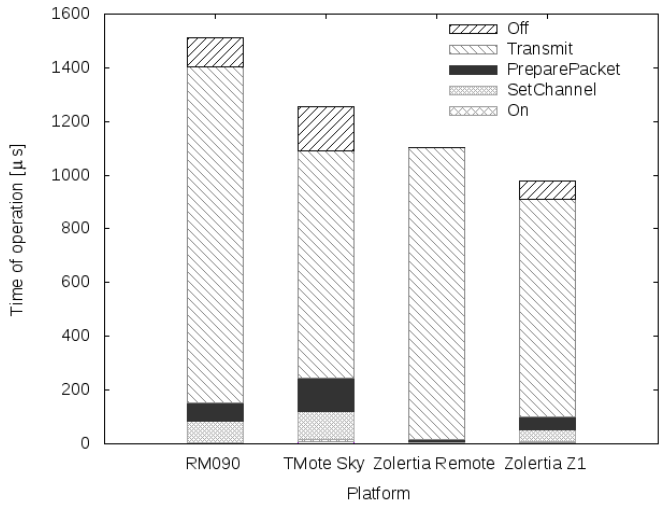



Figure 3: Comparison of the duration of basic radio instructions. The instruction timings vary significantly between different platforms.

- Conversely, developers can include margins in their timings, to counter clock drift or to ensure multi-platform compatibility. An example can be found in the CSMA/CA implementation in Contiki, which waits to receive an acknowledgment. Even if the acknowledgment has already been received the radio remains in the receive state until the AFTER_ACK_DETECTED_WAIT_TIME has passed. Not only is this wait period too long, the MAC protocol also does not react to events happening at the physical layer. A radio should go to idle or LPM when the acknowledgment is received. Although this choice might ensure the protocol runs on multiple radio platforms, it results in performance degradations.

**Solution.** The above problems can be alleviated by having more knowledge regarding the timing of different instructions for different hardware platforms. These timings are hard to calculate theoretically since they depend on many external factors (clock speed, transition type, etc.). Instead, we propose to benchmark the execution time of different instruction timings (e.g. CCA timings, duty cycle, packet transmission timings, etc.) automatically before compile

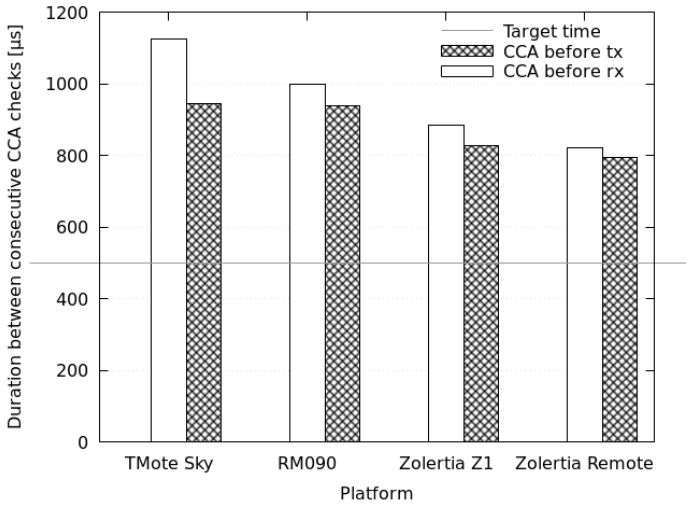



Figure 4: Time between consecutive ContikiMAC CCA checks for different platform, demonstrating that the timing varies significantly depending on the platform. The figure also illustrates that the default ContikiMAC duration (horizontal line) is too strict for the considered platforms. Fifty measurements were performed per plaform/CCA type combination. The variance of the measurements was negligible, since for the slowest devices it was measured to be only 0.025µs.

time, so that these values can be included in the MAC code.

Injecting real radio instruction execution durations measured on the device ensures the protocol logic will not break due to wrongly chosen timings, and removes the need to include large margins to reach multi-platform compatibility. From a practical point of view, measuring the instruction timings (rather than code block timings) is more useful since the data can then be reused for future MAC implementations. Measuring the instruction timings could be done in two ways. (i) Either the instructions are executed, and the timings are measured by a second device. In this case, the authors advice to use designated devices (e.g. a logic analyzer) which are more accurate and precise. (ii) Alternatively, a pre-compiler can automatically execute each used individual instructions sufficient times to derive statistically relevant information about its execution times. These instruction timings can be stored locally, or could be shared through an

automated repository with external database. Using these values, the duration of logical blocks can be calculated out of the individual instruction timings. The burden of selecting optimal timing information is now shifted from the MAC developer to the pre-compiler, allowing the MAC developer to focus on implementing the protocol logic without manually choosing the platform-specific timings.

To estimate the duration of each instruction, the benchmarking framework needs to be able to predict the duration of instructions with variable execution times, caused by several factors.

- The first factor of variability is caused by configurable function parameters, for example the number of bytes to be transmitted. This can be solved by measuring how long it takes to execute a single unit (e.g. the transmission duration of a byte is 32 µs), which can be used to calculate the duration of larger transmissions.

- The second factor is the occurrence of interrupts during function execution. If an interrupt occurs, the instruction timing is dependent on the duration of the interrupt service routine (ISR). This can partly be solved by running the MAC protocol in the highest possible interrupt context, minimizing the possibility that the execution of the MAC protocol can be interrupted or delayed.

- Lastly, the instruction execution timing can be influenced by clock drift. To handle this, a number of measurements should be performed to capture the distribution of instruction durations (e.g. min, max, average duration). The evaluation in this paper assumes a worst case scenario, to make sure that the instruction is always completed in the defined amount of time. Alternatively, the performance could be increase by assuming an instruction duration which is valid in e.g. 95% of the time (95 percentile).

**Evaluation.** To evaluate these concepts, we extended TAISC with an automated benchmarking framework for time duration. For every evaluated platform

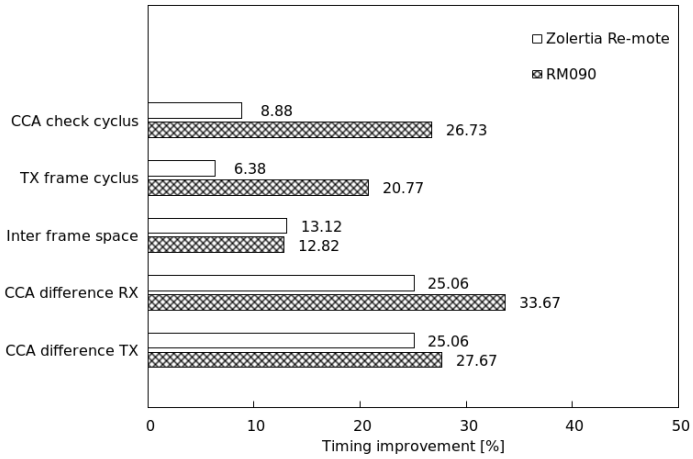
an XML is created with the instruction times for all instructions of all three hierarchical HAL layers of Section 3. This multi-platform XML document is then used to support cross-platform compilation of MAC code. This information is used for scheduling the next instruction: at what point in the future should a next instruction be executed to make sure that execution does not start before the current one has finished, e.g. the off-instruction should not be executed as a tx-instruction is still active. The next instruction should be scheduled as close as possible to the end of the current instruction[5]. This allows to approach the limits of the system in a controlled manner: the result is a MAC implementation that can be compiled to multiple radio platforms with limited, or even non-existing margins. Next the performance gains will be evaluated for the MAC protocols: ContikiMAC and IEEE805.15.4g TSCH.

Figure 5a shows the timing improvements of the automatically calculated timings, compared to the default Contiki implementation, for basic ContikiMAC operations. For every operation, the automated timing calculation outperforms the default implementation. The performance benefits from the shorter acquired timings. the CCA times are a lot closer towards the physical limit of the device, up to 33.67% better compared to the default implementation. Also the IFS is 13.32% better compared to the default implementation. Thus, by including automated time annotations, the performance significantly improves and the protocol logic can efficiently be reused on multiple radio platforms. The better timings achieved through time-annotation also have an impact on radio energy consumption as the radio is more likely to be in the correct state at the right moment, e.g. the radio is less in the energy-consuming 'on'-state. Figure 5b shows the improvement in energy consumption of several basic ContikiMAC operations. The RM090 benefits with a major improvement in energy consumption as the radio goes into lower power mode if no operations need to be executed, while in the default implementation it only goes to idle-mode. This difference

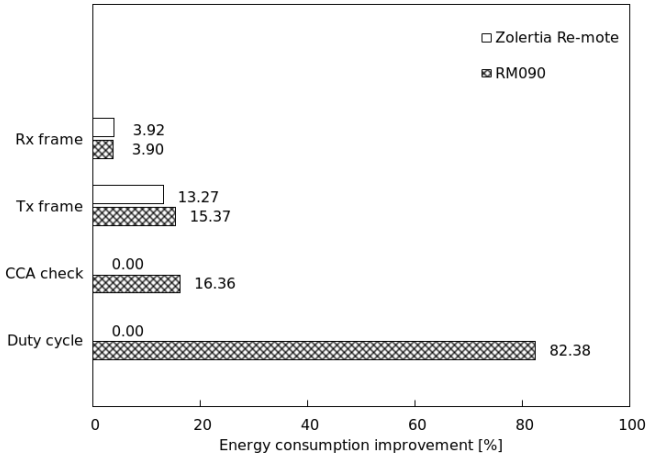
[5]It should be noted that the provided MAC API instructions have a very predictable performance (variance is less than 1µs).

becomes apparent in the duty cycle and CCA check operations in figure 5b as the energy consumption for the RM090 is considerably lower. In figure 5b it can be seen that the same improvement does not happen on the Zolertia Re-mote, since the radio can only put into a lower power mode if the MCU is also put in low power mode making it impossible to control from the MAC layer.

In contrast to ContikiMAC, TSCH is a synchronized TDMA MAC protocol. The default slot sizes described in the IEEE802.15.4 and IEEE802.15.4e standards include large margins for two main reasons: (i) to compensate clock drift on the devices, and (ii) to use multiple device types within the same network. Current TSCH implementations are compatible because the slot duration is chosen so slower radio platforms can adhere to these timings, regardless of the actual presence of such platforms in the network. The default fixed slot duration (TsSlotDuration) of 15ms is not optimal for the performance: a reduced slot duration would improve the network performance (energy consumption and throughput). Hence, the current approach towards portability does not result in optimal performance. The XML containing instructions was used to automatically calculate the minimal TSCH slot duration for a number of platforms (not including clock drift), for which the result can be observed in figure 6. The timings are either significantly shorter, or might require more then 15ms depending on the selected modulation. This demonstrates the limitations of imposing manual slot durations, which can be overcome by automatically calculating the actually needed instruction timings.




(a) Several timings of ContikiMAC were measured on both the the time annotated instruction version and the default version. This graph represents the improvement (in %) which the time annotated instructions offer compared to the default ContikiMAC.




(b) For several operations, the energy consumption was measured on both the the time annotated instruction version and the default version. This graph represents the improvement (in %) which the time annotated instructions offer compared to the default ContikiMAC.

Figure 5: Performance evaluation (timings and energy consumption) from including instruction timing information in the ContikiMAC protocol: the MAC protocol is automatically optimized for the capabilities of each individual radio platform.

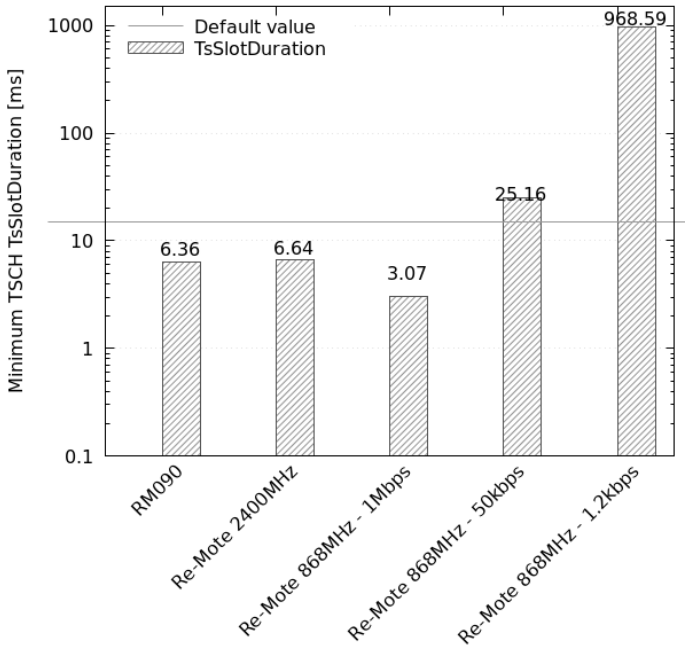



Figure 6: Automatically calculated minimum TSCH slot duration per platform. The default TSCH slot duration (15ms) is indicated by a horizontal line.

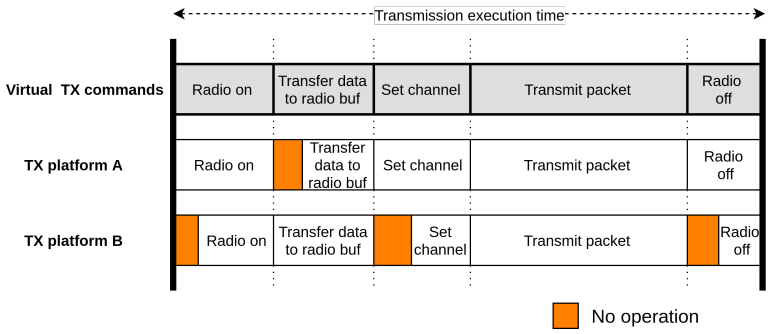
## 5. Challenge 3: Multi-platform networks exhibit unpredictable behavior.

**Context.** Previous sections focused on problems related to running the same MAC protocol code on different devices. This section describes problems that arise from having multiple devices of a different hardware type in the same network. Even when these devices run the same MAC protocol code, the performance might still differ due to time differences introduced by a different physical layer implementation and a different CPU clock speed.

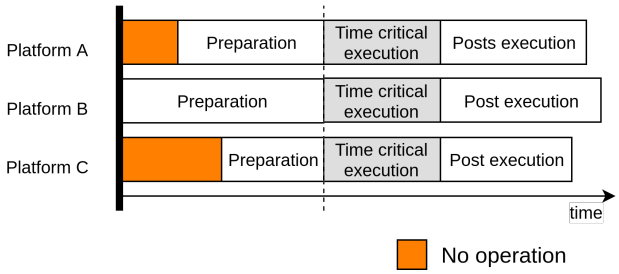
**Problem.** To demonstrate the performance impact of different platforms running the same code, an experiment was conducted where different hardware platforms send data to a central server. All nodes form a star topology, whereby the leaves were 5 RM090 and 5 Zolertia Remote devices, all using the default implementation of the ContikiMAC protocol (with a cycle time of 125ms) and the same configuration (channel, transmission power, clear channel assessment threshold, etc). All nodes were configured to transmit a single packet per second to a central node. The experiment was repeated twice, each time with a different type of device as central node. Figure 8a shows the percentage of successfully sent packets over the total per platform. Although the same software code was used, the percentage of successful packet transmissions of the Zolertia Remote is significantly higher than the ones of the RM090, since the Zolertia Remote can access the medium more quickly due to faster instruction timings (see Figure 3). As such, there is a need to ensure fairness in the presence of different hardware platforms, since this is not inherently offered by the use of the same MAC protocol.

**Solution.** To counter cross-platform incompatibility the execution duration of the different instructions should be equalized across the different platforms.

- One solution is to add wait periods to ensure that instructions require the exact same amount of time on all platforms in the network (Figure 7a). For each instruction, the corresponding duration of the worst platform in the network is used as the target duration. Figure 7a shows how this




(a) An additional margin is added after each instruction to ensure all instructions have the same duration across all platforms.




(b) A single margin is added before the full instruction sequence to ensure that the execution of the critical time components (for example TX or CCA) happens simultaneously across all platforms.

Figure 7: Approaches to equalize the MAC performance across multiple device types within the same network, thereby ensuring fairness and cross-platform compatibility.

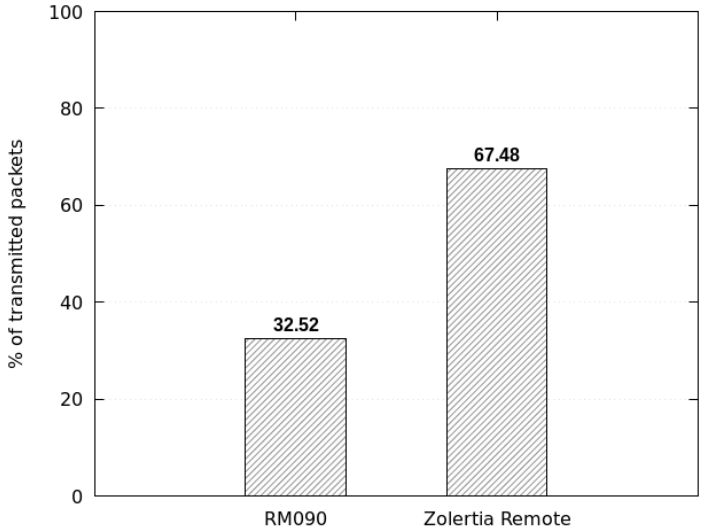
approach ensures that all platforms now finish the packet transmission in the same amount of time. By using the time-annotated commands to calculate the additional wait periods, the performance of all platforms is equalized, at the cost of extending the duration of some states on faster platforms[6].

- Alternatively, it is possible to use the timing information to reschedule essential instructions (such as TX instructions) so that they start at the

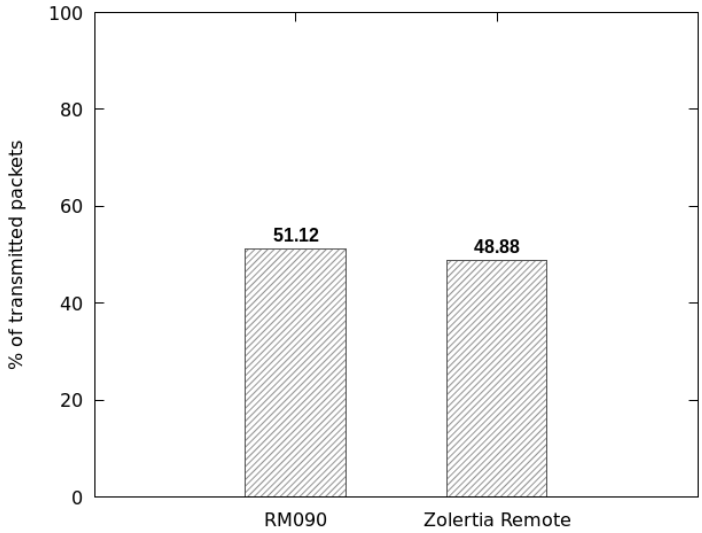
[6]For example in 7a, platform B goes to the radio on state faster compared to platform A. By doing so, the radio is already consuming energy when it is not yet strictly necessary.

same time across different platforms. Most MAC logic blocks consist of three phases: (i) a preparation phase, (ii) a critical execution, and (iii) parsing of the execution results. By adding wait periods or margins at the beginning of the MAC block, the instructions can be scheduled so that the time critical execution occurs simultaneously (Figure 7b). This approach ensures fairness and does not negatively impact energy efficiency of the individual platforms, but it does require knowledge about which parts of the execution should be aligned.

**Evaluation.** The cross-platform compatibility solutions have been evaluated by extending the TAISC framework with a "sync"-command that can be used for multi-platform designs. Every MAC protocol instruction sequence can be annotated with one such sync command indicating the critical execution part. Using these extensions time-critical parts of the MAC protocol (e.g. a frame transmission) are scheduled immediately after the sync, and the preparation for the time-critical execution (e.g. copying a packet into the radio buffer, putting the radio in the correct state) are scheduled before. It becomes possible to align critical parts of instruction blocks, e.g. put the radio in receive mode just before another node transmits a packet, without the need to calculate the timings/margins by the MAC implementer. The impact of these changes is shown in Figure 8b. Compared to the same experiment in default Contiki from Figure 8a, the platforms now share the spectrum fairly due to the alignment of the time critical execution parts.




(a) Percentage of transmitted packets over the total for two platforms running the same MAC code in a single network. The performance variation between different platforms is significant.




(b) Percentage of transmitted packets over the total for two platforms running the same MAC code in a single network. By using time annotations (Section 4) the overall performance increases. By also aligning the critical execution parts (Section 5), the fairness increases.

Figure 8: Impact of different platforms in a single network running the same MAC code on different hardware.

## 6. Conclusions

Scientific literature regarding MAC protocols currently focuses on the design and evaluation of new MAC protocol prototypes, most often without considering reuse of code towards future platforms. As a consequence, the availability of MAC protocols designed for a specific device is often very limited. Even though multi-platform MAC protocol implementations exist, performance degradation and offered functionality differing from defined/designed behavior caused by hardware differences and implementation mistakes are common. In addition, multi-platform network deployments suffer from unfairness and unpredictable performance since, as it was proven, the same MAC protocol logic on multiple devices behaves differently in terms of timing due to hardware differences. Consequently, different devices running the same MAC protocol logic can become incompatible with one another.

Table 1: Summary of how the three proposed solutions in this paper contribute to the portability, compatibility and reuse of multi-platform MAC protocols.

| **Solution** | **Portability** | **Compatibility** | **Reuse** |
|---|---|---|---|
| 1. Hierarchical framework | x | | x |
| 2. Automatic benchmarking | x | | |
| 3. Instruction aligning | | x | |

This paper described in detail the challenges related to multi-platform MAC development, and experimentally quantified how the above issues impact the MAC protocol performance when deploying and executing MAC protocols on multiple radio chips. (i) Firstly, it was shown that current interfaces are either too radio chip specific, thereby hindering portability of MAC protocols, or too general, thereby forcing developers to generate inefficient code. By providing a hierarchical framework of interfaces, our proposed methodology allows efficient code implementation while allowing fall-backs to alternative implementations for portability. (ii) Secondly, it was shown that MAC timings depend strongly on the underlying hardware. By automatically benchmarking the duration of

MAC instructions, our approach is able to optimize the MAC towards the specific radio chip timings, thereby exhibiting better performance in terms of radio energy consumption (up to 82% improvement), timings (up to 33% improvement) and throughput compared to the existing multi-platform implementations. (iii) Thirdly, we proved that the same MAC protocol implementation acts differently per platform, resulting in unfair/unpredictable behavior when multiple platforms are combined in the same network. By adding margins to equalize the duration of all radio instructions, or by aligning the critical instructions across multiple platforms, these unpredictable effects are removed and fairness is restored. Table 1 summarizes which purpose each the proposed solutions serves in the overall goal of achieving portable, multi-platform MAC protocols.

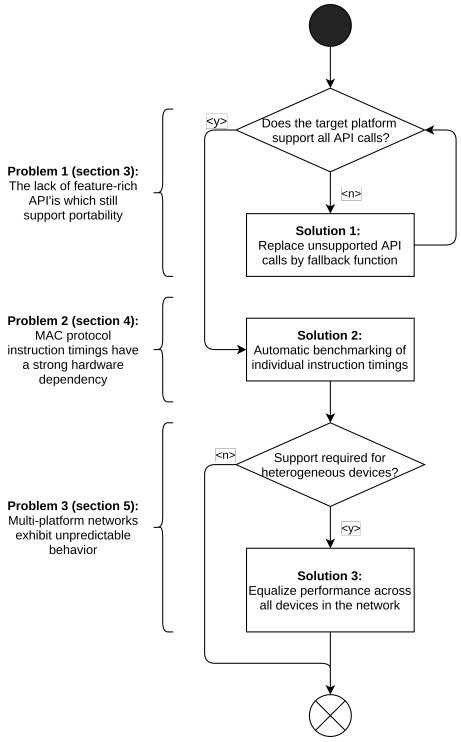



Figure 9: Flow diagram summarizing the three solutions to multi-platform issues in MAC protocols.

It is worth noting that all three solutions can be applied individually, or the can be combined with each other towards an overall methodology for the design of portable MAC protocols. The different steps of the methodology are visualized in Figure 9. By following the described steps it becomes possible to achieve a MAC design which (i) is more developer friendly, by moving the burden of time annotation towards the pre-compiler, (ii) is more efficient compared to traditional approaches, significantly improving MAC performance and energy efficiency and (iii) allows efficient reuse and combination of MAC protocols over a wide range of IoT platforms without additional development efforts.

## Acknowledgment

This work was partially supported by project SAMURAI: Software Architecture and Modules for Unified RAdIo control, and European Commission Horizon 2020 Programme under grant agreement no. 645274 (WiSHFUL).

## References

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, M. Hoffmann, Industry 4.0, Business & Information Systems Engineering 6 (4) (2014) 239–242. doi:10.1007/s12599-014-0334-4.

[2] P. Huang, L. Xiao, S. Soltani, M. W. Mutka, N. Xi, The evolution of mac protocols in wireless sensor networks: A survey, IEEE communications surveys & tutorials 15 (1) (2013) 101–120. doi:10.1109/SURV.2012.040412.00105.

[3] J. R. Cordeiro, D. F. Macedo, L. F. Vieira, Fs-mac: A flexible mac platform for wireless networks, in: Wireless Communications and Networking Conference (WCNC), 2018 IEEE, IEEE, 2018, pp. 1–6. doi:10.1109/WCNC.2018.8376981.

[4] P. Thubert, An Architecture for IPv6 over the TSCH mode of IEEE 802.15.4, Internet-Draft draft-ietf-6tisch-architecture-12, Internet Engi-

neering Task Force, work in Progress (Aug. 2017).
URL https://datatracker.ietf.org/doc/html/draft-ietf-6tisch-architecture-12

[5] J. A. Gutierrez, E. H. Callaway, R. Barrett, IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks, IEEE Standards Office, New York, NY, USA, 2003. doi:10.1109/WCNC.2003.1200605.

[6] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, T. C. Schmidt, Riot os: Towards an os for the internet of things, in: Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on, IEEE, 2013, pp. 79–80. doi:10.1109/INFCOMW.2013.6970748.

[7] A. Dunkels, B. Gronvall, T. Voigt, Contiki-a lightweight and flexible operating system for tiny networked sensors, in: Local Computer Networks, 2004. 29th Annual IEEE International Conference on, IEEE, 2004, pp. 455–462. doi:10.1109/LCN.2004.38.

[8] 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, Transactions on Emerging Telecommunications Technologies 23 (5) (2012) 480–493. doi:10.1002/ett.2558.

[9] J. Bauwens, B. Jooris, E. De Poorter, P. Ruckebusch, I. Moerman, towards a mac protocol app store, in: EWSN 2016, the International Conference on Embedded Wireless Systems and Networks, 2016, pp. 1–2. doi:1854/LU-7172061.

[10] J. G. Ko, N. Tsiftes, A. Dunkels, A. Terzis, Pragmatic low-power interoperability: Contikimac vs tinyos lpl, in: Sensor, mesh and ad hoc communications and networks (SECON), 2012 9th annual IEEE communications society conference on, IEEE, 2012, pp. 94–96.

[11] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-Haggerty, J.-P. Vasseur, M. Durvy, A. Terzis, A. Dunkels, D. Culler, Industry: beyond interoperability: pushing the performance of sensor network ip stacks, in: Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems, ACM, 2011, pp. 1–11.

[12] J. Vanhie-Van Gerwen, E. De Poorter, B. Latré, I. Moerman, P. Demeester, Real-life performance of protocol combinations for wireless sensor networks, in: 2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, IEEE, 2010, pp. 189–196.

[13] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-Haggerty, A. Terzis, A. Dunkels, D. Culler, Contikirpl and tinyrpl: Happy together, in: Workshop on Extending the Internet to Low Power and Lossy Networks (IP+ SN), 2011.

[14] S. Bouckaert, W. Vandenberghe, B. Jooris, I. Moerman, P. Demeester, The w-ilab. t testbed., in: TridentCom, Vol. 46, 2010, pp. 145–154. doi:10.1007/978-3-642-17851-1_11.

[15] M. Uwase, M. Bezunartea, J. Tiberghien, J. Dricot, K. Steenhaut, Experimental comparison of radio duty cycling protocols for wireless sensor networks, IEEE Sensors Journaldoi:10.1109/JSEN.2017.2738700.

[16] M. Brzozowski, P. Langendoerfer, Is cross-platform protocol stack suitable for sensor networks? empirical evaluation, in: Wireless and Mobile Networking Conference (WMNC), 2013 6th Joint IFIP, IEEE, 2013, pp. 1–8. doi:10.1109/WMNC.2013.6548983.

[17] M. Brzozowski, H. Salomon, K. Piotrowski, P. Langendoerfer, Cross-platform protocol development for sensor networks: lessons learned, in: Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications, ACM, 2011, pp. 7–12. doi:10.1145/1988051.1988054.

[18] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, Mantis os: An embedded mul-

tithreaded operating system for wireless micro sensor platforms, Mobile Networks and Applications 10 (4) (2005) 563–579. doi:10.1007/s11036-005-1567-8.

[19] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, Wireless mac processors: Programming mac protocols on commodity hardware, in: INFOCOM, 2012 Proceedings IEEE, IEEE, 2012, pp. 1269–1277.

[20] B. Jooris, J. Bauwens, P. Ruckebusch, P. De Valck, C. Van Praet, I. Moerman, E. De Poorter, Taisc: A cross-platform mac protocol compiler and execution engine, Computer Networks 107 (2016) 315–326. doi:10.1016/j.comnet.2016.03.027.

[21] A. Dunkels, The contikimac radio duty cycling protocol, Swedish Institute of Computer Science, 2011.