Lessons Learned through Implementation and Performance Comparison of Two MAC/RDC Protocols on Different WSN OS

Kévin Roussel, Ye-Qiong Song, Olivier Zendra

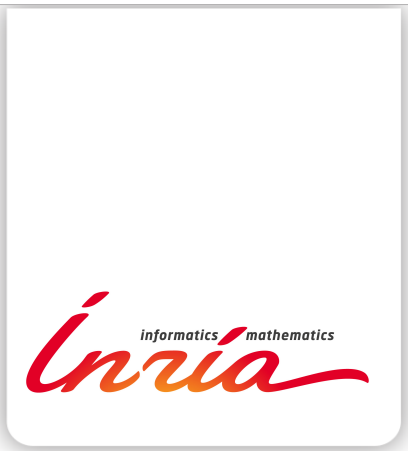
RESEARCH REPORT

N° 8777

March 2015

Project-Team Madynes

ISSN 0249-6399 ISRN INRIA/RR--8777--FR+ENG



Lessons Learned through Implementation and Performance Comparison of Two MAC/RDC Protocols on Different WSN OS

Kévin Roussel, Ye-Qiong Song, Olivier Zendra

Project-Team Madynes

Research Report n° 8777 — March 2015 — 25 pages

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Abstract: Implementing new, high-performance MAC/RDC protocols on WSN/IoT motes is a complex and requiring task; to do it efficiently, one has to encounter many challenges that need to be overcome using the best trade-off between various and often contradictory objectives.
A first key point is the software platform used for implementation: many specialized OSes for WSN motes are available, each one with its own set of features. What are the most important features an OS can offer when trying to implement a new protocol into its network stack? What software platform did we choose according to these requirements?
A second point is the availability of development tools facilitating implementation and debugging. Emulators and simulators are such tools. They dramatically help to develop and debug WSN/IoT software. Are they also adequate for performance evaluation of MAC/RDC protocols?
Finally, what is the impact of implementation choices on the performance of the final software? To what extent does optimization influence actual results during evaluation?
We propose, in this report, our answers to these questions; answers we had to give while building and testing an implementation of our own MAC/RDC protocol.

Key-words: MAC/RDC protocols, WSN, dedicated OS, network stacks, implementation, performance, evaluation

Leçons apprises via l'implémentation et la comparaison des performances de deux protocoles MAC / RDC sur différents systèmes d'exploitation pour réseaux de capteurs sans-fil

Résumé : L'implémentation de nouveaux protocoles MAC / RDC à hautes performances pour les noeuds de réseaux de capteurs sans-fil (WSN) et de l'Internet des Objets (IoT) est une tâche complexe et exigeante ; pour la réaliser efficacement, il est nécessaire de faire face à de nombreux défis à surmonter en trouvant le meilleur compromis entre des objectifs souvent contradictoires.

Un premier point-clé est la plate-forme logicielle utilisée pour l'implémentation : de nombreux OS spécialisés pour les WSN sont disponibles, chacun ayant son propre ensemble de fonctionnalités. Quelles sont les fonctionnalités les plus importantes qu'un OS doit offrir pour faciliter l'implémentation d'un nouveau protocole dans sa pile réseau ? Quelle plate-forme logicielle avons nous choisie en fonction de ces besoins ?

Un deuxième point-clé est la disponibilité d'outils de développement facilitant l'implémentation et le déboguage. Les émulateurs et simulateurs font partie de cette catégorie d'outils. Ils aident grandement au développement et au déboguage de logiciels pour les WSN et l'IoT. Sont-ils également adéquats pour l'évaluation des performances des protocoles MAC / RDC ?

Enfin, quel est l'impact des choix d'implémentation sur les performances du logiciel terminé ? Jusqu'à quel point l'optimisation influence-t-elle les résultats réels obtenus durant l'évaluation ?

Nous proposons, dans ce rapport, nos réponses à ces questions ; réponses que nous avons dû donner durant l'implémentation et les tests de notre propre protocole MAC / RDC.

Mots-clés : protocoles MAC / RDC, réseaux de capteurs sans-fil, OS dédiés, piles réseau, implémentations, performances, évaluation

1 Introduction

In the present report, we describe the various difficulties we had to encounter when implementing a new *Radio Duty Cycle* (RDC) protocol (S-CoSenS, described hereafter), as well as the solutions we found for these problems.

The questions we had to answer were the following:

- *Software Platform:* what features are needed in WSN/IoT operating systems to implement efficiently new high-performance MAC/RDC protocols?
- *Development Tools:* Emulators and simulators are great tools for helping in the development and debugging of such protocols. Are they also fit for other uses, like performance evaluation of software implementations?
- *Implementation Best Practices:* what trade-offs have to be made between optimization on the one hand, and safety and modularity on the other hand when actually implementing these protocols? What is the impact of such trade-offs on the performances of the implementation?
- *Evaluation of Implementations:* what are these implementations' performances, especially in terms of QoS and energy savings?

We will—as our contribution—analyze our answers to these various questions.

While evaluation and comparison of MAC/RDC protocols is quite common, the S-CoSenS protocol itself having been already compared in previous literature [1], the originality of the present paper is to list the obstacles that had to be overcome to fulfill the implementation and specific evaluation of this protocol against ContikiMAC.

Thus the remainder of the paper is organized as follows:

- Section II will discuss about software platforms, the properties we wish them to provide us, and which one we choosed.
- Section III describes the two RDC protocols we wished to evaluate and compare.
- Section IV describes the experimental setup used for our simulations.
- Section V focus on Cooja, and explains why it is not suited as an performance evaluation tool.

- Section VI shows the results we obtained during our simulations, details some problems we encountered during them, and how they can be avoided or overcome.
- Finally, we will conclude by summarizing the contributions made in the present paper.

2 WSN Software platforms

Specialized OSes for the resource-constrained devices that constitute wireless sensor networks have been designed, published, and made available for quite a long time.

The current reference OS in the domain of WSN and IoT is *Contiki* [2]. It is an open-source OS, which was first released in 2002. It is also at the origin of many assets: we can mention, among many others, the low-power Rime network stack [3], or the Cooja advanced network simulator [4].

Contiki is very lightweight and well adapted to motes and resource-constrained devices. It is coded in standard C language, which makes it relatively easy to learn and program. It offers an event-based kernel, implemented using cooperative multithreading, and a complete network stack. All of these features and advantages have made Contiki widespread, making it the reference OS when it comes to WSN.

Contiki developers also have made advances in the MAC/RDC domain: many of them have been implemented as part of the Contiki network stack, and a specifically developed, ContikiMAC, has been published in 2011 [5] and implemented into Contiki as the default RDC protocol (designed to be used with standard CSMA/CA as MAC layer).

Consequently, we naturally decided, as a comparison reference our first tests, to use the Contiki software platform: the ContikiOS, the ContikiMAC RDC protocol, the standard CSMA/CA MAC layer, and the Rime network layer, used on the IEEE 802.15.4 physical wireless/radio protocol.

However, Contiki's extremely compact footprint and high optimization comes at the cost of some limitations that prevented us from using it as our software platform.

Contiki OS is indeed not a real-time OS: the processing of "events"—using Contiki's terminology—is made by using the kernel's scheduler, which is based on cooperative multitasking (using "protothreads"; this also implies restrictions on the use of the `switch` construct of the C language). This scheduler only triggers at a specific, pre-determined rate; on the platforms we're interested in, this rate is fixed to 128 Hz: this corresponds to a time skew

of up to 8 milliseconds (8000 microseconds) to process an event, interruption management being one of the possible events. Such a large granularity is clearly a huge problem when implementing high-performance MAC/RDC protocols, where a time granularity of 320 microseconds is needed, corresponding to one backoff period (BP). The only "real-time-oriented" feature of the system, `rtimer` does not solve these problems, because of its deep limitations:

- only one instance is available;
- it is impossible to call most Contiki functions—like those from the kernel or the network stack—from the `rtimer` callbacks, trying to do so provoking crashes or unexpected behavior.

Lesson 1. *For high performance MAC/RDC protocols, we have to synchronize precisely neighbouring nodes: the leaf nodes with their router, and the router with its sink.*

To that end, we definitely need an OS with true real-time features (which basically implies pre-emptive multitasking), as well as fine granularity (the 32 KHz rate offered by `rtimer` is just fine) for event handling mechanism. Being able to leverage many instances of such real-time mechanisms would be a great advantage.

For all these reasons, we were unable to use Contiki OS to develop and implement our high-performance MAC/RDC protocols.

Consequently, we focused our interest on *RIOT OS* [6].

This new system—first released in 2013—is also open-source and specialized in the domain of low-power, embedded wireless sensors. It offers many useful features, that we will now describe.

It provides the basic benefits of an OS: portability (it has been ported to many devices powered by ARM, MSP430, and—more recently—AVR microcontrollers) and a comprehensive set of features, including a network stack.

Lesson 2. *Moreover, it offers a set of key features that is otherwise uncommon in the WSN/IoT domain, the most important for us being:*

- *an efficient, interrupt-driven, tickless micro-kernel providing preemptive multitasking;*
- *a highly efficient use of* hardware timers*: all of them can be used concurrently (especially since the kernel is tickless), offering the ability to schedule actions with high precision; on low-end devices, based on MSP430 architecture, events can be scheduled with a resolution of 30 microseconds, instead of 8000 microseconds for Contiki OS.*

These features are what make RIOT a full-fledged real-time *operating system, that fulfills our needs.*

Thus, RIOT is a powerful real-time operating system, adapted to the limitations of deeply embedded hardware microcontrollers, while offering state-of-the-art techniques (preemptive multitasking, tickless scheduler, optimal use of hardware timers) that—we believe—makes it one of the most suitable OSes of the embedded and real-time world.

3 Protocols

To evaluate our work, we first tried to evaluate and compare two implementations of MAC/RDC protocols. The first is the well-known implementation of the default RDC protocol for the currently most used OS in the WSN/IoT literature. The second is the implementation of one of our new designs in RDC protocols on the software platform we choosed.

3.1 The ContikiMAC RDC protocol

It is currently the default RDC protocol proposed by the Contiki OS. This protocol is designed to work under the classical CSMA/CA MAC layer, and is based on the *Low-Power Listening* (LPL) paradigm.

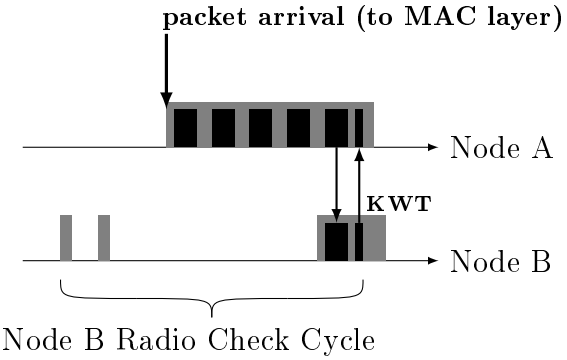


Figure 1: Transmission of a data packet with the ContikiMAC protocol.
Gray boxes represent radio transceiver activation, while black boxes represent packets' transmissions.
KWT is the keep-in-wakeup timeout delay used for handling burst transmissions reception

As shown in figure 1, it is technically a derivative of the classical X-MAC protocol [7]: its basic principle, fundamentally asynchronous, is that every

node keeps its radio transceiver off as much as possible, only waking it up for short delay of radio medium listening at a fixed rate (by default, eight times per second in ContikiMAC). If, during the short amount of time the radio transceiver is on, data is sensed on the medium, the radio transceiver is kept on until the whole transmission is done; otherwise, the node goes back into "radio silence mode" to save energy. This ability to automatically adapt the duration of transceiver activity to the live traffic on radio channel is what makes ContikiMAC a *traffic auto-adaptative* (or *self-adaptative*) protocol, at least up to a certain extent.

When a node has to emit a packet, it continuously emits it until the intended receiver nodes "wakes up" its radio transceiver, receives the transmitted data, and sends back an acknowledgement.

The first specificity of ContikiMAC is that, during transmission, instead of emitting a specific preamble until the receiver is ready, then sending it the actual data packet, the data packet is directly emitted by the sender—it is its own preamble—so that receiver(s) can directly receive and acknowledge the data packet as soon as its radio is ready.

The second specificity is the presence of an optimization mechanism named *"transmission phase lock"*: senders note the time when they receive acknowledgement from receivers, and according to the known rate of radio wake-up, deduce the subsequent expected times when these receivers are supposed to be ready to receive packets. Thus, senders can synchronize with nodes that have previously received their transmissions, and can send their next packets at the optimal period to minimize the useless transmission of packets when the intended receiver is offline. The result of this mechanism is shown in figure 2.

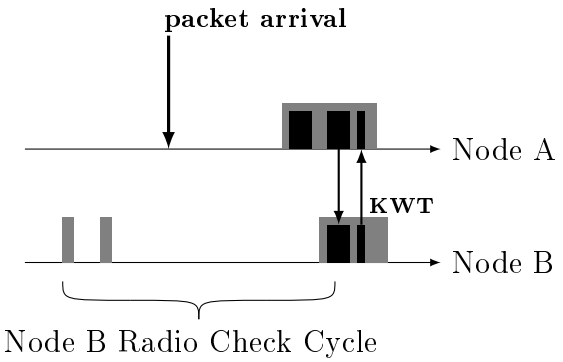


Figure 2: "Transmission Phase Lock": After the transmission in figure 1 occured, node A knows when to send packets so that node B is ready to receive them, even though both nodes have different radio check cycle phases.

The third feature specific to ContikiMAC is the ability to send in a single *burst* a whole queue of packets when those are all destined to a same receiver. This is implemented thanks to a keep-in-wakeup timeout period (KWT) observed by ContikiMAC after each packet reception (as shown in figures 1 and 2).

3.2 Our first design: the S-CoSenS RDC protocol

The first RDC protocol we wanted to implement is S-CoSenS [8], which is designed to work with the IEEE 802.15.4 MAC and physical layers. Like ContikiMAC, it is designed to work with the classical (i.e.: beacon-less) CSMA/CA MAC layer.

It is an evolution of the already published CoSenS protocol [9]: it adds to the latter a sleeping period for energy saving. Thus, the basic principle of S-CoSenS is to delay the forwarding (routing) of received packets, by dividing the RDC cycle in three periods: a sleeping period (SP), a waiting period (WP) where the radio medium is listened by routers for collecting incoming 802.15.4 packets, and finally a burst transmission period (TP) for emitting adequately the packets enqueued during WP.

The main advantage of S-CoSenS is its ability to adapt dynamically to the wireless network throughput at runtime, by calculating for each radio duty cycle the length of SP and WP, according to the number of relayed packets during previous cycles. Note that the set of the SP and the WP of a same cycle is named *subframe*; it is the part of a S-CoSenS cycle whose length is computed and known *a priori*; on the contrary, TP duration is always unknown up to its very beginning, because it depends on the amount of data successfully received during the WP that precedes it.

Moreover, the computation of WP duration follows a "sliding average" algorithm, where WP average for each cycle is computed as:

$$\overline{\mathrm{WP}_n} = \alpha \cdot \overline{\mathrm{WP}_{n-1}} + (1-\alpha) \cdot \mathrm{WP}_{n-1}$$
$$\mathrm{WP}_n = \max(\mathrm{WP}_{min}, \min(\overline{\mathrm{WP}_n}, \mathrm{WP}_{max}))$$

where $\overline{\mathrm{WP}_n}$ and $\overline{\mathrm{WP}_{n-1}}$ are respectively the average WP length at n^{th} and $(n-1)^{\mathrm{th}}$ cycle, while WP_n and WP_{n-1} are the actual length of respectively the n^{th} and $(n-1)^{\mathrm{th}}$ cycles; α is a parameter between 0 and 1 representing the relative weight of the history in the computation, and WP_{min} and WP_{max} are high and low limits imposed by the programmer to the WP duration.

The local synchronization between a S-CoSenS router and the leaf nodes that take it as parent is done thanks to a beacon packet, that is broadcasted by the router at the beginning of each cycle. This beacon contains the duration (in microseconds) of the SP and WP for the currently beginning cycle.

The whole S-CoSenS cycle workflow for a router is summarized in figure 3 hereafter.

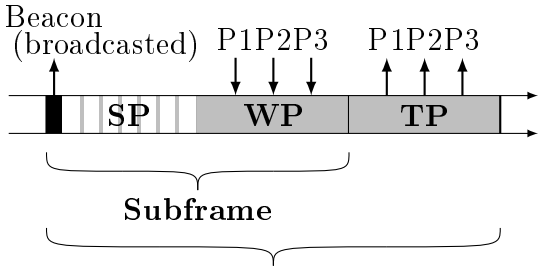


Figure 3: A typical S-CoSenS router cycle.
The gray strips in the SP represents the short wake-up-and-listen periods used for inter-router communication.

Note that in contrast to ContikiMAC, S-CoSenS is based on the *Receiver Initiated* (RI) paradigm, whose the classical RI-MAC protocol [10] is one of the better known application. Local synchronization between a S-CoSenS router and its leaf nodes, done as seen hereabove with the regular emission of a beacon, relies on the *Low-Power Probing* (LPP) paradigm. On the other hand, synchronization and communication between different S-CoSenS routers, made thanks to short regular wake-up-and-listen periods, is based on the Low-Power Listening (LPL) paradigm (like, among others, the B-MAC protocol [11]).

An interesting property of S-CoSenS is that leaf (i.e.: non-router) nodes always have their radio transceiver off, except when they have packets to send. When a data packet is generated on a leaf node, the latter wakes up its radio transceiver, listens and waits to the first beacon emitted by an S-CoSenS router, then sends its packet using CSMA/CA at the beginning of the WP described in the beacon it received. A leaf node will put its transceiver offline during the delay between the beacon and that WP (that is: the SP of the router that emitted the received beacon), and will go back to sleep mode once its packet is transmitted. All of this procedure is shown in figure 4.

We thus need to synchronize with enough accuracy different devices (that can be based on different hardware platforms) on cycles whose periods are dynamically calculated at runtime, with resolution that needs to be in the sub-millisecond range. This is where RIOT OS advanced real-time features really shine, while the other comparable OSes are for that purpose definitely lacking.

We have implemented S-CoSenS under RIOT OS, and evaluated it by comparing to the ContikiMAC + CSMA/CA couple under Contiki OS.

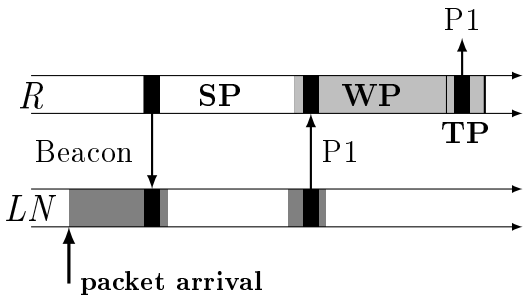


Figure 4: A typical transmission of a data packet with the S-CoSenS protocol between a leaf node and a router.

As there was no standard implementation in RIOT OS network stack of the CSMA/CA described in the 802.15.4 protocol, our implementation of S-CoSenS currently also includes that MAC layer. This shall be changed in future production-ready implementations.

4 Experimental Setup

For our first experiments, we used, for practical reasons, the Cooja simulator rather than actual hardware. All the simulated nodes are virtual Zolertia Z1 motes, well-known industrial MSP430-based devices.

We have implemented test applications under both Contiki and RIOT OS, and made first tests by performing simulations of a basic 802.15.4 PAN (Personal Area Network) constituted of a "router", and ten motes acting as "leaf nodes". The ten nodes regularly send data packets to the router, that retransmits these data packets to a nearby "sink" device. Both the router and the ten nodes use exclusively the S-CoSenS RDC/MAC protocol. This is summarized in figure 5.

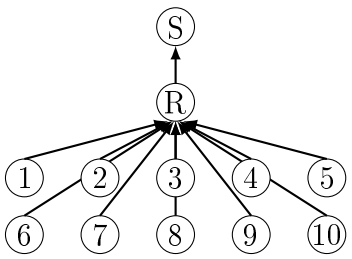


Figure 5: Functional schema of our virtual test PAN.

We then performed simulations on this virtual PAN, varying several pa-

rameters of the ContikiMAC RDC protocol, under moderately-heavy to extreme network loads. The loads were generated by the application loaded on the 10 leaf nodes: they were programmed to generate large 802.15.4 network packets (90 bytes of payload, which translates to an actual packet size of 110 bytes) at a fixed rate, with random jitters in the interval of [0.5 PAI, 1.5 PAI], with PAI being the targeted Packet Arrival Interval period (i.e.: the inter-arrival between two consecutive packets sent by each mote). The different setups are described in table 1: the table gives the targeted average PAI, the average number of packets emitted every second for all of the 10 leaf nodes, and the expected resulting data rate.

Setup	Inter-Arrival	Packets/s	Data Rate
Moderate	1500 ms	6.7	5,867 bit/s
High	1000 ms	10	8,800 bit/s
Very High	500 ms	20	17,600 bit/s
Extreme	100 ms	100	88,000 bit/s

Table 1: Transmission data rates used on leaf nodes.

Considering the overhead of MAC/RDC layer, the two-hop transmission route—which means that the router has at a given time to communicate with either leaf nodes, either sink; thus effectively dividing bandwidth by two—, the large size of our 802.15.4 packets, we expect the "very high" scenario to be near the maximum effective data rate possible, and the "extreme" scenario to be well beyond channel exhaustion.

5 Cooja: a tool for development, not evaluation

The Cooja tool provided by the Contiki project [4] is very handy for developement and debugging. It is a convivial GUI that allows to perform simulations of whole networks or wireless devices, and embeds emulators that allow to mimic quite accurately motes based on MSP430 and AVR architectures.

It has been, during our development effort, extremely helpful, especially for advanced debugging purposes, where it has proven to be at least as useful as hardware JTAG debuggers. We can thus regret that there are no such emulators for other microcontroller architectures—like ARM—as they would also undoubtly dramatically facilitate development for such platforms.

Lesson 3. *However, we discovered Cooja is not suited for precise and accurate performance evaluations.*

We constated, during our simulations, an anomaly in the MSPSim software (the MSP430 device simulator used by Cooja to emulate MSP430-based motes at the cycle level [12]): *the delay used by the simulator to load our large—110 bytes—IEEE 802.15.4 packets into the CC2420 transceiver TX buffer is too long;* Table 2 shows the differences of delay for loading our packets between Cooja simulations and actual hardware executions.

Operating System / SPI Driver	Cooja Sim.	Hardware Test
RIOT OS / Standard (Safe)	89 ticks	32 ticks
RIOT OS / Fast SPI Write	36 ticks	14 ticks
Contiki OS / Fast SPI Write	14 ticks	7 ticks

Table 2: Delays observed for loading packets into CC2420 TX buffer, using various software platforms and implementations.
Values are given in "ticks" of `rtimer/hwtimer`, both of these timers incrementing at a rate of 32,768Hz. The unit is thus a fixed period of time equal to about 30.5 microseconds.

We have thus determined that this problem occurs whatever OS and implementation is used on the motes. This problem seems to be caused by an overestimation of the delays caused by the operation of the SPI bus that links the MSP430 microcontroller to the CC2420 radio transceiver on the Z1 hardware.

6 Results and Discussion

6.1 Packet Reception Rate (PRR)

We use the PRR as the first indicator to quantify the QoS obtained by using the examined protocols.

Thus, we were able to determine that *the only parameter common to both protocols* that has actually influence on the PRR is the length of the subframe on S-CoSenS, whose equivalent in the ContikiMAC RDC protocol is the *wake-up interval* (or *check interval*). This interval is actually configured, at the implementation level, by changing the rate at which the radio medium is sensed (this parameter, in Contiki source code, is a `#define` named `NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE`). Its default value is 8, meaning the radio channel is checked eight times per second; this parameter is actually given in Hertz.

S-CoSenS also offers another parameter that allows to finely tune the balance between QoS and duty cycle: the WP_{min} and WP_{max} values that

allow to limit the WP weight in the S-CoSenS cycles to stay within a specific interval. More specifically, increasing the value of WP_{min} allows to ensure that PRR will never go below a high level. For our simulations, since we're primarily interested in maintaining an high QoS, we systematically set $\mathrm{WP}_{max} = \mathrm{PAI}$ and $\mathrm{WP}_{min} = 0.5 \cdot \mathrm{PAI}$, with PAI being the packet arrival interval for the given simulation.

Note that we also allowed, on Contiki, the the standard CSMA/CA MAC layer to perform up to seven retries for a given data packet, by setting the `CSMA_CONF_MAX_MAC_TRANSMISSIONS` parameter to 8 in Contiki source code. This was done in order to put it on par with S-CoSenS default configuration, where a packet can by default be transmitted up to 8 times before being cancelled. Attention: the "retry" term does not have the same signification under S-CoSenS—where it specifies the reemission of *one* instance of a packet—than on ContikiMAC where it specifies a new cycle during which the packet is reemitted an unspecified number of times (equivalent to the number of strobes in the X-MAC protocol). This is the consequence of the fundamental difference in design between the two protocols (see section 3 above).

This default value of 8 Hz/125 ms only gives very poor results for ContikiMAC. To obtain fairer results for this latter protocol, we changed that parameter's value, and doubled it up to 32 Hz/31 ms.

The results we obtained are shown in figures 6 to 9. These figures show the rates of packets successfully arrived to their destination, according to the cycle duration (with ContikiMAC: via changing the channel check rate parameter). These results are obtained with fixed packet number simulations.

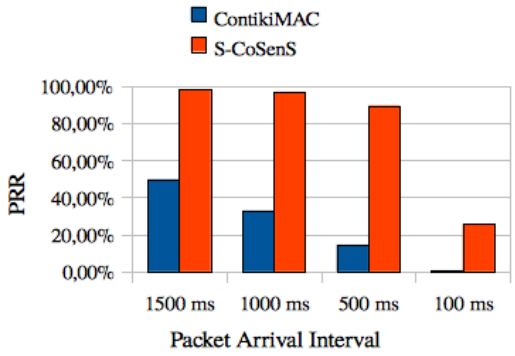


Figure 6: PRR results for 125 ms subframe/check interval

The data shown in figures 6 to 9 show us many facts that constitute the next learned lesson:

Lesson 4. *We observe that:*

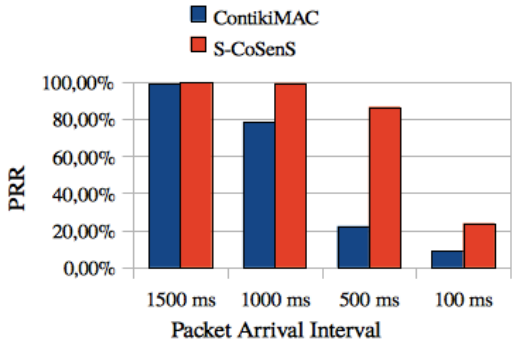


Figure 7: PRR results for 62 ms subframe/check interval

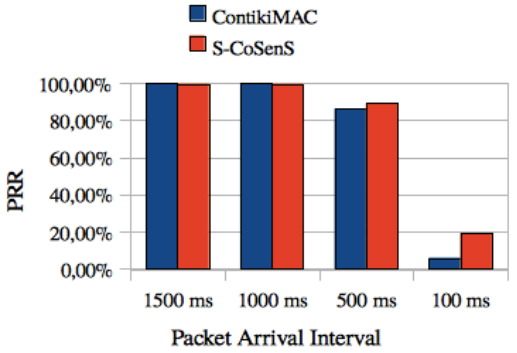


Figure 8: PRR results for 31 ms subframe/check interval

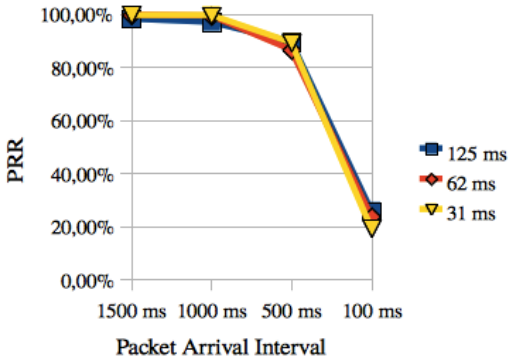


Figure 9: Combined PRR results for S-CoSenS, according to subframe duration

- *ContikiMAC PRR increases significantly with the Channel Check Rate (that is: when Channel Check Interval diminishes).*

- *S-CoSenS PRR results are either similar or better than ContikiMAC's in every scenario.*

- *S-CoSenS PRR results are much more stable when the subframe duration changes than ContikiMAC PRR when changing check rate: this stability is clearly visible on figure 9; thus, S-CoSenS performance is much more predictable, and gives the insurance of a fair PRR in most situations—that is: when radio channel is not saturated like in our "extreme" scenario).*

Analysis of the detailed results including the number of retries needed to transmit each successful packet (not given in this report for space reasons) has also shown us that :

- With ContikiMAC, the packet losses are mainly due to the router.

- With S-CoSens, the router never loses the packets it receives: all losses occur between leaf nodes and router.

This is not surprising, since S-CoSenS has an explicitly defined transmission period (TP) in every cycle for packet (re)transmission, while ContikiMAC doesn't provide such a feature.

During that TP, a "probabilistic priority" is given by the protocol to routers by giving them a lower base interval for backup calculation (i.e.: the $MacMinBE$ parameter defined by 802.15.4 standard) than for other (leaf) nodes: for routers, $MacMinBE = 2$ while for other nodes $MacMinBE = 3$. During TP, routers are thus priviledged for transmitting the packets in their output queue. Moreover, the leaf nodes that have received the beacon of a given router will restrain from emitting packets during that router's TP, since they have the knowledge of the period during which the latter is in reception during its waiting period (WP).

We can clearly understand that on the contrary, under heavy network load, a ContikiMAC router will have data to receive almost every time it wakes up its radio transceiver; thus, it will become increasingly difficult for this router to be able to retransmit the packet it receives with each increase of the network traffic. That's why under such scenarii, the router node is clearly the weak point for ContikiMAC setups, while S-CoSenS manages to avoid these difficulties by design.

6.2 End-to-end Transmission Delays

We computed the delays that takes each successfully transmitted packet to go from its originating leaf node up to the sink—that is: the duration of its two-hop transmission. This is the second indicator for quantifying QoS we studied in our experiments.

The results we obtained are shown in figures 10 to 13. These figures show the mean delay for a successfully transferred packet to make all of its two-hop trip. Note that figures 10, 11 and 12 show the delays—in their vertical axis—in logarithmic scale, for better readability.

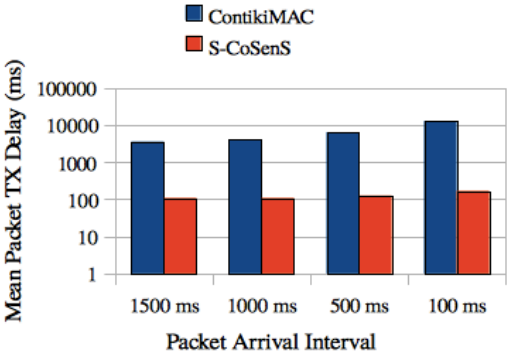


Figure 10: Transmission delay means for 125 ms subframe/check interval

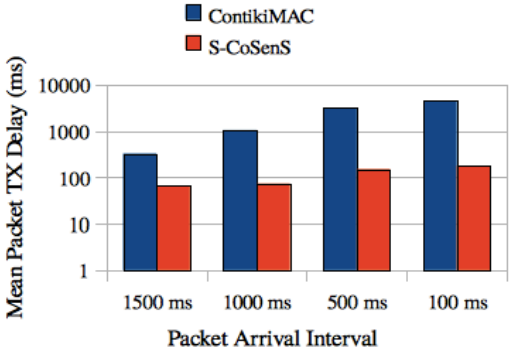


Figure 11: Transmission delay means for 62 ms subframe/check interval

Lesson 5. *These results also indicate that:*

- *ContikiMAC delays shorten with Channel Check Rate.*
- *S-CoSenS delays are always similar or shorter than ContikiMAC's.*

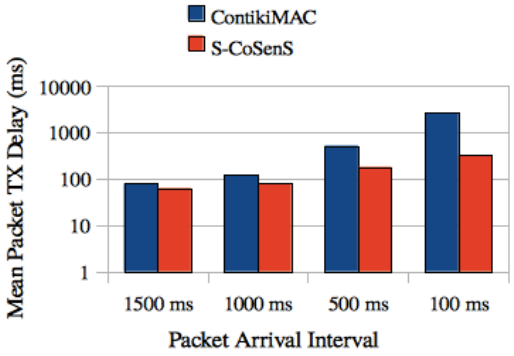


Figure 12: Transmission delay means for 31 ms subframe/check interval

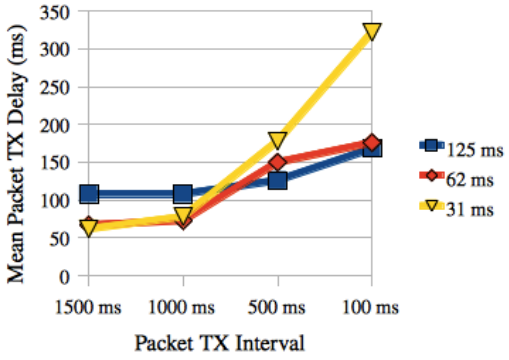


Figure 13: Combined Transmission delay averages for S-CoSenS, according to subframe duration

- *S-CoSenS delays are much more stable when the subframe duration changes. In most cases (i.e.: when the radio channel is not saturated), S-CoSenS gives the insurance that packet transmission delays will keep under a reasonable limit.*

These observation are strictly similar to those made for PRR results.

This very large increase in end-to-end transmission delays for ContikiMAC can be interpreted as "overzealousness": a packet can be retransmitted during a fixed number of retries (8 in our setups, 3 by default in Contiki); since for ContikiMAC+CSMA/CA, an attempt to transmit a packet can be delayed up until the radio medium is available, and consists in trying to send a packet repetitively up to a whole check interval cycle, a enormous amount of time is elapsed before the protocol finally gives up on transmitting a given packet.

While such an "obstinacy" can improve PRR by increasing transmission success rate, it also eventually greatly increases the mean delay for end-to-end packet transmission. In the end, one can wonder about the interest of transmitting packets with such a great age, at the expense of transmitting packets with more recent and meaningful data.

The configuration of the amount of retries to perform for a given transmission is—like many other parameters—a balance between two contradictory goals: the aim is to have a sufficient PRR, while also allowing the loss and discarding of too old data that have lost its interest. It seems that in the scenarii we studied, S-CoSenS manages to obtain a better trade-off between these two goals than ContikiMAC.

6.3 Duty Cycles

Cooja can easily compute the accurate duty cycle statistics for all of the virtual motes simulated. These statistics are a strong indicator for quantifying energy consumption, since the radio transceiver is, by far, the most energy-consuming hardware in the motes that constitute WSNs.

The results we obtained for duty cycle statistics are presented in tables 3 and 4. Note that due to the differences in PRR, only comparing duty cycle results for a channel check interval/subframe duration of 31 ms makes sense.

Lesson 6. *These duty cycle results clearly show that ContikiMAC has here a significant advantage. Transceiver activity rates and reception time ("radio RX") are always lower—that is: better—for ContikiMAC than for S-CoSenS. Especially for transceiver activity on router nodes, the difference is particularily dramatic.*

	PACKET ARRIVAL INTERVAL		
	1500 ms	1000 ms	500 ms
Channel Check Interval = 31 ms			
ROUTER			
Transceiver Active	12.77%	17.85%	31.13%
Radio TX	2.64%	4.00%	6.88%
Radio RX	2.78%	4.18%	8.24%
Radio Interference	0.16%	0.40%	2.64%
NODES (average)			
Transceiver Active	5.61%	7.45%	14.27%
Radio TX	0.90%	1.48%	3.89%
Radio RX	0.66%	1.03%	1.96%
Radio Interference	0.11%	0.21%	1.13%

Table 3: Duty Cycle Statistics for ContikiMAC.

	PACKET ARRIVAL INTERVAL		
	1500 ms	1000 ms	500 ms
Subframe Duration = 31 ms			
ROUTER			
Transceiver Active	66.22%	67.56%	67.91%
Radio TX	4.96%	5.48%	8.43%
Radio RX	3.84%	4.43%	8.90%
Radio Interference	1.04%	1.48%	8.39%
NODES (average)			
Transceiver Active	6.15%	7.68%	40.04%
Radio TX	0.58%	0.70%	2.64%
Radio RX	0.68%	0.93%	7.08%
Radio Interference	0.16%	0.22%	3.98%

Table 4: Duty Cycle Statistics for S-CoSenS.

ContikiMAC proves here that it is indeed a *very low power protocol.*

The only mitigation to this constatation comes from the transmission time rate ("radio TX"): for the leaf nodes—but not the routers—they are systematically lower/better for S-CoSenS than for ContikiMAC. This is easily explained by the design of both these protocols: S-CoSenS being designed using the RI paradigm, it naturally implies less packet transmissions than ContikiMAC, which is designed using the LPL paradigm (and more especially, the continuous emission of the whole data packets until acknowledgement by the receiver).

6.4 Memory Constraints and Stability

During the development and the evaluation of our protocol, we encountered some crashes, especially when using large values for outgoing packet queues. These crashes were due to system stack overflowing and going into the packet queue memory space: when the outgoing queue was full, packet data were thus regularly overwriting system stack, resulting of course in unrecoverable software failures.

Lesson 7. *The crashes observed with S-CoSenS/RIOT are due to the very limited amount of data RAM in the MSP430 used in Z1 motes (8 Kb RAM total), combined with the absence of* Memory Protection Unit *(MPU) in this microcontroller architecture, thus allowing for silent and undetectable memory corruption problems.*

While using microcontrollerss with more RAM can probably avoid this kind of problem from occuring, detecting and handling it correctly and systematically needs either:

- *a hardware mechanism* to detect it and provoke an exception when unwanted memory accesses occur: this is the role, in microcontrollers, of MPU; some architectures (like ARM) offer such a feature as an option which should be seen as necessary for the design of safe systems;

- for architectures devoid of such mechanisms, a *software support* for surveying the value in criticial registers—such as the stack pointer—simply if nothing more to ensure it is within a permitted range; such a mechanism is hard to implement at the OS level without hardware support: a more easy way could be to add automaticallty at the compilation stage some "boilerplate" code to check stack pointer evolution at every subroutine entry (since almost always during such an event that stack pointers are decremented towards new stack tops).

For our tests, we just reduced the outgoing packet queue size to leave enough room to system stack.

6.5 Influence of optimization in implementations

One can also clearly see on table 2 that independently of the overestimation due to the simulation inaccuracies, RIOT OS delays observed for loading packets into CC2420 TX buffer are always significantly longer to those observed under Contiki OS for the same task. Our investigations on the subject have shown that these large differences are due to the way SPI communication is handled at the OS level: under RIOT, when a byte is transmitted over the SPI bus, the result of that transmission is always waited for, by polling the flags indicating the success or the failure of that transmission; on the other hand, Contiki OS uses for loading the transceiver buffer a "fast SPI write" procedure that doesn't care for loading these flags, and only synchronizes for the next byte transfer by waiting the SPI bus to be ready. While Contiki solution is clearly much faster, it is also unsafe since unable to detect correctly any failure that may occur on the SPI bus; RIOT implementation is on the contrary much safer, but slower.

Lesson 8. *There is clearly a trade-off between performance optimization and safety. For loading the CC2440 TX buffer, it depends less on the software platform (OS) itself than on the implementation choices for the SPI bus driver.*

We tried to implement the "unsafe" SPI optimization on RIOT, and measured the speed difference between both versions under the same software platform. The results are shown in table 2 in section 5.

We can still note, however, that the speed penalty imposed to S-CoSenS by RIOT OS SPI subsystem did not prevent it from obtaining superior QoS performance.

6.6 Limitations and Possible Future Improvements

The current work has of course its limitations. Its main drawback is that *it is only based on simulations*. To be truly accurate, we would need to run these tests on actual hardware.

Due to the delays overestimation by the emulator, we can expect results on real hardware to be better than those we have observed in our simultations. Since the delay overestimation seems quite larger for RIOT OS operation than for Contiki (for a reason that is still unexplained to us), we also think these results on real hardware will improve further for RIOT OS

than for Contiki compared to what we obtained by simulation; this would then reinforce the advantage we noticed for S-CoSenS for QoS matters (PRR and TX delays).

Concerning duty cycle statistics, ContikiMAC has a large advantage over S-CoSenS. Here again, there is a balance between the two contradictory objectives that are obtaining a high QoS and low duty cycles. By design, ContikiMAC is clearly more oriented to the latter objective than S-CoSenS, even it is possible to lower the WP_{min} parameter to push our protocol more towards that same objective (cf section 3).

Still, observing ContikiMAC and its working makes us think that we could achieve lower (better) duty cycles for S-CoSenS leaf nodes by implementing a mechanism similar to ContikiMAC's "phase lock" described above. This shall be the subject of one of our future experiments.

One must also accept that there is no way to keep the very low power consumption feature when dealing with such heavy network loads, the motes' radio transceivers being heavily activated and solicited to handle the traffic.

7 Conclusion and Future Works

We described two implementation of complete and functional RDC protocols, the well-known ContikiMAC and our new S-CoSenS, explaining their differences in design and implementation. We then built simulation scenarios to test whether the two studied protocols could handle heavy network loads, trying to push the envelope the farthest possible—knowing anyway that approaching the theoretical limit of 250 kbit/s of bandwith for the underlying IEEE 802.15.4 radio medium is just impossible because of delays that are imposed both by the 802.15.4 standard (SIFS, LIFS), hardware limits of the transceivers, and overheads due to the various stacks of software layers (especially MAC and RDC).

We have, during the implementation of one of our new RDC protocols, overcome many problems that we have discussed in the present paper.

Our contributions thus consist in the following:

- We determined that real-time features are necessary to implement efficient new MAC/RDC protocols—namely: fine grained timing resolution ($<$ 1 ms, ideally in the order of 30 μs/32 KHz), availability of more than one instance of high-resolution (preferably hardware) timers—and have presented one software platform—the RIOT OS—offering such features.

- We demonstrated that the Cooja emulator, while being a very valuable development and debugging tool, is not suited for performance evaluation because of timing inaccuracies—specifically: overstimation of delays related to SPI bus operation.

- We showed—with the limits and preventions discussed hereabove—that the implementation of our protocol, S-CoSenS, seems to have an advantage in terms of QoS (PRR, end-to-end delays) while ContikiMAC as implemented under Contiki OS release has largely the upper hand in duty cycle statistics. We then see that it's important for a MAC/RDC protocol to be able to carry heavy/burst traffic when needed, while still keeping low duty cycle during light traffic periods.

- We encountered crash related to memory corruption—precisely: stack overwritten by other data, because of very strong constraints on basic WSN motes and the absence of a MPU mechanism—and envisioned the counter-measures needed to handle these problems—i.e.: on MPU-less architectures, the need to survey constantly stack use and memory corruption, using code added at the compilation stage.

- We evaluated the influence of optimizations on implementation performance—specifically, in the management of SPI bus in an optimized or safe manner, which impacts the speed of loading packets into radio transceiver by a factor of 2 to 3—at the expense of stability.

In the future, we would like to try other, more powerful hardware platforms with the same scenarii, and try using different MAC and RDC layers in the network stack of the RIOT operating system.

References

[1] S. Zhuo, Z. Wang, Y.-Q. Song, Z. Wang, and L. Almeida, "iQueue-MAC: A traffic adaptive duty-cycled MAC protocol with dynamic slot allocation," in *IEEE 10th Conference on Sensor, Mesh, and Ad Hoc Communications and Networks*, ser. SECON 2013. IEEE Communications Society, 2013, pp. 95–103.

[2] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *IEEE 29th Conference on Local Computer Networks*, ser. LCN '04. IEEE Computer Society, 2004, pp. 455–462, `http://www.contiki-os.org/`.

[3] A. Dunkels, "Rime — a lightweight layered communication stack for sensor networks," in *EWSN, Poster/Demo session*, 2007.

[4] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with Cooja," in *IEEE 31st Conference on Local Computer Networks*, ser. LCN '06. IEEE Computer Society, 2006, pp. 641–648.

[5] A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol," Swedish Institute of Computer Science, Tech. Rep. T2011:13, 2011.

[6] O. Hahm, E. Baccelli, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *INFOCOM 2013, Poster Session*, April 2013, `http://www.riot-os.org/`.

[7] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. ACM, 2006, pp. 307–320.

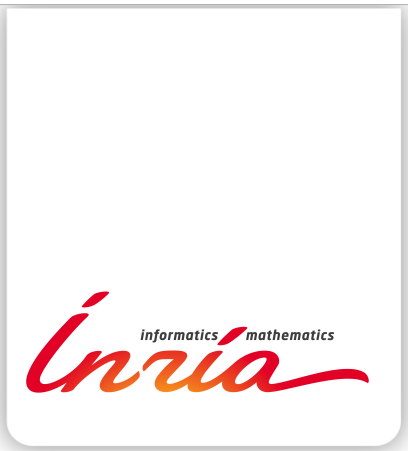
[8] B. Nefzi, "Mécanismes auto-adaptatifs pour la gestion de la Qualité de Service dans les réseaux de capteurs sans fil," Ph.D. dissertation, Networking and Internet Architecture. Institut National Polytechnique de Lorraine (INPL), 2011.

[9] B. Nefzi and Y.-Q. Song, "CoSenS: A Collecting and Sending Burst Scheme for Performance Improvement of IEEE 802.15.4," in *IEEE 35th Conference on Local Computer Networks*, ser. LCN '10. IEEE Computer Society, 2010, pp. 172–175.

[10] Y. Sun, O. Gurewitz, and D. B. Johnson, "RI-MAC: A Receiver-initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks," in *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '08. ACM, 2008, pp. 1–14.

[11] J. Polastre, J. Hill, and D. Culler, "Versatile Low Power Media Access for Wireless Sensor Networks," in *Proceedings of the 2bd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. ACM, 2004, pp. 95–107.

[12] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and J. Marrón, "COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks," in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, ser. Simutools '09. ICST, 2009, pp. 27:1–27:7.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique

CS20101

54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399