RIOT OS Paves the Way for Implementation of High-Performance MAC Protocols

Kévin Roussel, Ye-Qiong Song, Olivier Zendra

RIOT OS Paves the Way for Implementation of High-Performance MAC Protocols

Kévin Roussel, Ye-Qiong Song and Olivier Zendra
LORIA/INRIA Nancy Grand-Est,
Université de Lorraine,
615, rue du Jardin Botanique,
54600 Villers-Lès-Nancy, France
{Kevin.Roussel,Ye-Qiong.Song,Olivier.Zendra}@inria.fr

Keywords:
Real-Time, Wireless Sensor Networks, Internet of Things, MAC protocols, RIOT OS

Abstract:
Implementing new, high-performance MAC protocols requires real-time features, to be able to synchronize correctly between different unrelated devices. Such features are highly desirable for operating wireless sensor networks (WSN) that are designed to be part of the Internet of Things (IoT). Unfortunately, the operating systems commonly used in this domain cannot provide such features. On the other hand, "bare-metal" development sacrifices portability, as well as the multitasking abilities needed to develop the rich applications that are useful in the domain of the Internet of Things.
We describe in this paper how we helped solving these issues by contributing to the development of a port of RIOT OS on the MSP430 microcontroller, an architecture widely used in IoT-enabled motes. RIOT OS offers rich and advanced real-time features, especially the simultaneous use of as many hardware timers as the underlying platform (microcontroller) can offer. We then demonstrate the effectiveness of these features by presenting a new implementation, on RIOT OS, of S-CoSenS, an efficient MAC protocol that uses very low processing power and energy.

1 INTRODUCTION

When programming the small devices that constitutes the nodes of the Internet of Things (IoT), one has to adapt to the limitations of these devices.

Apart from their very limited processing power (especially compared to the current personal computers, and even mobile devices like smartphones and tablets), the main specificity of the devices is that they are operated on small batteries (e.g.: AAA or button cells).

Thus, one of the main challenges with these motes is the need to reduce as much as possible their energy consumption. We want their batteries to last as long as possible, for economical but also practical reasons: it may be difficult—even almost impossible—to change the batteries of some of these motes, because of their locations (e.g.: on top of buildings, under roads, etc.)

IoT motes are usually very compact devices: they are usually built around a central integrated chip that contains the main processing unit and several basic peripherals (such as timers, A/D and D/A converters, I/O controllers...) called microcontroller units or MCUs. Apart from the MCU, a mote generally only contains some "physical-world" sensors and a radio transceiver for networking. The main radio communication protocol currently used in the IoT field is IEEE 802.15.4. Some MCUs do integrate a 802.15.4 transceiver on-chip.

Among the various components that constitute a mote, the most power-consuming block is the radio transceiver. Consequently, to reduce the power consumption of IoT motes, a first key point is to use the radio transceiver only when needed, keeping it powered-off as much as possible. The software element responsible to control the radio transceiver in an adequate manner is

the *MAC / RDC (Media Access Control & Radio Duty Cycle)* layer of the network stack.

A efficient power-saving strategy for IoT motes thus relies on finding the better trade-off between minimizing the radio duty cycle while keeping networking efficiency at the highest possible level. This is achieved by developing new, "intelligent" MAC / RDC protocols.

To implement new, high-performance MAC / RDC protocols, one needs to be able to react to events with good reactivity (lowest latency possible) and flexibility. These protocols rely on precise timing to ensure efficient synchronization between the different motes and other radio-networked devices of a *Personal Area Network (PAN)*, thus allowing to turn on the radio transceivers *only* when needed.

At the system level, being able to follow such accurate timings means having very efficient interruption management, and the extensive use of hardware timers, that are the most precise timing source available.

The second most power-consuming element in a mote, after the radio transceiver, is the MCU itself: every current MCU offers "low-power modes", that consist in disabling the various hardware blocks, beginning with the CPU core. The main way to minimize energy consumption with a MCU is thus to disable its features as much as possible, only using them when needed: that effectively means putting the whole MCU to sleep as much as possible.

Like for the radio transceiver, using the MCU efficiently while keeping the system efficient and reactive means optimal use of interruptions, and hardware timers for synchronization.

Thus, in both cases, we need to optimally use interruptions as well as hardware timers. Being able to use them both efficiently without too much hassle implies the use of a specialized operating system (OS), especially to easily benefit from multitasking abilities. That is what we will discuss in this paper.

2 PREVIOUS WORK AND PROBLEM STATEMENT

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

2.1 TinyOS

The first widely used system in this domain was *TinyOS* (Levis et al., 2005). It is an open-source OS, whose first stable release (1.0) was published in september 2002. It is very lightweight, and as such well adapted to limited devices like WSN motes. It has brought many advances in this domain, like the ability to use Internet Protocol (IP) and routing (RPL) on 802.15.4 networks, including the latest IPv6 version, and to simulate networks of TinyOS motes via TOSSIM (Levis et al., 2003).

Its main drawback is that one needs to learn a specific language—named nesC—to be able to efficiently work within it. This language is quite different from standard C and other common imperative programming languages, and as such can be difficult to master.

The presence of that specific language is no coincidence: TinyOS is built on its own specific paradigms: it has an unique stack, from which the different components of the OS are called as statically linked callbacks. This makes the programming of applications complex, especially for decomposing into various "tasks". The multitasking part is also quite limited: tasks are run in a fixed, queue-like order. Finally, TinyOS requires a custom GNU-based toolchain to be built.

All of these limitations, plus a relatively slow development pace (last stable version dates back to august 2012) have harmed its adoption, and it is not the mainly used OS of the domain anymore.

2.2 Contiki

The current reference OS in the domain of WSN and IoT is *Contiki* (Dunkels et al., 2004). It's also an open-source OS, which was first released in 2002. It is also at the origin of many assets: we can mention, among others, the uIP Embedded TCP/IP Stack (Dunkels, 2003), that has been extended to uIPv6, the low-power Rime network stack (Dunkels, 2007), or the Cooja advanced network simulator (Österlind et al., 2006).

While a bit more resource-demanding than TinyOS, Contiki is also very lightweight and well adapted to motes. Its greatest advantage over TinyOS is that it is based on standard, well-known OS paradigms, and 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 (Dunkels, 2011) and implemented into Contiki as the default RDC protocol (designed to be used with standard CSMA/CA as MAC layer).

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. 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, knowing that the transmission of a full-length 802.15.4 packet takes bout 4 milliseconds (4000 microseconds), a time granularity of 320 microseconds is needed, corresponding to one backoff period (BP).

To address this problem, Contiki provides a real-time feature, `rtimer`, which allows to bypass the kernel scheduler and use a hardware timer to trigger execution of user-defined functions. However, it has very severe limitations:

- only one instance of `rtimer` is available, thus only one real-time event can be scheduled or executed at any time; this limitation forbids development of advanced real-time software—like high-performance MAC / RDC protocols—or at least makes it very hard;
- moreover, it is unsafe to execute from `rtimer`, even indirectly, most of the Contiki basic functions (i.e.: kernel, network stack, etc.), because these functions are not designed to handle pre-emption. Contiki is indeed based on cooperative multithreading, whereas the `rtimer` mechanism seems like a "independent feature", coming with its own paradigm. Only a precise set of functions known as "interrupt-safe" (like `process_poll()`) can be safely invoked from `rtimer`, using other parts of Contiki's meaning almost certainly crash or unpredictable behaviour. This restriction practically makes it very difficult to write Contiki extensions (like network stack layer drivers) using `rtimer`.

Also note that this cooperative scheduler is designed to manage a specific kind of tasks: the *protothreads*. This solution allows to manage different threads of execution, without needing each of them to have its own separate stack (Dunkels et al., 2006). The great advantage of this mechanism is the ability to use an unique stack, thus greatly reducing the needed amount of RAM for the system. The trade-off is that one must be careful when using certain C constructs (i.e.: it is impossible to use the `switch` statement in some parts of programs that use protothreads).

For all these reasons, we were unable to use Contiki OS to develop and implement our high-performance MAC/RDC protocols. We definitely needed an OS with efficient real-time features and event handling mechanism.

2.3 Other options

There are other, less used OSes designed for the WSN/IoT domain, but none of them fulfilled our requirements, for the following reasons:

SOS (Han et al., 2005) This system's development has been cancelled since november 2008; its authors explicitly recommend on their website to "consider one of the more actively supported alternatives".

Lorien (Porter and Coulson, 2009) While its component-oriented approach is interesting, this system seems does not seem very widespread. It is currently available for only one hardware platform (TelosB/SkyMote) which seriously limits the portability we can expect from using an OS. Moreover, its development seems to have slowed down quite a bit, since the latest available Lorien release was published in july 2011, while the latest commit in the project's SourceForge repository (r46) dates back to january 2013.

Mantis (Abrach et al., 2003) While this project claims to be Open Source, the project has made, on its SourceForge web site, no public release, and the access to the source repository (`http://mantis.cs.colorado.edu/viewcvs/`) seems to stall. Moreover, reading the project's main web page shows us that the last posted

news item mentions a first beta to be released in 2007. The last publications about Mantis OS also seems to be in 2007. All of these elements tend to indicate that this project is abandoned...

LiteOS (Cao et al., 2008) This system offers very interesting features, especially the ability to update the nodes firmwares over the wireless, as well as the built-in hierarchical file system. Unfortunately, it is currently only available on IRIS/MicaZ platforms, and requires AVR Studio for programming (which imposes Microsoft Windows as a development platform). This greatly hinders portability, since LiteOS is clearly strongly tied to the AVR microcontroller architecture.

MansOS (Strazdins et al., 2010) This system is very recent and offers many interesting features, like optional preemptive multitasking, a network stack, runtime reprogramming, and a scripting language. It is available on two MCU architectures: AVR and MSP430 (but not ARM). However, none of the real-time features we wanted seems to be available: e.g. only software timers with a 1 millisecond resolution are available.

In any case, none of the alternative OSes cited hereabove offer the real-time features we were looking for.

On the other hand, "bare-metal" programming is also unacceptable for us: it would mean sacrificing portability and multitasking; and we would also need to redevelop many tools and APIs to make application programming even remotely practical enough for third-party developers who would want to use our protocols.

We also envisioned to use an established real-time OS (RTOS) as a base for our works. The current reference when it comes to open-source RTOS is *FreeRTOS* (`http://www.freertos.org/`). It is a robust, mature and widely used OS. Its codebase consists in clean and well-documented standard C language. However, it offers only core features, and doesn't provide any network subsystem at all. Redeveloping a whole network stack from scratch would have been too time-consuming. (Network extensions exist for FreeRTOS, but they are either immature, or very limited, or proprietary and commercial software; and most of them are tied to a peculiar piece of hardware, thus ruining the portability advantage offered by the OS.)

2.4 Summary: Wanted Features

To summarize the issue, what we required is an OS that:

- is adapted to the limitations of the deeply-embedded MCUs that constitute the core of WSN/IoT motes;
- provides real-time features powerful enough to support the development of advanced, high-performance MAC / RDC protocols;
- includes a network stack (even a basic one) adapted to wireless communication on 802.15.4 radio medium.

However, none of the established OSes commonly used either in the IoT domain (TinyOS, Contiki) nor in the larger spectrum of RTOS (FreeRTOS) could match our needs.

3 THE RIOT OPERATING SYSTEM

Consequently, we focused our interest on *RIOT OS* (Hahm et al., 2013).

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 interesting 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.

Moreover, it offers key features that are otherwise yet unknown in the WSN/IoT domain:

- an efficient, interrupt-driven, tickless *microkernel*;
- that kernel includes a priority-aware task scheduler, providing *pre-emptive 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 granularity; on low-end devices, based on MSP430 architecture, events can be scheduled with a resolution of 32 microseconds;

- RIOT is entirely written in *standard C language*; but unlike Contiki, there are no restrictions on usable constructs (i.e.: like those introduced by the protothreads mechanism);
- a clean and *modular design*, that makes development with and *into* the system itself easier and more productive.

The first three features listed hereabove make RIOT a full-fledged *real-time* operating system.

We also believe that the tickless kernel and the optimal use of hardware timers should make RIOT OS a very suited software platform to optimize energy consumption on battery-powered, MCU-based devices.

A drawback of RIOT, compared to TinyOS or Contiki, is its higher memory footprint: the full network stack (from PHY driver up to RPL routing with 6LoWPAN and MAC / RDC layers) cannot be compiled for Sky/TelosB because of overflowing memory space. Right now, constrained devices like MSP430-based motes are limited to the role of what the 802.15.4 standard calls *Reduced Function Devices (RFD)*, the role of *Full Function Devices (FFD)* being reserved to more powerful motes (i.e.: based on ARM microcontrollers).

However, we also note that, thanks to its modular architecture, the RIOT kernel, compiled with only PHY and MAC / RDC layers, is actually lightweight and consumes little memory. We consequently believe that the current situation will improve with the maturation of higher layers of RIOT network stack, and that in the future more constrained devices could also be used as FFD with RIOT OS.

When we began to work with RIOT, it also had two other issues: the MSP430 versions were not stable enough to make real use of the platform; and beyond basic CSMA/CA, no work related to the MAC / RDC layer had been done on that system. This is where our contributions fit in.

4 OUR CONTRIBUTIONS

For our work, we use—as our main hardware platform—IoT motes built around MSP430 microcontrollers.

MSP430 is a microcontroller (MCU) architecture from Texas Instruments, offering very low-power consumption, cheap price, and good performance thanks to a custom 16-bit RISC design. This architecture is very common in IoT motes. It is also very well supported, especially by the Cooja simulator (Österlind et al., 2006), which makes simulations of network scenarios—especially with many devices—much easier to design and test.

RIOT OS has historically been developed first on legacy ARM devices (ARM7TDMI-based MCUs), then ported on more recent microcontrollers (ARM Cortex-M) and other architectures (MSP430 then AVR). However, the MSP430 port was, before we improved it, still not as "polished" as ARM code and thus prone to crash.

Our contribution can be summarized in the following points:

1. analysis of current OSes (TinyOS, Contiki, etc.) limitations, and why they are incompatible with development of real-time extensions like advanced MAC / RDC protocols;
2. add debugging features to the RIOT OS kernel, more precisely a mechanism to handle fatal errors: crashed systems can be "frozen" to facilitate debugging during development; or, in production, can be made to reboot immediately, thus reducing unavailability of a RIOT-running device to a minimum;
3. port RIOT OS to a production-ready, MSP430-based device: the Zolertia Z1 mote (already supoorted by Contiki, and used in real-world scenarios running that OS);
4. debug the MSP430-specific portion of RIOT OS—more specifically: the hardware abstraction layer (HAL) of the task scheduler—making RIOT OS robust and production-ready on MSP430-based devices.
   Note that all of these contributions have been reviewed by RIOT's development team and integrated into the "master" branch of RIOT OS' Github repository (i.e.: they are now part of the standard code base of the system).
5. running on MSP430-based devices also allows RIOT OS applications to be simulated with the Cooja simulator; this greatly improves speed and ease of development.
6. thanks to these achievements, we now have a robust and full-featured software platform offering all the features needed to develop high-performance MAC/RDC protocols—such as all of the time-slotted protocols.

As a proof of concept of this last statement, we have implemented one of our own designs, and obtained very promising results, shown in the next section.

5 USE CASE: IMPLEMENTING THE S-COSENS RDC PROTOCOL

5.1 The S-CoSenS Protocol

The first protocol we wanted to implement is S-CoSenS (Nefzi, 2011), which is designed to work on top of the IEEE 802.15.4 physical and MAC (i.e.: CSMA/CA) layers.

It is an evolution of the already published CoSenS protocol (Nefzi and Song, 2010): 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 radio duty 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.

The computation of WP duration follows a "sliding average" algorithm, where WP duration for each duty cycle is computed from the average of previous cycles 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 length of the whole subframe being a parameter given at compilation time, SP duration is simply computed by subtracting the calculated duration of WP from the subframe duration for every cycle.

The local synchronization between a S-CoSenS router and its leaf nodes 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 1 hereafter.

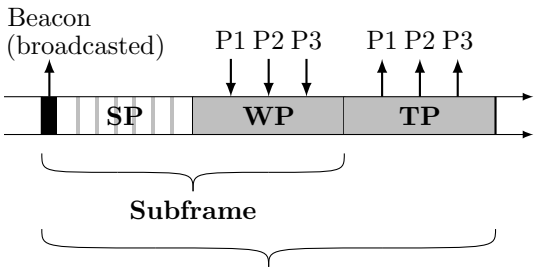

Figure 1: A typical S-CoSenS router cycle.
The gray strips in the SP represents the short wake-up-and-listen periods used for inter-router communication.

An interesting property of S-CoSenS is that leaf (i.e.: non-router) nodes always have their radio transceiver offline, 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 2.

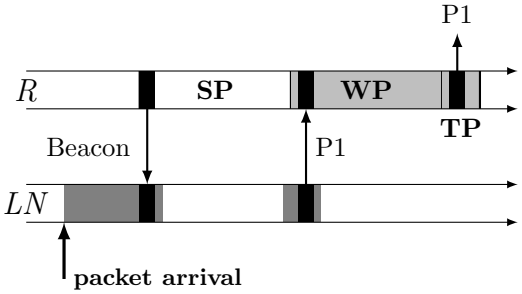

Figure 2: A typical transmission of a data packet with the S-CoSenS protocol between a leaf node and a router.

We thus need to synchronize with enough accuracy different devices (that can be based on different hardware platforms) on cycles whose peri-

ods 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.

5.2 Simulations and Synchronization Accuracy

We have implemented S-CoSenS under RIOT, and made first tests by performing simulations—with Cooja—of a 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 3.

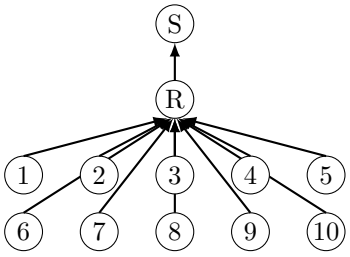

Figure 3: Functional schema of our virtual test PAN.

Our first tests clearly show an excellent synchronization between the leaf nodes and the router, thanks to the time resolution offered by RIOT OS event management system (especially the availability of many hardware timers for direct use). This can be seen in the screenshot of our simulation in Cooja, shown in figure 4. For readability, the central portion of the timeline window of that screenshot (delimited by a thick yellow rectangle) is zoomed on in figure 5.

On figure 5, the numbers on the left side are motes' numerical IDs: the router has ID number 1, while the leaf nodes have IDs 2 to 11. Grey bars represent radio transceiver being online for a given mote; blue bars represent packet emission, and green bars correct packet reception, while red bars represent collision (when two or more devices emit data concurrently) and thus reception of undecipherable radio signals.

Figure 5 represents a short amount of time (around 100 milliseconds), representing the end of a duty cycle of the router: the first 20 milliseconds are the end of SP, and 80 remaining milliseconds the WP, then the beginning of a new duty cycle (the TP has been disabled in our simulation).

In our example, four nodes have data to transmit to the router: the motes number 3, 5, 9, and 10; the other nodes (2, 4, 6, 7, 8, and 11) are preparing to transmit a packet in the next duty cycle.

At the instant marked by the first yellow arrow (in the top left of figure 5), the SP ends and the router activates its radio transceiver to enter WP. Note how the four nodes that are to send packets (3, 5, 9, and 10) do also activate their radio transceivers *precisely* at the same instant: this is thanks to RIOT OS precise real-time mechanism (based on hardware timers), that allows to the different nodes to precisely synchronize on the timing values transmitted in the previous beacon packet. Thanks also to that mechanism, the nodes are able to keep both their radio transceiver *and* their MCU in low-power mode, since RIOT OS kernel is interrupt-driven.

During the waiting period, we also see that several collisions occur; they are resolved by the S-CoSenS protocol by forcing motes to wait a random duration before re-emitting a packet in case of conflict. In our example, our four motes can finally transmit their packet to the router in that order: 3 (after a first collision), 5, 10 (after two other collisions), and finally 9. Note that every time the router (device number 1) successfully receives a packet, an acknowledgement is sent back to emitter: see the very thin blue bars that follow each green bar on the first line.

Finally, at the instant marked by the second yellow arrow (in the top right of figure 5), WP ends and a new duty cycle begins. Consequently, the router broadcasts a beacon packet containing PAN timing and synchronization data to all of the ten nodes. We can see that all of the six nodes waiting to transmit (2, 4, 6, 7, 8, and 11) go idle after receiving this beacon (beacon packets are broadcasted and thus not to be acknowledged): they go into low-power mode (both at radio transceiver and MCU level), and will take advantage of RIOT real-time features to wake up precisely when the router goes back into WP mode and is ready to receive their packets.

5.3 Performance Evaluation: Preliminary Results

We will now present the first, preliminary results we obtained through the simulations we described hereabove.

Important: note that *we evaluate here the im-*

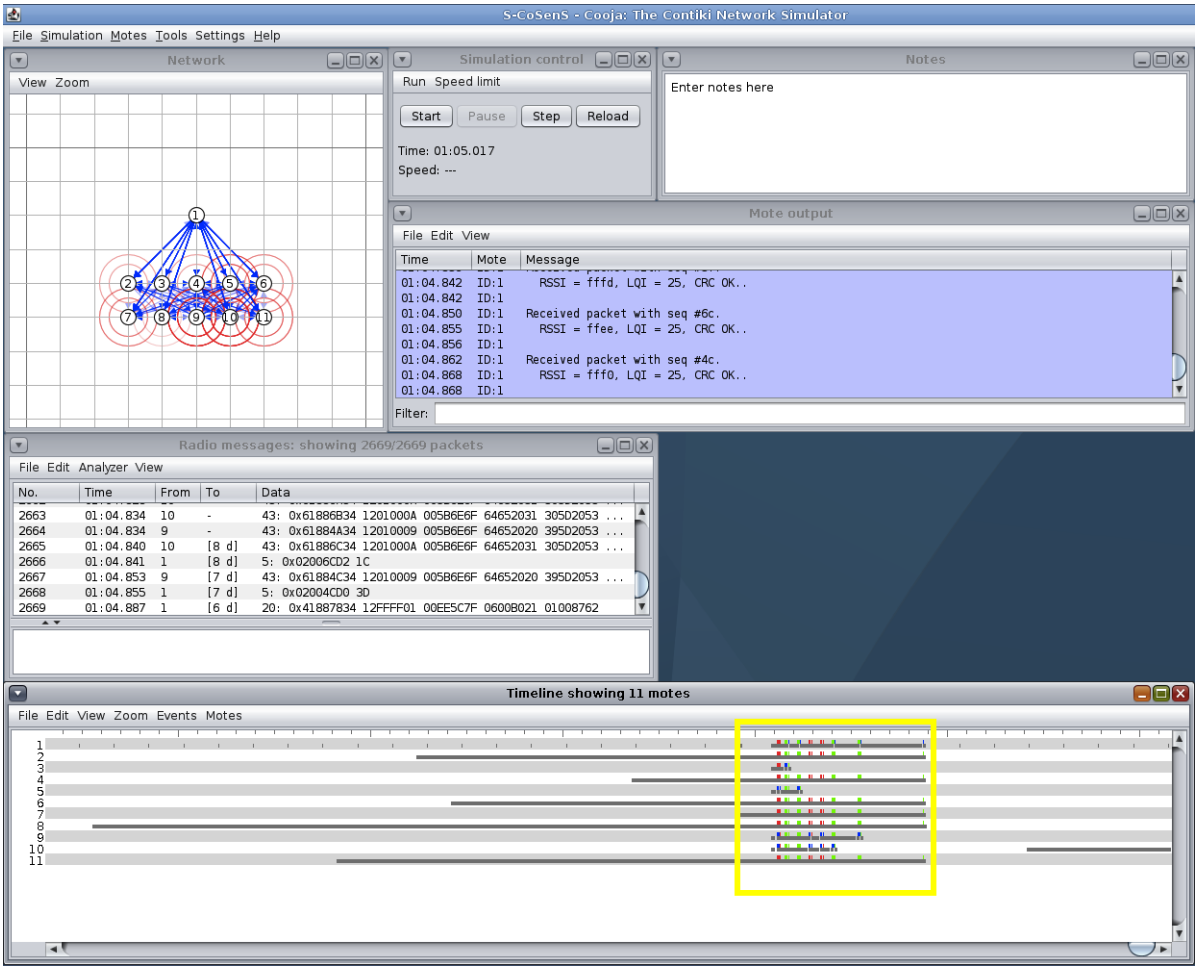

Figure 4: Screenshot of our test simulation in Cooja. (Despite the window title mentioning Contiki, the simulated application is indeed running on RIOT OS.)

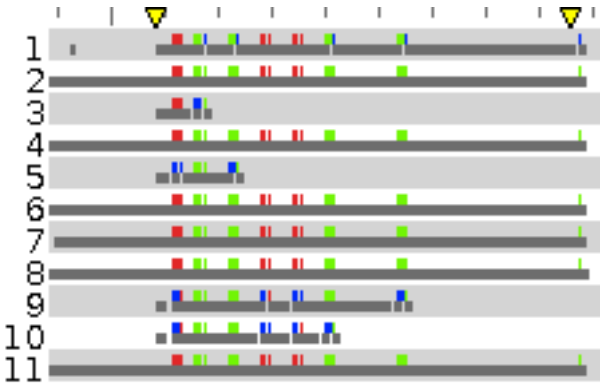

Figure 5: Zoom on the central part of the timeline of our simulation.

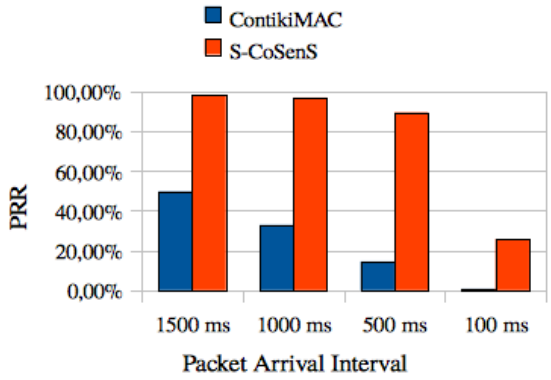

Figure 6: PRR results for both ContikiMAC and S-CoSenS RDC protocols, using default values for parameters.

PAI \ Protocol	ContikiMAC	S-CoSenS
1500 ms	49.70%	98.10%
1000 ms	32.82%	96.90%
500 ms	14.44%	89.44%
100 ms	0.64%	25.80%

Table 1: PRR results for both ContikiMAC and S-CoSenS RDC protocols, using default values for parameters.

plementations, and not the intrinsic advantages or weaknesses of the protocols themselves.

We have first focused on QoS results, by computing Packet Reception Rates and end-to-end delays between the various leaf nodes and the sink of the test PAN presented earlier in figure 3, to evaluate the quality of the transmissions allowed by using both of the protocols.

For these first tests, we used default parameters for both RDC protocols (ContikiMAC and S-CoSenS), only pushing the CSMA/CA MAC layer of Contiki to make up to 8 attempts for transmitting a same packet, so as to put it on par with our implementation on RIOT OS. We have otherwise not yet tried to tweak the various parameters offered by both the RDC protocols to optimize results. This will be the subject of our next experiences.

5.3.1 Packet Reception Rates (PRR)

The result obtained for PRR using both protocols are shown in figure 6 as well as table 1.

The advantage of S-CoSenS as shown on the figure is clear and significant whatever the packet arrival interval constated. Excepted for the "extreme" scenario corresponding to an oversaturation of the radio channel, S-CoSenS achieve an excellent PRR ($\gtrsim 90\%$), while ContikiMAC's PRR is always $\lesssim 50\%$.

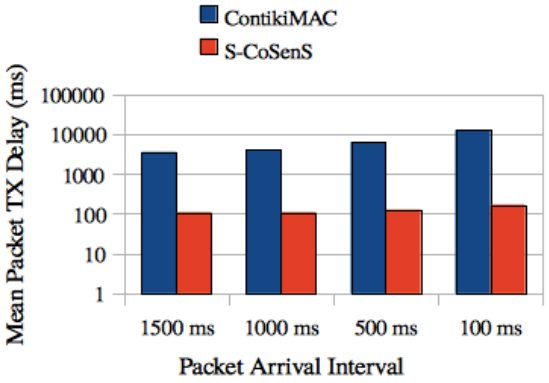

Figure 7: End-to-end delays results for both ContikiMAC and S-CoSenS RDC protocols, using default values for parameters; note that vertical axis is drawn with logarithmic scale.

PAI \ Protocol	ContikiMAC	S-CoSenS
1500 ms	3579 ms	108 ms
1000 ms	4093 ms	108 ms
500 ms	6452 ms	126 ms
100 ms	12913 ms	168 ms

Table 2: End-to-end delays results for both ContikiMAC and S-CoSenS RDC protocols, using default values for parameters.

5.3.2 End-To-End Transmission Delays

The result obtained for PRR using both protocols are shown in figure 7 and table 2.

S-CoSenS has here also clearly the upper hand, so much that we had to use logarithmic scale for the vertical axis to keep figure 7 easily readable. The advantage of S-CoSenS is valid whatever the packet arrival interval, our protocol being able to keep delay below an acceptable limit (in the magnitude of hundreds of milliseconds), while ContikiMAC delays rocket up to tens of seconds when network load increases.

5.3.3 Summary: QoS Considerations

While these are only preliminary results, it seems that being able to leverage real-time features is clearly a significant advantage when designing and implementing MAC/RDC protocols, at least when it comes to QoS results.

6 FUTURE WORKS AND CONCLUSION

We plan, in a near future:

- to bring new contributions to the RIOT project: we are especially interested in the portability that the RIOT solution offers us; this OS is indeed actively ported on many devices based on powerful microcrontrollers based on ARM Cortex-M architecture (especially Cortex-M3 and Cortex-M4), and we intend to help in this porting effort, especially on high-end IoT motes we seek to use in our works (e.g.: as advanced FFD nodes with full network stack, or routers);
- to use the power of this OS to further advance our work on MAC/RDC protocols; more precisely, we are implementing other innovative MAC/RDC protocols—such as iQueue-MAC (Zhuo et al., 2013)—under RIOT, taking advantage of its high-resolution real-time features to obtain excellent performance, optimal energy consumption, and out-of-the-box portability.

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 for the embedded and real-time world.

While we weren't able to accurately quantize energy consumption yet, we can reasonably think that lowering activity of MCU and radio transceiver will significantly reduce the energy consumption of devices running RIOT OS. This will be the subject of some of our future research works.

Currently, RIOT OS supports high-level IoT protocols (6LoWPAN/IPv6, RPL, TCP, UDP, etc.). However, it still lacks high-performance MAC / RDC layer protocols.

Through this work, we have shown that RIOT OS is also suitable for implementing high-performance MAC / RDC protocols, thanks to its real-time features (especially hardware timers management).

Moreover, we have improved the robustness of the existing ports of RIOT OS on MSP430, making it a suitable software platform for tiny motes and devices.

REFERENCES

Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. (2003). Mantis: System Support for Multimodal Networks of In-Situ Sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA 2003, pages 50–59. ACM.

Cao, Q., Abdelzaher, T., Stankovic, J., and He, T. (2008). the LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, pages 233–244. IEEE Computer Society. `http://www.liteos.net/`.

Dunkels, A. (2003). Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 85–98. ACM.

Dunkels, A. (2007). Rime — a lightweight layered communication stack for sensor networks. In *EWSN, Poster/Demo session*.

Dunkels, A. (2011). The ContikiMAC Radio Duty Cycling Protocol. Technical Report T2011:13, Swedish Institute of Computer Science.

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

Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. (2006). Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42. ACM.

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

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). SOS: A Dynamic Operating System for Sensor Nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176. ACM. `https://projects.nesl.ucla.edu/public/sos-2x/doc/`.

Levis, P., Lee, N., Welsh, M., and Culler, D. (2003). TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137. ACM.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D.

(2005). TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg. `http://www.tinyos.net/`.

Nefzi, B. (2011). *Mécanismes auto-adaptatifs pour la gestion de la Qualité de Service dans les réseaux de capteurs sans fil*. PhD thesis, Networking and Internet Architecture. Institut National Polytechnique de Lorraine (INPL).

Nefzi, B. and Song, Y.-Q. (2010). CoSenS: a Collecting and Sending Burst Scheme for Performance Improvement of IEEE 802.15.4. In *IEEE 35th Conference on Local Computer Networks*, LCN '10, pages 172–175. IEEE Computer Society.

Österlind, F., Dunkels, A., Eriksson, J., Finne, N., and Voigt, T. (2006). Cross-Level Sensor Network Simulation with Cooja. In *IEEE 31st Conference on Local Computer Networks*, LCN '06, pages 641–648. IEEE Computer Society.

Porter, B. and Coulson, G. (2009). Lorien: a Pure Dynamic Component-Based Operating System for Wireless Sensor Networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 7–12. ACM. `http://lorienos.sourceforge.net/`.

Strazdins, G., Elsts, A., and Selavo, L. (2010). MansOS: Easy to Use, Portable and Resource Efficient Operating System for Networked Sensor Systems. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, Sensys '10, pages 427–428. ACM. `http://mansos.edi.lv/`.

Zhuo, S., Wang, Z., Song, Y.-Q., Wang, Z., and Almeida, L. (2013). 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*, SECON 2013, pages 95–103. IEEE Communications Society.