2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)

Integration of the ultra-low power WiseMAC with the µ111 Real Time Operating System: a performance evaluation

Robin Berguerand, Lorenzo Bergamini, Philippe Dallemagne, Edoardo Franzi

Centre Suisse d'Électronique et Microtechnique – Rue Jaquet-Droz 1, Neuchâtel - Switzerland
robin.berguerand, lorenzo.bergamini, philippe.dallemagne, edoardo.franzi @csem.ch

***Abstract*— Many communication solutions for low power wireless sensor networks (WSN) rely on monolithic implementations instead of Operating System (OS) features, properties and services. Such OS-less approach implements the timings, state-machines and functions into a single software including scheduling and synchronization. This means that porting a protocol on a new hardware requires the rewriting of major software parts and at least the low-level drivers and implies full re-testing. Using a low-resource, low footprint Real Time Operating System (RTOS) would instead allow for an easy and efficient implementation of applications, whose complexity is reduced by using the OS features and services who acts as a convenient abstraction layer. The general feeling is that, however, the presence of the OS would impact the performance of the wireless communication, in particular with respect to latency and power consumption. This paper describes the results obtained by integrating the µ111 RTOS with the ultra-low power Medium Access Control (MAC) protocol for Wireless Sensor Networks WiseMAC [1], [2], showing that the impact of the OS in terms of energy consumption and hardware resources utilization is minimal if not null.**

Keywords— Wireless Sensor Networks, Network Protocols, Real-time operating systems, Performance analysis.

I. INTRODUCTION

Initially introduced at the beginning of years 2000, WSN can be defined as a self-configured wireless network composed of low-power embedded devices with limited computational and hardware resources commonly called *Sensor Nodes* or simply *nodes*. Each node can serve a variety of purposes, such as physical data collection (e.g., temperature, humidity), partial processing or actuation (e.g., start fan, open door). The data recorded from the different nodes is generally routed to a single point of collection (*sink*) that usually acts as a gateway to a computer or central server where the data is stored and analysed. Minimizing the energy consumption is of critical importance in WSN design, since replacing or recharging batteries is often impractical. Radio communication is often the main source of energy consumption, so a big number of dedicated communication protocols have been developed during the years to reduce energy usage while guaranteeing adequate performance in terms of packet delivery reliability and latency. Differently from the huge effort on the design and evaluation of communication protocols, the WSN research community never adopted a common framework for portable and easily repeatable solutions, in spite of the initial popularity of TinyOS [3], Contiki [4] or, more recently, FreeRTOS [5]. Most of the proposed protocols are based on monolithic programs that run on a specific hardware and are able of executing a single application. The main argument to avoid the use of an OS could be that it would increase the power consumption, or it would cause a delay in the protocol operations, which can result in an increase in the number of lost packets or in the latency.

This project is partly supported by the European Commission under the Horizon 2020 EIC Fast Track to Innovation program (H2020-EIC-FTI-2018-2020, GA #878950)

In this work we want to show that this is not the case. We implemented WiseMAC as both a monolithic program and as a process running on µ111 RTOS on the same hardware platform. We will show experimentally that differently from what one could expect, the introduction of an OS does not impact the power consumption nor the overall performance of the MAC protocol. Throughout the paper, we will put in evidence the several advantages of using an RTOS instead of an OS-less implementation, like the possibility of running multiple applications in parallel on the same processor (e.g.: a communication protocol can be run at the same time as a data collection application) or the reduction of the risk of introducing bugs thanks to the encapsulation of applications in a controlled environment.

After this introduction, we review some of the past works on the subject of Mac protocols for WSN (section II.A) and we discuss why we chose WiseMAC for our test; an overview of operating Systems for WSN will be given in section II.B. Section III describes in a nutshell the µ111 RTOS we selected to run WiseMAC protocol, while the details of the protocol implementations on the OS-less environment and on µ111 are given in section IV. Section V contains our experimental analysis and performance evaluation of the protocol in its OS-based version when compared to the OS-less version, while section VI concludes the paper and proposes some future research directions.

II. RELATED WORK

To the best of our knowledge, this paper is one of the first to compare the same WSN protocol in an OS and OS-less environment on the same hardware platform. In this section, we quickly describe the subject of MAC protocols, we motivate why we chose WiseMAC, and we review available RTOS for embedded systems and.

A. MAC Protocols for WSN

As mentioned in the introduction, WSN have been around for more than two decades and saw an intense research activity, leading to the development of efficient communication protocols and low power solutions. The literature on the subject of WSN protocols is vast ([6] to [11]) and includes solutions ranging from the management of the MAC layer to the routing layer for multi-hop topologies. Our paper is not about a novel WSN protocol, so we believe that an exhaustive review of the existing solutions is beyond the scope of this paper. Interested readers can refer to the cited works and in particular to the most recent ones [6, 11] for more detailed information on the subject.

In this section we rather explain the reasons why we chose WiseMAC for our experiments.

Among the different families of MAC protocols, namely Time Division Multiple Access (TDMA), Frequency Division Multiple Access (FDMA) and Carrier-Sense Multiple Access (CSMA). We decided to focus on the latter

978-1-6654-2854-5/21/$31.00 ©2021 IEEE

category because when compared to TDMA and FDMA solutions, CSMA protocols present little constraints in terms of network synchronization; since this is a first experience in porting to an RTOS, we wanted to minimize the impact of external factors (e.g., clock management) to the performance of the protocol.

CSMA refers to a family of protocols where nodes sense the channel before transmitting any data to avoid packet collisions. To send a packet, the node senses the channel and, if free, it enters a back-off phase before the transmission starts. The duration of the back-off is taken randomly in an interval that varies depending on the traffic intensity. Since the probability that two nodes will have the same duration of the back-off period is small, a collision is unlikely. The size of the interval is increased (usually multiplied by 2) with each collision to minimize the likelihood of a subsequent collision. The key advantage of this solution is the simplicity and scalability of implementation. The node can enter the network without any configuration. In addition, the CSMA solution has almost no overhead and full bandwidth can be used to send data. On the other hand, the radio of each device must be switched on continuously to receive packets and this causes a lot of passive (idle) listening as well as overhearing and has a negative impact on power consumption. Different solutions have been proposed (see [6] or [11] for detailed descriptions), and WiseMAC belongs to this family of protocols. The description of how WiseMAC works in principle will be given in section IV. As for the reasons why we selected this protocol, WiseMAC is a solution widely accepted by the WSN community as one of the reference protocols in the domain for low traffic applications (according to Langendoen [9] "*The WiseMAC protocol showed a remarkable consistent behaviour across a wide range of operational conditions, always achieving the best or second-best performance*") and its complete software implementation is well-known and available to the authors, thus allowing a more complete analysis of the RTOS-related aspects. Finally, WiseMAC in its OS-less version, was already used and validated in several real-time scenarios and thus a good amount of data for comparison was available.

B. Operating Systems for WSN

As mentioned in the introduction, many wireless communication protocols for WSN were developed as monolithic solutions, following an OS-less approach. Early approaches to develop protocols in an OS-like way supporting time-critical tasks date back to the initial existence of WSN and include TinyOS [3], or SensorOS [12]. The interest for these solutions has dropped with the advent of significantly more powerful hardware and richer software environments which allows the execution of more complex systems usually indicated as RTOS.

The term real-time often means that the operating system guarantees that the application it runs respects temporal constraints, such as durations, deadlines, synchronization, etc. In addition, it often provides the tools for building time aware services, which can for example notify the application in case the system does not meet the temporal requirements.

The literature classifies RTOS based on characteristics such as scheduling algorithms, number of priority levels, threading model, interrupt and event handling, objects and mechanisms for inter-process communication & synchronisation, time constraint specification, memory management, etc. Beyond the core characteristics of an RTOS, other features make the life of the developers and users easier, such as upgradability, data communication, control, monitoring, device interfacing, data acquisition and signal processing, database interfaces and level of standardisation. Some applications require the compliance to specific standards such as POSIX, ISO or domain specific standards such as ECSS for space or RTCA/DO-17x for aeronautics.

An interesting survey of potentially relevant OS for embedded devices is [18]. A partial list is also given below, to illustrate the diversity of solutions found on the market and that of the addressed requirements, both from commercial vendors and open-source initiatives.

- ITRON is an open RTOS specification for embedded systems for which many implementations exist from Fujitsu, Hitachi, Mitsubishi, Miyazaki, Morson, NEC, Sony Corp., Three Ace Computer Corp., Toshiba, etc.
- OSEK-VDX is an RTOS specification adopted by the automotive industry for their embedded systems.
- QNX is a real-time, extensible POSIX compliant OS based on a micro-kernel, which was very popular in the 90s and years 2000 and found a new home in IoT.
- VRTX is an RTOS certified to RTCA/DO-178B standard for mission-critical aerospace systems. It is based on a Nanokernel.
- Wind River VxWorks is a commercial RTOS with a long history in the market [11].
- Microsoft Azure ThreadX (formerly Express Logic ThreadX) is an RTOS available for ST, Microchip, NXP and Renesas microcontrollers and others, which smallest instance 2KB of code and 1KB of RAM. It is certified for IEC-61508 SIL 4, IEC-62304 SW Safety Class C, ISO 26262 ASIL D and EN 50128.
- TI-RTOS, also known as SYS/BIOS features IPv4 and IPv6-compliant TCP/IP stack and tools such as DNS, HTTP, and DHCP. It includes the support for the wireless connectivity using Wi-Fi, Bluetooth LE and ZigBee.
- LynxOS is a UNIX-compatible, POSIX-conformant real-time operating system for embedded applications. RTAI takes a unique approach of running Linux as a task (lowest priority) that competes with other real-time tasks for the CPU.

We could continue this list (RTEMS, Nut/OS, µC/OS-II, embOS, TNKernel, ChibiOS/RT, RTLinux, pSOS….), but going further is beyond the scope of this work.

III. U111 RTOS IN A NUTSHELL

µ111 RTOS (simply µ111 in the rest of the paper) is a multitasking operating system developed at CSEM. µ111 is the latest incarnation of the fully customisable µKOS microkernel OS for offering a multitasking environment to low resource embedded systems. Its development started in 1992. µKOS was originally designed and used for the Khepera mobile robot [13], [14] with the aim of controlling it. µKOS operated these robots in advanced space applications as early as 1997 at JPL [15] and numerous other miniature and low power systems at CSEM. The first implementation was written in the CALM assembler tool developed at EPFL, then ported to C in 1996 (with MPW in OS/9 on Macintosh, then with GNU-GCC). µ111 has been ported to many micro-

controllers over the years and it now runs on the most recent RISC-V and ARM platforms, such as the NRF52 chip on which the present experiments have been performed. The main purpose of µ111 is to offer tools and platforms to develop real-time applications. The OS is also already available for a wide range of microcontrollers (allowing simple application porting) and is highly customizable; a distribution of the OS is available for researchers or developers upon request (contact one of the authors).

A. General architecture

µ111 can run multiple tasks (processes) in parallel and offers then a multithreading environment. Its architecture is based on hierarchical layers of modules. At the top of the architecture there is the Application layer (the high-level modules capable of exploiting the hardware and the software resources of the system). This layer is organized into families of modules. Under the Application layer there is the Library layer. This layer allows the interface between the Application layer and the different managers (BIOS). The Manager layer is the HAL (Hardware Abstraction Layer) implementing the interface between the system call functions and the physical world (chips, interfaces, electronics, etc.). Moreover, it reduces the effort for porting the system onto other platforms. In parallel, this also increases the ease of portability and debugging of additional components of the RTOS, such as communication protocols.

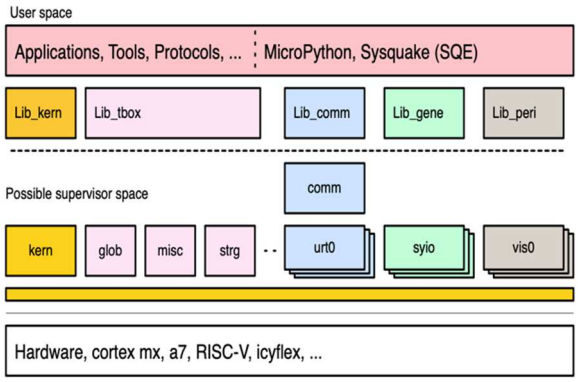

Figure 1: system architecture of µ111

A detailed description of a the different components is beyond the scope of this paper, but a few details about the Application Layer, which is the one we mostly used to realize our porting, are reported in the remainder of this section.

The Application layer is composed of families of modules designed to have a specific behaviour. Essentially, the boot procedure deals with starting the kernel processes (called Daemons) needed to run the basic operations of the OS. Once the kernel is up and running, the user can start from the Startup Console (typically a UART connection to a terminal on a PC, or in some cases a wireless console) any needed process, including wireless protocols like WiseMAC.

Process execution is governed by a system of priorities and in normal conditions the switch between processes occurs at a timeout or when a particular event happens (e.g., peripheral interruption). The OS kernel provides different mechanisms so that processes can interact with each other and with the kernel (presented in Section III.B). Peripherals can generate interruptions that are executed at a high priority level and interrupt the execution of any process.

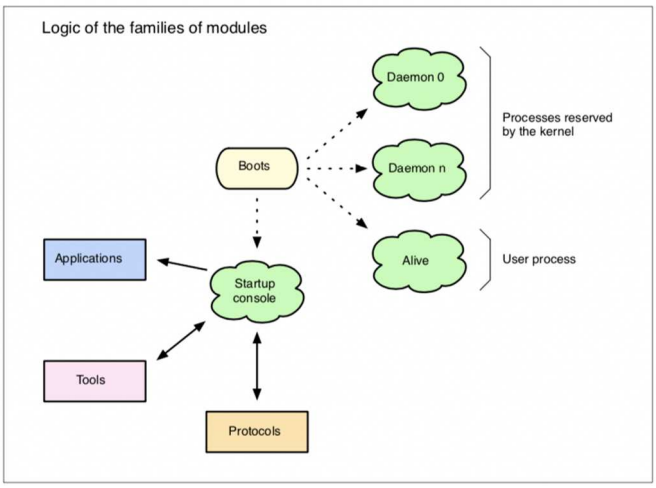

Figure 2: µ111 families of modules

B. Kernel functionalities for wireless operations

Among the services proposed by the RTOS, there are two functionalities which are of critical importance for our wireless protocol:

1. *Semaphore*: a functionality designed to guarantee exclusive access to shared resources. They are used to prevent two or more processes to simultaneously access the same resource. Since our wireless protocol needs to co-exist with other processes running on the OS, this functionality is essential to guarantee that critical resources like for example the radio are available and its usage is guaranteed to be exclusive.
2. *Signal and precise signal*: Signals allow processes to be synchronized with one or more events. A process can wait for a signal coming from another process or an interruption. A special class of signals is called precise signals. They allow the process to set up a signal that will be signaled after a specified amount of time. This functionality is maintained by means of a node's real-time clock that allows good precision. The event is signaled by an RTC interruption. The process that waits for a precise signal should have a high priority in order to ensure that it is scheduled by the OS when the event is signaled.

IV. WISEMAC: PRINCIPLES AND IMPLEMENTATIONS

In this section we describe in general terms how WiseMAC works and how it was implemented first on its OS-less original implementation and then on its µ111 version, focusing on the main difference between the two versions and the expected impact on the overall behaviour of the protocol.

A. WiseMAC principles

As mentioned before, WiseMAC [1] is a low-power protocol for WSN introduced in 2004 which belongs to the CSMA family of protocols with periodic preamble sampling.

The basic idea is that all the nodes belonging to the network periodically sample the channel (an extremely low power operation) and if there is no traffic for them to receive, they go back to their lowest consumption mode (sleep). The sampling schedule offsets are independent, meaning that when a transmitter wants to send a packet to an unknown node, it must use a so-called *Long Preamble*, i.e., it should transmit multiple copies of the same packet for a time interval

greater than the defined preamble sampling interval. On the receiving side, as soon as a node detects a transmission intended for it, it stays awake until the reception of the last copy of the message; if the packet is not intended for it, a listening node can detect this after the first received copy and go to sleep thus reducing the time of overhearing. This communication scheme is pretty expensive on the transmitter side, because a transmission should last for a relatively long time. To mitigate this issue, when the receiver receives the last copy of the packet, it acknowledges the correct reception to the transmitter and it additionally includes an information about its next wake-up interval. With this information, the original sender can now include the receiver in its local list of known neighbours, and it will be able to send an eventual future packet using a shortened wake-up preamble named *Short Preamble*, i.e., multiple copies of the packet for an interval which is only taking into account the drift of local clocks and not the entire preamble sampling interval (*Short Preamble* << *Long Preamble*). The knowledge of the sampling scheme of each direct neighbour is updated during every data exchange, and since a node is expected to have just a few direct neighbours, even nodes with limited memory are capable of storing this information. This communication scheme is depicted in Figure 3.

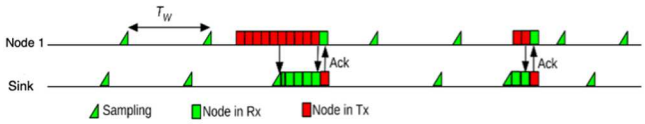

Figure 3: WiseMAC basic operations

When *Node 1* sends a packet to the *Sink* for the first time, it uses a Long Preamble (leftmost red squares); when performing a second communication to the same node, a Short Preamble is used (rightmost red squares).
For the interested reader, the work in [16] contains a detailed analysis of the power consumption needed to perform the main operations of WiseMAC.

B. OS-Less Implementation

WiseMAC operations are based on a state machine, with a state transition happening in one of two cases: a timer firing or a radio interruption. Since its first version in 2004, WiseMAC was implemented in an OS-less fashion, with a basic scheduler, essentially implemented as two FIFO (First in First Out) queues: a high priority queue and a standard priority one. This code is entirely based on interrupts to manage the radio operations and there is no possibility of multithreading. Pre-emption is possible by a proper configuration of interrupt priorities.
When running the code, the OS-less implementation features an infinite loop where the system periodically runs the next event present in the queues (it first checks the high priority queue, then the standard one) and if there is no event scheduled, then the processor goes in sleep mode. All the transactions required by the state machine of the protocol are represented as events to be placed in the queue that will be run at a later time by the scheduler. In case of an interrupt (timer, radio, etc.), the relative interrupt routine will be executed according to the state machine indications.

C. µ111 Implementation

The key point to address when porting the code to µ111 was to rewrite part of the core functionalities of the protocol to make them interact in the correct way with the RTOS blocks. This implied, namely

- Rewriting the radio driver according to µ111 format and keep it compatible with the protocol state machine.
- Protect the radio operations from concurrent access from other processes (Section III.B – Semaphores)
- Replace the timers of the OS-less version with the system calls available on µ111 (Section III.B – Signals)
- Interface the protocol with µ111 scheduler to have it work in a multithreading environment.

These operations need to respect three key constraints imposed by the protocol itself, without which the porting would be a useless exercise:

- Respect the timing constraints imposed by WiseMAC operations
- Limit the code executed in the interruption routines to not disrupt the multithreading structure of the RTOS
- Keep CPU operations and power consumption at their lowest possible value.

In the current implementation, WiseMAC protocol is executed as a µ111 process; it consists essentially of an infinite loop that waits for signals. While the process is waiting for the next signal, the OS scheduler can run another process or put the system in sleep mode to avoid unnecessary power consumption. Each signal has an associated callback that is executed by the process. Similarly, to the OS-less implementation, the state transitions are signaled by timeout and radio interruption. The timers are implemented with the precise signals presented in section III.B. The process is set to the highest level of priority to ensure its execution when a signal is signaled. Ideally other applications that run in parallel should have a lower priority and minimized the code that is executed with disabled interruption. The main difference with the OS-less version is the absence of the FIFO queues, which have been replaced by the OS-Scheduler, while WiseMAC became a stand-alone process which is part of the OS environment.

V. Performance Evaluation

The purpose of our analysis is to quantify the impact of using an OS on the performance of WiseMAC.
Our analysis is performed by running a series of identical tests on the same reference hardware, and by collecting and comparing the results of 4 metrics:

1. *Energy Consumption:* what is the difference (if any) between the two versions of the protocol?
2. *Memory Footprint:* in terms of memory space, is there any difference between the two versions?
3. *Latency:* measured from the moment when the packet is placed in the transmission queue to when it is delivered to the application layer of the receiver, what is the difference between the two versions?
4. *Packet Delivery Ratio (PDR):* is the overall number of correctly delivered packets affected by the RTOS or not?

Our experiments were performed on a *Wisenode VXI* board (WN), a custom hardware designed at CSEM. The WN features a Nordic Semiconductor NRF52840 CPU with integrated 2.4 GHz radio transmitter [17], a few onboard

sensors and an external 868MHz radio (not used either). To keep the evaluation as simple as possible in terms of energy consumption, we implemented our protocol to use the internal radio of the NRF52840, while all the other elements on the board are switched off, so the only component consuming energy is the processor itself with its integrated radio.
To avoid concurrent processes biasing the evaluation, during the experiments we executed WiseMAC as the only running process.

A. Energy Consumption

The consumption of the WN was measured using a Keysight CX3324A Current Analyzer, with an external power supply voltage of 3V. The radio transmission power is set to 0dBm.
We identified the following 5 main operations of WiseMAC expected to consume most of the energy:

- *Idle*, which does not consume a lot but it is the base operation running for > 99% of the time (IDLE)
- *Sampling,* short periodic operation implying the activation of the radio (SAMPLE)
- *Transmission* using a long preamble (TX-Long)
- *Transmission* using a short preamble (TX-Short)
- *Reception* of short preambles (RX)

We decided to not consider the energy needed for the reception of a long preamble because its value strongly depends on the random moment when the receiver starts listening to the transmitted packet with long preamble, thus its value is always biased by this unpredictable factor.
In the following table, we report the values we measured for the 4 operations we mentioned above. For each operation we indicate the average power consumption expressed in µA over a duration of 1 second. Each value is the result of an average over 10 different measurements.

Table 1: Energy consumption comparison

Consumption (µA)	OS-less	µ111
IDLE	46.7	46.7
SAMPLE	184.2	186.9
TX-Long	3397.7	3381.5
TX-Short	592.4	599.4
RX	372.3	380.5

As we can see, the consumption is very similar in both the OS-less and µ111. The difference we recorded between the two versions are so small that they are imputable more to randomness rather than indicate a real pattern, so we can conclude that the introduction of the RTOS has no impact on the overall energy consumption of WiseMAC.

B. WiseMAC performance

To evaluate the PDR and the latency we considered two different scenarios. We remind that WiseMAC is meant to be used in scenarios implying low traffic and quick reaction, as for example periodic monitoring applications with the possibility of quickly raising alarms in case of abnormal/dangerous values.
For both scenarios, each run of the experiment corresponds to the transmission of 100 packets, and we executed 5 runs for each version (OS-less or OS-based) to average results. The WiseMAC sampling period for each node is set to 250ms. The measures were performed in a controlled environment without any forced interference, apart from the Wi-Fi network of the office (one access point, one laptop connected to it) which has a negligible impact on the WiseMAC experiments.
In the first scenario, 2 WN are used with 1 node acting as the transmitter and the other as receiver. The transmitter emits packets at the rate of 1 packet/second. Results are summarized below.

Table 2: WiseMAC performance for scenario 1

Settings	Packet Delivery Ratio	Latency (ms)
µ111 WiseMAC	100%	546
OS-Less WiseMAC	100%	626.2

As we could expect in such a simple scenario, the PDR was perfect in each test, thanks to the reliability of WiseMAC and to the fact that there are no interference or collisions because only 2 nodes where considered. There is on the other hand an interesting indication for what concerns the latency: the OS version is quicker in delivering packets (about 80ms less). We will discuss this result in more details after the presentation of the second set of experiments.
In the second scenario, we used 1 node as receiver (sink) and 4 transmitter nodes. Each transmitter generates packets at the rate of 0.5 packets/second.

Table 3: WiseMAC performance for scenario 2

Settings	Packet Delivery Ratio	Latency (msec)
µ111 WiseMAC	99.9%	615.5
OS-Less WiseMAC	99.7%	652.1

As we can see by comparing the results from the two scenarios, the PDR remains pretty high in both cases, with the OS version performing slightly better than the OS-less one, but the difference (0.2%) is not significant.
For what concerns latency, on the other hand, we can notice that the use of an OS allows a reduction of the latency of about 12% in the first scenario and about 5% in the second.
We can explain the lower improvement in the second case as follows. The overall latency is due to two factors: the rapidity of the protocol and all the software application in managing the packets and the communications and the charge of the whole WSN which causes collisions and thus retransmissions of packets. In the first scenario, we do not expect any collision, so the latency value is essentially due to the responsiveness of the application. In the second scenario, the latency value is increased by the charge of the network. We can conclude that the RTOS application is about 12% quicker in managing packets and delivering them to the intended destination. This advantage diminishes when the impact of the charge of the network becomes the predominant factor in the overall latency. The lower processing latency of the OS can be explained by the fact that the OS guarantees a higher efficiency in terms of scheduling and inter-process communication.

C. *Memory footprint*

VI. Conclusions and Future Work

In this work we presented our porting of the well-established WSN protocol WiseMAC on a low-power RTOS (µ111). The vast majority of proposed WSN protocols were normally presented to the community as monolithic blocks of code, with no or little possibility to perform multiple operations at the same time. It is a common belief that an operating system will generally introduce a significant overhead to the core functions of the WSN nodes. Our experience shows that such a statement is at least misleading, since the performance of WiseMAC combined with µ111 is similar than that of an OS-less implementation. An interesting observation, which confirms the aims of an OS is that the OS implementation exhibits a lower latency and higher packet delivery ratio than the OS-less implementation. The reason is still under investigation, our assumption being that the µ111 RTOS is more thoroughly tested and debugged from a temporal point of view.

The possibility of running a protocol in a multitasking environment opens the possibility to run other application in parallel, like for example an automatic tool to analyse the network traffic or any other automatic process that could improve the quality of the proposed solution.

Apart from the additional memory required by the OS, using the µ111 OS has little to no impact to the protocol, both in terms of energy consumption and in terms of protocol performance.

After this promising outcome, µ111 will include in the future other types of WSN protocols in addition to WiseMAC, such as WiseMAC-HA (high availability, developed to guarantee an improved responsivity [19]), a TDMA protocol and hybrid solutions like WiseTOP (automatic switching between CSMA and TDMA [16]). WiseMAC itself will be improved by introducing the novelties brought by recent asynchronous WSN protocols, such as FAWR [20].

References

[1] A. El-Hoiydi, J.-D. Decotignie, C. Enz and E. Le Roux, “Wise-MAC: An Ultra-Low Power MAC Protocol for the WiseNET Wireless Sensor Network”, ACM 1-58813-707-9/03/0011, November 5–7, 2003, Los Angeles, California, USA

[2] El-Hoiydi, A.; Decotignie, J.-D.: WiseMAC: An Ultra-Low Power MAC Protocol for Multihop Wireless Sensor Networks, algosensors, First International Workshop on Algorithmic Aspects of WSN,2007.

[3] Levis P. et al. (2005) TinyOS: An Operating System for Sensor Networks. In: Weber W., Rabaey J.M., Aarts E. (eds) Ambient Intelligence. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-27139-2_7 http://www.tinyos.net/

[4] Dunkels, Adam (2004), "Contiki – a lightweight and flexible operating system for tiny networked sensors", Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks., pp. 455–462.

[5] FreeRTOS, https://www.freertos.org/

[6] Ramasamy, Anubhama & Rajendran, T. (2017). A Survey on Mac Protocols for Wireless Sensor Networks. 121-126. 10.15439/2017R26.

[7] Ye, Wei & Heidemann, John & Estrin, Deborah. (2001). An Energy-Efficient MAC protocol for Wireless Sensor Networks. IEEE INFOCOM. 2002.

[8] Dae T. & Langendoen K, An adaptive energy-efficient MAC protocol for wireless sensor networks, The 1st ACM Conference on Embedded Net- worked Sensor Systemes (Sensys 03), Los Angeles, CA, USA, November, 2003.03.

[9] Koen Langendoen and Andreas Meier. 2010. Analyzing MAC protocols for low data-rate applications. ACM Trans. Sen. Netw. 7, 1, Article 10 (August 2010), 34 pages. DOI=http://dx.doi.org/10.1145/1806895.1806905

[10] Chandel, Anita, Chouhan Vikram Singh, Sharma Sunil. 2021. A Survey on Routing Protocols for Wireless Sensor Networks. Chapter of book “Advances in Information Communication Technology and Computing”. Springer Singapore

[11] Farhana Afroz and Robin Braun, Energy-efficient MAC protocols for wireless sensor networks: a survey, Int. J. Sensor Networks, Vol. 32, No. 3, 2020

[12] Kuorilehto M., Alho T., Hännikäinen M., Hämäläinen T.D. (2007) SensorOS: A New Operating System for Time Critical WSN Applications. In: Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS 2007.LNCS, vol 4599. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73625-7_44

[13] Francesco Mondada, Edoardo Franzi, and Paolo Ienne , Mobile robot miniaturisation: A tool for investigation in control algorithms Experimental Robotics III, Proceedings of the 3rd International Symposium on Experimental Robotics, Kyoto, Japan, October 28-30, 1993, Springer Verlag, London, 1994, pp501-513.

[14] F. Mondada , E. Franzi , A. Guignard, The development of khepera, In Proc. of the 1st International Khepera Workshop, pp. 7—13, 1999

[15] Alberto Behar, George Bekey, George Friedman, Rajiv Desai, SUB-KILOGRAM INTELLIGENT TELE-ROBOTS (SKIT) FOR ASTEROID EXPLORATION AND EXPLOITATION, Space Manufacturing 11, Proceedings of the Thirteenth SSI/Princeton Conference on Space Manufacturing, May 8-11, 1997,

[16] Bergamini Lorenzo, Decotignie Jean-Dominique, Dallemagne, Philippe. (2018). WiseTOP: a multimode MAC protocol for wireless implanted devices. 41-50. 10.1145/3273905.3273919

[17] https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840

[18] O. Hahm, E. Baccelli, H. Petersen and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," in IEEE Internet of Things Journal, vol. 3, no. 5, pp. 720-734, Oct. 2016, doi: 10.1109/JIOT.2015.2505901.

[19] [19] J. F. M. Gerrits, J. Rousselot, J. R. Farserotu, C. Hennemann, M. Hübner, J.-D. Decotignie,, FM-UWB and WiseMAC-HA for Medical BAN Applications. , CSEM Scientific and Technical Report 2009.

[20] A. Pegatoquet, T. N. Le and M. Magno, "A Wake-Up Radio-Based MAC Protocol for Autonomous Wireless Sensor Networks," in IEEE/ACM Transactions on Networking, vol. 27, no. 1, pp. 56-70, Feb. 2019, doi: 10.1109/TNET.2018.2880797.