Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 4185650, 10 pages
https://doi.org/10.1155/2018/4185650

Research Article

An Adaptive Scheduler for Real-Time Operating Systems to Extend WSN Nodes Lifetime

Roberto Rodriguez-Zurrunero, Ramiro Utrilla, Elena Romero, and Alvaro Araujo

B105 Electronic Systems Lab, ETSI Telecomunicación, Universidad Politécnica de Madrid, Avenida Complutense 30, 28040 Madrid, Spain

Correspondence should be addressed to Roberto Rodriguez-Zurrunero; r.rodriguezz@b105.upm.es

Received 25 July 2017; Revised 29 December 2017; Accepted 9 January 2018; Published 6 February 2018

Academic Editor: Giovanni Pau

Wireless Sensor Networks (WSNs) are a growing research area as a large of number portable devices are being developed. This fact makes operating systems (OS) useful to homogenize the development of these devices, to reduce design times, and to provide tools for developing complex applications. This work presents an operating system scheduler for resource-constraint wireless devices, which adapts the tasks scheduling in changing environments. The proposed adaptive scheduler allows dynamically delaying the execution of low priority tasks while maintaining real-time capabilities on high priority ones. Therefore, the scheduler is useful in nodes with rechargeable batteries, as it reduces its energy consumption when battery level is low, by delaying the least critical tasks. The adaptive scheduler has been implemented and tested in real nodes, and the results show that the nodes lifetime could be increased up to 70% in some scenarios at the expense of increasing latency of low priority tasks.

1. Introduction

An operating system (OS) is a software layer that provides hardware abstraction and allows the developer to manage hardware resources. An OS also provides the developer standard mechanisms and services to ease and unify application development.

Therefore, the main advantages of using an OS are the software portability over heterogeneous hardware platforms and the ability to build application level developments regardless of the hardware used. OSes also provide other features such as multithreading capabilities or memory management. Wireless Sensor Networks (WSNs) are one of the most OSes demanding fields as these networks are composed by heterogeneous nodes where efficient hardware management is a main issue.

On the other hand, using an OS usually implies an overload in memory, CPU cycles, or energy consumption. This overload could be a critical issue in autonomous resource-constraint systems such as nodes present in WSNs. For this reason, OSes for WSNs must fulfil some specific requirements and features:

(i) *energy efficiency*, so the battery of autonomous wireless sensor nodes could last long periods;

(ii) *memory management tools*, in order to develop dynamic applications that use memory efficiently;

(iii) *real-time capabilities*, as most applications require bounded processing latencies of sensor data;

(iv) *wireless protocol stack*, which allows reliable and efficient communications in the nodes, while consuming low resources;

(v) *adaptability to the environment*, as WSN applications are heterogeneous and they usually operate in dynamic environments.

The scheduler is considered the core of an OS as it manages the tasks execution and could provide real-time management capabilities to the developer. Optimizing the scheduler is mandatory in OSes for WSNs in order to provide real-time multithread capabilities while using the lowest resources possible.

Our work proposes a scheduler that changes the task scheduling depending on environment conditions, which are

treated as inputs. Energy efficiency could be improved with this algorithm as the scheduler adapts dynamically to the environment when it changes. In this work, we use the device battery level and the tasks priorities as environment inputs in order to reduce energy consumption when the battery is running low, while maintaining minimum latencies for high priority tasks.

This paper is organized as follows. Section 2 presents the related works in the WSN OS field. In Section 3, the architecture of the scheduler is described. Section 4 shows the algorithm used for making scheduling decisions, while Section 5 describes the implementation of the algorithm in real nodes and the test scenario used. In Section 6, results are presented and discussed. Finally, the paper is finished in Section 7 with the work conclusion.

2. Related Work

Operating systems for WSNs are a highly studied area in the last decade since the first networks were deployed. Many of them have been developed during the last years, with TinyOS [1] and Contiki OS [2] being the most extended ones in WSN applications.

These OSes fit the requirements of WSN OSes, as they have a very small memory footprint while providing development abstraction. They also provide full network stack, simple memory management, and some multithreading capabilities. These OSes can run well in resource-constraint low-power microcontrollers, such as the Texas Instruments MSP430 used in TelosB, running at 8 MHz with 10 KB RAM.

However, new microcontrollers, such as low-power ARM Cortex-M ones, have increased available resources while maintaining very low energy consumption, reaching up to 120 MHz clock speed and 320 KB RAM. Therefore, these new devices allow the usage of more advanced OSes that employ fully preemptive threads and other features such as mutexes, semaphores, timers, or queues. Real-time operating systems (RTOS), such as open sourced FreeRTOS, may be used for WSN on these new microcontrollers. A priority scheduler or a round-robin scheduler may be used to implement a real-time OS. Several studies have been conducted to compare performance of both and to decide the best situation for using each method [3]. A round-robin scheduler shares the executing time with all active tasks, while a priority scheduler executes first higher priority tasks, reducing their latencies. Mixed strategies could be used as FreeRTOS does, where round-robin schedule is applied for same-priority tasks. However, for these OSes the main drawback is RAM usage, so several memory optimization techniques are presented by authors [4] to reduce it. Both thread optimization and memory allocation techniques could be useful for multithread RTOS.

Recent studies demonstrate that these real-time operating systems are being used in WSN monitoring systems [5], showing that a sensor network could be implemented even with real-time constraint over a wireless channel.

Other open issues regarding OSes are also being studied in the last years such as their steep learning curve and their power management features. RIOT OS [6] was developed in order to reduce the learning curve when programming IoT applications. This OS also provides real-time and built-in energy capabilities and energy-efficiency features. However, it uses a priority scheduler that does not share executing time between same-priority tasks and does not adapt their properties dynamically. On the other hand, improving power management of a multitasking WSN is also proposed by Brandolese et al. [7]. This management infrastructure and optimization model improves energy saving exploiting hibernation modes dynamically without memory retention. However, this method could cause losing real-time capabilities, as sensing tasks are grouped to improve energy efficiency, and they are not processed till a later time. Finally, CerberOS [8] presents a method to facilitate third party application design, by providing resource-secure capabilities in the nodes, allowing sharing them for different applications.

The distributed OSes for WSN field are also targeted by several research works in order to provide better management features and a highly transparent interface for the network developer [9, 10]. Load balancing of the nodes in distributed architectures has also been studied by Zoican et al. [11]. This work proposes a method for centralized task migration resulting in a final load near the average over all nodes of the network. Therefore, node cooperation and context aware methods will be critical issues for future WSN OSes developments.

Finally, other works target dynamic reconfiguration and operation of OSes. Lorien OS [12] was proposed as a fully component based operating system allowing efficient dynamic modules loading. On the other hand, an OS reconfiguration mechanism is proposed by Gasmi et al., [13] in order to provide efficient middleware that solves decision-making problems during reconfiguration stage. Besides, improving TinyOS tasks throughput while reducing energy consumption is achieved using a dynamic priority scheduler [14]. In this scheduler, the energy consumption is 1.14 times lower than the original TinyOS.

All these works show the interest in dynamic reconfiguration of OSes for WSNs. However, there are still open issues getting an adaptive scheduler that modifies its properties dynamically on changing environments. The main target of our work consists in improving WSN nodes lifetime in dynamic battery-operated environments by adapting dynamically a round-robin scheduler.

3. Scheduler Architecture

In this section, we explain an architecture for an adaptive scheduler that could modify its behaviour in changing environments. The global architecture is shown in Figure 1. The main idea consists in a module which accepts some environment inputs and makes scheduling decisions to modify scheduler properties. Some of the scheduler properties that could be dynamically modified are the duty cycle, the tasks priorities, and the scheduler system timer (Systick) period.

(i) The duty cycle is the portion the node CPU is running with respect to total time; the rest of the time the node is in low-power mode, also called sleep mode.

(ii) The tasks priorities allow managing tasks execution and real-time capabilities, as higher priority tasks are

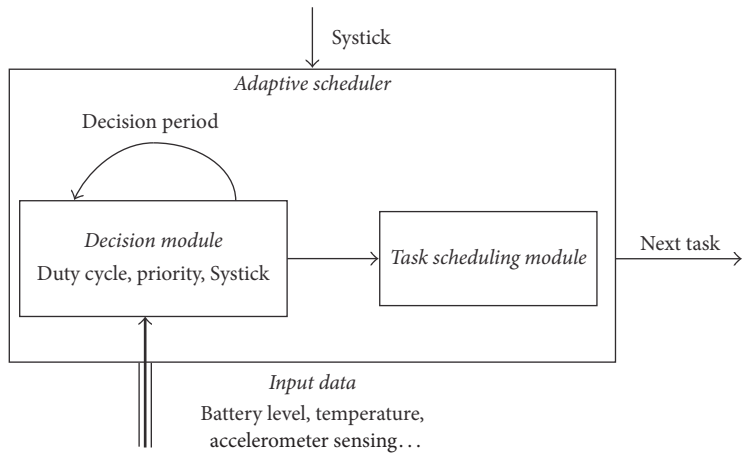


Figure 1: Adaptive scheduler architecture.

usually executed with lower latencies compared with low priority ones.

(iii) The Systick is the main timer of the scheduler, so the execution could be changed from one task to another in every system timer interrupt.

In order to make decisions, the scheduler architecture proposed uses a decision period. At the starting time of this period a decision is made in order to change scheduler properties. This decision period is a multiple of the Systick so we use only one timer for OS kernel management. The Systick period and the CPU clock frequency are constant in our scheduler, so the CPU does not change its executing speed dynamically.

This architecture is scalable to adapt dynamically any scheduler property although in this work we manage only the scheduler duty cycle. Our target consists in extending lifetime through scheduler duty cycling control while maintaining low latency for real-time tasks. Duty cycle represents the time when the microcontroller is active, so it is directly proportional with energy consumption.

In most schedulers, the active duty cycle is set by the tasks load, so all energy management is expected to be done by the programmer of each task. This way, a badly programmed task that never sleeps causes the CPU always to be active, so the duty cycle will be 100%. In the scheduler proposed, the node active time is fixed by the duty cycling decision module independently of the task load. A task which is in ready state could not be executed if the fixed active time has lapsed. This could cause increasing latencies for some tasks but save large amount of energy in some situations. To avoid this effect for real-time tasks the scheduler allows them to execute even if the active time has lapsed. Therefore, the adaptive scheduler saves energy by delaying low priority tasks. It should be noted that this is done without slowing them down as the CPU clock frequency does not change.

The flowchart of the adaptive scheduler process with duty cycling decision is shown in Figure 2. The schedule process is executed each Systick interrupt. First, it checks whether there is any active task or not. If not, it goes to sleep mode until the next Systick interrupt is triggered, so the process would start again.

On the other hand, if there are active tasks, it checks if the decision has lapsed. If it does, a decision must be taken in order to set a new duty cycle for the next decision period. This duty cycle sets the maximum available executing time for the tasks during this period. For example, if we set the Systick timer to 1 time unit and the decision period to 5 time units, we will have 2 time units as available active time if the decision-making process sets the duty cycle to 40%.

Whether or not a decision is made, the next step consists in checking if there is any available active time to execute tasks during this period. If not, the system checks if there is any task with highest priority, as we need them to be executed even if there is not available time for this period. If there are not highest priority tasks, the system goes to sleep mode until the current period finishes.

Finally, if there is any available time or there is any highest priority task, the next task to be executed will be scheduled in a priority-based round-robin way. The task will be executed during the Systick time, and when it lapses the process will start again.

In Figure 3 a time diagram example is presented comparing a priority round-robin scheduler with our adaptive scheduler. There are 3 tasks, with task 1 and task 2 being low priority tasks, having equal priority, and task 3 being the highest priority task. In this example, the decision period is 5 time units, while the decisions made set duty cycle to 40% for the three first ones and 20% for the last decisions. This way, the active time results in 2 and 1 time units, respectively. The Systick timer for both schedulers are set to 1 time unit.

The example shows the behaviour of our proposed scheduler. While round-robin scheduler executes all available tasks as soon as possible, the adaptive scheduler only executes the fixed duty cycle for each period. This causes low priority tasks to be delayed compared to round-robin scheduler. The result over large time scheduler operation will be a lower number of executions of these tasks which will lead to a large energy saving. On the other hand, high priority tasks, like task 3, execute the same way they do in a round-robin scheduler,

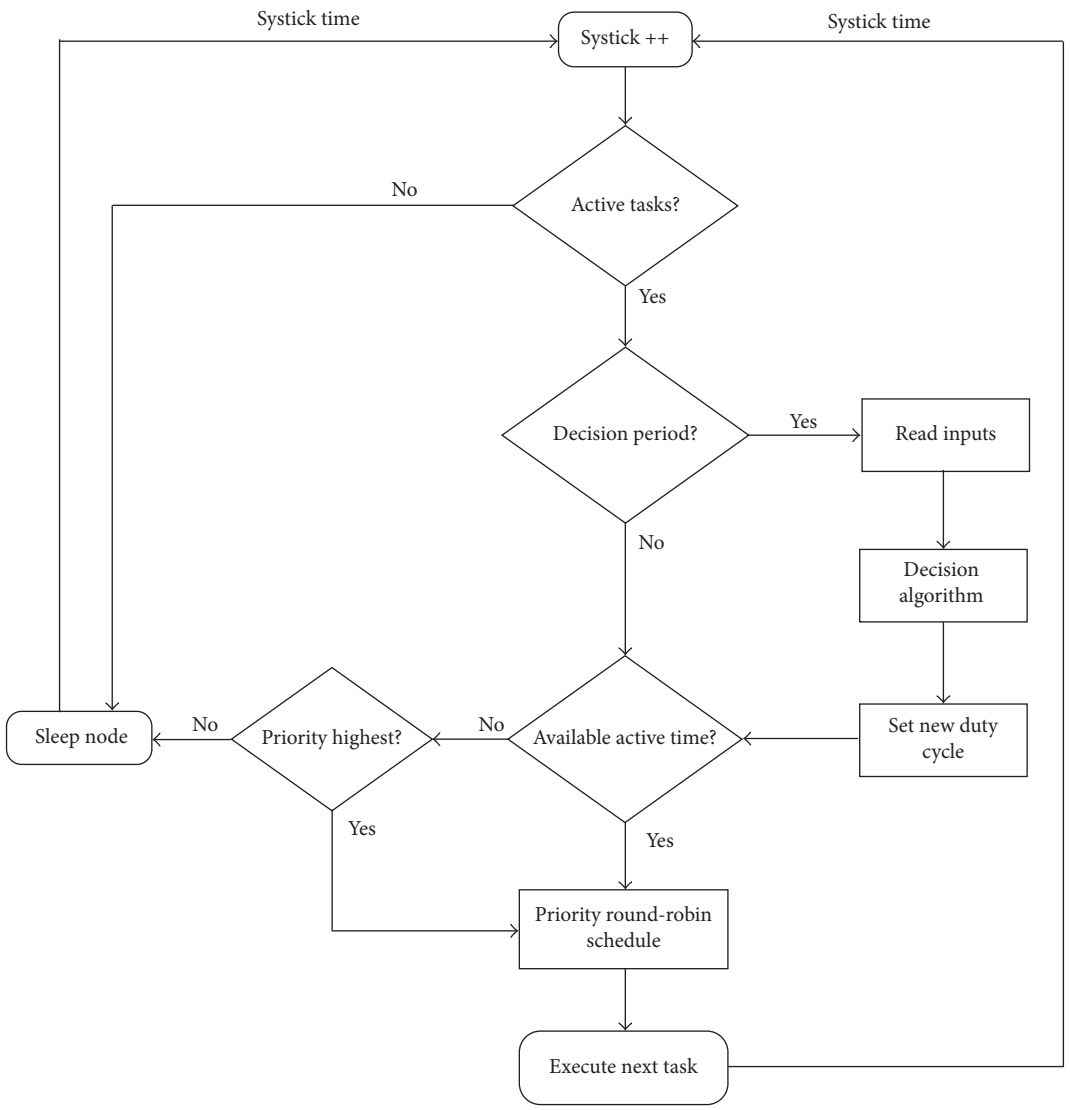
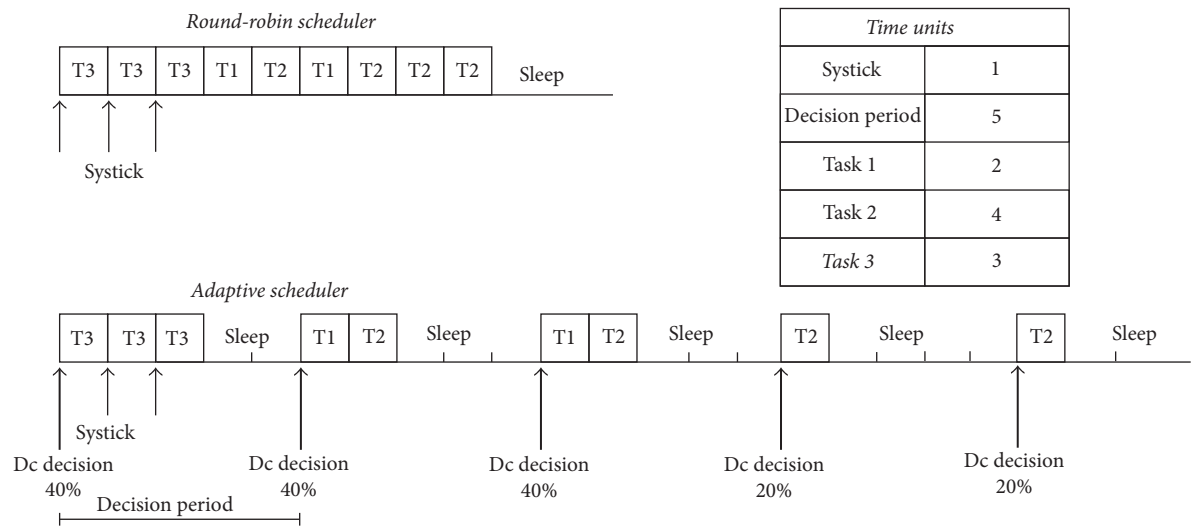


Figure 2: Adaptive scheduler flowchart.



Time units	
Systick	1
Decision period	5
Task 1	2
Task 2	4
Task 3	3

Figure 3: Time diagrams for round-robin and adaptive schedulers.

meaning no extra latency for them. In this example task 3 executes during 3 time units in both schedulers, so it is not delayed. On the other hand, task 1 lasts 6 time units in round-robin scheduler to complete execution, while it needs 11 time units in our scheduler. This delay in executing low priority tasks results in large energy saving as the system is in sleep state for a longer time.

In this work, we make duty cycle decisions, so node active time is changed dynamically. Input data used on this model could be either environmental parameters such as temperature, humidity, and RSSI or node parameters like battery level, energy consumption, tasks priorities, and execution state. This data could be collected each time a decision is made or could be stored in node memory and accessed by the decision module.

4. Duty Cycle Decision Algorithm

In this section, we present a duty cycle decision algorithm targeted at improving nodes lifetime. We use the approach proposed by Sirakoulis and Karafyllidis [15] which uses Public Goods Games (PGG) as a model to make decisions in power-aware embedded systems. This approach is based on Game Theory, which is a large field that studies mathematical models for making decisions in scenarios where rational players must use a shared resource. Players will take different decisions depending on the outcome of each one. On the PGG model, players compete for a shared resource and they cooperate optimizing their global outcome. This work [15] studies the effects of cooperation using a PGG applied to embedded systems on changing environments and presents a complete theoretical approach to these games. Besides, a global overview to Game Theory is also presented.

As described by authors of [15], the PGG is the most appropriate model for a scenario where there are power-aware jobs considered as players that should compete or cooperate for energy resources. Therefore, PGG provides a standardized formulation to solve the decisions proposed in our scheduler.

In our work, we propose a variation of the standard PGG problem in order to get a duty cycle value for each decision period. First, we define the global game parameters. The players of our game are each active task for the decision period and the shared resource is the execution time. The players could cooperate investing part of their available time in the decision cycle resulting in a global lower execution time for all tasks. This way, the lifetime could be extended when tasks decide to invest part of their time.

The investment done by each task in a decision cycle $I_i(t+1)$ is calculated in (1), where t and $t+1$ are indexes denoting the time step and i is the index denoting the current player (task). $I_i(t)$ is the investment done by a task in the previous decision cycle, adding a memory component to the algorithm, while F is a function of the reward $r_i(t)$ obtained each round for each task. The investment I_i is limited between 0 and 1 and represents the portion of time a task invests in order to save energy. The function F is then bounded in order to maintain the investment on its limits as shown in (2):

$$I_i(t+1) = I_i(t) + F\left(r_i(t)\right), \tag{1}$$

$$-I_i \leq F\left(r_i(t)\right) \leq 1 - I_i. \tag{2}$$

The parameter r_i represents the reward a task will obtain when investing part of its executing time. It is defined by the difference between the gain $g_i(t)$ obtained and the investment done in the previous decision period:

$$r_i(t) = \left(g_i(t) - I_i(t)\right). \tag{3}$$

The gain g_i depends on the state of the inputs defined for our scheduler decision module. Therefore, the gain changes in every cycle depending on the input values, so it defines the behaviour of the scheduler in changing environments. In our algorithm, we use the node battery level, the tasks priority, and a user-defined multiplication factor (MF) as input values.

In order to get the desired behaviour, the gain function is defined increasing with the multiplication factor and decreasing with task priority and battery level. This way, for high battery level or high task priority the gain value is low since the task is less likely to invest its executing time. We have defined the gain function in (4), where $M(t)$ is the multiplication factor, P_i is the normalized task priority, and $E(t)$ is the battery level. On the other hand, the parameters a, b, c, and k are fixed weights used to calibrate the behaviour of the scheduler:

$$g_i(t) = M(t)\left(\frac{a}{P_i} + \frac{b}{\left(E(t)+k\right)^2} + c\right). \tag{4}$$

It is important to note that the gain function defines the behaviour of the scheduler depending on the input data. This function could be modified in order to get a different behaviour or if we had other input data sources. Thereby both the function and its weights a, b, c, and k could be tuned depending on the desired behaviour. In our work, the gain function and its weights have been fixed to empirical values, which leads to a reasonable behaviour, in order to increase the gain when battery is low and the task priority is also low.

Finally, the reward function $F(r_i(t))$ is defined in (5). Its maximum and minimum values are F_{MAX} and $-F_{\mathrm{MAX}}$, respectively, that must meet the bounds set in (2). So, using the F_{MAX} value of (6) we can make sure the bounds are never exceeded. The reward function is linear between R_1 and R_2, which are fixed user-defined limits, and constant beyond these bounds. This function and the memory component of (1) make the scheduler response to changes slower, so the duty cycle will not change abruptly even if input values do:

$$F\left(r_i(t)\right) = \begin{cases} F_{\mathrm{MAX}} & r_i(t) \geq R_2 \\ \dfrac{2F_{\mathrm{MAX}}}{R_2 - R_1}\left(r_i(t) - R_2\right) + F_{\mathrm{MAX}} & R_1 \leq r_i(t) < R_2 \\ -F_{\mathrm{MAX}} & r_i(t) \leq R_1, \end{cases} \tag{5}$$

$$F_{\mathrm{MAX}} = I_i(t)\left(1 - I_i(t)\right). \tag{6}$$

Once the PGG has been formulated, the algorithm steps are presented in order to get the duty cycle of each period:

(1) Check the number of active tasks.

(2) For each active task,

(i) calculate reward value $r_i(t)$ from previous gain and investment values (3);

(ii) calculate reward function value $F(r_i(t))$ (5);

(iii) obtain the inversion of this cycle for this task $I_i(t+1)$ (1);

(iv) compute the gain value of this cycle $g_i(t)$ (4), which will be used in the next cycle.

(3) Calculate the arithmetic mean investment over all active tasks (7).

(4) Obtain the period duty cycle from the mean investment (8):

$$I_{\text{mean}}(t+1) = \frac{\sum_i^{N_{\text{tasks}}} I_i(t+1)}{N_{\text{tasks}}}, \tag{7}$$

$$D_T(t+1) = 1 - I_{\text{mean}}(t+1). \tag{8}$$

The duty cycle calculated has values between 0 and 1 as the investment has. The greater the investment made by all tasks is, the shorter the duty cycle of this period is. Therefore, by using this decision module in our adaptive scheduler architecture, the duty cycle is reduced when tasks decide to cooperate investing part of their time. This allows extending lifetime at the price of delaying low priority tasks that have decided to reduce their executing time.

5. Materials and Methods

The adaptive scheduler and the duty cycle decision algorithm proposed have been implemented in the YetiMote WSN node developed in the B105 Electronic Systems Lab which is shown in Figure 4. It is a custom-designed node composed by a high-performance low-power STM32L4 [16] microcontroller. The node runs up to 80 MHz with high memory capabilities (512 KB Flash, 128 KB RAM) and supports several low-power modes. In our test scenario, the microcontroller has 48 MHz system clock frequency. The node also has 2 accelerometers, a temperature sensor, an air quality sensor, a power management module, and 3 radio interfaces for 433 MHz, 868 MHz, and 2.45 GHz bands. A full version of FreeRTOS operating system is implemented on these nodes, which uses a priority-based round-robin scheduler. The tests performed are run on FreeRTOS scheduler in order to compare results with our adaptive scheduler.

The test scenario consists of 16 periodic tasks running a fixed time of 60 seconds for each test. The tasks periods are all different as well as their executing time in order to get the most realistic scenario possible when the tasks do not execute synchronously. In our tests, three scenarios have been defined depending on average task load. The task load is defined as the sum of the tasks active times divided by the total test time. Therefore, the tests have been performed using low task load (5%), medium task load (10%), and high task load (25%). Although 25% may not seem as a high executing load for most systems, in an energy constrained WSN scenario this task load is considered very high.

Figure 4: YetiMote WSN node used for testing the adaptive scheduler.

The input values used in our adaptive scheduler are the battery level $E(t)$, the normalized task priority P_i, and the user-defined multiplication factor $M(t)$. The battery level is limited to 0 when battery is discharged and 1 when it is fully charged. The scheduler implemented has 6 different task priorities, from 1 to 6, so the normalized task priority P_i is the quotient of the task priority and the total number of priorities. We have set the maximum task priority to 6 and the lowest task priority to 1. In all the scenarios 3 tasks are defined as high priority tasks, with priority levels 5 and 6, and the remaining 13 tasks have random priority values from 1 to 4. Finally, the multiplication factor allows the user to tune the scheduler behaviour dynamically, and it could have any positive value.

The adaptive scheduler parameters have been set to fixed values for all tests as well as the duty cycle decision algorithm parameters. The decision period value is 10 ms, while the Systick time value is 250 μs. On the other hand, gain function (4) parameters are $a = 0.5$, $b = 3.5$, $c = -2.5$, and $k = 0.7$, while reward function (5)bounds are $R_1 = -0.6$ and $R_2 = 1.2$. These values were empirically obtained after numerous tests in order to get a specific scheduler behaviour. Different values could be used to tune the scheduler if other behaviour is desired.

Each test measures the energy consumption and each one lasts 60 seconds. Therefore, different battery level or multiplication factor values are fixed for each test so we can evaluate energy saving on different input conditions. The energy consumption is obtained counting the time the microcontroller is in sleep mode during the test time. For that reason, we need to suppose 30 mW average power consumption when microcontroller is running and zero milliwatts when it is sleeping.

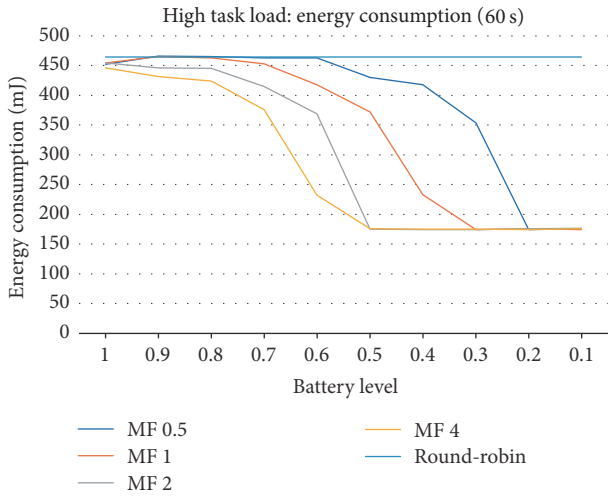


Figure 5: High task load test: energy consumption.

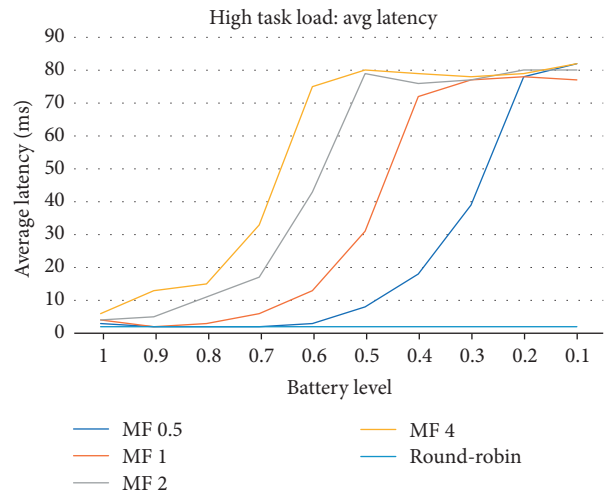


Figure 6: High task load test: average latency.

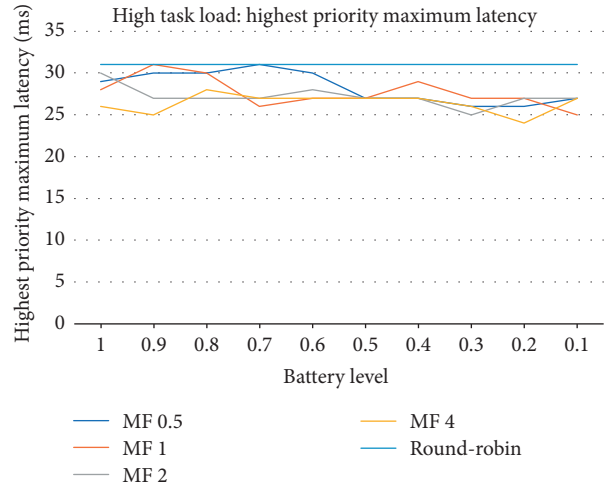


Figure 7: High task load test: maximum latency achieved by highest priority tasks.

The tasks latencies are also measured in order to evaluate how much the tasks execution is delayed in the proposed scheduler. We have measured the maximum and average latencies reached over all tasks during a test as well as the maximum and average latencies reached only by highest priority tasks, which should not be delayed in our adaptive scheduler.

For these tests, we use a modified version of FreeRTOS with most OS functionalities—in addition to the scheduler—such as memory management, tasks management, tasks communications, device drivers, and wireless stack. The tests are performed using the default FreeRTOS priority round-robin scheduler and using our adaptive scheduler in order to compare the performance of both.

6. Results and Discussion

For each of the three proposed scenarios, with different task load, tests have been performed varying the battery level from 1 to 0, with a step of 0.05. Therefore, up to 20 tests are executed for each scenario with different battery level values. Besides, the tests have been carried out with different multiplication factor values: 0.5, 1, 2, and 4. The priority round-robin scheduler has also been tested in order to compare the results with our scheduler.

First, we discuss the high task load scenario results. In Figure 5, the energy consumption is presented for this scenario over different battery levels and multiplication factor values. It can be seen that energy consumption is reduced when battery is discharging. That allows saving energy in low battery charge situations. The effect of the multiplication factor can also be noticed, represented in the figures as MF. Different MF values maintain the global behaviour. However, the scheduler starts saving energy at different battery level depending on MF. This way, the multiplication factor input may be used from user level to dynamically tune the scheduler behaviour.

Moreover, Figure 6 represents the average latency over all tasks and it can be seen that latencies are highly increased when battery level is low. However, Figure 7 shows that for highest priority tasks the maximum latencies are not increased as they have almost the same values as they do in the round-robin scheduler.

From now on, we will present the results only for multiplication factor 1, as this factor just tunes the scheduler behaviour maintaining the same functionality. For medium task load and low task load the results are quite similar, but moving the average energy consumption and task latencies to lower levels.

The results for medium task load are presented in Figures 8 and 9. The energy consumption is reduced when the battery level runs low and the task latencies are increased. The same behaviour can be seen in Figures 10 and 11 when the task load is low but displaced to lower values. Therefore, the results show that the scheduler behaviour is the same for different tasks sets, making the scheduler suitable for various applications with different tasks loads.

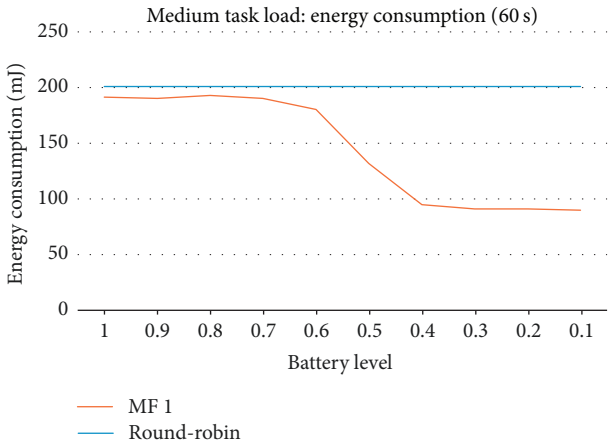


Figure 8: Medium task load test: energy consumption.

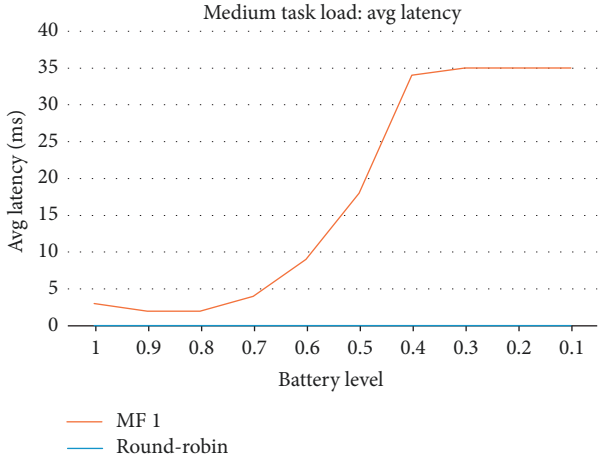


Figure 9: Medium task load test: average latency.

The latencies for highest priority tasks stand in the same level as in the round-robin scheduler, so real-time jobs could be performed with our scheduler even at low battery levels.

We also measure the overhead introduced by our scheduler in order to compare it to the overhead of a round-robin scheduler. In the tests performed the round-robin scheduler expends 58 milliseconds in the scheduling routines over 60-second tests. This time supposes 0.098% of the time which is despicable over the total time. On the other hand, our adaptive scheduler takes 83.4 ms during the scheduling routines and duty cycle decision algorithm. That means 0.14% of total time, which could be still considered despicable.

Finally, we obtain the expected lifetime of a node running our scheduler supposing it is powered by a 3000 mAh battery. Figure 12 shows the battery discharge rate of the round-robin scheduler compared with our proposed one for the three test scenarios and with a multiplication factor value of 1.

In this case the lifetime is extended from 48 days to 82 days, which means up to 71% increment. The lifetime is also obtained for medium and low task loads, which leads to 57% and 21% improvement, respectively. That means that our scheduler performs better with a higher task load.

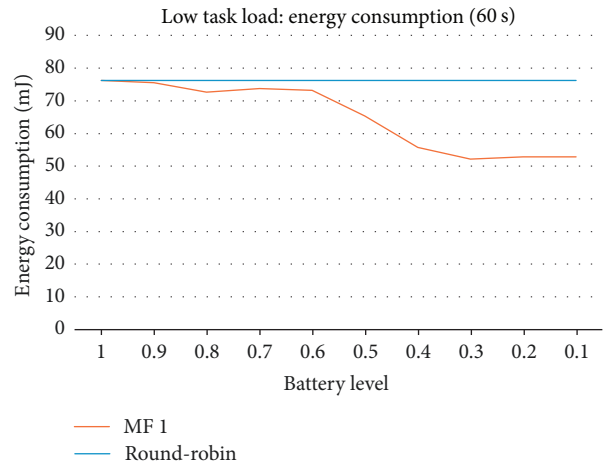


Figure 10: Low task load: energy consumption.

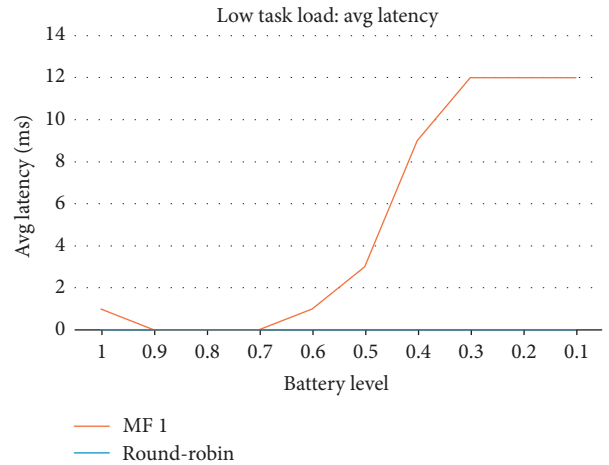


Figure 11: Low task load: average latency.

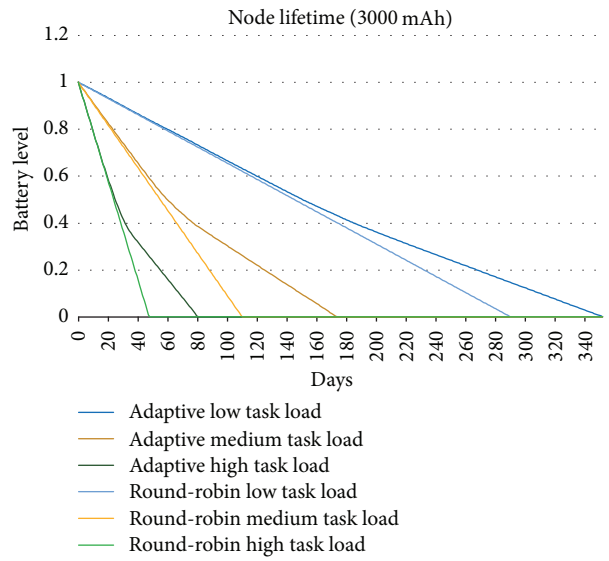


Figure 12: Node lifetime for test scenarios.

As the results obtained show, this scheduler could be highly suitable for battery-operated scenarios with energy harvesting sources. For example, if the nodes have solar panels as energy source, the battery is expected to be full during daylight hours, so the scheduler will run similar to a priority round-robin scheduler and low priority tasks will not be delayed. However, during night hours, the battery level will decay and the scheduler will start saving energy by delaying low priority tasks. This way, we could prevent the node running out of battery in cloudy days or in winter station when the night lasts longer than the day.

7. Conclusion

In this paper, we have proposed an adaptive scheduler architecture which makes possible change the task scheduling dynamically depending on the environment conditions. This could be very useful for WSN applications where changing environments are common. Specifically, we have targeted our scheduling algorithm at improving nodes lifetime, while it could be used for other optimization techniques in future works. The proposed scheduler changes dynamically its active duty cycle depending on battery level and tasks priorities. This leads to a large energy saving when battery charge is low and normal operation when battery is charged. For this duty cycle decisions, a PGG based algorithm is used and it is integrated in our scheduler architecture.

The scheduler proposed delays low priority tasks to achieve lower energy consumption, so they are executed with a higher period during low battery level states, which gives large energy saving. However, this latency does not affect high priority tasks as they are executed in all conditions, even when battery level is low.

Finally, the adaptive scheduler presented has been implemented and tested in real WSN nodes. The results show higher latencies when using our scheduler compared to a round-robin for low priority tasks. On the other hand, large energy saving is achieved and we can increase nodes lifetime up to 71% depending on the scenario.

The OS scheduler proposed is useful in many WSN scenarios to prevent nodes running out of battery by delaying noncritical tasks, while keeping high priority tasks running. This could lead to controlled degradation mechanisms for network nodes as they could maintain just critical functionality before the nodes run out of battery.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was partially funded by the Spanish Ministry of Economy and Competitiveness, under RETOS COLABORACION program (Reference Grants SONRISAS: RTC-2015-3601-3, All-in-One: RTC-2016-5479-4 and EASYSAFE RTC-2015-3893-4), and the Spanish Ministry of Industry, Energy, and Tourism through the Strategic Action on Economy and Digital Society (AEESD) under DEPERITA: TSI-100503-2015-39 and SENSORIZA: TSI-100505-2016-10 projects.

References

[1] P. Levis, S. Madden, J. Polastre et al., "TinyOS: an operating system for sensor networks," in *Journal of Ambient Intelligence and Smart Environments*, pp. 115–148, Springer, Berlin, Germany, 2005.

[2] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th IEEE Annual International Conference on Local Computer Networks (LCN '04)*, pp. 455–462, November 2004.

[3] M. Chovanec and P. Šarafín, "Real-time schedule for mobile robotics and WSN aplications," in *Proceedings of the Federated Conference on Computer Science and Information Systems, FedCSIS 2015*, pp. 1199–1202, Poland, September 2015.

[4] X. Liu, K. M. Hou, C. De Vaulx, H. Zhu, and J. Liu, "Memory optimization techniques for multithreaded operating system on wireless sensor nodes," in *Proceedings of the 2014 2nd IEEE International Conference on Progress in Informatics and Computing, PIC 2014*, pp. 503–508, China, May 2014.

[5] S. P. Patil and S. C. Patil, "A real time sensor data monitoring system for wireless sensor network," in *Proceedings of the 2015 IEEE International Conference on Information Processing, ICIP 2015*, pp. 525–528, India, December 2015.

[6] O. Hahm, E. Baccelli, H. Petersen, M. Wählisch, and T. C. Schmidt, "Demonstration abstract: Simply RIOT - Teaching and experimental research in the Internet of Things," in *Proceedings of the 13th IEEE/ACM International Conference on Information Processing in Sensor Networks, IPSN 2014*, pp. 329-330, Germany, April 2014.

[7] C. Brandolese, W. Fornaciari, and L. Rucco, "Power management support to optimal duty-cycling in stateful multitasking wsn," in *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013*, pp. 1123–1132, Australia, July 2013.

[8] S. Akkermans, W. Daniels, G. S. Ramachandran, B. Crispo, and D. Hughes, "CerberOS: a resource-secure OS for sharing IoT devices," in *EWSN 2017*, 2017.

[9] A. Sleman and R. Moeller, "SOA distributed operating system for managing embedded devices in home and building automation," *IEEE Transactions on Consumer Electronics*, vol. 57, no. 2, pp. 945–952, 2011.

[10] B. Pasztor and P. Hui, "OSone: a distributed operating system for energy efficient sensor network," in *Proceedings of the 2013 25th International Teletraffic Congress, ITC 2013*, China, September 2013.

[11] S. Zoican, R. Zoican, and D. Galatchi, "Improved load balancing and scheduling performance in embedded systems with task migration," in *Proceedings of the 12th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services, TELSIKS 2015*, pp. 354–357, Serbia, October 2015.

[12] B. Porter and G. Coulson, "Lorien: A pure dynamic component-based operating system for wireless sensor networks," in *Proceedings of the MidSens'09 - 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks, Co-located with the 10th ACM/IFIP/USENIX International Middleware Conference*, pp. 7–12, USA, December 2009.

[13] M. Gasmi, O. Mosbahi, M. Khalgui, L. Gomes, and Z. Li, "R-Node: New Pipelined Approach for an Effective Reconfigurable Wireless Sensor Node," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–14.

[14] Z. Jing, X. Leng, H. Fan, and C. Yi, "TQS-DP: A lightweight and active mechanism for fast scheduling based on WSN operating system TinyOS," in *Proceedings of the 27th Chinese Control and Decision Conference, CCDC 2015*, pp. 1470–1475, China, May 2015.

[15] G. C. Sirakoulis and I. G. Karafyllidis, "Cooperation in a power-aware embedded-system changing environment: public goods games with variable multiplication factors," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 42, no. 3, pp. 596–603, 2012.

[16] STMicroelectronics, http://www.st.com/en/microcontrollers/stm32l476re.html.