

YetiOS: an Adaptive Operating System for Wireless Sensor Networks

Roberto Rodriguez-Zurrutero
B105 Electronic Systems Lab
Universidad Politécnica de Madrid
Madrid, Spain
r.rodriguezz@b105.upm.es

Francisco Tirado-Andrés
B105 Electronic Systems Lab
Universidad Politécnica de Madrid
Madrid, Spain
firta@b105.upm.es

Alvaro Araujo
B105 Electronic Systems Lab
Universidad Politécnica de Madrid
Madrid, Spain
araujo @b105.upm.es

Abstract— Recent decades have experienced a breakthrough in the study of Wireless Sensor Networks (WSNs) as a key technology for Internet of Things (IoT). Operating Systems (OSes) for resource-constraint WSN devices have been developed during the last years, facilitating application development for these networks. However, these OSes are static software modules and do not take into account environmental changes to optimize themselves. In this paper we present YetiOS, a novel OS that can change its properties dynamically through an adaptive core. This way, the OS adapts itself to the environment and can optimize the performance, power consumption, security or tasks latency on different scenarios while maintaining the same developing interface. YetiOS also includes a dynamic layered network stack architecture, allowing different sensor network protocols to be implemented. Besides, it features Linux-like device drivers, a FreeRTOS based process management module, an I/O interface module and an energy management module. Performance tests presented in this work demonstrate YetiOS suitability for low-power resource-constraint WSN nodes. In addition, a use case of the adaptive core is presented, and its performance test demonstrates the dynamic optimization achieved.

Keywords— *Operating System, WSN, resource-constraint, adaptive, network stack, wireless, low-power*

I. INTRODUCTION

IoT systems have been a boost to WSNs research in recent years [1], [2]. The growing number of wireless devices has also caused an increasing interest in Operating Systems (OSes) for WSN to facilitate and improve application development. WSN OSes allow using a common development interface for different hardware platforms. In addition, efficient memory management, resource sharing or task concurrency are features that an OS can provide.

However, implementing an OS implies a processing and memory overhead which is an important issue in WSNs because of the resource constraints. Therefore, these OSes must be optimized in order to have the lowest memory footprint as well as the lowest CPU usage possible. Other important features like memory management, real-time support, networking support and energy efficiency must be taken into account when designing a WSN OS [3], [4]. These issues have been widely developed in existing OSes for WSN.

Since many heterogeneous IoT services and applications are emerging, future WSNs are also expected to self-adapt dynamically to environment and context changes to perform optimally in different scenarios. For example, cognitive radio algorithms [5] are adaptive methods that have long been applied to improve security, reliability and energy consumption issues by sensing the wireless channel and dynamically modifying wireless radio parameters. Other adaptive techniques have also been targeted in different WSN fields to optimize several features like routing protocols or Quality of Service (QoS) [6], [7]. Therefore, including adaptive capabilities has demonstrated the ability to improve the performance of WSNs, but this has not been applied to existing resource-constraint OSes.

In this paper, we present YetiOS, a novel OS for WSNs which includes an adaptive management module allowing the modification of its behavior depending on the environment. This module allows optimizing the execution of heterogeneous services and applications in different scenarios. YetiOS also features an advanced memory management module, real-time process management, tick-less idle management and a Linux-like device driver interface. Besides, a layered network stack architecture is included in YetiOS, and thus any layer can be

implemented independently. The adaptive core can also dynamically change network stack layers and their properties to optimize wireless communications, similar to cognitive radio.

This paper is organized as follows. Section II presents the related work in WSN OS field. Section III contains a full description of YetiOS architecture and its features. Section IV presents the performance tests done on the OS while a use case of the adaptive core is explained in Section V. Finally, Section VI presents this work's conclusions.

II. RELATED WORK

WSN Operating Systems have been widely studied in the last decades. This research has led to a classification of target devices depending on the RAM and ROM memory they have to support an OS [8]. Class 0 devices support OSes with RAM memory lower than 10 KB and ROM memory lower than 100 KB. Class 1 devices are larger and provide up to some tens of KB of RAM memory while class 2 devices provide up to 50 or 100 KB of RAM memory to implement an OS.

The first WSN OS developed was TinyOS [9], an event-driven OS that fits in tiny microcontrollers with 16 KB of program memory and 4 KB of RAM memory. It has been long used but its developing and multi-threading capabilities are limited. Contiki OS [10] was later released fitting 2 KB of RAM memory and 40 KB of program memory. Its programming model is event-driven and can also support some preemptive tasks. Both OSes include a network stack for wireless sensors that have long been used and improved over the years. Besides, some functionalities and features were added to these OSes such as program loaders, user interfaces, network simulators, IPv6 networking and file systems. Both TinyOS and Contiki OS are designed to run on class 0 devices whose RAM memory is lower than 10 KB. As multi-threading preemptive scheduling consumes high RAM memory, other OSes such as Mantis OS [11] and LiteOS [12] were developed to include this feature while fitting in class 0 devices.

Wireless devices with RAM memory larger than 1 MB allow using more advanced OSes such as iOS, Windows Phone and embedded Linux distributions like Android or Ubuntu Touch. However, these OSes cannot be used in most resource-constraint WSN nodes. This has led to the emergence of new OSes for IoT devices like Android Things [13] or Ubuntu Core [14]. They have enhanced networking and development capabilities, although their memory usage of some tens of MB is still too high even for class 2 WSN devices.

The emergence of new low-power microcontrollers with larger resources has allowed using more advanced OSes than Tiny OS or Contiki OS. Therefore, full real-time OSes like FreeRTOS can be used in devices with tens or hundreds of KB of RAM memory. This OS has been traditionally used in real-time embedded systems with large resources available, but nowadays it can be also used in class 1 and class 2 WSN nodes. This has led to the development of FreeRTOS extensions for the integration of WSN devices with IoT services, such as Amazon FreeRTOS. However, this OS just integrates devices with Amazon services and it is not interoperable with other IoT services. Other OS developed in the last years with large

features is RIOT OS [15] which allows real-time scheduling and claims to have zero learning curve when developing IoT applications. In addition, Mbed OS [16] includes security tools and preemptive multithreading for ARM devices while using static modular libraries. Most of these OSes include static networking capabilities with 6LoWPAN and RPL protocols. Finally, Zephyr [17] is a highly configurable real-time OS with a large developer community. Its main drawback is the compile-time configuration definition which does not allow loading and unloading OS modules dynamically.

WSNs are expected to be located in completely heterogeneous environments depending on their various possible applications and services. Therefore, existing static OSes are not optimized for all WSN applications, which has led to the introduction of adaptive techniques in the WSN field. In a previous work [18], we developed an adaptive task scheduler for real-time operating systems which demonstrates the benefits of using an adaptive algorithm to extend nodes lifetime up to 71%. Moreover, dynamic reconfiguration and decision making problems have also been addressed in other works [19] enabling adaptive capabilities for WSN nodes.

As stated, even through much work has been done in WSN OS field over the last years there are some features that can be improved for future WSNs. Wireless communications stacks are included in almost every WSN OS, but the tiniest class 0 OSes lack real-time capabilities and have limited developer features, while larger class 1 and 2 OSes lack any adaptive capability and do not allow dynamic changes. Therefore, in this paper we present YetiOS, which provides real-time scheduling capabilities, a layered network stack, Linux-like device drivers, an energy management module and fits in class 2 devices. In addition, unlike other OSes, it supports implementing adaptive algorithms that the literature has demonstrated to optimize different WSNs properties in heterogeneous scenarios [6], [7], [18], [19].

III. YETIOS DESCRIPTION

YetiOS has been developed to meet three key requirements for current and future WSNs. It must have a reconfigurable layered network stack. In addition, the development time must be minimized which is done by including real-time scheduling,

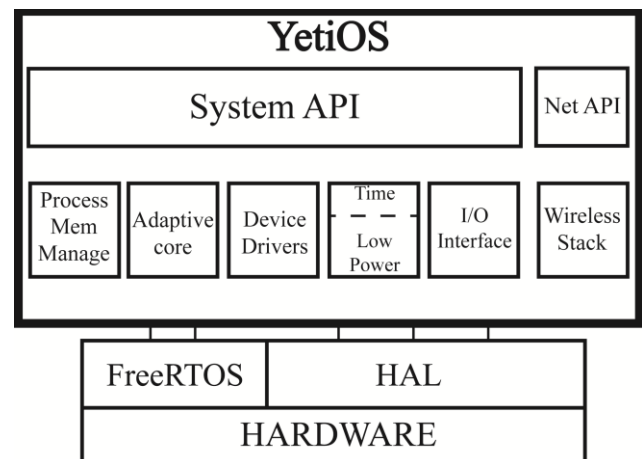


Fig. 1. YetiOS architecture

hardware management and energy management. Finally, it must provide an adaptive management module which allows implementing adaptive and context aware techniques to optimize network features.

A. YetiOS architecture

YetiOS follows a modular architecture which allows managing each module separately and to be dynamically enabled, disabled or modified. As a result, this architecture is more flexible than a layered one which has static relationships between layers.

The global architecture of the OS is represented in Fig. 1. YetiOS is based on FreeRTOS which constitutes the core of the system. FreeRTOS microkernel is implemented in YetiOS so the preemptive real-time scheduler and all FreeRTOS features are kept. This way, we maintain a reliable and well-known basis to build-up YetiOS. Besides, different microcontroller Hardware Abstraction Layers (HALs) can be used to manage their peripherals and develop device drivers.

YetiOS architecture provides two Application Programming Interfaces (APIs): the system API, that integrates the functions of every module to develop user applications and the Net API, that contains the functions needed to develop wireless communications. These two APIs integrate the functions that an application developer may need, which greatly reduces the learning curve and development time. This approach completely isolates the application from the hardware platform used, that becomes transparent for developers.

The main modules that make up the YetiOS core are:

- Process and memory manager module.
- Adaptive core.
- Device driver manager module.
- Wireless stack module.
- Time and low-power manager module.
- I/O interface module.

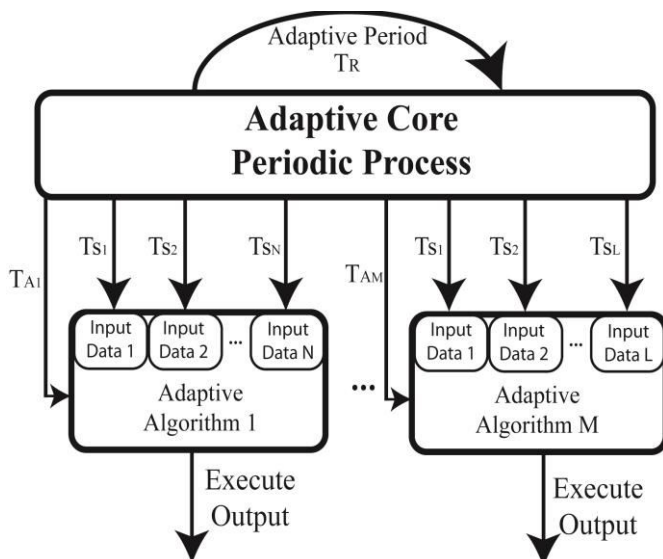


Fig. 2. YetiOS adaptive core architecture

All the modules except the process and memory management ones can be disabled by the developer to save resources when they are not needed. Besides, each device driver and application process are compiled as a module that may be dynamically loaded.

B. Device drivers

YetiOS implements a simplified Linux-like device driver interface to manage peripherals, sensors and other devices. This allows a faster developer learning curve by using a well-known Linux-like interface. Each driver is implemented in the OS kernel as an independent module, and can be loaded at runtime or at system initialization.

The functions that an application developer can use to manage device drivers are only five: “open”, “close”, “read”, “write”, and “ioctl”. The “open” function allocates and prepares a device instance, while “close” de-allocates the instance. The “read” and “write” functions receive and transmit, respectively, data stored in a buffer to or from an opened device. Finally, the “ioctl” function allows configuring the opened device. Therefore, new device drivers can also be developed using a well-known interface by implementing these functions for each particular device.

C. Energy management

Low-power operation is a key element of WSN devices, where node batteries should last as long as possible working as autonomous devices. Therefore, an OS for WSNs must optimize the energy consumption of the nodes by managing microcontroller low-power modes and network stack operation. In addition, the CPU load of the OS has to keep the energy consumption overhead minimum.

YetiOS performs tick-less idle scheduling which allows the microcontroller to enter low-power or “sleep” state – about 10 μ A of current consumption with only low-power clock domains activated – when it has no tasks to execute. The OS allows managing a low-power timer and a Real-Time Clock (RTC) allowing the applications to achieve high timing accuracy even when the microcontroller is in “sleep” mode. In addition, YetiOS includes an energy management utility that allows the developers to block entering in low-power modes in order to prevent DMA or peripheral transfer errors to happen.

D. Adaptive core

As mentioned in the related work section, adaptive techniques have proven to allow WSNs optimization in heterogeneous environments. However, these techniques have not been applied in WSNs OSEs, so we have included an adaptive module in YetiOS to implement these techniques in an efficient way.

The adaptive core is the main contribution of YetiOS. This module allows implementing and testing algorithms that dynamically change the system behavior to optimize its operation. Its architecture is shown in Fig. 2. The adaptive core uses a dedicated periodic process with the highest priority possible which is triggered every adaptive period T_R . The adaptive core also includes engines to create and manage generic input data modules and adaptive algorithms modules.

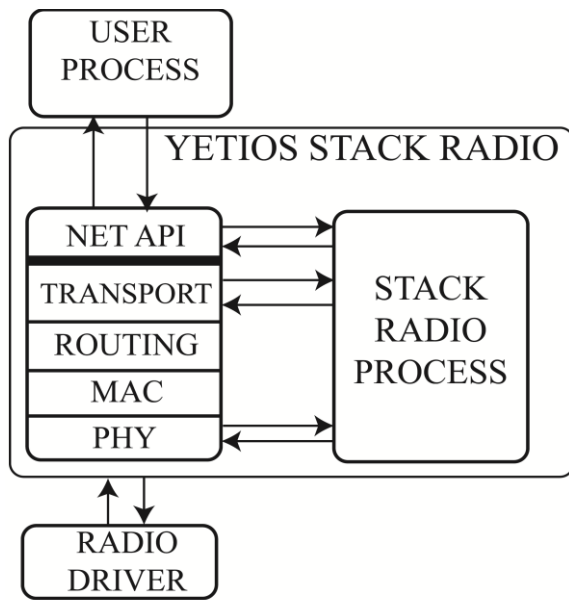


Fig. 3. YetiOS network stack architecture

This way, any number of adaptive algorithms can be loaded in the adaptive core, which allows configuring the algorithms used for each specific application. In addition, each hardware platform can implement different data input modules depending on its sensing capabilities.

Each input data module is defined by its sampling period T_{S_i} , its sampling function and a configurable buffer to store the samples. When an input module is loaded in the adaptive core, the adaptive process executes the sampling function at each time T_{S_i} , and stores the samples in the buffer. Therefore, the resolution of the sampling period is T_R , which is the period of the adaptive core process. It is important to note that these input data modules can represent any data source like sensor readings (humidity, temperature, accelerometry), node parameters (battery level, current consumption), radio and network parameters (RSSI level, data rate, packet error rate), or output data from adaptive modules used as feedback to implement learning algorithms.

The adaptive core also allows creating different adaptive algorithms modules. Each one is defined by its executing period T_{A_i} and its algorithm function. Besides, any number of input data modules can be linked to an adaptive algorithm. Therefore, when an adaptive algorithm module is loaded, the adaptive process runs the algorithm function each time T_{A_i} . This function uses the data sources linked to perform the necessary calculations and depending on the results an output action may be executed. This way, the OS can dynamically modify its operation by executing different output actions to change parameters like scheduling duty cycle, tasks priorities, radio duty cycle, or to load and unload OS modules.

It is important to note that the algorithms implemented must be as lightweight as possible to reduce their overhead. Using large T_R , T_{S_i} and T_{A_i} values reduces the adaptive core overhead but increases the algorithms response time, so a compromise value must be chosen depending on the application. Therefore,

the developer has to evaluate the dynamic optimization obtained by each algorithm individually and the optimization benefits must compensate the overhead added.

E. Network Stack

YetiOS implements a network module which contains a layered stack architecture to perform wireless communications. This way, each layer can be configured and designed independently facilitating new protocols development. As shown in Fig. 3 there are four layers defined: the physical layer, the Media Access Control (MAC) layer, the routing layer and the transport layer. The physical layer manages the hardware radio transceiver to allow communicating through it. The MAC layer manages the access to the wireless channels and also manages the duty cycle of the radio interface. The time the radio transceiver is active or in low-power state is defined by this radio duty cycle so it is a critical issue in WSNs applications. The routing layer can provide multi-hop wireless communications as well as broadcasting, unicasting and flooding communications. The transport layer allows establishing point to point communications while providing some services like fragmentation, flow control, reliability and multiplexing.

As it can be seen in Fig. 3, there is a net API on top of the network stack that provides functions to establish both connectionless-oriented and connection-oriented communications. Besides, the API includes the functions needed to manage routes and network addressing. This API has the same interface independent of the protocols implemented in the lower layers, so it can be used as a common interface for developing a networking testbench.

YetiOS uses a dedicated radio process to manage all the network stack operations. This means that the user processes do not actually run the network stack protocol when it calls a net API function. Instead the net API triggers the stack radio process and this process executes the routines necessary in the network stack. This mechanism has two main advantages. First, it isolates user processes from stack radio operation which could prevent errors. In addition, there is only one process needed for all network stack operations, reducing the overhead compared with a network stack that uses multiple processes.

The current YetiOS release provides custom lightweight protocol layers. The physical layer implemented allows managing CC2500 and SPIRIT1 low-power transceivers. The MAC layer implements address filtering and basic radio duty cycling, while the routing layer implements multi-hop mesh networking with auto-route discovery. Finally, the transport layer provided implements reliable TCP-like connections and unreliable UDP-like communications.

F. I/O Interface and developing features

The OS presented in this paper also features an I/O interface to ease applications developing and testing. This interface includes a Shell that can be configured by developers to add or delete new commands. This Shell communicates using a Standard Input-Output (STDIO) interface that may be set to use a USB port or a UART port. There also exist functions to print or read data that uses the STDIO interface.

TABLE I. YETIOS MEMORY FOOTPRINT

Enabled features	Flash memory	RAM memory
All ON	93 KB	15 KB
Stack OFF	75 KB	9 KB
Drivers and stack OFF	59 KB	7 KB
All OFF	51 KB	6 KB

Finally, it is important to note that the YetiOS is built using the Eclipse Integrated Development Environment (IDE). Development time is highly reduced by using this IDE instead of using complex Makefile systems as other WSN OSes do.

IV. YETIOS PERFORMANCE

This section presents some performance tests done on YetiOS. All the tests have been performed on a YetiMote WSN platform [18], which uses an STM32L476RE microcontroller with 512 KB of Flash memory and 128 KB of RAM memory. This microcontroller runs up to 80 MHz, but we use the 48 MHz low power clock to run our tests. YetiMote platform includes three radio transceivers for 433 MHz, 868 MHz and 2.45 GHz bands, one accelerometer sensor and one temperature sensor.

First, memory footprint of YetiOS has been measured depending on the modules enabled. All the tests were performed with one periodic blinking led process running in the node and the program compilation was done using GCC with size optimization. No adaptive algorithm module or data input module was loaded in these tests. Table 1 shows the complete memory footprint of four test programs, which also includes the memory used by FreeRTOS and HAL firmware. The test program with all modules enabled fits in 93 KB of Flash which is 18% of STM32L476RE available memory, and 15 KB of RAM which is 11%. The table also shows the footprint when the network stack is disabled, when the device drivers and network stack is disabled, and when all core modules are disabled except the process manager module. Disabling different modules allows greatly reducing the memory used by YetiOS.

These results show that microcontrollers with RAM memory larger than 20 KB have enough resources for running YetiOS and several user applications. In the same way, microcontrollers with Flash memory larger than 256 KB are recommended when developing YetiOS applications.

A runtime test has also been performed to measure the processing overhead when using YetiOS. To gather these data we have used Percepio Tracealyzer software [20] that provides runtime information of FreeRTOS based programs. For this test, all modules are enabled and there is one application sending one radio packet every two seconds. The packet is sent in application level containing 64 bytes of data and using a 433 MHz radio interface with 250 Kbps data rate. In Fig. 4, the CPU load of YetiOS is shown when running the test process for 60 seconds. The graph shows 7 tasks running in our system. “Startup” task is only executed in the system initialization.

“TzCtrl” is the Tracealyzer task used to acquire and store stats, so this task would not exist in a real application. “StdioProcess” is used by STDIO subsystem to allow sending and receiving data on the standard interface. “TestProcess” is the test application developed which sends a packet every 2 seconds. “NetstackProcess” is the task that runs the network stack and actually sends the radio packet of the “TestProcess” task. “TmrSvc” is the task that manages and processes the internal software timers used by the system. Finally, “AdaptCoreProcess” is the process in charge of managing the adaptive core and running the adaptive algorithms. Besides, this task manages the system watchdog and handles some error events of the OS. For this test, no adaptive algorithms or input data modules have been included as the CPU load and resource usage can vary a lot from one algorithm to another. Therefore, performance tests should be done to each algorithm individually.

The CPU load results of the runtime test presented in Fig. 4 show that YetiOS CPU load is below 1% as “TzCtrl” should not exist in a real application. This extremely low CPU load means that the microcontroller is in “sleep” state 99% of the time, which minimizes the power consumption and makes YetiOS suitable for WSNs low-power applications. Also, low CPU load adds low tasks wait times facilitating real-time applications development. Besides, it can be seen that the “TestProcess” has a very low CPU load demonstrating that the packet sending is done in the network stack task which is isolated from user applications.

V. USE CASE OF THE ADAPTIVE CORE

Finally, a simple example of a use case is presented in order to demonstrate the dynamic performance improvement achieved when using an adaptive algorithm. The use case proposed consists on a WSN application in which nodes have to provide a warning message when an event occurs. Some examples of this kind of application could be a wearable device that sends a message when it detects a fall by using an accelerometer, or a surveillance device that warns when a perimeter intrusion happens by using any kind of surveillance sensor.

In our use case we have used an ADXL355 accelerometer to sense the environment, like a wearable device does. When

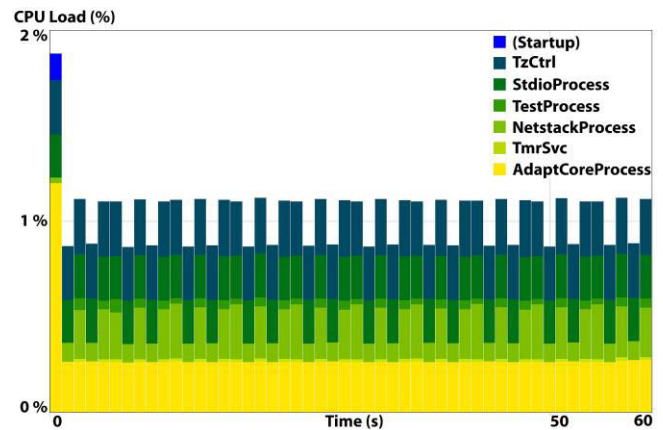


Fig. 4. YetiOS CPU load performance test

the peak-to-peak acceleration value sensed exceed a defined threshold, warning messages are sent using STDIO interface. These sensing and warning tasks are performed by a single process called “TestProcess”.

In addition, we have added an input data module to the adaptive core which takes a sample of the accelerometer with a sampling time T_{S_1} of 30 milliseconds. An adaptive algorithm module has also been loaded with the accelerometer input data module linked to it. The algorithm execution period T_{A_1} is 150 milliseconds. Besides, the adaptive period T_R has been set to 10 milliseconds. These values have been chosen for this example as a compromise between CPU load and response time. The larger these times are, the lower the average CPU load is, but the larger the response time is. In our use case example, there is a maximum response time of 150 milliseconds as it is enough for our demonstrating purposes. However, these values must be carefully chosen in real-world applications so they can meet their time requirements.

The basic adaptive algorithm loaded calculates the peak-to-peak value of the input data samples stored. When a threshold is exceeded, the OS netstack and STDIO modules are enabled, and a timer is started. This allows the “TestProcess” application to send the warning messages through the STDIO or the netstack interfaces. When the timer timeout (3 seconds) is reached, the netstack and STDIO modules are disabled as they are not expected to be used to send more warning messages. This way, resources used by these modules are only allocated when they are used, and they are released the rest of the time.

A performance test has been done for this use case to prove

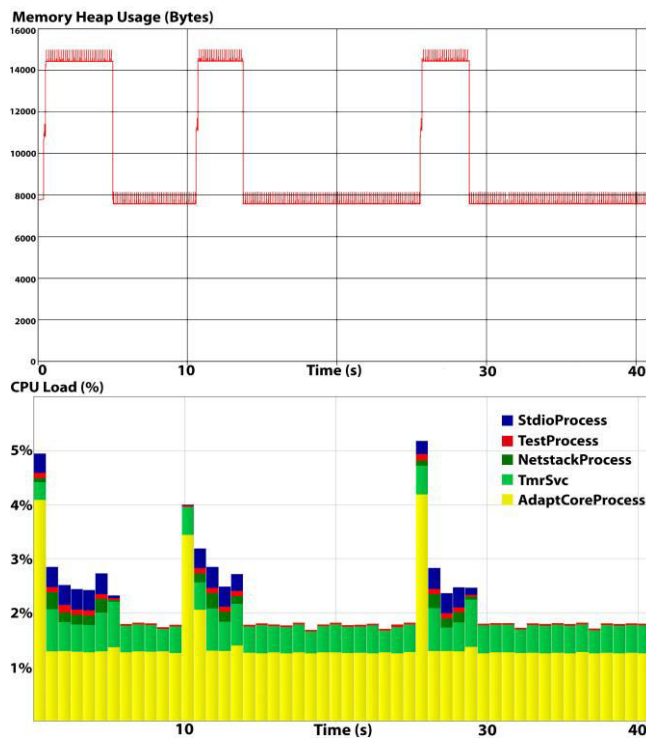


Fig. 5. Adaptive core use case performance test. Memory heap used and CPU load of the system.

the dynamic optimization achieved. This test lasted 40 seconds, and two accelerometer excitations were externally provoked at approximately 10 and 25 seconds. The test was performed on a YetiOS WSN node and the Percepio Tracealyzer software was used to get the runtime data. In Fig. 5 the memory heap used – dynamic memory – and the CPU load of the system are presented.

Regarding the memory heap used, it can be seen (Fig. 5 top) that at the beginning more than 14 Kbytes of memory were used by the system, as all the modules of the OS were loaded. After a few seconds, when the application startup is finished, the netstack and STDIO modules are unloaded so their reserved memory is freed. Therefore, the memory used falls to approximately 8 Kbytes at this moment. When the first excitation is provoked in the accelerometer (at 10 seconds) the adaptive core initializes the netstack and the STDIO modules to allow the Test Process to send the warning messages. Then, the system memory used increases again to more than 14 Kbytes for 3 seconds, which is the timeout fixed for this test. The same behaviour can be seen when the second excitation is provoked after 25 seconds.

On the other hand, the CPU Load graph (Fig. 5 bottom) presents the load of each process when executing this test. In this graph, the “Startup” and “TzCtrl” processes loads have not been displayed to facilitate its visualization. It can be seen that the “AdaptCoreProcess” load is higher than the one presented in Fig. 4. This is because for this test an input data module is used to sample the accelerometer and an adaptive algorithm is used to enable or disable netstack and STDIO modules, which increases this process CPU load to above 1%. In addition, the “TmrSvc” process also increases its CPU usage compared to the one presented in Fig. 4, as the ADXL355 driver implemented uses a timer to manage the peripheral. When an accelerometer excitation is generated, the global CPU load rises to about 2.5%, as the “StdioProcess” and the “NetstackProcess” are enabled. In addition, it is important to note that initializing the netstack and STDIO modules implies a large CPU usage for a short time by the process of the adaptive core. This causes an increase of the global CPU load to above 5% in those moments. The rest of the time, when no acceleration values above the threshold are detected, the global CPU load is about 1.5%.

These results demonstrate the suitability of the adaptive core to achieve dynamic optimization when developing WSNs applications, even when using a very simple adaptive algorithm. In the WSN warning application proposed as use case, it is expected that no warning events are detected the majority of the time. Therefore, the memory usage and the CPU load are reduced most of the time, as the STDIO and netstack resources are only allocated when they are needed.

VI. CONCLUSION

In this work, we have presented YetiOS, a novel operating system based on FreeRTOS that provides several new features for future WSNs. Existing works have demonstrated that WSNs can be optimized in different ways in heterogeneous changing environments by using adaptive algorithms like cognitive radio techniques, adaptive routing and QoS

techniques and adaptive OS scheduling duty-cycle techniques. The main contribution of YetiOS is an adaptive core that allows dynamically changing the system behavior as well as managing different adaptive algorithms to take advantage of their improvements. The adaptive core architecture proposed provides high flexibility developing applications and designing new dynamic optimization techniques. As far as we know, these adaptive capabilities are not included in any existing WSN OS.

A layered network stack architecture has also been implemented in YetiOS, providing network developers and researchers a very useful tool to evaluate and implement different wireless protocols. Besides, the layers of the network stack can be dynamically reconfigured which allows optimizing network properties depending on the environment. The modular architecture of YetiOS provides high flexibility for developers, allowing to implement processes and device drivers as independently loadable modules. Other features of YetiOS like the I/O Interface module, the process management module, the energy management module, the unified system API and the IDE based on Eclipse are key elements for reducing the learning curve and the development time.

Finally, tests performed have proven that a YetiOS full-version CPU load is lower than 1% and that it fits in less than 100 KB of Flash memory and less than 20 KB of RAM memory. In addition, the use case presented demonstrates that the adaptive core allows implementing algorithms to highly improve OS performance in changing environments. Therefore, YetiOS is suitable for WSN resource-constraint nodes while providing advanced features for researchers and developers.

ACKNOWLEDGMENT

The authors would like to thank George Alexandrov for his valuable comments. 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] L. Mainetti, L. Patrono, and A. Vilei, "Evolution of wireless sensor networks towards the Internet of Things: A survey," in *SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks*, 2011, pp. 1–6.
- [2] P. Rawat, K. D. Singh, H. Chaouchi, and J. M. Bonnin, "Wireless sensor networks: a survey on recent developments and potential synergies," *J. Supercomput.*, vol. 68, no. 1, pp. 1–48, Apr. 2014.
- [3] G. Strazdins, A. Elsts, K. Nesenbergs, and L. Selavo, "Wireless Sensor Network Operating System design rules based on real-world deployment survey," *J. Sens. Actuator Networks*, vol. 2, no. 4, pp. 509–556, Aug. 2013.
- [4] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: a survey," *Sensors (Basel)*, vol. 11, no. 6, pp. 5900–30, 2011.
- [5] D. Cavalcanti, S. Das, Jianfeng Wang, and K. Challapali, "Cognitive radio based Wireless Sensor Networks," in *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, 2008, pp. 1–6.
- [6] S. Ezdiani, I. S. Acharyya, S. Sivakumar, and A. Al-Anbuky, "Wireless Sensor Network softwarization: Towards WSN adaptive QoS," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1517–1527, Oct. 2017.
- [7] G. Serpen, J. Li, and L. Liu, "AI-WSN: Adaptive and intelligent Wireless Sensor Network," *Procedia Comput. Sci.*, vol. 20, pp. 406–413, Jan. 2013.
- [8] Arrow, "IoT Operating Systems | Arrow.com," Arrow, 2016. [Online]. Available: <https://www.arrow.com/en/research-and-events/articles/iot-operating-systems>. [Accessed: 21-Mar-2018].
- [9] P. Levis *et al.*, "TinyOS: An Operating System for Wireless Sensor Networks," in *Ambient Intelligence*, Springer, 2005, pp. 115–148.
- [10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, 2004, pp. 455–462.
- [11] S. Bhatti *et al.*, "MANTIS OS: An embedded multithreaded Operating System for wireless micro sensor platforms," *Mob. Networks Appl.*, vol. 10, no. 4, pp. 563–579, Aug. 2005.
- [12] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS Operating System: Towards Unix-Like abstractions for Wireless Sensor Networks," in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, 2008, pp. 233–244.
- [13] Google, "Android Things." [Online]. Available: <https://developer.android.com/things/index.html>. [Accessed: 15-Mar-2018].
- [14] Ubuntu, "Ubuntu Core documentation." [Online]. Available: <https://docs.ubuntu.com/core/en/>. [Accessed: 15-Mar-2018].
- [15] E. Baccelli, O. Hahm, M. Gunes, M. Wahlsch, and T. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHOPS)*, 2013, pp. 79–80.
- [16] ARM MBED, "Mbed OS | Mbed." *ARM MBED*. [Online]. Available: <https://www.mbed.com/en/platform/mbed-os/>. [Accessed: 21-Mar-2018].
- [17] Zephyr Project, "Introducing Zephyr — Zephyr Project Documentation." [Online]. Available: http://docs.zephyrproject.org/introduction/introducing_zephyr.html. [Accessed: 27-Jul-2018].
- [18] R. Rodríguez-Zurrutero, R. Utrilla, E. Romero, and A. Araujo, "An adaptive scheduler for real-time Operating Systems to extend WSN nodes lifetime," *Wirel. Commun. Mob. Comput.*, vol. 2018, pp. 1–10, Feb. 2018.
- [19] M. Gasmí, O. Mosbahi, M. Khalgui, L. Gomes, and Z. Li, "R-Node: New Pipelined Approach for an Effective Reconfigurable Wireless Sensor Node," *IEEE Trans. Syst. Man, Cybern. Syst.*, vol. PP, no. 99, pp. 1–14, 2016.
- [20] Percepio, "Tracealyzer for FreeRTOS documentation - Tracealyzer for FreeRTOS." [Online]. Available: <https://percepio.com/docs/FreeRTOS/manual/>. [Accessed: 17-Mar-2018].