

Comparison of IoT constrained devices operating systems : A Survey

Challouf Sabri, Kriaa Lobna, Saidane Leila Azzouz
 CRISTAL Laboratory, National School of Computer Sciences
 University of Manouba
 Manouba, Tunisia
 {sabri.challouf, lobna.kriaa, leila.saidane}@ensi-uma.tn

Abstract—As it refers to the ever-growing network of physical objects that feature an IP address for internet connectivity, the Internet of Things (IoT) is becoming very challenging for research community and domain. Operating systems for IoT devices play a vital role in this regard. These low-end devices can not use traditional operating system (OS) such as Windows or Linux due to stringent resource constraints. A lot of efforts have been made to design operating systems for these concerned devices. The main contribution of the paper is the comparative study of the most recent operating systems for low-end IoT devices. The comparison will focus on the major OS keys as architecture, scheduling, real-time capabilities, programming model, memory footprint, network connectivity, hardware support and energy efficiency. We highlight these OS keys values at the selected OSs : FreeRTOS, Mbed, Contiki, TinyOS and RIOT.

Index Terms—internet of things; operating systems; comparison; constrained devices, freeRTOS; mbed; contiki; tiny; riot

I. INTRODUCTION

IoT is an additional layer of information, interaction, transaction and action which is added to the Internet thanks to devices, equipped with data sensing, analysis and communication capabilities, using Internet protocols [16]. It is rapidly evolving and statistics estimate by 2020 that IoT devices will achieve 50 billion connected devices.

The architecture of IoT systems involves a large number of end-nodes (constrained devices) connected to gateways as aggregating devices, which in turn are connected to remote cloud platforms [2].

IoT needs complexity to be managed in order to scale up to billions of devices. The OS acts as a resource manager for complex systems. In a typical system, these resources include processors, memories, timers, disks, network interfaces, etc.

According to a VDC Research report [21], the IoT and embedded OS market is forecasted to grow from just under \$1.5B in 2014 to \$1.7B in 2019 at a 5 year compound annual growth rate (CAGR) of 2.9%. Also, VDC forecasts a surge in worldwide unit shipments (with commercial and non-commercial OSs combined) from 11.0B units to over 17.6B at a 5-year CAGR of 9.8% [21].

Leading vendors are pushing further down into small-footprint OSs and focusing on Real-Time Operating System (RTOS) and Fully-Functional Operating System (FFOS). With the RTOS and FFOS markets coexisting in parallel, the embedded OS industry has remained relatively static over the past

few years. Currently, the IoT OS market is experiencing a rapid paradigm shift essentially due to the maturation of Linux and the need to rethink value creation and revenue generation [21].

The next figure portrays survey results when respondents are asked to identify which operating systems they employ for IoT [12], [13], [14].

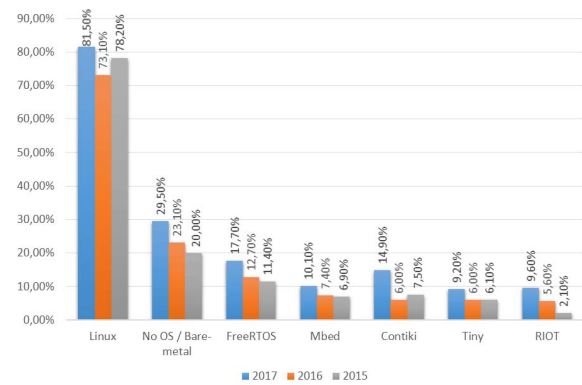


Fig. 1. Survey results for Operating Systems used for IoT Devices [12], [13] and [14].

More than 70% of IoT devices use Linux and most other mainstream OS range in the single digit percentages. Linux dominates the IoT OS market for gateways devices. For low-end IoT devices, traditional OSs are not suitable as they cannot run on the limited resources afforded by the hardware. Simultaneously, some functions like wireless support are difficult to build from scratch without an operating system. For these situations, low footprint real-time operating systems have become the answer in recent years, and they are now increasingly being deployed in applications that previously had no formal OS.

The advantages of openness have fuelled the growth of several open source variants of IoT OSs. Basing on surveys [12], [13] and [14], this article present a comparative analysis of most used open source low-end IoT OSs in last three years and will be organized into two parts. The first one details the major concerns for an IoT operating system which are architecture, scheduling, real-time capabilities, programming model, memory footprint, network connectivity, hardware

support and energy efficiency. The second part is dedicated to the description and comparison of the OSs : FreeRTOS, Mbed, Contiki, Tiny and RIOT according to the characteristics presented in the first part.

II. MAJOR CONCERNS IN IoT OS DESIGN

Operating systems is the interface between applications and physical resources. Mainly, the OS comprises of a kernel, utility software and a system shell. The kernel is the most essential and core part of the OS. It contains modules for managing various resources and it is the real master and controller of the hardware. Utility software supports the OS's operation and provides the kernel with essential software like debuggers, compilers, etc. System shells provide users with an interface to the kernel.

The key characteristics and requirements that differentiate an IoT OS are described below :

A. Architecture

The kernel is the core of an OS. The organization of the kernel constitutes the OS structure which has an influence on the size as well as on the way it provides services to the application programs. There are many OS architectures. Some of the well known ones are the monolithic and the micro-kernel architectures which cover the studied OSs.

A monolithic architecture does not have any structure. It is a single large process running entirely in a single address space. Its services are implemented separately and each service provides an interface for others. All system services are bundling together into a single system image.

A micro-kernel architecture has a minimalistic kernel. In that architecture the kernel is broken down into separate processes, known as servers. Some of these servers run in kernel space and some run in user-space. The kernel size is significantly reduced due to that only minimum functionalities are provided inside it [3].

Discuss :

Although the monolithic OS modules interactions costs are low, the system is hard to understand and modify, unreliable, and difficult to maintain, this make them a poor OS design choice for low-end IoT devices. In the other side, despite its poor memory performance compared to the monolithic kernel, a microkernel is the design choice for many embedded OS due to the small kernel size and the few number of context switches. Furthermore, this kind of architecture provides better reliability, ease of extension and customization since most of the OS functionalities are provided via user-level servers, like memory server and time server. In this case, if one server fails, the whole system does not crash unlike monolithic.

B. Scheduling & Real-time capabilities

Another crucial part of any OS is the scheduler, which affects other important properties such as energy efficiency, real-time capabilities, system performance, the programming model and latency, all depend on the scheduling algorithm.

There are two types of schedulers, preemptive and cooperative. A preemptive scheduler such as RTOS one assigns

CPU time to each task, while the different tasks have to yield themselves in the cooperative model.

For real-time capabilities, the scheduler should respect strict time constraints. This kind of scheduling should reach the processes deadlines so that tasks are completed within given time limits.

Moreover, the schedulers should be energy efficient and multitasking in IoT systems.

C. Programming model

The programming model defines how an application developer can model the program. The typical programming models for IoT OSs can be divided into event-driven and multi-threaded systems.

In an event-driven system, every task has to be triggered by an external event, such as an interrupt. This approach is often accompanied by a shared-stack model and a simple event loop instead of more complex scheduler. While a programming model based on multi-threading gives the opportunity to run each task in its own thread context, and communicate between the tasks by using an Inter Process Communication (IPC) APIs [3].

Discuss :

Thanks to their properties, multithreaded OSs can offer greater intellectual manageability through a simpler code, syntactic support for sequential service algorithms and the ability to take advantage of compile-time enhancements to improve performance. That is why thread-based designs seems to be the better choice for the implementation of highly available and scalable IoT applications.

D. Memory footprint

IoT devices are much more resource constrained than other connected machines especially in terms of memory. IoT devices provide typically a few kilobytes of memory, million times less than standards connected machines (laptops , smartphones, etc). This observation holds both for RAM and Flash memory [22].

Memory footprint values depend upon architecture of hardware platform, compiler settings and most importantly OS configurations which include kernel size and size of run-time libraries [32].

On the year 2014, the Internet Engineering Task Force (IETF) standardized in the RFC-7228 a classification of such devices in three subcategories based on memory capacity which are presented in table 1:

TABLE I
MEMORY CLASSES OF CONSTRAINED DEVICES [22]

Name	Data size (RAM)	Code size (Flash)
Classe 0	<< 10 Kib	<< 100 Kib
Classe 1	~ 10 Kib	~ 100 Kib
Classe 2	~ 50 Kib	~ 250 Kib

^{Kib}Memory value expressed in KiloByte = 1024 bytes.

Devices within the C0 classification usually consist of an MCU built on an 8-bit architecture, while C1 and C2 devices can range from 8-bit up to 32-bit architectures.

E. Network Connectivity & Protocol Support

The main point of having IoT devices is that they can interconnect, and communicate with other devices or with the Internet, thus IoT devices are typically equipped with one or more network interfaces.

IoT Communication techniques use not only a wide variety of low-power radio technologies like IEEE 802.15.4 and Bluetooth Low Energy (BLE) but also various wired technologies like ethernet.

The IoT stack must be flexible, so that with minimal changes it can be configured to meet the demands of a wide range of IoT applications. The support to Ipv6 is mandatory in IoT systems to have unique identities in tremendously large networks. An IoT OS should support for low power network protocols like 6LoWPAN (Low-power Wireless Personal Area Network over IPV6), RPL (IPv6 Routing Protocol for Low-Power and Lossy Networks) and CoAP (Constrained Application Protocol). Header compression and inclusion of minimal features help in keeping the protocols more viable [9].

According to the last developer surveys [12], [13], [14] and other network standards surveys for IoT like [9], most of the network protocols and standards are IEEE 802.15.4, Bluetooth (BLE), ZigBee, LPWAN (LoRa, Sigfox, etc), 6Lowpan, RPL, Coap and MQTT. These standards will be the focus in our network comparison.

F. Hardware support

From an architectural perspective, today's Internet hardware and protocols are not too diverse. For IoT the degree of heterogeneity explodes and a generic OS should support this. The large variety of IoT use cases led to the development of a large variety of hardware and communication technologies. IoT low-end devices are based on various microcontroller architectures and families, including 8-bit processor architecture like Intel 8051 and Atmel AVR, 16-bit architecture like TI MSP430, 32-bit architecture like ARM Cortex-M and MIPS32 and 64-bit architectures might also appear in the future.

G. Energy efficiency

Reducing power is not only green, it saves money and increase product life. Energy efficiency becomes essential for IoT devices powered by batteries and it should be taken into account when designing an IoT OS.

Unlike conventional network protocols, the Internet of things needs power-efficient protocols designed with power-efficiency in mind. In low-power wireless systems, the radio transceiver is typically the most power-consuming component, so power-efficiency translates into keeping the radio off as much as possible by using low power protocols like 6lowpan, RPL, LoRaWan, etc.

For the microcontroller consumption side, well-engineered designers work on minimizing power whenever practical. So

devices running off of batteries need power management for long battery runtimes. In this case to achieve an operation time of over one year in idle mode the device should use less than 10 μ A [18]. In the current technology used for microcontrollers, the primary source of power consumption occurs during switching phase.

Processors are able also to increase clock frequency to run faster as integrated circuits have become smaller. A faster clock boosts performance but also increases power levels. So slowing down or turning off the clock whenever possible reduce power levels. Many processors support to vary the clock frequency or even turn off the clock in Sleep mode. Some reduced static power levels will still be required even with the clock off to save the values in registers and volatile RAM memory, so that the device can wake up without a full reboot.

III. IOT OPERATING SYSTEMS COMPARISON

Our article cover the most used OSs for low-end devices during the years 2015, 2016 and 2017 according to Eclipse developer surveys [12], [13], [14]. These OSs are FreeRTOS, RIOT, Mbed, Contiki and Tiny OS. The next sub-section compare generally (usage and license) the selected OSs and the rest of subsections will compare to the major concerns detailed in section II.

A. General

All five OSs are open source nevertheless not all have the same license. FreeRTOS has Modified GPL license, Mbed has Apache v2.0 license, Contiki and TinyOS are made BSD license and RIOT is under LGPL v2.1.

According to [12], [13] and [14], RIOT and Contiki OSs have the high growth rate (7,5% and 7,4% respectively) in years between 2015 and 2017 then it comes to FreeRTOS with 6,3%. But if we focus on the growth usage every year, we observe that FreeRTOS and Contiki slopes are disturbed while RIOT has always increasing linear slope thinks to its rich features.

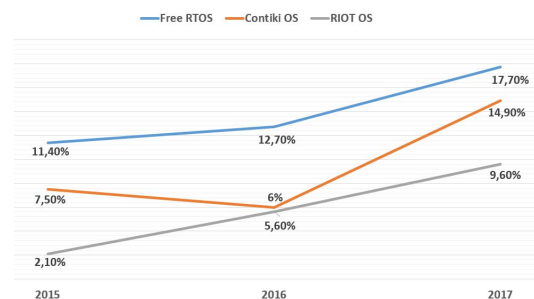


Fig. 2. FreeRTOS, Contiki and RIOT growth rate between years 2015 and 2017.

B. Architecture

Compared Operating systems are based on monolithic or microkernel architecture, table 2 shows every OS architecture.

TABLE II
OPERATING SYSTEMS ARCHITECTURES

OS	Architecture
FreeRTOS	MikroKernel RTOS
Mbed OS	MikroKernel RTOS
Contiki OS	Monolithic
Tiny OS	Monolithic
RIOT OS	MikroKernel RTOS

Most OSs are based on microkernel due to the small kernel size and the number of context switches. In addition, the real-time capabilities are very depending on the kernel architecture. This why OSs FreeRTOS, Mbed and RIOT have microkernel architecture and they are all full Real-Time OSs unlike Tiny and Contiki.

C. Scheduling & Real-time capabilities

The scheduling strategy of an OS is a key factor for the system's performance. Some IoT systems might have strict real-time constrains and an IoT OS must be able to provide the scheduling needed for these types of systems. IoT RTOS systems are generally based on a multitasking scheduler, therefore preemptive scheduler is extremely preferred.

TABLE III
OPERATING SYSTEMS SCHEDULING AND REAL-TIME CAPABILITIES

OS	Scheduling	Real-time
FreeRTOS	Preemptive	Supported
Mbed OS	Preemptive	Supported
Contiki OS	Cooperative	Partial Support
Tiny OS	Cooperative	Not Supported
RIOT OS	Preemptive	Supported

Due to the need of preemptive systems, Contiki tried to implement the preemptive behavior by using a hybrid model. This model is based on an event-driven kernel where preemptive multithreading is implemented as an application library linked optionally with programs that compulsorily require it.

To provide some real-time capabilities, Contiki added the *rtimer* library that provides scheduling of real-time tasks. This library pre-empts any running Contiki process in order to let the real-time tasks execute at the scheduled time. But comparing to RTOS, there is no implementation of any real-time process scheduling algorithm.

D. Programming model

The Programming model of an OS plays an important role in both performance and productivity. It should aid in increasing productivity for developers as well as utilizing an abstraction to the underlying system. The table 4 presents every OS programming model.

Tiny and Contiki OSs are Event-driven systems so they can be more memory efficient, while FreeRTOS, Mbed and RIOT OSs are multi-threading systems so more suitable in IoT application design.

In this context, the event driven Contiki kernel does not provide multi-threading by itself but their multithreaded applications use protothreads through optional libraries.

TABLE IV
OPERATING SYSTEMS PROGRAMMING MODELS

OS	Programming model
FreeRTOS	Multi-threading
Mbed OS	Multi-threading
Contiki OS	Protothreads and events
Tiny OS	Event Driven and TOS
RIOT OS	Multi-threading

As Contiki use protothreads to join multithreading, TinyOS version 2.1 provides also partial support for multithreading via TOS Threads which use a cooperative threading approach.

E. Memory footprint

Generally, IoT devices have less resources than conventional connected objects, therefore one requirement for a generic IoT OS is to be able to function on small memory devices. An IoT OS needs to provide efficient memory management in order to abstract the memory management for the programmer.

As shown in the next figure, for event-driven OSs Tiny OS still have the typical memory configuration and on multithreaded OSs, RIOT is the better choice.

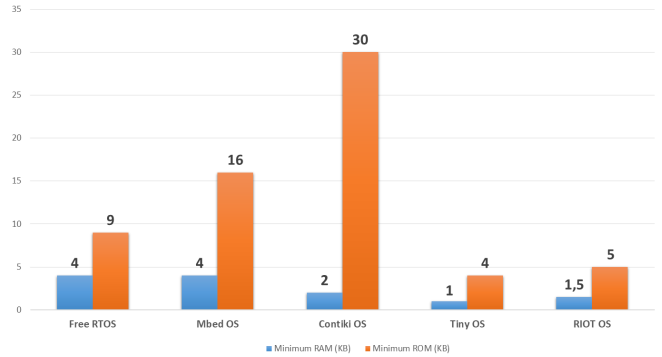


Fig. 3. Operating systems memory footprint in KiloBytes [8].

Basing on memories needs, FreeRTOS, Mbed OS and RIOT are all under class memories 1 and 2. Contiki is classified under classes 1 and 0 and the most memory footprint OS is the TinyOS belonging to the class 0.

F. Hardware support

A key challenge for a generic IoT OS is the ability to run on different hardware platforms. MCU's from 8-bit up to 64-bits are available on the market. Ideally, a generic IoT OS should have support for all of these different Hardware platforms. The next table shows the most known processors families covered by every OS.

TABLE V
OPERATING SYSTEMS MCUS ARCHITECTURE SUPPORT [8].

OS	Supported processors families or vendors
FreeRTOS	AVR, MSP430, ARM, x86, 8052, Renesasc
Mbed OS	ARM Cortex-M
Contiki OS	AVR, MSP430, ARM7, ARM Cortex-M, PIC32, x86, 6502
Tiny OS	AVR, MSP430, px27ax
RIOT OS	AVR, MSP430, ARM7, ARM Cortex-M, x86 , TI MSP430 devices

As shown in table 6, every operating system supports large variety of hardware platforms nevertheless, we can note a real limit of the Mbed OS so it is limited only on ARM Cortex-M variants and especially on the 32bits processors, unlike other OSs.

G. Network Connectivity & Protocol Support

IoT networking is of high importance for an IoT OS since Internet connectivity is its key aspect. However, traditional IP-stacks are too power consuming or too complex for small highly constrained devices. Hence, an IoT OS must support light-weight and reliable network stacks. The next table shows exhaustively the support of the selected protocols by studied OSs.

TABLE VI
OPERATING SYSTEMS NETWORK PROTOCOLS

Protocols	802 · 15 · 4	B L E	Z i g B e e	L P W A N	6 L o w p a n	R P L	C o a p	M Q T T
FreeRTOS	x		x		x	x	x	x
Mbed	x	x	x	x	x	x	x	x
Contiki	x	x			x	x	x	x
Tiny	x		x		x	x	x	x
RIOT	x	x	x	x	x	x	x	x

Every one of the studied OSs supports a significant number of protocols. According to the selected ones in our comparison, RIOT and Mbed operating systems seems to be the best choice.

H. Energy efficiency :

About power management, the five OSs are all well energy efficient and we can't specify exactly who is the most one unless if we run a benchmark program on every OS using the same hardware platform. All OSs treat the power consumption similarly and have their own library. Everyone tries to put the processor on sleep mode as longer as possible and wake up the transmission peripheral at least as possible. The next paragraphs show how every OS manage its power.

• FreeRTOS

FreeRTOS, uses the Idle task hook to place the microcontroller into a low power state to reduce the consumed power. This simple method limits the power saving due to the necessity to exit periodically and then re-enter the low power state to

process tick interrupts. When there are no application tasks to execute, the FreeRTOS tickless idle mode stops the periodic tick interrupt during idle periods to allows the microcontroller to remains in a deep power saving state until either an interrupt occurs, or it is time for the RTOS kernel to transit a task into the Ready state [11].

FreeRTOS built in tickless idle implementation uses the WFI (Wait For Interrupt) instruction to suspend execution after stopping the tick interrupt. It provides some macros to allow application using specific power saving code near the WFI instruction. Some of these macros are performed immediately before the Wait For Interrupt instruction to turn peripheral clocks off and activate any microcontroller specific low power functionality.

• Mbed OS

For the Mbed OS, Silicon Labs and ARM® collaborated to deliver the mbed power management APIs, unlocking the ARM Cortex®-M power saving capabilities for low power applications within the mbed community. The APIs achieve even greater power savings on the Silicon Labs EFM32 Gecko MCUs. Basing on the EFM32 architecture nuances, the APIs automatically determine and enable optimal sleep modes based on peripherals in use. The APIs also saves energy by enabling EFM32 I/O operations to be executed even when the EFM32 core is in sleep mode or doing other processing tasks. The next table shows the average current consumption with and without Low Power APIs in an example that combine automatic sleep mode selection and background I/O operations.

TABLE VII
AVERAGE CURRENT CONSUMPTION WITH/WITHOUT LOW POWER APIs [19].

Without Low Power APIs	With Low Power APIs
1.03 mA	0.100 mA

In addition to Sleep mode, there is also a DeepSleep mode, a PowerDown mode, and a DeepPowerDown mode. These modes save some additional power, nevertheless, they also take longer to power up again and require a somewhat more complex setup using WatchDog (Deep Sleep only), real time clock (RTC), or external interrupts to wake up.

Mbed provides also the possibility to vary the clock rate from 48MHz to 128MHz with some complexity since some I/O device clock rates will also need to be changed. At a clock rate of 48Mhz performance is reduced by 50%, but the power level is only reduced to 86% of that with the default 96Mhz clock [24].

• Contiki OS

Contiki scheduler exposes the size of the event queue so that applications can put the microcontroller or peripheral hardware in sleep modes. Therefore power saving process could put the CPU in sleep mode when the event queue is empty.

The default mechanism for attaining low-power operation of the radio is called ContikiMAC. With this MAC protocol, nodes can be running in low-power mode and still be able to

receive and relay radio messages. It uses an efficient wake-up mechanism to attain a high power efficiency: with a wake-up frequency of 8 Hz, the idle radio duty cycle is only 0.6% [5]. The ContikiMAC principle of operation is shown in the next figure.

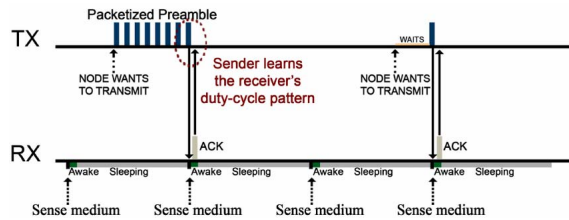


Fig. 4. ContikiMAC [23].

Periodically, nodes wake up to check the radio channel for a reception from a neighbor. If a radio signal is sensed, the node keeps the radio on to listen for the packet. When receiving a data frame, the receiver sends an acknowledgment frame. In the other side, the sender repeatedly sends the data frame until it receives an acknowledgment, or until the packet was sent for an entire wake-up period without an acknowledgment being received [4].

• Tiny OS

To keep the microcontroller in a low power sleep state as long as possible, TinyOS copes with this by using the split-phase and event-driven execution model. As long as there are no tasks in the task queue the scheduler puts the CPU in sleep mode. So in combination with the split-phase operation, the CPU does not waste energy while waiting for other hardware components.

About the peripherals use, TinyOS has a programming convention that allows subsystems to be put in a low-power idle state. Components have an interface, which exposes commands to tell the component to try to minimize its power consumption. TinyOS almost try automatically to put systems and hardware into the lowest power state possible. For peripherals, this typically works through power locks. So for example, when an application writes or reads from a flash chip, TinyOS will automatically power it on, perform the operation, then power it down when done [7].

• RIOT OS

Riot OS scheduler works without periodic events and introduces a tickless scheduler able to function on constrained devices in order to achieve a maximum energy-efficiency. Most schedulers use timer ticks to wake up periodically and check if something needs to be done. However, if the processor is in the idle state, it has to wake up from its power-saving sleep state every timer tick, even when there is nothing to do which is not desirable for energy constrained systems. Moreover, with most microcontrollers, only external interrupt sources can wake up the processor from deep-sleep mode. Hence, using a timer tick prevents using deep-sleep mode. By avoiding timer-tick dependency with its tickless scheduler, RIOT significantly decreases the energy consumption of the system by increasing

the time spent in deep-sleep mode. Also RIOT have several ways at the microcontroller level to reduce power usage. Some methods for reducing power consumption are slow down the clocks, especially the main clock for the core of the device, reduce the supply voltage to the core of the device, stop the main clock, turn off power to the core and turn off power to the clock generator(s) [15].

Regarding peripherals, the system turns off clocks to unused peripherals and the bus controllers and it can turn off power to unused peripherals when necessary. RIOT provide methods that can put the memory into low-power or not instantly accessible mode. These methods can also turn off power to unused sections of memory or to the entire device except an external 'wake-up' pin.

IV. CONCLUSION

Considered as the third wave in the development of the Internet, IoT is rapidly challenging. Neither traditional operating systems currently running on Internet hosts, nor typical OS for sensor networks are capable to fulfill all at once the diverse requirements of such a wide range of devices.

In this survey, different OSs have been presented and selected according to their newness and usage in the IoT domain last three years. The selected OSs are Tiny OS and Contiki OS as most known and used event driven model and FreeRTOS, Mbed OS and RIOT OS as RTOS OSs.

Many open challenges can motivate an IoT OS. Among these challenges the light-weight kernel architecture that should reduce OS services and have a small footprint with an efficient task execution, the Real-Time compatibility and the minimising of the energy consumption by using different power modes. We presented seven major concerns for an IoT OS which are OS architecture, scheduler and realtime capabilities, programming model, memory footprint, network connectivity, hardware support and energy efficiency. After that, we compared choosed OSs according to every concern.

This work differs from other work in that the choice of OSs selection is based on the lasts three years surveys did with Elcipse IoT developers team [12], [13], [14].

While the question 'Which is the most used IoT OS's ?' is clear, the answer is rather vague for a few good reasons. First IoT is a new field of specialization although the idea is old. Second, IoT OS's are new and limited because the field is new and third IoT specialists are few and most developers, researchers, and even companies are in the learning phase. Leading with these three reasons, it is difficult to find a correct answer for the question. While the answer is difficult, it could be misleading to assume that :

- Linux is the most used kernel, Linux-based OS's are the dominant IoT OS's and minimalistic OS's will dominate the embedded world of Internet of Things.
- There are mainly three design categories of OSs that have the potential to become the most suitable for IoT. For multithreaded OSs, RIOT is identified as the most promising one, while for Event-driven OSs, Contiki is

proposed as the best choice and FreeRTOS is selected as the most real-time IoT OS [8][18].

Although the optimal operating system expected to the IoT is not yet born, it must take into account that many other challenges either studied in this article must be considered as security in operating system level and personalization of the OS with some specific OS services for Internet of Things applications.

REFERENCES

- [1] T. B. Chandra, P. Verma, and A. K. Dwivedi, "Operating Systems for Internet of Things," Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies - ICTCS 16, 2016.
- [2] "IoT Operating Systems," Arrow.com, 31-Sep-2016. [Online]. Available: <https://www.arrow.com/en/research-and-events/articles/iot-operating-systems>. [Accessed: 13-Apr-2017].
- [3] M. O. Farooq and T. Kunz, "Operating Systems for Wireless Sensor Networks: A Survey," Sensors, vol. 11, no. 12, pp. 5900–5930, 2011.
- [4] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A Low-Power CoAP for Contiki," 2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems, 2011.
- [5] A. Dunkels, L. Mottola, N. Tsiftes, F. Österlind, J. Eriksson, and N. Finne, "The Announcement Layer: Beacon Coordination for the SensorNet Stack," Lecture Notes in Computer Science Wireless Sensor Networks, pp. 211–226, 2011.
- [6] P. Levis and D. Gay, "Device drivers and the hardware abstraction architecture (HAA)," TinyOS Programming, pp. 206–220.
- [7] "FAQ," FAQ - TinyOS Wiki. [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/FAQ#What_hardware_does_TinyOS_support.3F. [Accessed: 03-Aug-2016].
- [8] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," IEEE Internet of Things Journal, vol. 3, no. 5, pp. 720–734, 2016.
- [9] P. Gaur and M. P. Tahiliani, "Operating Systems for IoT Devices: A Critical Survey," 2015 IEEE Region 10 Symposium, 2015.
- [10] G. Hwaiyu, "Internet Of Things And Data Analytics In The Cloud With Innovation And Sustainability," Internet of Things and Data Analytics Handbook, pp. 1–28, 2016.
- [11] "Low Power Support Tickless Idle Mode," Tickless Low power features in FreeRTOS. [Online]. Available: <http://www.freertos.org/low-power-tickless-rtos.html>. [Accessed: 24-Apr-2017].
- [12] I. Skerrett, "IoT Developer Survey 2015," LinkedIn SlideShare, 07-Apr-2015. [Online]. Available: <https://www.slideshare.net/IanSkerrett/iot-developer-survey-2015>. [Accessed: 21-Feb-2017].
- [13] I. Skerrett, "IoT Developer Survey 2016," LinkedIn SlideShare, 14-Apr-2016. [Online]. Available: <https://www.slideshare.net/IanSkerrett/iot-developer-survey-2016>. [Accessed: 21-Feb-2017].
- [14] I. Skerrett, "IoT Developer Survey 2017," LinkedIn SlideShare, 18-Apr-2017. [Online]. Available: <https://www.slideshare.net/IanSkerrett/iot-developer-survey-2017>. [Accessed: 21-Apr-2017].
- [15] E. Baccelli, "Task Forces," GitHub, 06-Feb-2017. [Online]. Available: <https://github.com/RIOT-OS/RIOT/wiki/Task-Forces>. [Accessed: 11-Apr-2017].
- [16] N. Windpassinger, "Internet of Things - the complete online guide to the IoT," i-SCOOP. [Online]. Available: <https://www.i-scoop.eu/internet-of-things-guide/>. [Accessed: 02-Apr-2017].
- [17] A. V. Dastjerdi and R. Buyya, Internet of things: principles and paradigms. Cambridge, MA: Morgan Kaufmann, 2016.
- [18] A. Elvstam and D. Nordahl, "Operating systems for resource constraint Internet of Things devices: An evaluation," MUEP, 2016. [Online]. Available: <https://dspace.mah.se/handle/2043/20810>. [Accessed: 2-Apr-2017].
- [19] S. Coorman, "Reducing Power Consumption with the ARM® mbed™ Low Power APIs and EFM32™ MCUs," Arm Community, 12-Mar-2015. [Online]. Available: <https://community.arm.com/iot/b/blog/posts/reducing-power-consumption-with-the-arm-mbed-low-power-apis-and-efm32-mcus>. [Accessed: 10-Jan-2017].
- [20] J. Bregell, "Hardware and software platform for Internet of Things," Chalmers studentarbeten, 30-Sep-2015. [Online]. Available: <http://studentarbeten.chalmers.se/publication/223403-hardware-and-software-platform-for-internet-of-things>. [Accessed: 03-Feb-2017].
- [21] R. Murdock, "Traditional IoT & Embedded Operating System Vendors Face Shifting Mark," PRWeb, 19-Nov-2015. [Online]. Available: <http://www.prweb.com/releases/2015/11/prweb13091582.htm>. [Accessed: 04-Apr-2017].
- [22] C. Bormann, A. Keranen, and M. Ersue, "Terminology for Constrained-Node Networks," IETF Tools, May-2014. [Online]. Available: <https://tools.ietf.org/html/rfc7228>. [Accessed: 01-May-2017].
- [23] R. Martinez-Sandoval, A.-J. Garcia-Sanchez, F. Garcia-Sanchez, J. Garcia-Haro, and D. Flynn, "A Comprehensive WSN-Based Approach to Efficiently Manage a Smart Grid," Sensors, vol. 14, no. 10, pp. 18748–18783, Oct. 2014.
- [24] "Mbed Power Management," Power Management - Cookbook | Mbed. [Online]. Available: <https://developer.mbed.org/cookbook/Power-Management>. [Accessed: 12-Nov-2016].