Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking

Miguel Silva, David Cerdeira, Sandro Pinto, and Tiago Gomes

***Abstract*—In the era of the Internet of Things (IoT), billions of wirelessly connected embedded devices rapidly became part of our daily lives. As a key tool for each Internet-enabled object, embedded operating systems (OSes) provide a set of services and abstractions which eases the development and speedups the deployment of IoT solutions at scale. This article starts by discussing the requirements of an IoT-enabled OS, taking into consideration the major concerns when developing solutions at the network edge, followed by a deep comparative analysis and benchmarking on Contiki-NG, RIOT, and Zephyr. Such OSes were considered as the best representative of their class considering the main key-points that best define an OS for resource-constrained IoT devices: low-power consumption, real-time capabilities, security awareness, interoperability, and connectivity. While evaluating each OS under different network conditions, the gathered results revealed distinct behaviors for each OS feature, mainly due to differences in kernel and network stack implementations.**

***Index Terms*—Benchmarking, embedded systems, Internet of Things (IoT), low-end devices, operating systems (OSes).**

I. Introduction

THE INTERNET of Things (IoT) is revolutionizing the Internet for the future by connecting billions of smart devices over a massive and collaborative network infrastructure. The most recent statistics estimate that by the year 2020 there will be over 50 billion Internet-enabled devices, motivating an increasing focus from both industry and academia on such multitrillion dollar market [1], [2]. The key-concept of the IoT is to enable people and things to be connected anytime, at anyplace, with anything, and anyone [3], which leads to a countless number of use cases, constraints, and requirements in order to satisfy all possible needs. The constrained nature of the IoT edge network often implies the deployment of battery-powered and resource-limited devices. Hence, hardware and software solutions must support low-power operations while providing the necessary system's performance [4]. Additionally, it is required from any IoT device to connect with others and the Internet, which is mainly done through a wireless interface, e.g., RFID, IEEE 802.15.4, Wi-Fi, bluetooth low-energy (BLE) [5], etc.

The massive heterogeneity of the existing embedded devices, combined with the connectivity requirement, calls for proper software solutions to efficiently manage and control the available hardware resources. This requirement has driven the development of a multitude of operating systems (OSes), which tend to be specially tailored to cover a specific application (e.g., automotive, wearables, healthcare, etc.) and development needs. Its design choices (e.g., the kernel architecture and the scheduling policy) have a direct and significant impact on the overall system's behavior, both in terms of performance, determinism, and power consumption. Among a broad list of open-source OSes, some of them have been widely deployed in several low-end IoT devices: Contiki, RIOT, Zephyr, TinyOS, Amazon FreeRTOS, and many more [6]–[10].

Motivated by this broad collection of OSes and since there is no "one size fits all" solution, the purpose of this article is to analyze and benchmark some of the most prominent open-source OSes for the IoT, taking into consideration the most important aspects when deploying devices at the very edge. By understanding their main differences and characteristics, they can be carefully selected and deployed according to the application needs, i.e., when real-time is required or a low-power solution is preferred. For this purpose, this article contributes to the state-of-the-art with: 1) an analysis on some of the most prominent open-source OSes for the IoT: Contiki-NG, RIOT, and Zephyr and 2) a complete benchmarking on the most important features when designing an IoT embedded device, such as performance, power consumption, real-time capabilities, and memory footprint. In our experiments, we had evaluated the effect of the network stack over the overall OS metrics, and we had concluded that it is a significant source of overhead and latency. To the best of our knowledge, this article goes behind the state-of-the-art [9]–[12], where mostly a theoretical approach and a literature review is provided.

II. IoT Ecosystem

The Internet engineering task force (IETF) standardized the classification of constrained devices into different sets of classes [13]. This classification is done according to the required memory footprint for both code and data.

1) *Class 0:* These devices have the smallest resources (less than 10 KB of RAM and less than 100 KB of Flash), e.g., tiny sensing motes.

Manuscript received July 22, 2019; revised August 26, 2019; accepted August 28, 2019. Date of publication September 4, 2019; date of current version December 11, 2019. This work has been supported by national funds through FCT - Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2019. *(Corresponding author: Miguel Silva.)*

The authors are with the Centro ALGORITMI, University of Minho, 4800-058 Guimaraes, Portugal (e-mail: miguel.silva@dei.uminho.pt; david.cerdeira@dei.uminho.pt; sandro.pinto@dei.uminho.pt; mr.gomes@dei.uminho.pt).

Digital Object Identifier 10.1109/JIOT.2019.2939008

2) *Class 1:* These devices have medium-level resources (around 10 KB of RAM and 100 KB of Flash), e.g., motes with routing capabilities and security features.
3) *Class 2:* Devices from this category have more resources than the previous ones, e.g., small gateways, but are still limited when compared to middle- and high-end devices.

Such classes are the result from the requirements imposed by the IoT ecosystem, where each low-end device must fulfil a set of requirements: ability to explore low-power modes with reduced duty-cycle operation, manage resource-constrained hardware, full connectivity and interoperability support, real-time capabilities, and security awareness (hardware, data handling, and secure data transmissions).

A. Low-Power Consumption

IoT low-end devices often resort to low-power hardware, which leads to a reduced energy consumption when all the on-board components are lowered to the bare minimal, and when processor's low-power modes (e.g., deep sleep) are explored. While both features are important, the latter is a common requirement for almost any IoT solution, as these usually only need to perform periodic tasks with decreased duty-cycle operation. The software running on such platforms must also be optimized and aware of the available power-saving features.

B. Resource-Constrained Embedded Devices

There is an ongoing trend in the industry to squeeze, as much as possible, the so-called size, weight, power, and cost (SWaP-C) budget. IoT devices are a great example of such paradigm. These devices are designed with minimal margins and are usually limited to accommodate only the target application, as oversized and unused components also contribute to an increased power consumption and overall cost. Another important aspect that greatly limits the resources available on these devices is their placement, frequently requiring footprints as small as possible. Hence, the number of components has to be kept to the bare minimum to avoid any waste of space. Taking into consideration the previous aspects, the size and type of available memory in a system can be also an indicator of the class of an IoT device, as these components are sometimes the most power-consuming elements of the final solution. Therefore, it is important to determine the size of the available memory, both for code and data, while also minimizing the amount of data that needs to be saved between sleep/wake-up cycles, as this may require memory to be permanently powered.

C. Connectivity and Interoperability

At the core of the IoT concept is the ability to connect everything. Hence, each device must include the necessary hardware and software for connectivity. Regarding the hardware, power consumption is a crucial aspect that led to the adoption of standard communication protocols for lossy communication links. Among them, the most common are the IEEE 802.15.4, BLE, and Wi-Fi [14]. The physical (PHY) and medium access control (MAC) layers, as well as the handling protocols, play a major role in the overall power consumption. Regarding the software, an embedded network stack must be provided in order to support such wireless interfaces, while seamlessly connecting to the Internet.

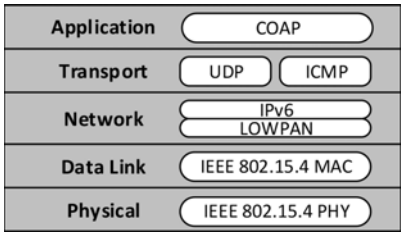

Fig. 1. Standard IoT network stack.

Such stack (depicted in Fig. 1), was derived from the traditional seven-layer OSI model stack and it is adopted by the majority of OSes for IoT low-end devices. It is divided into five independent layers, which allows full interoperability and eases its portability among heterogeneous devices. For each layer, several standards and protocols, e.g., UDP/TCP, IPv6, CoAP, and 6LoWPAN, were carefully selected to fulfil the tight constraints of the target devices. Regarding the network layer, by using a 128-bit addressing scheme, IPv6 is the key to connect billions of devices to the Internet. In order to allow its support over different MAC and PHY standards, it is necessary an adaptation layer protocol, e.g., 6LoWPAN for the IEEE 802.15.4. Due to the portability and interoperability requirements, the network stack is usually provided by the OS. Several open-source network stacks for low-end devices, like OpenWSN [15] or lwIP [16], provide different characteristics making them suitable for different sets of applications. Additionally, OSes usually offer the network stack tightly intertwined with the kernel, where both kernel and network stack are developed and optimized together.

D. Real-Time

Several IoT applications, e.g., healthcare and automotive systems [17], [18], require strict real-time guarantees. While the microcontroller and the remaining hardware play a role on system's determinism and predictability, the system software is also a major player. Several important aspects, such as implementation, optimization, architecture, programming model, scheduling algorithm, and whether or not it supports real-time events (e.g., interrupts), defines how the OS is able to attend critical tasks with high predictability and determinism [19].

OS architectures can be classified in either monolithic or microkernel. The first approach assumes that all components of the system are developed in tandem, leading to a simpler and more efficient design. The microkernel approach is usually designed with minimum kernel functionalities, implementing several services in userland. Microkernel-like architectures strive for simplicity, modularity, and minimality, usually at the cost of performance. The scheduler is also directly related to the OS architecture, and therefore, represents another key aspect when designing the OS, as it is the component that manages how and when tasks are executed by the processor.

The scheduler should aim to optimize the system's throughput, energy efficiency, and resources utilization, while ensuring determinism on each thread execution. While there are several algorithms, most schedulers can be classified as either cooperative or preemptive. The former requires each thread to yield its execution to bring other threads in context, while the latter relies on the principle that at some scheduling point the running thread is interrupted and halted by the scheduler to start the execution of the next thread. Achieving real-time on a resource-constrained device is a goal that must be also met by the system software, and thus, choosing the right OS and configuration for the application is a key aspect on whether the entire solution will be able to meet the deadlines.

E. Security and Safety

The ongoing cat-and-mouse game of increasing hacks and software patches has raised significant concerns on the need of securing IoT devices. Moreover, the ever-growing volume of sensible data that is being processed by IoT nodes following their proliferation; examples range from personal or health-related data acquired from sensors. Therefore, in order to ensure privacy and security of critical data through communications on the IoT ecosystem, security mechanisms must ensure confidentiality, integrity, authenticity, and nonrepudiation of the whole information's life cycle. This can be achieved through the protocols implemented on the network stack or other external mechanisms [20].

Safety is another important aspect of IoT devices as most of them tend to implement an ever-growing number of mixed-critically features. Furthermore, safety and security stand hand-by-hand—there is no safety without security and vice versa. Any kind of malfunction or attack that seizes control of on-board actuators can directly influence the normal behavior of the system or even cause hazards to users. For example, a very common attack to any device connected to a network or to the Internet is called a denial of service (DoS) attack. Despite it can be achieved through several ways, the main idea is to deprive the system of its resources (e.g., processing time), preventing it from performing as expected. This proves to be a problem, for example, in automotive applications, when critical tasks need to be executed under bounded and deterministic deadlines. While the DoS attack is only a generic example, there are several other types of attacks that can be performed by simply adapting the same concept from classical attacks on network-based systems to IoT devices.

Another major concern around security relies on the hardware itself. For instance, Arm TrustZone is a system-on-chip (SoC) and CPU security solution, which highly increases the system security and reduces the attack surface by providing system-wide hardware isolation for trusted software [21]. TrustZone was recently extended to Cortex-M-based systems, enabling robust levels of protection at all cost points [22]. Embedded software developers can now enhance their productivity by developing TrustZone-based systems. Such tendency is already being adopted by prominent embedded OSes such as Zephyr [8]. Moreover, hardware solutions with embedded system "Root-of-Trust" provide enhanced security features in low-end systems traditionally deprived of security.

TABLE I
OSES COMPARISON

OS	*Architecture*	*Programming Model*	*Scheduler*	*Supported Architectures*	*Network Support*
Contiki-NG	Monolithic	Event-driven	Cooperative	AVR, MSP430, ARM Cortex-M	uIP and RIME
RIOT	Microkernel	Multithreading	Tickless Preemptive	AVR, MSP430, ARM Cortex-M, x86	gnrc
Zephyr	Nanokernel+ Microkernel	Multithreading	Tick-based Preemptive	ARC, AVR, ARM Cortex-M, x86, RISC-V	Native implementation

III. OSES ANALYSIS

In this article, we evaluate three of the most prominent OSes for IoT, which apart from being completely open source, are currently enjoying widespread applicability and continuous support in the context of low-end IoT applications: Contiki-NG, RIOT, and Zephyr. They were selected regarding the main characteristics discussed in Section II, their programming model (event-driven and multithreading), kernel architecture (monolithic and microkernel), scheduling policy (cooperative and preemptive), and native support of a network stack for low-end devices (at least for class 0 and/or 1). These characteristics are summarized in Table I.

A. Contiki-NG

Contiki was originally proposed by Dunkels *et al.* as an OS for wireless sensor networks (WSNs) targeting resource-constrained wireless node, and only later adapted for more powerful devices [6]. The OS follows an event-driven programming model based on a cooperative scheduling approach using protothreads, a lightweight mechanism for pseudo-threading, from which the programmer is abstracted. On its current version, these are seen as statically defined *Processes* and do not support priorities, since this OS implements a cooperative scheduler. From a developer perspective, aside from its declaration, each *Process* must be implemented as a function that at some point yields the execution time to the next *Process*. Failing to yield, e.g., stopping in an infinite loop, would cause the whole system to halt. When deployed in low-end devices, it is common to have a *Process* sleeping or waiting for events that trigger their execution. While such a feature is supported by the OS, fast response times to event occurrence may not be achieved due to its scheduling policy.

To support the OS main requirements, Contiki includes features that aim at communication-based low-power systems, for instance, sleep mode managing, and support for several network stacks, e.g., uIP. These stacks offer support for standard and well-known protocols, such as IPv6, RPL, 6LoWPAN, and CoAP, while supporting several PHY technologies, such as IEEE 802.15.4, Wi-Fi or BLE. Contiki is mainly focused on dependable (secure and reliable) low-power communication and standard protocols for modern IoT platforms based on 32-bit microcontrollers, mostly supporting Arm architectures. On its latest version, Contiki-NG, the overall code structure was revised and optimized with new

configurations and a major cleanup of the code base (obsolete protocols and standards were removed), minimizing the final binary size.

B. RIOT

Initially designed with the IoT ecosystem as the main target, its main characteristics comprise real-time capabilities and low-power efficiency [7]. This OS employs a multithread programming model and follows a microkernel-like architecture, which steams for a simpler and shorter development cycle due to its modular nature. RIOT implements a tickless preemptive scheduler, which means that there are no periodic events as scheduling points. This method tries to optimize the time spent in low-power modes, i.e., sleep or deep sleep modes, by forcing the system into these states whenever there are no threads to be executed, i.e., when the idle thread is active. The OS further guarantees the execution of kernel tasks and interprocess communication, in order to fulfill real-time requirements. Additionally, this OS includes its own implementation of a full IoT stack named *gnrc*, which adds the support to new protocols, such as the 6TiSCH, the IPv6 over the TSCH mode of IEEE 802.15.4 standard.

C. Zephyr

Zephyr is an ongoing project from the Linux Foundation designed for resource-constrained systems [8]. Similarly to RIOT, it follows a multithreading programming model with a microkernel-based architecture. Zephyr uses a scheduler based on a tick system to schedule each thread in a periodic fashion. There are two major types of threads: 1) fiber, which is a lightweight non-preemptible thread, usually with small execution times and designed to be used in critical contexts and 2) task, that implements the common concept of a task that can be preempted. While both can be prioritized among themselves, fibers are inherently prioritized over tasks, and no task will be scheduled when there are fibers waiting to execute.

Zephyr implements both a nanokernel and microkernel architectures. The former is a high-performance, multithreaded execution environment with a minimalist set of kernel features conceived for highly constrained devices, while the latter complements this with a set of richer and more complex features, such as network stack and device drivers for more complex devices. Regarding networking features, Zephyr integrates its own network stack implementation, including support for low-power devices that require IEEE 802.15.4 or BLE radio interfaces to communicate. Therefore, the 6LoWPAN adaption layer is also supported in order to provide IPv6 connectivity, leading to a highly modular and flexible implementation of the network stack. Finally, another important characteristic of Zephyr is that it already supports RISC-V, an open source processor architecture that recently has been given a lot of attention from both academia and industry [23].

D. Conclusion

From Table I it is possible to observe that Contiki-NG is representative of a different kernel implementation paradigm, while both RIOT and Zephyr are similar in most aspects.

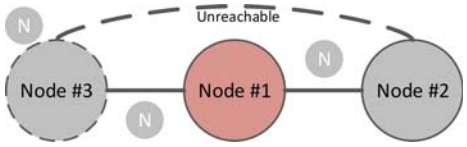

Fig. 2. Network topology used in the experiments.

Despite both Zephyr and RIOT following a multithreading programming model, and to some extent, sharing the same architecture principles, they follow a distinct scheduling policy: RIOT is deprived of the notion of time and Zephyr uses periodic events to iterate over the waiting threads. Aside from the support to the main platforms on the market from all the OSes, Zephyr is the one that already supports the next-generation hardware architectures, i.e., TrustZone-M and RISC-V. Finally, and regarding the supported network stack, Contiki-NG and Zephyr provide their own monolithic implementations, while RIOT uses an external stack from the OpenWSN project.

IV. Evaluation

In this article, we benchmark the three aforementioned IoT OSes by running the same set of benchmarks under the same hardware and network conditions. The performed experiments aimed at assessing memory footprint, performance, real-time, and power consumption. Security-related aspects are, however, out of the scope of this article.

A. Experimental Setup

All experiments were performed on an STM32L476G-DISCOVERY, connected to a TI CC2520 radio through a serial peripheral interface (SPI). This development board features an Arm Cortex-M4 processor running at a clock speed of 80 MHz. The Arm Cortex-M4 is one of the most widespread microprocessor architectures on the embedded systems market and, therefore, it is widely supported by the developers community. The radio used in this setup supports the IEEE 802.15.4 standard, which is one of the most widely used protocols in IoT applications, such as home automation and industrial monitoring systems. In such systems, it is common to have several nodes connected to each other and the Internet. Due to the intrinsic difficulty in recreating a big-scale network, a smaller topology was deployed (depicted by Fig. 2) composed by three nodes, which evaluates a device under test on different network conditions. Despite simple, it still recreates the desired scenario where a node in operation receives random connections from surrounding nodes, and the incoming data is intended to the node itself or another node in the network. Node 1 represents the device under test. This device is able to establish a connection with both other nodes. However, Nodes 2 and 3 cannot directly communicate with each other, thus, they rely on the node being tested to forward their message to its destination. Given that the nodes are role-independent, the communication to and from the device under test is established through a UDP connection without a specific protocol at the application layer. As a result, the entire network is being used with the following configuration: UDP,

IPv6/RPL, 6LoWPAN, and IEEE 802.15.4 (MAC and PHY). Whenever Node 1 receives a message from the network, it must either accept, reject, or forward it.

Experiments were performed by sending a fixed-size UDP message of 24 bytes (which corresponds to 58 bytes of the MAC data frame) through a socket on local/remote ports 8080/8081. The number of frames being sent to the device under test was kept at an average rate of 230 packets per second, reaching a point where the network is almost overloaded but the receiving node is still able to attend each packet that is being received. Finally, all the evaluated OSes already provided support for the target board and only slight modifications were required to run them. Furthermore, the integration of the radio driver with the network stack, as well as with the benchmark suit to each OS, just required a small porting effort. Also, each OS was kept with its default or suggested settings for a similar application to the one under evaluation. Therefore, the conducted benchmarks evaluate each OS as an off-the-shelf solution, representing the conditions the developers will expect when using each OS without modifications.

On all performed experiments, the network stack is always initialized, whether network traffic exists or not. Depending on the stack implementation and its integration with the kernel, there may be threads related to the network stack initiated before each benchmark starts executing. We strive for this approach as we have the intention of understanding the influence of the stack, even if it is on an idle state, while benchmarking for each OS. Additionally, based on Fig. 2, four different states of the network were used.

1) *Idle:* In this state, there is no traffic in the network and only essential tasks are being executed. It is expected that the system does not perform any network-related tasks.
2) *Accept:* All the network traffic is intended to Node 1. Data must be accepted and processed accordingly, requiring the intervention of all the network stack layers.
3) *Reject:* Contrarily to the previous test case scenario, all the traffic is intended to be rejected, e.g., data is intended to another node and must not be forwarded. In this case, the packet is rejected as soon as possible (it can be done either at the MAC or upper layers).
4) *Forward:* This last test assumes that the destination of the packet is unreachable by the original sender but belongs to the same network and the selected routing scheme is aware of the existence of this neighbor. In such case, the packet is forwarded before reaching the application layer of the stack.

For measuring the performance of each OS, the Thread-Metric Benchmark Suite was used. This suite aims at evaluating the most common RTOS services and interrupt processing mechanisms, encompassing a total of eight benchmarks.

1) *Basic Processing (T1):* A single thread performs mathematical operations in a loop and counts the number of times the operation was done. It serves as the baseline for the remaining tests.
2) *Cooperative Context Switching (T2):* Five threads execute concurrently, each of them counting the number of times they run. The result is the sum of all counters from each thread.
3) *Preemptive Context Switching (T3):* It consists of five threads with different priorities, each resuming the next thread with a higher priority before suspending and counting the number of times they run. The result is the sum of the count values of each thread.
4) *Interrupt Processing (T4):* A single thread is executed, interrupted, and resumed afterward. The result is the sum of the number of times the interrupt was attended and the number of times the thread was executed.
5) *Interrupt Processing With Preemption (T5):* It consists of two threads with different priorities, where one of which triggers an interrupt that is responsible for resuming the other suspended thread. The value obtained is the sum of the number of times each thread was executed and the interrupt was attended.
6) *Message Passing (T6):* A single thread sends a message to itself through a queue, and upon receiving, a counter is incremented.
7) *Semaphore Processing (T7):* A single thread gets and releases a semaphore in a loop cycle, counting the number of times this process is executed.
8) *Memory Allocation and Deallocation (T8):* A thread consecutively allocates and deallocates memory blocks of 128 bytes, counting the number of times it is done.

Each benchmark, after running for a certain number of iterations based on a 30-s cycle execution time, outputs a score value, representing the OS impact on the running application—higher scores express a smaller impact, i.e., higher performance. For the purpose of these experiments, it was decided that all traffic should be either accepted, rejected, or forwarded. It was not taken into account test case scenarios that encompass a mix of these network states, as they would lead to application-biased conclusions.

In order to evaluate the power consumption, the development board was powered with a precision power supply of 3.3 V, while an ammeter was used to measure the current that was only consumed by the SoC. This way, the power consumption of all the external peripherals present on the board, as well as the radio IC, were not considered.

B. Performance

Table II presents the performance results gathered from all experiments. Each value corresponds to the average of 1000 collected samples. The three major rows correspond to each OS for the different network operations. Given that the hardware is the same in all tests, it is fair to put all OSes into perspective and establish some comparisons among them. Notwithstanding, we start by breaking down the results of the various experiments for each OS.

1) Contiki-NG: In Contiki-NG, a *Process* is scheduled following a cooperative policy, while the only form of preemption is used by interrupt handlers in device drivers. For this reason, preemptive-related benchmarks T3 and T5 were slightly modified to be supported by Contiki, which means that the results do not necessarily express a preemptive behavior. Their score resembles more a cooperative result since all processes run at the same priority level, and in order to keep the benchmark realistic, each thread controls the execution of the next one.

TABLE II
THREAD-METRIC BENCHMARK SUITE RESULTS

OS	Net. Status	T1	T2	T3	T4	T5	T6	T7	T8	PD (%)
Contiki	*Idle*	252 568	13 056 369	13 316 018	20 707 686	12 858 805	12 483 662	12 103 265	9 205 842	**(baseline)**
	Accept	236 938	12 212 617	12 448 708	19 355 980	12 022 817	11 664 648	11 308 671	8 592 961	**-6.52%**
	Reject	240 215	12 382 610	12 621 778	19 634 155	12 194 618	11 829 477	11 469 341	8 716 317	**-5.20%**
	Forward	237 706	12 253 965	12 489 632	19 426 463	12 058 879	11 698 686	11 348 671	8 619 847	**-9.36%**
RIOT	*Idle*	260 213	8 955 046	4 178 175	20 689 193	8 780 255	9 677 227	14 370 013	11 650 253	**(baseline)**
	Accept	233 256	8 018 193	3 740 900	19 068 063	7 728 775	8 661 825	12 870 150	10 417 357	**-10.33%**
	Reject	236 160	8 126 312	3 836 133	19 287 248	7 813 942	8 784 321	13 039 798	10 564 781	**-9.03%**
	Forward	234 939	8 084 983	3 770 557	18 889 409	7 882 572	8 843 571	13 198 882	10 512 343	**-12.83%**
Zephyr	*Idle*	195 196	6 089 882	3 351 826	36 079 757	5 432 733	7 291 447	16 545 946	2 662 762	**(baseline)**
	Accept	166 825	5 202 160	2 884 377	30 910 487	4 653 708	6 225 813	14 135 562	2 274 075	**-14.47%**
	Reject	167 476	5 217 418	2 873 679	30 930 780	4 660 294	6 249 194	14 187 074	2 282 132	**-14.26%**
	Forward	147 821	4 622 752	2 511 241	27 424 242	4 024 194	5 581 969	12 618 843	2 052 703	**-24.18%**

Furthermore, with the exception of T4, all experiments have led to similar results regarding the cooperative context switching benchmark (T2). This is mainly due to the cooperative OS nature and the differences observed among the results reflect the influence of the different APIs being used on each benchmark, e.g., semaphores, suspending, or resuming threads, etc. Finally, a noteworthy outlier in the assessed results, occurs in T4, where a single thread is interrupted and its execution is returned after the interrupt service routine (ISR) is finished. This behavior results in a significant boost of performance, as no context switches are performed.

2) RIOT: Due to its microkernel architecture, the values gathered from the experiments reveal that in this OS, the cooperative related benchmarks tend to have better performance when compared to the preemptive ones. This is related to the fact that the executing task yields itself, instead of resuming a different one, as observed in preemptive scheduling. The latter involves more system calls and more effort from the scheduler since the tasks have different priorities, resulting in a considerable degradation of performance. Regarding the interrupt-based benchmarks, the cooperative one reveals the best results among all the others OSes, since the return from the ISR does not involve a full context reschedule, once the task that is resuming its execution is the same task that was previously executing. On the other hand, in T5, a scheduling point is forced after the interrupt since a different task resumes its execution. Likewise the results obtained from the other OSes, from T6 to T8, the obtained scores also reflect the direct APIs influence on the system's performance.

3) Zephyr: Similarly to RIOT, Zephyr follows a microkernel approach, and therefore, the achieved results follow a similar pattern. However, in T4, when the system is going to leave the ISR context, unlike RIOT, Zephyr does not trigger any context-switching operation. This obviously results in a higher performance. Accordingly to Table II, the main performance bottleneck from Zephyr lies on its memory management subsystem. When comparing the results with T6 and T7, it is possible to conclude that the process of allocating and deallocating memory introduces a significant overhead on the system due to the implemented memory management schema.

4) Summary: Performance is a requirement in almost every application, specially when network traffic needs to be handled by the OS. Considering this, Contiki-NG provides the best results. However, the application will only run in a cooperative scheduling policy. Between the other OSes, which implement a preemptive scheduler, RIOT presents better performance.

C. Time Predictability

In order to evaluate the time predictability of each OS, we performed a set of micro-benchmarks which encompassed four typical thread-management APIs: 1) *Thread Create*, which allocates TCB resources and puts the thread in the ready state; 2) *Thread Resume*, which forces a scheduling point to the appointed thread; 3) *Thread Suspend*, which suspends the execution of a specific task; and 4) *Thread Delete*, which reverses what was done in thread creation. For measuring the execution time of each API, we have configured a timer to start counting on the exact instruction before the API call, and to stop counting the instruction after the function return. However, in some cases, a scheduling point is forced by the API, and therefore, the system resumes its execution in a different location. In such cases, the timer stops counting right after the context restoring operation and before jumping to the next thread. Additionally, due to the cooperative nature of Contiki-NG, some of the previous APIs are not provided by the kernel. Their implementation is mostly based on preprocessor macros and polling mechanisms, making the time measurement incoherent with the remaining OSes. For this reason the *Thread Suspend* API was not tested for Contiki-NG.

In the context of these experiments, we have used the preemptive context switching benchmark (T3) from the Thread-Metric Suite to evaluate the effect of changing parameters, such as the number of tasks (from 5 to 20) the priority of tasks (from 1 to 32), and the priority gap between tasks (from 1 to 5, when possible). Finally, all experiments were repeated for different network configurations, i.e., idle (without any traffic) and active (all packets are accepted). The reason to present results for the active network state is because it represents the worst case scenario, i.e., the system has a higher workload. Fig. 3 depicts the achieved results, where the bars represent the number of clock cycles that each specific API takes in its execution time, while the lines represent the minimum and maximum measured values (jitter).

1) Contiki-NG: As aforementioned, for Contiki-NG the *Suspend* API was not evaluated. According to Fig. 3, for the idle network configuration, Contiki-NG presents the worst

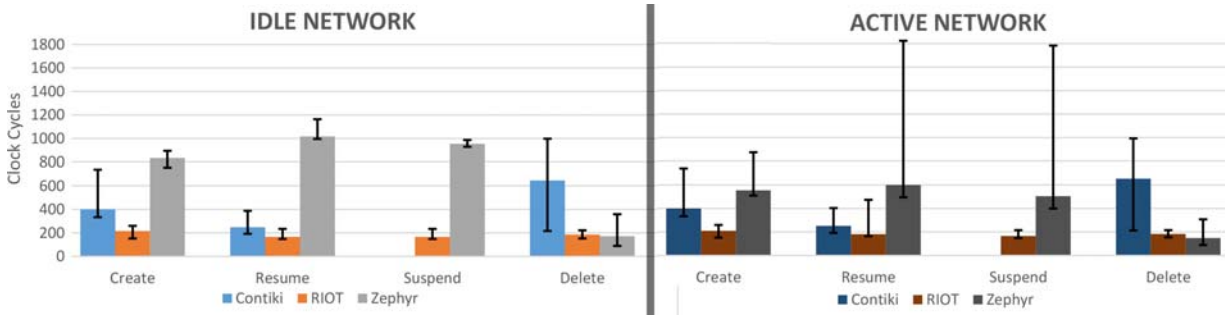

Fig. 3. Time predictability evaluation results.

predictability among the three OSes under evaluation. This is due to the cooperative nature of Contiki-NG, which leads to higher amounts of code that can be preempted by interrupts. Another interesting conclusion is that the predictability of the OS is not affected by the network configuration, as the variation is similar for the active network configuration.

2) RIOT: RIOT presents the best time predictability (less variation) among all experimented OSes, with the exception for one test case scenario: the *Suspend* API in the idle network configuration. In this case, the variation is minimal, when compared with the best results (Zephyr) on the same scenario. It happens because the scheduler implementation on RIOT uses a circular list of threads and Zephyr uses a generic linked list. Another fact worth to highlight is that the variation is kept small and constant among both network configurations for the *Create*, *Suspend*, and *Delete* APIs. For the *Resume* API there is a significant lack of timing predictability when all packets are intended to be accepted.

3) Zephyr: Zephyr presents, on average, the highest amount of variation across all OSes. This happens because every event on Zephyr, e.g., system calls or interrupts, is handled by the kernel in a privileged mode, which implies a processing mode switch every time it happens. Whenever an interrupt needs to be attended, the kernel is invoked to manage the event. However, the system not always requires a context-switch, which is the place that by default stops the timer and measures the API execution time. Such condition leads to an exceedingly high jitter at run-time, as the time is not measured when it is supposed to. While this is not fully intended to be measured, any workaround would require major changes in the internals of the kernel to modify its default behavior.

4) Summary: For use case applications that impose hard real-time deadlines, e.g., industrial IoT, a deterministic and predictable system is mandatory. Therefore, among the evaluated OSes, RIOT have shown the best time predictability, which lead us to conclude that this OS is the best option for applications that require real-time.

D. Memory Footprint

To assess the memory footprint of each OS, we measured the required amount of RAM and Flash memory needed by both the kernel and application. By assessing the memory footprint we aim at classifying each OS accordingly to the IETF device class required by each OS for this application.

TABLE III
MEMORY FOOTPRINT (IN KB)

OS	*RAM*	*Flash*
Contiki-NG	29.8	50.6
RIOT	33.0	59.9
Zephyr	52.9	130.5

Table III summarizes the memory required by each OS, including the generated firmware when all OS services are selected along with the network stack protocols. The obtained results show that on average, the required amount of RAM for Contiki-NG, RIOT, and Zephyr, are, respectively, around 29.8 KB, 33.0 KB, and 52.9 KB. By its turn, the required Flash memory is 50.6 KB, 59.9 KB, and 130.5 KB. Such values are not only related to the OS itself but also with network stack implementation (unique in each OS).

Summary: For the given setup and network stack configuration, Contiki-NG and RIOT can be used in a class 1 device while Zephyr demands for a class 2. This is due to the required amount of RAM needed to accommodate both system and user data. Such requirement is highly related to the network stack features that for the selected configuration, enables a set of protocols needed for the experiments with network.

E. Power Evaluation

Finally, we managed to assess the power consumption, at the SoC level (excluding on-board peripherals and the radio transceiver), that each OS requires for a specific benchmark. Table IV shows the average power consumption by the SoC during a fixed time period, in which the system is switching among five threads, similarly to T3. Additionally, all the tests were performed with the network in both idle and active states. According to the data-sheet, for the same clock frequency and voltage supply, the SoC should consume around 37 mW. Hence, all the results depicted in Table IV are coherent with the results presented in the previous sections. RIOT and Contiki-NG have similar power consumption, with RIOT being nearly 5% better, corroborating the results from the previous evaluations. On the other hand, Zephyr presents a higher power consumption, also emphasizing the results from all other experiments, where the lower performance and increased memory footprint are here reflected. Across all OSes, a slight increase in the power consumption is observed

TABLE IV
POWER CONSUMPTION (IN MW)

OS	Idle Network	Active Network
Contiki-NG	44,652	47,691
RIOT	46,022	46,144
Zephyr	54,483	54,658

when there is traffic on the network. Despite this increment being minimal, 6.8% on Contiki, 2.6% on RIOT, and 0.4% on Zephyr, it is due to the number of times the system is being interrupted to attend network requests and run the related code.

Summary: The power consumption of an IoT device is a major concern, specially when powered by batteries. Despite resorting to sleep modes, the final application scenario often seeks for software that does not incur in additional power wastes. In this regard, Contiki-NG and RIOT provide the best results, when compared with Zephyr.

V. CLOSING DISCUSSION

Across all OSes, it was possible to observe that when the packets need to be forwarded to another node, the performance and determinism degradation reaches its peak (application-specific tasks that could be also performance-consuming, such as sensors reading and heavy-processing algorithms, are not being considered). This effect is caused by the fact that not only the packet needs to be rejected but also a new packet needs to be created, with all necessary changes regarding addressing and protocol data. Another point that is valid to all OSes is that the influence of rejecting packets is always smaller than accepting. When a packet is meant to be rejected, it is mainly discarded in the lower layers, e.g., MAC or IP, which means that instead of being forwarded through the entire stack, fewer software tasks need to be executed, and thus, the system is free to perform other tasks earlier. Contrarily, when a packet is to be accepted, it is the upper layer that checks its validity, inherently requiring more processing cycles. Table V depicts a subjective comparison of the evaluated OSes.

1) Performance Evaluation: Regarding the performance, and considering the results from T1 as the baseline with the network configuration in *idle*, it is possible to compare the three OSes among each other. This is due to the fact that in this case, a single thread is performing simple operations without resorting to any kernel service. Among all OSes, Zephyr provides approximately 25% less performance.

Comparing the data from T6–T8, it is possible to argue that: 1) for the conducted experiments, the message queue system on the microkernel-based OSes perform slightly worse than in Contiki-NG; 2) semaphores on Zephyr are considerably better than the other two, despite this OS having an overall worse performance; and 3) the dynamic memory management system of Zephyr is far worse than the other two, due to the overhead induced by its implementation. The last column of Table II shows the average performance degradation associated with the different network states. Each percentage refers to its own OS baseline illustrating the degradation relative to the system without network traffic. Taking this into consideration, it is possible to understand the influence of the default network stack on each system, and the kernel's ability to attend the stack demands. Zephyr reveals the worst performance when the network is active, due to the increased overhead of its kernel operations, leading to an overall performance worse than other OSes. Contrarily, Contiki-NG has the least degradation under the same conditions, given the fact that its scheduling mechanisms do not involve extensive operations of context-switching. On the other hand, RIOT stands in the middle, balancing its performance with the advantages of a microkernel architecture.

TABLE V
QUALITATIVE COMPARISON OF THE EVALUATED OSES

OS	Network State	Performance	Time Predictability	Memory Footprint	Power Consumption
Contiki-NG	Idle	●●●	●●○	●●●	●●●
	Active	●●●	●●○		●●○
RIOT	Idle	●●●	●●●	●●○	●●●
	Active	●●○	●●●		●●●
Zephyr	Idle	●●○	●●○	●○○	●●○
	Active	●○○	●○○		●●○

2) Real-Time Evaluation: Regarding the determinism, both initialization and clean-up stages of the majority of systems can be neglected since these occur only once. Hence, it is important to focus on the analysis of the kernel functions that are used constantly throughout the system's life cycle. Moreover, the presence of network traffic greatly affects the determinism of all OSes, given that the system is continually being interrupted when new packets are received by the radio interface. Additionally, on all OSes, the network stack is implemented in a monolithic fashion with the kernel, which can cause interference with other OS services.

3) Memory Footprint Evaluation: The resources consumed by the OS are directly related to the system's power consumption. Memory is usually the most power-consuming component on embedded devices, which is reflected in the performed evaluations. However, OSes that are mostly implemented with static resources, usually incur on a higher amount of memory, which greatly increases their power consumption. Contrarily, using dynamic management of resources, and despite increasing their memory footprint, can be a determining factor in reducing the system's energy consumption.

VI. CONCLUSION

In this article, we presented the requirements and characteristics of the IoT ecosystem from a low-end embedded system point of view. These requirements are reflected on features that are expected from the system software running on such devices, and therefore, not every OS fits all device types. Three different OSes (Contiki-NG, RIOT, and Zephyr), chosen by their availability, openness, and their kernel internals singularities, were thoroughly studied and benchmarked considering the IoT-related requirements. Such benchmarks, and other important experiments (real-time, power consumption, and memory footprint), were performed by running the Thread-Metrics Benchmark Suite over the default configuration of each OS while varying the network state (idle, packets to be accepted, and packets to be rejected).

Obtained results reflect their differences and the influence of the kernel-specific implementation, along with the default network stack support. Such differences can affect the overall system's performance, memory footprint, and power consumption. Considering the target application devices, i.e., edge low-end devices, we can roughly conclude that when real-time is not a requirement, Contiki-NG can be a great choice for applications, where the power consumption and the memory footprint are a priority. By its turn, when real-time is a demand, RIOT proved to provide a good balance between performance and real-time capabilities. Hereafter, as proposed by the several development stages of ChamelIoT [24], the future work will encompass a deep study on each OS network stack implementation and the exploration of hardware acceleration for what has been deemed a bottleneck. This kind of approach has somehow been proposed with CUTE mote, where network-related tasks are already being accelerated, such as the IEEE 802.15.4 and 6LoWPAN standards [25], [26]. However, new modules that hinder the system might emerge and allow new solutions to be deployed.

REFERENCES

[1] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen, "A survey on Internet of Things from industrial market perspective," *IEEE Access*, vol. 2, pp. 1660–1679, 2014.

[2] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, "Internet of Things (IoT): Research, simulators, and testbeds," *IEEE Internet Things J.*, vol. 5, no. 3, pp. 1637–1647, Jun. 2018.

[3] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, "Vision and challenges for realizing the Internet of Things," in *Cluster of European Research Projects on the Internet of Things*, EU Commission, Apr. 2010.

[4] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 854–864, Dec. 2016.

[5] P. Narendra, S. Duquennoy, and T. Voigt, "BLE and IEEE 802.15.4 in the IoT: Evaluation and interoperability considerations," in *Internet of Things. IoT Infrastructures*. Cham, Switzerland: Springer, 2016, pp. 427–438. [Online]. Available: https://citation-needed.springer.com/v2/references/10.1007/978-3-319-47075-7_47?format=bibtex&flavour=citation

[6] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—A lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annu. IEEE Int. Conf. Local Comput. Netw.*, Tampa, FL, USA, Nov. 2004, pp. 455–462.

[7] E. Baccelli *et al.*, "RIOT: An open source operating system for low-end embedded devices in the IoT," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018.

[8] Zephyr Project. *Zephyr OS*. Accessed: Jan. 30, 2019. [Online]. Available: https://www.zephyrproject.org/

[9] F. Javed, M. K. Afzal, M. Sharif, and B. Kim, "Internet of Things (IoT) operating systems support, networking technologies, applications, and challenges: A comparative review," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 2062–2100, 3rd Quart., 2018.

[10] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the Internet of Things: A survey," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 720–734, Oct. 2016.

[11] Y. B. Zikria, H. Yu, M. K. Afzal, M. H. Rehmani, and O. Hahm, "Internet of Things (IoT): Operating system, applications and protocols design, and validation techniques," *Future Gener. Comput. Syst.*, vol. 88, pp. 699–706, Nov. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18317710

[12] T. B. Chandra, P. Verma, and A. K. Dwivedi, "Operating systems for Internet of Things: A comparative study," in *Proc. 2nd Int. Conf. Inf. Commun. Technol. Competitive Strategies*, 2016, pp. 1–47.

[13] M. E. C. Bormann and A. Keranen, "Terminology for constrained-node networks," Internet Eng. Task Force, RFC 7228, May 2004. [Online]. Available: https://tools.ietf.org/html/rfc7228#page-11

[14] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.

[15] T. Watteyne *et al.*, "OpenWSN: A standards-based low-power wireless development environment," *Trans. Emerg. Telecommun. Technol.*, vol. 23, pp. 480–493, Aug. 2012.

[16] A. Dunkels, *Design and Implementation of the LWIP TCP/IP Stack*, Swedish Inst. Comput. Sci., Stockholm, Sweden, 2001.

[17] A. Milenković, C. Otto, and E. Jovanov, "Wireless sensor networks for personal health monitoring: Issues and an implementation," *Comput. Commun.*, vol. 29, nos. 13–14, pp. 2521–2533, 2006.

[18] X. Krasniqi and E. Hajrizi, "Use of IoT technology to drive the automotive industry from connected to full autonomous vehicles," in *IFAC PapersOnLine*, vol. 49, no. 29, pp. 269–274, 2016.

[19] T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares, "Bringing hardware multithreading to the real-time domain," *IEEE Embedded Syst. Lett.*, vol. 8, no. 1, pp. 2–5, Mar. 2016.

[20] J. Granjal, E. Monteiro, and J. S. Silva, "Security for the Internet of Things: A survey of existing protocols and open research issues," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 3, pp. 1294–1312, 3rd Quart., 2015.

[21] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surveys*, vol. 51, no. 6, pp. 1–130, Jan. 2019.

[22] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voilà!" in *Proc. IEEE Real Time Embedded Technol. Appl. Symp. (RTAS)*, Montreal, QC, Canada, Apr. 2019, pp. 293–304.

[23] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 27–29.

[24] M. Silva, A. Tavares, T. Gomes, and S. Pinto, "ChamelIoT: An agnostic operating system framework for reconfigurable IoT devices," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 1291–1292, Feb. 2019.

[25] T. Gomes, F. Salgado, A. Tavares, and J. Cabral, "CUTE mote, A customizable and trustable end-device for the Internet of Things," *IEEE Sensors J.*, vol. 17, no. 20, pp. 6816–6824, Oct. 2017.

[26] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, "A 6LoWPAN accelerator for Internet of Things endpoint devices," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 371–377, Feb. 2018.

Miguel Silva is currently pursuing the Ph.D. degree with the University of Minho, Guimaraes, Portugal.

During his master's thesis, he was with the Development and Testing of Automotive Instrument Clusters for a major industry company, and later he joined a research project related with video and multimedia. He refined his knowledge of embedded systems, system software, and connectivity on low-end devices. His current research interests include development of an agnostic operating system in hardware for the Internet of Things.

David Cerdeira is currently pursuing the Ph.D. degree with the University of Minho, Guimaraes, Portugal.

He was a Researcher in developing innovative human machine interfaces for vehicles, granting him a strong background in embedded systems and system programming. During the master's thesis, he was specialized in embedded systems and studying development of secure system. His current research interests include leveraging trusted execution environments for edge computing and Internet of Things.

Sandro Pinto received the Ph.D. degree in electronics and computer engineering from the University of Minho, Braga, Portugal.

He is a Research Scientist and an Invited Assistant Professor with the University of Minho, Guimaraes, Portugal. He was a Visiting Researcher with AIT, Kasendorf, Germany, and the University of Wurzburg, Würzburg, Germany. He has a deep academic background and several years of industry collaboration focusing on operating systems, virtualization, and security for embedded and Internet of Things-based systems. He has published several scientific papers on top-tier conferences/journals.

Tiago Gomes received the master's degree in telecommunications engineering and the Ph.D. degree in electronics and computers engineering from the University of Minho, Guimaraes, Portugal.

He is a Research Scientist and an Invited Professor with the University of Minho. His current research interests include embedded systems hardware/software co-design for resource constrained wireless devices, wireless protocols for low-rate wireless personal area networks, and network protocols for the Internet of Things low-end devices.