RIOT: One OS to Rule Them All in the IoT

Emmanuel Baccelli, Oliver Hahm

RIOT: One OS to Rule Them All in the IoT

E. Baccelli, O. Hahm

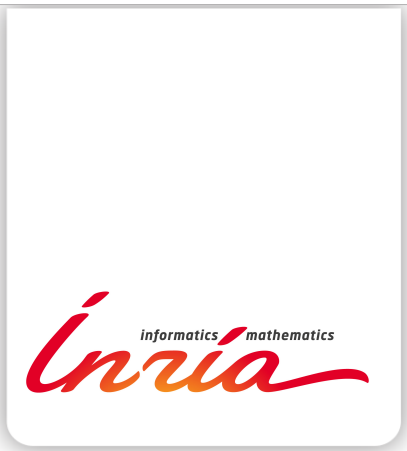
RESEARCH REPORT

N° 8176

December 2012

Project-Team HiPERCOM

ISSN 0249-6399 ISRN INRIA/RR--8176--FR+ENG



RIOT: One OS to Rule Them All in the IoT

E. Baccelli, O. Hahm *

Project-Team HiPERCOM

Research Report n° 8176 — December 2012 — 10 pages

Abstract: The Internet of Things (IoT) embodies a wide spectrum of machines ranging from sensors powered by 8-bits microcontrollers, to devices powered by processors roughly equivalent to those found in entry-level smartphones. Neither traditional operating systems (OS) 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. Hence, in order to avoid redundant developments and maintenance costs of IoT products, a novel, unifying type of OS is needed. The following analyzes requirements such an OS should fulfill, and introduces RIOT, a new OS satisfying these demands.

Key-words: Network, internet, things, objects, IoT, routing, OS, energy, efficient, operating system, protocol, IPv6, wireless, radio, constrained, embedded

* INRIA Saclay-?le-de-France

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

Parc Orsay Université
4 rue Jacques Monod
91893 Orsay Cedex

RIOT: un système d'exploitation pour les gouverner tous dans l'Internet des Objets

Résumé : Ce document analyse les propriétés qu'un système d'exploitation doit avoir pour être approprié pour l'Internet des Objets dans son ensemble, rassemblant des appareils communicants de capacités très diverses. Un nouveau système d'exploitation, RIOT, rassemblant ces propriétés, est ensuite introduit.

Mots-clés : réseaux, internet, objets, routage, radio, sans-fil, capteurs, IPv6, système d'exploitation, protocole, OS, efficace, énergie, contraint, embarqué

1 Introduction

Wireless networks of microelectromechanical systems have been envisioned since the 1990s, when early concepts such as Smart Dust introduced the idea of computers equipped with sensors and simple radio transceivers so cheap and tiny that massive use is possible [1]. A wide range of potential use cases includes scattering such devices from an airplane over an area to monitor, or injecting them directly into the cement of buildings to be monitored. To accomplish their purpose, nodes must have a very small form factor and must be able to function long enough on battery supply. To fulfill these requirements, nodes are usually very constrained in terms of computing power, available memory, communication and energy capacities. Furthermore, nodes are also very constrained in the way they communicate: nodes resort to radio transmissions as seldom as possible to save energy, and otherwise use energy efficient wireless communication technologies, which typically offer little bandwidth and limited packet sizes.

To match such constraints, specialized proprietary software and protocols were designed. In particular, standard Internet protocols, like IP, TCP, or HTTP, were then deemed inappropriate in this context. However, since the 1990s, other distributed embedded systems have emerged in various domains including home and building automation, healthcare automation, or intelligent transport systems. Power line communication (PLC) and spontaneous wireless networks are now expected to interconnect heterogeneous devices including sensors, home appliances, smartphones, vehicles, thus giving birth to the Internet of Things (IoT) [2]. Over the last decade, the IoT has come to embody the low-end of computers on the current and future Internet: a wide spectrum of billions of devices ranging from those based on 8-bit or 16-bit microcontrollers, to devices powered by new generations of energy-efficient 32-bit processors. Thus, IoT systems have not only to fulfill the requirements of wireless sensor networks (WSNs), but also the more complex ones of cyper-physical systems (CPS) as envisioned by projects like the German *Industrie 4.0* initiative [3]. These requirements comprise (i) reliability, (ii) real-time behavior, and (iii) an adaptive communication stack. IPv6 is nowadays considered a viable and desirable solution to run on IoT devices, due to its larger address space compared with IPv4, its more appropriate packet header design and some inherent features for bootstrapping and neighbor discovery. Appropriate IPv6 adaptation layers and protocols are thus being standardized [4–6].

For an OS it is extremely challenging to satisfy all the heterogeneous demands of the diverse IoT hardware platforms and communication stacks that typically compose such networks. Neither a lightweight OS targeting WSNs nor a traditional full-fledged OS (Windows, UNIX, Linux) is capable to fulfill requirements so diverse. Existing WSN OS like Contiki or TinyOS were designed for typical sensing applications. They implement an event-driven approach and are developed to run on very constrained, but energy-efficient hardware. In order to be able to do that however, they lack several key features and compromises they make affect their API which is much less developer-friendly compared to full-fledged OS such as Unix, Windows, or Linux. Such full-fledged OS originally aimed to fulfill the requirements of Internet applications and were designed to run on quite powerful servers, workstations, or personal computers (PCs) without stringent energy constraints, and hence are not applicable on constrained devices such as low-end IoT nodes power by cheap micro-controllers. Hence, in order to avoid redundant developments and maintenance costs, a new, unifying type of OS is needed, which is the subject of this report. In this report, we introduce RIOT OS, an operating system for the IoT, which was designed to fulfill the varying requirements of the heterogeneous platforms that compose such networks. Due to its modularity, RIOT can be stripped down to fit even on very constrained nodes and provide only the required feature set. Furthermore, RIOT's scheduler is optimized for energy-efficiency, even on more powerful devices than typical WSN platforms. On the other hand, RIOT comprises a full TCP/IP stack, offers support for C and C++ programming languages, real multi-threading,

and dynamic memory management, and an API

The remainder of this report is structured as follows. In section 2 we describe the requirements for OS in the IoT. Then we consider existing OS for WSNs and Internet hosts in section 3. Subsequently, we introduce RIOT OS as an OS to fulfill the requirements for IoT devices section 4. The paper closes in section 5 with conclusions and future work.

2 Requirements for an IoT OS

2.1 From WSN to IoT

During the last decade the formerly separated fields of embedded systems and Internet systems overlap increasingly. Wireless Sensor Networks, Cyber-physical Systems, and the Internet of Things take key roles in this merge. We will describe the concepts, main characteristics, and requirements of these systems in the following.

2.1.1 Wireless Sensor Networks

Wireless Sensor Networks cover a wide range of applications: wildlife observation, security surveillance, and monitoring of disaster scenarios are just some examples. WSNs usually consists of large amount of tiny, memory and CPU constrained systems that run on battery supply. They use low-power radio transceivers to transmit sensed data from the nodes to a data sink. Network protocols are mostly designed to fit exactly the particular requirements of the application scenario. One of the main goals in the development of WSNs is maximizing the network lifetime by minimizing the energy consumption of each node and distribute load over the network. Variations of WSN include Wireless Sensor and Actor Networks, Underwater Sensor Networks, and Wireless Multimedia Sensor Networks.

2.1.2 Cyber-Physical Systems

A cyber-physical system (CPS) is a system featuring a tight combination of, and coordination between, the system's computational and physical elements. Today, a precursor generation of cyber-physical systems can be found in areas as diverse as aerospace, automotive, chemical processes, civil infrastructure, energy, healthcare, manufacturing, transportation, entertainment, and consumer appliances. This generation is often referred to as embedded systems. In embedded systems the emphasis tends to be more on the computational elements, and less on an intense link between the computational and physical elements.

2.1.3 Internet of Things

The Internet of Things refers to uniquely identifiable objects (things) and their virtual representations in an Internet-like structure. The concept of the Internet of Things first became popular through the Auto-ID Center and related market analysts publications. Radio-frequency identification (RFID) is often seen as a prerequisite for the Internet of Things. If all objects and people in daily life were equipped with radio tags, they could be identified and inventoried by computers. However, unique identification of things may be achieved through other means such as barcodes or 2D-codes as well. Equipping all objects in the world with minuscule identifying devices could be transformative of daily life. For instance, business may no longer run out of stock or generate waste products, as involved parties would know which products are required and consumed. One's ability to interact with objects could be altered remotely based on immediate or present needs, in accordance with existing end-user agreements

2.2 Requirements for the Operating Systems

The mentioned systems create the following demands for the OS:

- minimal requirements to memory (RAM and program memory) and computing power
- ability to run on constrained hardware without more advanced components like a Memory Management Unit (MMU) or a Floating-Point Unit (FPU)
- support to a variety of hardware platforms
- a high degree of energy-efficiency
- a standard programming interface
- support for high level programming languages
- an adaptive and modular network stack
- reliability

From this list, we can derive that an OS for IoT devices has to strike the balance between the constraints of the hardware and the usability for developers. For example, it is inevitable to implement some parts of the OS and drivers in an assembly language, while at least C or C++ should be available for the application developer. Also, there is usually no need for an user interface on these devices.

Ideally, the capabilities of a full-fledged OS (*e.g.* Linux, Unix, BSD, or Windows) are desirable on all IoT devices. These OS are appealing because they are developer-friendly: numerous available system libraries, network protocols or algorithms, and near-zero learning curve in the sense that developers can code in standard C or C++. However, their minimal requirements in terms of CPU and memory do not fit constrained IoT devices powered by small micro-controllers. While efforts have attempted at pushing down these requirements [7], Linux will never truly be energy efficient unless it is fully redesigned with this goal in mind. For these reasons Linux unfortunately cannot be the one OS to rule them all in the IoT.

On the other hand, the trade-offs that enable a typical lightweight OS targeting WSNs to run on the most constrained IoT devices make it significantly less developer-friendly and/or inappropriate on IoT devices that are less constrained. In the following, we will focus on the dominant WSN operating systems: Contiki [8] and TinyOS [9]. These operating systems follow an event driven design, that is useful for typical WSN scenarios, but is disadvantageous for efficient and functional networking implementations.

An alternative system architecture is desirable, providing capabilities more similar to those of a modern, full-fledged OS such as native multi-threading, hardware abstraction, dynamic memory management. However, these types of systems have so far been considered too complex for IoT devices.

3 Comparing Operating Systems

Fundamentally, an OS is characterized by a few key design aspects: the structure of the kernel, the scheduler, and the programming model. The kernel can either (i) follow a layered approach, (ii) be built in a monolithic fashion, or (iii) implement the microkernel architecture. While the

OS	Min RAM	Min ROM	MCU w/o MMU	16-bit processors
Contiki	<2kB	<30kB	✓	✓
Tiny OS	<1kB	<4kB	✓	✓
Linux	~1MB	~1MB	✘	○
RIOT	~1.5kB	~5kB	✓	✓

Table 1: Hardware requirements for Contiki, TinyOS, Linux, and RIOT. (✓) full support, (○) partial support, (✘) no support. The table compares the OS in minimum requirements in terms of RAM and ROM usage for a basic application as well as support for MCUs without memory management unit (MMU).

OS	C Support	C++ Support	Multi-Threading	Modularity	Real-Time
Contiki	○	✘	○	○	○
Tiny OS	✘	✘	○	✘	✘
Linux	✓	✓	✓	✓	○
RIOT	✓	✓	✓	✓	✓

Table 2: Key characteristics of Contiki, TinyOS, Linux, and RIOT. (✓) full support, (○) partial support, (✘) no support. The table presents if the OS support for C and C++ programming languages, multi-threading, modularity, and real-time behavior.

OS	IPv6	TCP	6LoWPAN	RPL	CoAP
Contiki	✓	○	✓	✓	✓
Tiny OS	✘	○	✓	✓	✓
Linux	✓	✓	✓	○	○
RIOT	✓	✓	✓	✓	✘

Table 3: Supported network protocols of Contiki, TinyOS, Linux, and RIOT. (✓) full support, (○) partial support, (✘) no support.

monolithic kernel is the simplest way to design the OS, it lacks modularity and often results in a complex structure that is hard to understand when the system exceeds a certain size. The layered model helps to organize the system in a hierarchical way. The developer has the choice where to draw the border between kernel and user space. In the microkernel design the whole OS is split up in small, well-defined modules and only a minimum set of functions runs in kernel mode. This approach increases the reliability of the system, as bugs in one of the components like a device driver or the file system will not crash the computer. The choice of the scheduling strategy is tightly bound with real-time support (or the lack thereof), the support for different task priorities, or the supported degree of user interaction. Finally, the programming model defines whether (i) all tasks are executed within the same context and have no segmentation of the memory address space, or (ii) every process can run in its own thread and has its own memory stack. The programming model is also linked to the selection of the programming language and, therefore, the available programming languages for application developers.

For traditional hosts in the Internet, the most widespread OS are Windows, UNIX, and Linux. We will use Linux as an example because it is open source system and a good example for this kind

of OS, but most of the statements are true for the Windows and UNIX derivatives as well. Now let's analyze Contiki, Tiny OS and Linux within this framework. On one hand, Contiki is built in a modular way close to the layered approach, while Tiny OS consists in a monolithic kernel, as does Linux. The scheduling in Contiki is purely event driven, similar to that in TinyOS, where a FIFO strategy is used. This event driven approach is useful for typical WSN applications. The nodes in these scenarios usually will either periodically measure data or sleep for most of the time and just wake up on interrupts triggered by a certain event. After the measurement – and maybe a simple processing of the data – the node will transmit the sensed data to a data sink. Linux on the other hand, uses the Completely Fair Scheduler (CFS) that guarantees a fair distribution of processing time based on a red-black-tree. The programming models in Contiki and TinyOS are based on the event driven model, in a way that all tasks are executed within the same context, although they offer partial multi-threading support. Contiki uses a subset of the C programing language, where some keywords cannot be used, while TinyOS is written in a C dialect called nesC [10]. Linux, on the other hand, supports real multi-threading, is written in standard C, and offers support for a wide range of different programming and scripting languages. Because of these design trade-offs, TinyOS and Contiki are thus lacking several key developer-friendly features: standard C programability, standard multi-threading, real-time support.

The advent of the IoT will connect all kinds of constrained devices to the Internet, sometimes not only as mere hosts, but also as routers or even servers. In order to do so, a suitable, generic OS is required. From the developer's perspective the ideal OS powering IoT devices should behave like any traditional OS and provide a similar application programming interface (API). At the same time it should be able to run on constrained platforms like MSP430 based sensor node with only a few kilobytes of RAM, while retaining the ability to run and exploit additional capabilities available on new energy-efficient 32-bit processors.

4 RIOT - Bridging the gap

RIOT OS aims at bridging the gap we observed between operating systems for WSNs and traditional full-fledged operating systems currently running on internet hosts. The key design goals for RIOT OS were energy-efficiency, small memory footprint, modularity, and a developer friendly programming interface, which make RIOT the best choice to power the widest spectrum of IoT devices.

4.1 Architecture

The system is based on a microkernel and offers real multi-threading. In order to provide maximum modularity, RIOT implements the micro-kernel architecture inherited from FireKernel [11], a kernel designed to fulfill strong real-time requirements for emergency scenarios. RIOT thus supports multi-threading and real-time in that it features (i) zero-latency interrupt handlers, and (ii) minimum context-switching times combined with thread priorities. In order to achieve a maximum energy-efficiency, RIOT introduces a tickless scheduler able to function on constrained devices. Most schedulers use timers to wake up periodically and check if something needs to be done: this is the *timer tick*. However, if the processor is idle, it has to wake up from its power-saving sleep state every timer tick, even when there is nothing to do. This behavior is thus not desirable for energy constrained systems. Moreover, with most microcontrollers, a timer interrupt cannot wake up the processor from deep-sleep mode (only external interrupt sources can). 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.

4.2 Implementation

For any interrupt, the kernel itself is registered as the main interrupt handler instead of a forwarding to the Vectored Interrupt Controller (VIC), that many MCUs provide. As soon as the kernel saved the context of the current thread, it calls a jump instruction to the VIC address, after setting the return address to instruction after the jump. Now, the interrupt services routine (ISR) is called by the VIC and returns to the kernel. In this manner, the kernel can perform the necessary tasks caused by an interrupt without the need of choosing the corresponding ISR itself. In order to achieve a maximum energy-efficiency, RIOT introduces a tickless scheduler able to function on constrained devices. In most operating systems, schedulers use timers to wake up periodically and check if something needs to be done. This periodic timer event, usually called the *timer tick*, is simple in its design, but has a significant drawback: the timer tick happens periodically, irrespective of the processor state, whether it is idle or busy. If the processor is idle, it has to wake up from its power-saving sleep state every timer tick, even when there is nothing to do. This behavior is thus not desirable for energy constrained systems. Moreover, with most microcontrollers, a timer interrupt cannot wake up the microcontroller from deep-sleep mode (only external interrupt sources can). Hence, using a periodical timer prevents the microcontroller from using deep-sleep mode and thus consumes much more power than needed over the whole runtime. 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.

For instance, assuming that every interrupt handler uses only `N` cycles, it is guaranteed that RIOT will switch to the corresponding thread (e.g. the device driver thread) within a maximum of $N+\epsilon$ clock cycles, with ϵ being a very low number of clock cycles consumed by the context switch itself. It is also guaranteed that the thread with the highest priority, that is not blocked or sleeping, will never be interrupted by more than $N+\epsilon$ cycles. The kernel is kept as simple as possible and comprises - besides the scheduler and threading system - only mutexes and inter-process communication (IPC). Following the microkernel paradigm all other system functionality such as device drivers or the file system run in threads. Comparing to other OS for WSN the only way to achieve anything that has guarantees, is to stay in the interrupt handler. So there is usually nothing that supports the notion of guarantees, neither by design, convention, API, or code.

5 Conclusion

Despite this sophisticated architecture and efficient scheduling, RIOT has a low memory footprint. Available open source RIOT code [12] requires less than 5 kByte of ROM and less than 2 kByte of RAM for a basic application built for a MSP430 platform, for instance. RIOT uses a multi-threaded programming model in combination with standard ANSI C code and a common POSIX-like API for all supported hardware – from 16-bit microcontrollers to 32-bit processors. Hence, for projects involving heterogeneous IoT hardware, it is possible to build the whole software system upon RIOT and easily adopt existing libraries. Moreover, the availability of several networking protocols including the latest standards of the IETF for connecting constrained systems to the internet like 6LoWPAN and RPL make RIOT fully IoT-ready. Ongoing work includes full POSIX compliance and porting to various ARM processors. This allows straightforward development and porting of IoT software. An advanced timer system and hardware abstraction allows platform independent kernel and higher layers implementations. RIOT

also supports software suites with or without dynamic memory management: the kernel itself uses only static memory allocation, but an implementation of *malloc* is provided. Moreover, the availability of several networking protocols including the latest standards of the IETF for connecting constrained systems to the internet like 6LoWPAN and RPL make RIOT fully IoT-ready. On-going work includes a CoAP implementation for RIOT and porting of the OS to additional popular IoT platforms.

References

[1] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "Next century challenges: mobile networking for "Smart Dust"," in *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA: ACM Press, 1999, pp. 271–278. [Online]. Available: http://dx.doi.org/10.1145/313451.313558

[2] K. Ashton, "That 'Internet of Things' Thing," *RFID Journal*, July 2009. [Online]. Available: http://www.rfidjournal.com/article/view/4986

[3] B. für Bildung und Forschung, "Zukunftsprojekt Industrie 4.0," October 2012. [Online]. Available: http://www.bmbf.de/de/19955.php

[4] G. Montenegro, N. Kushalnagar, J. Hui, D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," in *Internet Engineering Task Force RFC 4944*, 2007.

[5] T. Winter, P. Thubert, "IPv6 Routing Protocol for Low-Power and Lossy Networks," in *Internet Engineering Task Force RFC 6550*, 2012.

[6] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained Application Protocol (CoAP)," December 2012. [Online]. Available: http://tools.ietf.org/id/draft-ietf-core-coap-13.txt

[7] D. McCullough, "uCLinux for Linux Programmers," in *Linux Journal*, 2004.

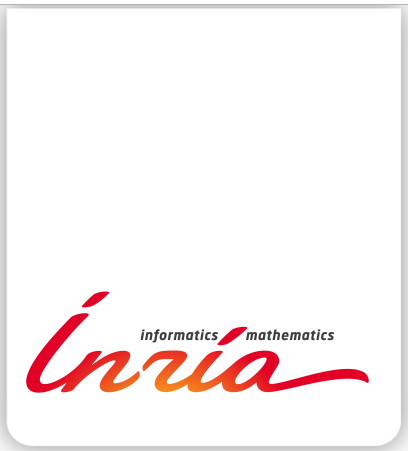
[8] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors." in *LCN*. IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: http://dblp.uni-trier.de/db/conf/lcn/lcn2004.html#DunkelsGV04

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Berlin/Heidelberg: Springer-Verlag, 2005, ch. 7, pp. 115–148. [Online]. Available: http://dx.doi.org/10.1007/3-540-27139-2_7

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/781131.781133

[11] H. Will, K. Schleiser, and J. H. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios," in *Proc. of the 34th IEEE Conference on Local Computer Networks (LCN)*, M. Younis and C. T. Chou, Eds. Piscataway, NJ, USA: IEEE Press, October 2009, pp. 834–841.

[12] "RIOT OS - An Operating System for the IoT," 2012. [Online]. Available: www.riot-os.org



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

Parc Orsay Université
4 rue Jacques Monod
91893 Orsay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399