RIOT OS: Towards an OS for the Internet of Things

Emmanuel Baccelli and Oliver Hahm
INRIA, France

Mesut Günes and Matthias Wählisch
Freie Universität Berlin, Germany

Thomas C. Schmidt
HAW Hamburg, Germany

***Abstract*—The Internet of Things (IoT) is characterized by heterogeneous devices. They range from very lightweight sensors powered by 8-bit microcontrollers (MCUs) to devices equipped with more powerful, but energy-efficient 32-bit processors. Neither a traditional operating system (OS) currently running on Internet hosts, nor typical OS for sensor networks are capable to fulfill the diverse requirements of such a wide range of devices. To leverage the IoT, redundant development should be avoided and maintenance costs should be reduced. In this paper we revisit the requirements for an OS in the IoT. We introduce RIOT OS, an OS that explicitly considers devices with minimal resources but eases development across a wide range of devices. RIOT OS allows for standard C and C++ programming, provides multi-threading as well as real-time capabilities, and needs only a minimum of 1.5 kB of RAM.**

I. Introduction

Billions of heterogeneous devices such as sensor nodes, home appliances, smartphones, and vehicles are expected to be interconnected by spontaneous wireless networks or power line communication, thus giving birth to the IoT [1]. Such devices, though extremely constrained in terms of computing power, available memory, communication, and energy capacities, are however expected to fulfill the requirements of cyper-physical systems: (i) reliability, (ii) real-time behavior, and (iii) an adaptive communication stack to integrate the Internet seamlessly.

An OS for IoT devices should fulfill these requirements and run on a wide spectrum of hardware, ranging from nodes based on low-power MCUs, to nodes powered by new generations of energy-efficient 32-bit processors. None of the existing OS – neither a lightweight OS targeting Wireles Sensor Networks (WSNs) nor a full-fledged OS – is capable to fulfill requirements so diverse. Hence, in order to avoid redundant developments and maintenance costs, a new, unifying type of OS is needed, which is the subject of this poster.

Problem Statement: Ideally, the capabilities of a full-fledged OS should be available on all IoT devices. The following will focus on the characteristics of Linux, since it is a good example for an open source OS among the major full-fledged OS. Compared to a typical lightweight OS targeting WSNs, Linux is more 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, Linux's minimal requirements in terms of CPU and memory do not fit constrained IoT devices powered by small MCUs. While efforts have attempted at pushing down these requirements [2], we argue that Linux has not been designed for the IoT and cannot fulfill strict energy efficiency. For these reasons Linux cannot be expected to become 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 its use on more powerful IoT devices will result in a less energy-efficient implementation, while not exploiting devices' full capabilities. The dominant WSN OS, Contiki and TinyOS [3], follow an event driven design, which is useful for typical WSN scenarios, but exhibit drawbacks for efficient and functional networking implementations. For example, in a typical WSN scenario it is sufficient to process the tasks sequentially in a loop, but doing so limits the windows sizes to at most one TCP connection, due to constrained memory. An alternative system architecture is desirable, providing capabilities 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.

II. RIOT: An OS for Smaller Things in the Internet

Design Aspects for an IoT OS: Fundamentally, an OS is characterized by the following key design aspects: the structure of the kernel, the scheduler, and the programming model. The kernel can either (i) be built in a monolithic fashion, (ii) follow a layered approach, or (iii) implement the microkernel architecture. The choice of the scheduling strategy is tightly bound to 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 available programming languages for application developers.

Comparing Existing Solutions: Based on these observations, we compare Contiki, Tiny OS, and Linux, and we derive first principles for an OS for the IoT. Contiki follows a modular concept close to the layered approach, while Tiny OS consists of a monolithic kernel, as Linux. The scheduling in Contiki is purely event driven, similar to that in TinyOS, where a FIFO strategy is used. Linux on the other hand, uses a scheduler, which guarantees a fair distribution of processing time. The programming models in Contiki and TinyOS are based on the event driven model, in a way that all tasks are executed

978-1-4673-5946-7/13/$31.00 ©2013 IEEE

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30kB	○	✘	○	✓	○	○
Tiny OS	<1kB	<4kB	✘	✘	○	✓	✘	✘
Linux	~1MB	~1MB	✓	✓	✓	✘	○	○
RIOT	~1.5kB	~5kB	✓	✓	✓	✓	✓	✓

TABLE I
KEY CHARACTERISTICS OF 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, SUPPORT FOR PROGRAMMING LANGUAGES, MULTI-THREADING, MCUS WITHOUT MEMORY MANAGEMENT UNIT (MMU), MODULARITY, AND REAL-TIME BEHAVIOR.

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. 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 and C++ programability, standard multi-threading, and real-time support (see Table I). An OS leveraging a successful, large-scale deployment of IoT devices should support these functions. From the developer perspective, this means a powerful, hardware-independent API.

RIOT Architecture Overview: RIOT OS aims at bridging the gap we observed between OS for WSNs and traditional full-fledged OS currently running on Internet hosts. It is based on design objectives including energy-efficiency, small memory footprint, modularity, and uniform API access, independent of the underlying hardware.

RIOT implements a microkernel architecture inherited from FireKernel [4], thus supporting multi-threading with standard API. In addition to FireKernel's original features, RIOT adds support for C++ – enabling powerful libraries such as the Wiselib algorithm framework – and provides a TCP/IP network stack. Advantages of the RIOT architecture thus include: (i) high reliability and (ii) a developer-friendly API. The *modular* microkernel structure of RIOT makes it robust against bugs in single components. Failures in the device driver or the file system, for example, will not harm the whole system. RIOT allows developers to create as many threads as needed and distributed systems can be easily implemented by using the kernel message API. The amount of threads is only limited by the available memory and stack size for each thread, while the computational and memory overhead is minimal.

RIOT Technical Details: To fulfill strong real-time requirements RIOT enforces constant periods for kernel tasks (e.g., scheduler run, inter-process communication, timer operations). An important prerequisite for guaranteed runtimes of $O(1)$ is the exclusive use of static memory allocation in the kernel. Yet, dynamic memory management is provided for applications. We achieve constant runtime of the scheduler by using a fixed-sized circular linked list of threads. Constant runtime of the timer operations is obtained by exploiting the fact that MCUs typically provide multiple compare registers.

It is mandatory to maximize the duration spent in deep-sleep modes, in order to implement energy-efficiency also for more powerful IoT devices, RIOT thus introduces a scheduler that works without any periodic events. Whenever there are no pending tasks, RIOT will switch to the *idle* thread, which determines the deepest possible sleep mode depending on peripheral devices in use. Only interrupts (external or kernel-generated) wake up the system from idle state.

Low complexity of kernel functions is a main factor for the energy efficiency of an OS. Therefore, the duration and occurrence of context switching have to be minimized. In RIOT context switching is performed in two cases: (i) a corresponding kernel operation itself is called, e.g., a mutex locking or creation of a new thread, or (ii) an interrupt causes a thread switch. The first case will occur rarely. For example, every thread is usually created once. Hence, it is important to reduce the processing time in case of a thread switch. Therefore, RIOT's kernel provides a minimized scheduler, when it gets called out of an interrupt service routine. In that case, saving the old thread's context is not required and thus a task switch can be performed in very few clock cycles.

III. AVAILABLE CODE & FUTURE WORK

Despite this sophisticated architecture and efficient scheduling, RIOT has a low memory footprint. Available open source RIOT code [5] requires less than 5 kByte of ROM and less than 1.5 kByte of RAM for a basic application on MSP430, 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 (6LoWPAN, RPL) make RIOT IoT-ready. On-going work includes full POSIX compliance and porting to various IoT platforms.

REFERENCES

[1] K. Ashton, "That 'Internet of Things' Thing," *RFID Journal*, 2009.
[2] D. McCullough, "uCLinux for Linux Programmers," in *Linux Journal*, 2004.
[3] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors Journal*, 2011.
[4] H. Will, K. Schleiser, and J. H. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios," in *IEEE LCN*, 2009.
[5] "RIOT Operating System," http://www.riot-os.org.