Operating Systems for the IoT — Goals, Challenges, and Solutions

Emmanuel Baccelli and Oliver Hahm
INRIA
Saclay, France

Mesut Günes and Matthias Wählisch
Freie Universität Berlin
Germany

Thomas C. Schmidt
HAW Hamburg
Germany

Abstract — The Internet of Things (IoT) embodies a wide spectrum of machines ranging from sensors powered by 8-bit microcontrollers to devices powered by processors equivalent to those found in smartphones. Neither traditional operating systems currently running on Internet hosts, nor typical operating systems 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 operating system is needed. This paper analyzes requirements of an operating system for the IoT. We introduce RIOT, an operating system that satisfies these demands.

I. 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 [1]. These computers are so cheap and tiny that massive use is possible. To accomplish the purpose of the various application scenarios, nodes must be cheap, have a very small form factor coupled with the ability to function long enough (typically months or years) on battery supply. These requirements lead to the development of very constrained nodes in terms of computing power and available memory. Furthermore, nodes are also very constrained in the way they communicate: they 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 very small payload per packet.

To match such constraints, specialized proprietary protocols have been designed and used so far. Standard Internet protocols, such as TCP and IP, were at first deemed inappropriate in this context [2]. However, as various distributed embedded systems have emerged recently (home automation, building automation, healthcare automation, and intelligent transport systems) power-line communications and spontaneous wireless networks are now expected to interconnect heterogeneous devices including sensors, home appliances, handhelds, vehicles, thus giving birth to the Internet of Things (IoT) [3].

Over the last decade the IoT has thus 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 processors roughly equivalent to those found in entry-level smartphones. The use of IP on these devices is increasingly considered as the cheapest alternative in the long-run, and contrary to IPv4, IPv6 is viewed as a viable and desirable solution to power IoT devices, due to its larger address space, its more appropriate packet header design and convenient features that enable bootstrapping and neighbor discovery.

The contribution of this paper is twofold. First, we analyze the generic requirements for software running on IoT devices. In particular, we derive from this analysis that none of the existing operating systems is capable to fulfill the diverse requirements of IoT systems, which include heterogeneous hardware constraints, as well as various network stack, autonomy and real-time constraints. Although several efforts aimed at adapting existing operating systems for the IoT, key features such as maximum energy efficiency or strong real-time guarantees cannot be efficiently implemented as add-ons to a pre-existing system, because such features impact every part of the system. The diversity of the requirements that need to be fulfilled by software running on IoT devices is too challenging for existing operating systems, which were not designed to run on the full range of hardware platforms that compose the IoT.

As a second contribution, we introduce RIOT OS, a microkernel-based operating system matching the various software requirements for IoT devices. We show how RIOT’s design and implementation deals with the diverse challenges in powering networks of constrained devices connected to the Internet. For this purpose, RIOT also features the implementation of an adaptive network stack, providing full-fledged IPv6 as well as protocols targeting more constrained networks, e.g. 6LoWPAN or RPL. By providing the same developer-friendly API across all platforms (from 16-bit microcontrollers to 32-bits processors) and by simultaneously providing key features such as real-time capabilities and energy efficiency, RIOT can power a wide spectrum of IoT devices. RIOT can thus be leveraged to avoid redundant code development and maintenance costs for IoT applications.

The remainder of this paper is structured as follows. In Section II, we will analyze the key software requirements for IoT devices, while subsection II-C will focus on a specific part of the software running on such devices: the network stack. Then, Section III will analyze, categorize and compare existing operating systems, while matching them with the identified requirements. Subsequently, we introduce RIOT in Section IV, an alternative OS fulfilling the heterogeneous requirements of IoT devices, while providing a standard, powerful API. The paper closes in Section V with conclusions and future work.
II. IoT SOFTWARE REQUIREMENTS

During the last decade the formerly separated fields of embedded systems and Internet systems converge increasingly. In parallel with the IoT, concepts such as Cyber-physical Systems (CPS) have emerged, based on a complex combination of, and the coordination between, computer software systems and mechanical or electronic objects, connected by a wired or wireless network, e.g. the Internet. Each CPS typically consists of very heterogeneous hardware platforms, which need to communicate with one another in order for the system to work.

A. Use Case

A vast research community and numerous projects currently address the IoT domain. One example is SAFEST, a French-German project funded by the ANR and BMBF, which studies and develops a CPS using sensors to provide better safety and security in public spaces and around critical infrastructure [4]. More specifically, SAFEST targets crowd control and area surveillance in airports by coupling two categories of systems:

1) A visual and audio surveillance system that monitors large crowds in order to provide guidance in case of unexpected events (e.g., mass panic).
2) A perimeter protection system that uses distributed event detection algorithms to detect unauthorized intrusions.

For the first task, rather powerful hardware is required: The system must be capable of audio-video processing, and the amount of data that has to be transferred can be substantial. For the second task on the other hand, hardware requirements are quite different: Light-weight nodes should be scattered over a large area, in which wired power supply may not be feasible, and the amount of data that has to be transferred is much less substantial. In order to fit these diverging requirements the project partners decided to use two different hardware platforms:

1) a powerful board based on an Intel® Atom™ and an ARM Cortex-A8 controller for the surveillance system, and
2) a constrained board based on an ARM7 TDMI-S micro-controller for the perimeter protection system.

The issue that arose at this point was that, as discussed in the following, existing operating systems are either (i) unable to leverage the capabilities of the powerful board, or (ii) unable to run on the constrained board. This implies the use of several operating systems in parallel, and thus forces the development and the maintenance of redundant application code running across the network of such devices. Therefore, in order to cut the cost of redundant code development and maintenance, a developer-friendly, generic operating system is desirable, able to run on constrained devices, as well as to leverage the capabilities of more powerful platforms.

B. Software Characteristics

As for the SAFEST project, typical IoT scenarios comprise very heterogeneous hardware handling tasks of various complexity. This implies strong requirements for the software running on such devices. A first category of requirements pertains to potentially constrained hardware. A second category deals with the demand for these systems to work autonomously. Finally, a third type of requirements focuses on the usability of the system from a developer’s perspective.

Heterogeneous Hardware Constraints

Memory Requirements
As many of typical IoT devices have very little memory (typically between 5kB and some hundreds of megabytes), the minimum memory requirement of the software has to be very low. This concerns RAM as well as persistent program storage.

CPU Requirements
The complexity of operations must be kept very low, because some of the MCUs in a IoT system will work at a very low clock cycle.

Limited Features
Software for IoT must be able to run on constrained hardware without more advanced components like a Memory Management Unit (MMU) or a Floating-Point Unit (FPU).

Platform Support
Software for IoT must support a variety of hardware platforms, run on constrained platforms, but also be able to leverage the capabilities of less constrained platforms.

Autonomy

Energy efficiency
The software must exploit the power saving features of the hardware and allow for large sleep cycles as much as possible.

Adaptive Network Stack
The network stack should provide full-fledged TCP/IP implementations as well as a 6LoWPAN stack aiming for more constrained devices. It should also be modular in a way that the protocols at each layer can be easily replaced.

Reliability
IoT systems are often deployed in critical applications in which physical access is difficult and related to high costs in many cases. For that reason, it is important that the system is robust and thus that the operating system runs very reliably.

Programmability

Standard API
In order to ease software development and simplify the porting of existing software, a standard programming interface such as POSIX or STL should be provided.

Standard Programming Languages
Support for standard high level programming languages, such as C++, is vital.

Following these generic insights concerning software running on IoT devices, we can derive the following conclusions concerning operating systems eligible to power the IoT. An
operating system for IoT devices must strike the balance between the constraints of the hardware and the usability for developers. On the one hand, it must perform well on multiple systems with a varying capacities and capabilities. On the other hand, it must be able to exploit the features provided by the platform in order to handle different complex tasks. Nevertheless, software should be easily portable between various systems, to minimize development and maintenance efforts while providing optimal interconnectivity. It is desirable to run the same operating system on all of the current platforms, as well as on future platforms. This operating system should however be as developer-friendly as possible. For example, while it is inevitable to implement some parts of the operating system and drivers in an assembly language, the OS should at least make C or C++ available for the application developer, a task which has remained a challenge so far, on constrained platforms.

C. IoT Network Stack

Let’s now study a key part of the software running on IoT devices: the network stack. In order to integrate constrained nodes into the Internet, protocols of the TCP/IP stack must indeed be adapted for these systems. For example, standard IP datagrams require a minimum payload of 576 bytes for IPv4 or 1280 bytes for IPv6, while typical IoT devices use a radio technology that offers a packet size of less than 150 bytes. In fact, if we compare the traditional IP network stack and the technology that offers a packet size of less than 150 bytes. In this domain, the IETF is currently developing CoAP (Constrained Application Protocol [9]), a web transfer protocol targeting machine-to-machine (M2M) applications, able to function on constrained nodes in LLNs, which can also natively interoperate with HTTP.

Various routing protocols such as OSPF, IS-IS, or BGP are used for routing within or between autonomous systems in the Internet. These protocols were designed to fulfill requirement that are extremely different from those of spontaneous wireless networks in the IoT with IEEE 802.15.4 MAC/PHY. Several alternative routing protocols have thus been proposed and designed. The IETF has, for example, recently standardized a routing protocol for low-power and lossy networks (LLNs) called RPL [7], targeting wireless sensor networks that focus primarily on data collection at the sink.

At the transport layer the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are mostly used in the Internet. While some efforts currently take place to adapt TCP for LLNs [8], TCP is often considered as too complex for LLNs. In addition, its elaborate congestion control mechanism usually decrease performance in wireless networks as packet loss does not indicate congestion. UDP is thus typically used on LLNs.

Similarly, at the application layer, while HTTP is used on the Internet, this protocol is considered too complex on constrained nodes and alternative mechanism have been developed for operation on LLNs. In this domain, the IETF is currently developing CoAP, a well-defined modules, and only a minimum set of functions implemented the microkernel architecture. This decision strongly affects the whole composition of the system. While a monolithic kernel is the simplest way to design an operating system, 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 better segment the system in a hierarchical way. The developer has to chose the level of separation between kernel and user space. In the microkernel, design goes even further in modularity: the whole operating system 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 individual components (such as device drivers or the file system) will not crash the system.

Another key design aspect is the scheduler. The choice of the scheduling strategy is tightly bound to capabilities of system to fulfill real-time properties, to support different priorities and degrees of user interaction.

### III. Operating Systems

In order to successfully adapt to the constraints of typical IoT devices, an operating system must be designed purposely in all its aspects, similarly to what we observed for the network stack in the previous section. In this section, we will thus analyze the key design aspects of operating systems in the IoT context and decompose the existing operating systems according to this analysis, in order to compare them.

A. Characteristics

There are multiple characteristics that categorize different types of operating systems [10]. One of the most fundamental design aspect concerns the structure of the kernel. The operating system can (i) be built in a monolithic way, (ii) follow a layered approach, or (iii) implement the microkernel architecture. This decision strongly affects the whole composition of the system. While a monolithic kernel is the simplest way to design an operating system, 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 better segment the system in a hierarchical way. The developer has to chose the level of separation between kernel and user space. In the microkernel, design goes even further in modularity: the whole operating system 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 individual components (such as device drivers or the file system) will not crash the system.

At the network layer IPv4 and IPv6 use ICMP for control messaging. While IP datagrams with a typical size of a few hundred bytes can be encapsulated into link layer frames traditionally used in the Internet, more effort is required on IoT networks, where an adaption layer (6LoWPAN) is used to deal with the constraints of the IEEE 802.15.4 MAC/PHY, such as packet fragmentation. This adaptation layer introduces methods for header compression decreasing datagram size and modifications to the neighbor discovery that reduces bootstrapping complexity [5], [6].
A third fundamental design aspect is the programming model. In some operating systems all tasks are executed within the same context and have no segmentation of the memory address space. Other systems offer multi-threading where every process can run in its one thread and has its own memory stack. The programming model is also linked to the selection of the programming language. It may have a strong impact on the chosen programming language for operating system implementation itself and therefore, define which programming languages are available for the application developers.

B. Comparison of Operating Systems

Taking into account the fundamental design aspects and the requirements derived from Section II, we now compare existing operating systems that are a priori eligible to power IoT devices. For that, we selected representative operating systems both among embedded WSAN operating systems, and among traditional full-fledged operating systems running on Internet hosts.

On WSAN devices, the two dominant operating systems are Contiki [11] and TinyOS [12]. Both provide implementations of various algorithms, protocols, device drivers, and helpful tools such as a file systems or a shell. On more traditional devices connected to the Internet, the most widespread operating systems are Windows, several UNIX derivatives, and Linux. In the following, we will thus comparatively analyze Tiny OS, Contiki, and Linux, the latter being chosen as the representative full-fledged OS, because it is open source and supports a wide spectrum of hardware platforms. Note that most of the statements below are true for the Windows and UNIX derivatives, too.

We will consider the characteristics of these operating systems with regards to the aforementioned categories of design aspects. TinyOS and Linux are implemented as a monolithic kernel, while Contiki is built in a modular way that corresponds to a layered system. In TinyOS a set of required components is glued together to build a single, static binary. The components expose one or more interfaces and communicate via commands and events. While the Linux kernel itself is monolithic, it is possible to configure device drivers as modules. In this way, a Linux system can be trimmed down to match exactly the particular needs for the application. But despite the fact that these modules can be loaded and unloaded during runtime, a failing driver might still crash the whole system. Contiki offers the operating system facilities, such as device drivers, communication, and sensor data handling as services. Besides the mandatory components, the Contiki core however comprises also the uIP stack, a device driver loader, and the protothreading system.

The scheduling in Contiki is purely event driven, similar to that in TinyOS where a FIFO strategy is used. Their scheduling strategies are optimized for simple event processing, such as handling interrupts from an asynchronous sensor. Linux currently uses the Completely Fair Scheduler (CFS) that guarantees a fair distribution of processing time based on a red-black-tree. The goals for this scheduler is maximization of overall CPU utilization as well as interactive performance.

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 some kind of multi-threading support. TinyOS version 2.1 introduces TOS Threads that use a cooperative threading approach, where the threads have to rely on an application to explicitly yield the CPU [13]. Contiki provide protothreads as a light-weight and stackless implementation of simple multi-threading [14]. Since events run to completion, no process synchronization between protothreads is possible.

Contiki uses a subset of the C programing language, where some keywords cannot be used. TinyOS is written in a C dialect called nesC. Linux, on the other hand, supports real multi-threading, is written in C and offers support for a wide range of different programming and scripting languages.

Let us now examine the capabilities of these operating systems with regards to the requirements derived from Section II in order to power a wide range of IoT devices. Contiki and TinyOS were designed to match the requirements of the very constrained WSANs. Thus, they have very little minimum requirements on memory and computational power. Although there exist specialized ports of Linux for embedded systems (like uClinux), the minimum requirements on memory and CPU are still much higher [15]. The same is true for systems with a limited feature set, where the WSAN operating systems will work on most common micro-controllers, uClinux is only available for more powerful micro-controllers. All modern operating systems aim to achieve a certain degree of energy-efficiency. However, due to the relatively complex architecture, Linux will never allow for very long sleep periods. But this architecture enables advanced programming interfaces, such as almost full POSIX compliance and vast amount of available programming and scripting languages. The WSAN operating systems, Contiki and TinyOS, provide only C or a C-like programming language, and the developer has to adapt to the particular programming paradigm. While the amount of available network protocols is similar for all of these operating systems, the focus is very diverse. Linux offers a highly configurable TCP/IP stack, capable to deal with very high data rates, but support for low-power protocols such as 6LoWPAN or RPL is only rudimentary. On the contrary, Contiki and TinyOS provide evolved implementations of these protocols for constrained devices, their variants for more complex protocols such as TCP is limited. Looking at the summary of these capabilities shown in Table I, we can conclude that TinyOS, Contiki and Linux each lack important features in order to fulfill the requirements of IoT devices in general.

IV. RIOT OS

RIOT OS aims exactly to fulfill the requirements mentioned in Section II and bridge the gap we observed between operating systems for WSANs and traditional full-fledged operating systems currently running on Internet hosts. The source code of RIOT is available online [16].

In order to achieve a minimum memory usage, the system is designed in a modular way. Thus, the configuration of the system can be customized to meet the particular specification. The size of the kernel itself is minimized, thus requiring only a few hundred bytes of RAM and program storage. Dependencies between the modules are reduced to an absolute minimum.

In contrast to many other operating systems, RIOT’s sched-
RIOT's kernel inherits from that of FeuerWare [17] which makes the IoT. The IoT is indeed very challenging concerning their features it comprises an highly adaptive network stack, and offer a standard compatible, and support for various platforms, it is possible to make the IoT. To counter this challenge, we propose a novel operating system, called System, which meets these requirements. RIOT provides a reliable microkernel architecture, where the kernel itself requires just a few hundreds bytes of ROM and RAM. Among its features it comprises an highly adaptive network stack, including the latest standards of the IETF for connecting constrained systems to the Internet like 6LoWPAN and RPL. Due to RIOT’s developer-friendly API, which is partly POSIX compliant, and support for various platforms, it is possible to build the whole software system upon RIOT OS and easily adapt existing libraries for projects involving heterogeneous IoT hardware.

**REFERENCES**


