Some experiences in building IoT platform

Aleksandar Milinković, Stevan Milinković, and Ljubomir Lazić

Abstract — **In this paper we give a short survey of some existing solutions and describe our attempt to build an Internet of Things platform independent of underlying hardware. It is concluded that the best way to do that is to virtualize hardware by using common microkernel-based operating system. Such an operating systems exists as an open source, however porting it to the particular hardware board turned out to be a non-trivial task. The proposed platform was tested in setup with full radio coverage and with star topology by using the standard socket API.**

Keywords — **constrained devices, Internet of Things, ARM, RIOT, MikroC.**

I. Introduction

INTERNET of Things is not a term that has a very clear definition, yet it so widely used, primarily for marketing purposes. One definition that would be acceptable is provided in the Strategic Research Agenda of the Cluster of European Research Projects on the Internet of Things: [1]: “Internet of Things (IoT) is an integrated part of Future Internet and could be defined as a dynamic global network infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual ‘things’ have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network.”

The IoT provides communication between different objects integrated into a global network as well as the people and these objects. In order to achieve this, it is necessary to ensure that the cost of such facilities is low, and a software that enables their operation should be adapted to the environment in terms of performance, memory size and power consumption. Today, the IoT can be associated with all that is called "smart", e.g. smart cities, smart metering, smart energy, home automation, eHeatlh, etc.

Simply put, the IoT is the network of physical objects containing embedded technology that can be accessed through the Internet.

Results presented in this paper are part of the research that is supported by Ministry of Education, Science and Technological Development of the Republic of Serbia, Grants No. III-45003 and TR-35026.

Aleksandar Milinković (amilinko@gmail.com), Belgrade University, School of Electrical Engineering, Bulevar kralja Aleksandra 73, 11120 Belgrade, Serbia.

Stevan Milinković (corresponding author) (smilinkovic@raf.edu.rs), Union University, School of Computing, Knez Mihailova 6, 11000 Belgrade, Serbia.

Ljubomir Lazić (ljubomir.lazic@metropolitan.ac.rs), Metropolitan University, Faculty of Information Technology, Tadeuša Košćuška 63, 11000 Beograd, Serbia.

II. Problem statement

The components that are building blocks of IoT have very limited resources, with more restrictions than it is usual for embedded systems. Therefore, the term constrained device [2] was recently introduced in order to define the difference between such a system and desktop computer. This especially applies to the significantly reduced energy, lower computing power, as well as significantly reduced amount of memory. Bearing in mind that the constrained devices are mainly based on microcontrollers, which usually don't have memory management units, their system software exclude solutions that are common in embedded systems (e.g., embedded Linux). In addition, there is a need to work in real time in some applications, which further narrows our choice.

Accordingly, our goal was to build IoT platforms with the following characteristics [2]:

First platform:

- Class of Constrained Devices: C2
- Class of Energy Limitation: E9 (no direct quantitative limitations to available energy)
- Strategy of Using Power for Communication: P9 (Always-on)

Second platform:

- Classes of Constrained Devices: C0 and C1
- Class of Energy Limitation: E1 (Period energy-limited)
- Strategy of Using Power for Communication: P1 (Low-power)

The operating system for the IoT must take into account all the constraints of hardware while maintaining a high usability for developers. It must function well on systems with different hardware capabilities and capacities. In addition, the operating system must have the ability to work in real time, and it is desirable to have some standard interface (e.g., POSIX), for good portability of applications, for minimizing maintenance, as well as to provide the ability to easily connect to other devices on the Internet. However, the ease of use for developers is crucial. In addition to the C language, the use of other programming languages and libraries is highly desirable, for example C ++ and STL, but they strongly depend on the development toolchain. Not all of them have this capability, and even in those cases where it exists, this task remained a challenge on IoT platforms.

III. Related work

Operating System (OS) for IoT must be designed purposely, which means that it must take into account all the constraints that exist in such devices.

Depending on their primary purpose, operating systems have different characteristics, but one of the main things which is necessary to take into account is the kernel structure. Kernel implementation may also have historical reasons, but in embedded systems they are of minor importance. Therefore, designers have monolithic, layered and microkernel architectures at their disposal.

Another important aspect in choosing or designing the operating system is scheduler. Scheduling strategy directly affects the system's ability to operate in real time, as well as to support different priorities and ways of interacting with the user. It also has a significant impact on energy consumption of the entire device.

Finally, the third aspect in the design is the programming model. On some operating systems, all tasks are executed in the same context and without partitioning of the memory address space. Other systems support multi-threading, where each task executes within its own thread and has its own stack. The programming model is closely related to selected programming language, because it can affect the implementation of the operating system itself, and also determine which programming language will be used by developers when working with this particular operating system.

Based on these observations, we compare Tiny OS [3], Contiki [4], RIOT [5], and Linux [6], in terms of operating system for IoT design aspects described above. Of course, there are other operating systems designed for IoT, and among them with real time capabilities are Nano-RK [7] and Enix [8]. Even WindRiver moved towards IoT with its famous VxWorks [9], however it is a commercial (and expensive) solution. Luckily, there is an open source IoT real-time operating system called NuttX [10]. It is comparable to VxWorks and is very powerful. At the same time, in our case this is a drawback, since it is hardly applicable to the C0 class devices.

Contiki and Tiny OS are not real time systems, but their presence on the market is dominant and therefore they are used for reference. The same reason applies to Linux.

Contiki operating system has a layered architecture, while Tiny OS is built upon a monolithic kernel, as is the case with Linux. The Contiki system is event driven and is similar to that of TinyOS, which uses a FIFO strategy. Linux on the other hand, uses a scheduler that guarantees fair schedule for all tasks, even allowing preemption by timer. Only Linux with real-time extensions has a scheduler for work in real-time. Programming models with Contiki and TinyOS are defined by events in a way that all tasks are executed in the same context, although they offer a partial multithreading support. Contiki uses a programming language similar to C, but can't use certain keywords. TinyOS is written in the language called nesC, which is similar but not compatible with the C language. Linux, on the other hand, supports true multithreading, it is written in standard C, and it offers support for various programming languages. Compared to Linux, TinyOS and Contiki do not possess several functionalities that would be a great relief to developers, such as programming in standard C and C ++, the standard multithreading, as well as support for real-time (see Table 1) [14] .

TABLE 1: KEY CHARACTERISTICS OF TINYOS, CONTIKI, RIOT, AND LINUX. (✓) FULL SUPPORT, (○) PARTIAL SUPPORT, (x) NO SUPPORT.

OS	*min RAM*	*min ROM*	*C support*	*C++ support*
TinyOS	< 1kB	< 4kB	x	x
Contiki	< 2kB	< 30kB	○	x
RIOT	~ 1.5kB	~ 5kB	✓	✓
Linux	~ 1MB	~ 1MB	✓	✓

OS	*multi-threading*	*MCU w/o MMU*	*modularity*	*real-time*
Tiny OS	○	✓	x	x
Contiki	○	✓	○	○
RIOT	✓	✓	✓	✓
Linux	✓	x	○	○

TinyOS version 2.1 introduces TOSThreads, fully preemptable user-level application threads library [11], in which programmer can use C and nesC APIs, but outside of that still has to use only nesC.

A TinyOS has to be present in the form of source code or as a library during the compilation of user programs (static linking), providing a common binary program which is then programmed into device. This approach simplifies some things, such as better resource usage analysis, or more efficient optimization. On the other hand, changes in customer applications require a re-distribution of the entire operating system. Unlike TinyOS, Contiki has the ability to load individual applications or services during the execution of the operating system on the device, which resembles the mechanisms on general purpose computers.

On top of Contiki basic event-driven kernel other execution models can be used. Instead of the simple event handlers processes can use Protothreads [12]. Protothreads are simple forms of normal threads in a multi-threaded environment. Protothreads are stackless so they save their state information in the private memory of the process. Like the event handler Protothreads can not be preempted and run until the Protothread puts itself into a waiting state until it is scheduled again.

Along with event-driven kernel, Contiki also contains a preemptive multithreading library. It is statically linked with application program only if the program explicitly calls some of its functions. However, each thread from this library must have its own stack, which is not the case in Protothreads [13].

RIOT (Real-time operating system for IoT) fills the gap between operating systems of wireless sensor networks and traditional operating systems. In addition, this operating system is designed to take care about energy efficiency of the device, to occupy as less memory space as possible, and to have a unique API, regardless of the underlying hardware.

RIOT is based on a microkernel architecture, which is inherited from FireKernel [15], thereby supporting multithreading using a standard API. Packed with features inherited from FireKernel, RIOT also provides support for C ++, enabling the use of powerful libraries, such as Wiselib [16]. As standard, it includes support for TCP / IP stack network. This modular approach makes RIOT robust against failures of its individual components, providing high reliability with a programmer-friendly API.

RIOT allows programmers to create as many threads as they need. The only constraint is the amount of available memory and the size of the stack for each thread. Thanks to the kernel message API and using these threads, it is possible to implement distributed systems in a simple way.

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. Constant runtime of the scheduler is achieved by using a fixed-sized circular linked list of threads.

Low complexity of kernel functions is a main factor for the energy efficiency of an OS. The duration and occurrence of context switching is minimized. In RIOT context switching is performed in two cases: (1) a corresponding kernel operation itself is called, e.g., a mutex locking or creation of a new thread, or (2) an interrupt causes a thread switch. Fortunately, the first case will occur rarely, since in majority of applications every thread is created once. On the other hand, when RIOT's kernel gets called out of an interrupt service routine, saving the old thread's context is not required and thus a task switch can be performed in very few clock cycles.

IV. Platform implementation

On the high end in terms of hardware CPU and memory capacities, RIOT competes mainly with Linux. Compared to Linux, RIOT can scale down to orders of magnitude less memory requirements and supports built-in energy efficiency and real-time capabilities. On the low end in terms of hardware CPU/memory capacities, RIOT competes mainly with Contiki, TinyOS, and FreeRTOS [17]. Compared to Contiki and TinyOS, RIOT offers real-time capabilities and multi-threading. In contrast to FreeRTOS, RIOT provides native energy efficiency and a full-featured OS including up-to-date, free, open-source interoperable network stacks (e.g., 6LoWPAN), instead of just a kernel. RIOT also offers standard POSIX APIs and the ability to code in standard programming languages (C and C++) using standard debugging tools, thus reduces the learning curve of developers and the software development lifecycle process.

Default protocol integration in RIOT is mainly driven by latest IETF/IRTF activities [18]. Currently, RIOT supports basic networking protocols including 6LoWPAN, RPL, IPv6, TCP, UDP, CoAP, and provides CCNlite to experiment with content-centric networking.

The C++ capabilities of RIOT enable powerful libraries such as the Wiselib, which includes algorithms for routing, clustering, timesync, localization, and security.

There are a number of other topics in current development, such as dynamic linking support, Python interpreter, CBOR (an alternative for JSON), energy profiler, etc.

RIOT currently supports the following microprocessor families: x86, ARM7, ARM Cortex M0, M3, and M4, AVR/ATmega and MPS430, on number of different platforms, such as Intel Galileo, Betty, STM Discovery, Arduino Mega2560, TelosB, etc.

In addition, to overcome the issue of specialized hardware availability, RIOT can also be run as a native process on Linux and MacOS. This facilitates development because such a native process can be analyzed using readily available tools (e.g., gdb, Valgrind). A RIOT process is accessible via shell, the UART interface, or the virtual link layer interface (TAP).

V. Results and discussion

At the moment, we describe our results only for the first platform (see section II). As a proof of concept, our target hardware for RIOT operating system is the development board EasyMx PRO v7 for STM32 ARM manufactured by Mikroelektronika. It contains many on-board modules necessary for development variety of applications, including multimedia, Ethernet, USB, CAN and other, as well as mikroProg programmer and debugger. Board is delivered with MCU card containing STM32F407VGT6 [19]. The board is in many aspects very similar to the already supported STM32F4discovery.

However, porting of the original source code for M4 platform (`RIOT/cpu/stm32f4/`) was not so straightforward as it should be. The main reason was the MikroC compiler. Directory `RIOT/cpu/arm_common/` contains hardware specific, yet common routines for Cortex M processors. Among the others, file named `common.s` contains the following low level routines for interrupt handling and context switching: `dINT`, `eINT`, `ctx_switch`, `task_return`, `cpu_switch_context_exit`, and `arm_irq_handler`. All of them are written in assembly language, however porting them to MikroC compiler was not trivial task. MikroC does not allow separate files containing only assembler functions. Instead, assembly language routines should be embedded in some C file, and within C function, which often unnecessary generates a stack frame. Their assembler is quite restrictive in terms of syntax, and it is documented only with a few sentences in Help file. For example, interrupt handler should be wrapped with C function, which in its declaration tells compiler where is the vector in interrupt vector table for that particular interrupt.

In assembly routines callable from C, care should be taken of compiler optimization level in order to avoid different methods of parameter passing (stack or registers). The same applies when using microprocessor registers within assembly code.

Other specific code we had to write were device drivers for radio modules. Two different radio boards were used: 802.15.4 on all boards, plus 802.11 on data sink board.

Mikroelektronika BEE Click is an accessory board featuring 2.4 GHz IEEE 802.15.4 radio transceiver module MRF24J40MA (Microchip). This module includes an integrated PCB antenna and matching circuitry and is connected to the microcontroller via a SPI interface. Virtual functions from RIOT radio interface driver are implemented directly by using routines given in BEE click example for ARM [20].

WiFi PLUS Click features Microchip MRF24WB0MA (2.4 GHz, IEEE std. 802.11) compliant module as well as MCW1001 (Microchip) companion controller with on-board TCP/IP stack and 802.11 connection manager. WiFi PLUS click communicates with target board via UART interface.

Due to highly modular design of the operating system (microkernel), it was much easier to write device drivers for these radio modules.

During our preliminary test, eight nodes (including the sink) were in the same room, all within radio range with each other, resembling the full mesh network. Nodes are programmed as servers using socket API (part of network related POSIX wrapper of RIOT), i.e. the application threads are blocked in `accept()`, until the request for connection is received. Only one connection is possible at the time. There is no payload in incoming packet. Immediately upon establishing connection, the server reads integer value from ADC channel 3 and sends it back to the sink node. The sink node application is programmed as client, i.e. it polls all nodes by executing instruction `connect()` to the particular address/port and waiting for the response. All IP addresses are assigned manually.

The main contribution of this work is the concept of building a portable Internet of Things platform on an arbitrary hardware. We have made analysis of some existing solutions and chose an open-source operating system which we ported and compiled in environment that was not originally written for. Namely, original RIOT is written by means of the GCC toolchain, which has a very little in common with our development environment, including the C compiler itself. On the other side, unique C language development tools from Mikroelektronika are made to support broad range of very different platforms, such as ARM STM32, PIC, PIC32, dSPIC, AVR, Tiva and even 8051 series of microcontrollers on number of development boards. That is much larger hardware base than originally provided by RIOT based on the GCC toolchain.

VI. CONCLUSION

The idea behind our work is to build an uniform hardware/software IoT platform that can allow us to make smart objects by just connecting them and building an application for them. The chosen operating system for our platform (RIOT) is a real-time multi-threading operating system aiming to ease development across a wide range of IoT devices. Designed for energy-efficiency, reliability, real-time capabilities, small memory footprint, modularity, and uniform API access, RIOT provides several libraries such as Wiselib, as well as a full IPv6 stack for connecting constrained systems to the Internet. Unfortunately, porting RIOT to Mikroelektronika development board turned out to be not so simple, mainly because their development tools. For example, MikroC for ARM uses a proprietary encoded (compressed) library format. Mikroelektronika is secretive and protective of their libraries and compiler inner workings so we cannot use, or make conversion from external object files. On the other hand, their development and add-on boards are very well documented, including detailed schematic diagrams. Therefore, they remain in our attention. Nevertheless, we have managed to get system working and have performed some initial testing with encouraging results.

REFERENCES

[1] Internet of Things Strategic Research Roadmap. Available: http://www.grifs-project.eu/data/File/CERP-IoT%20SRA_IoT_v11.pdf.

[2] C. Bormann, M. Ersue, and A. Keranen, *Terminology for Constrained-Node Networks*, IETF, RFC 7228, May 2014.

[3] P. Levis, D. Gay, V. Handziski, J-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz, "T2: A second generation OS for embedded sensor networks," Telecommunication Networks Group, Technische Universität Berlin, Tech. Rep. TKN-05-007, Nov. 2005.

[4] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors", in Proc. of the 29th Annual IEEE International Conference on Local Computer Networks, Tampa, FL, USA, 2004. pp. 455–462.

[5] E. Baccelli, O. Hahm, M. Wählisch, M. Günes, T. C. Schmidt, "RIOT: One OS to Rule Them All in the IoT", INRIA, *Research Report*, No. RR--8176, Dec. 2012.

[6] D. Abbott, *Linux for Embedded and Real-time Applications*, Newnes, 2012.

[7] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks," in Proc. of the 26th IEEE Real-Time Systems Symposium (RTSS), Miami, FL, USA, 2005, pp. 256–265.

[8] Y-T. Chen, T-C. Chien, and P. H. Chou,"Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms," in Proc. of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys '10), Zürich, Switzerland, 2010, pp. 183–196.

[9] Wind River, "VxWorks: The Real-Time Operating System for the Internet of Things," Product Overview, 2014. Available: http://www.windriver.com/vxworks/reinvented/

[10] Available: http://www.nuttx.org/

[11] W. P. McCartney, and N. Sridhar, "Stackless preemptive multi-threading for TinyOS," in 2011 International Conference on Distributed Computing in Sensor Systems and Workshops, Barcelona, Spain, 2011, pp. 1–8,

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in Proc. of the 4th International Conference on Embedded Networked Sensor Systems, Boulder, Colorado, USA, pp. 29–42.

[13] T. Reusing, "Comparison of Operating Systems TinyOS and Contiki," Seminar SN SS2012, Network Architectures and Services, August 2012, pp. 7–13.

[14] O. Hahm, E. Baccelli, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," Proc. of the 32nd IEEE International Conference on Computer Communications (INFOCOM), Poster Session, April 2013.

[15] H. Will, K. Schleiser, and J. H. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios", in Proc. of IEEE Conference on Local Computer Networks (LCN), Zürich, Switzerland, 2009, pp. 834–841.

[16] T. Baumgartner, I. Chatzigiannakis, S. Fekete, C. Koninis, A. Kröller, and A. Pyrgelis, "Wiselib: A Generic Algorithm Library for Heterogeneous Sensor Networks," Wireless Sensor Networks, LNCS, vol. 5970, 2010, pp 162-177.

[17] http://www.freertos.org/

[18] A. Y. Ding, J. Korhonen, T. Savolainen, M. Kojo, J. Ott, S. Tarkoma, and J. Crowcroft, "Bridging the Gap Between Internet Standardization and Networking Research," *SIGCOMM Comput. Commun. Rev*., vol. 44, no. 1, pp. 56–62, Jan. 2014.

[19] http://www.mikroe.com/easymx-pro/stm32/

[20] http://www.libstock.com/projects/view/242/bee-click-example/