OpenWSN: A Standards-Based Low-Power Wireless Development Environment

8 AUTHORS, INCLUDING:

RESEARCH ARTICLE

OpenWSN: a standards-based low-power wireless development environment

Thomas Watteyne[1,2]*, Xavier Vilajosana[1,3], Branko Kerkez[4,1], Fabien Chraim[1], Kevin Weekly[1], Qin Wang[1,5], Steven Glaser[4] and Kris Pister[1]

[1] BSAC, University of California, Berkeley, CA, USA
[2] Dust Networks/Linear Technology, Hayward, CA, USA
[3] Universitat Oberta de Catalunya, Barcelona, Spain
[4] Civil and Environmental Engineering, University of California, Berkeley, CA, USA
[5] University of Science and Technology, Beijing, China

ABSTRACT

The OpenWSN project is an open-source implementation of a fully standards-based protocol stack for capillary networks, rooted in the new IEEE802.15.4e Time Synchronized Channel Hopping standard. IEEE802.15.4e, coupled with Internet of Things standards, such as 6LoWPAN, RPL and CoAP, enables ultra-low-power and highly reliable mesh networks, which are fully integrated into the Internet. The resulting protocol stack will be cornerstone to the upcoming machine-to-machine revolution.

This article gives an overview of the protocol stack, as well as key integration details and the platforms and tools developed around it. The pure-C OpenWSN stack was ported to four off-the-shelf platforms representative of hardware currently used, from older 16-bit microcontroller to state-of-the-art 32-bit Cortex-M architectures. The tools developed around the low-power mesh networks include visualisation and debugging software, a simulator to mimic OpenWSN networks on a PC, and the environment needed to connect those networks to the Internet.

Experimental results presented in this article include a network where motes operate at an average radio duty cycle well below 0.1% and an average current draw of 68 μA on off-the-shelf hardware. These ultra-low-power requirements enable a range of applications, with motes perpetually powered by micro-scavenging devices. OpenWSN is, to the best of our knowledge, the first open-source implementation of the IEEE802.15.4e standard. Copyright © 2012 John Wiley & Sons, Ltd.

*Correspondence

T. Watteyne, BSAC, 403 Cory Hall 1774, University of California Berkeley, CA 94720-1774, USA.

E-mail: watteyne@eecs.berkeley.edu

1. INTRODUCTION

The Internet of Things and machine-to-machine revolutions are quietly coming, and with them is an epochal turning point in the way people interact with the 'things' surrounding them: appliances in a smart home, snow-level sensors in a smart ski resort, overflow sensors in a smart refinery and so on. Standardisation bodies are playing a key role in this revolution. Different working groups are finalising the protocols running at different levels of this communication stack, and the stack depicted in Figure 2 is becoming the *de facto* protocol stack for tomorrow's capillary networks.

At the foundation of this protocol stack is the new IEEE802.15.4e [1] 'Time Synchronized Channel Hopping' standard, which achieves high reliability through frequency agility (channel hopping) and low power through tight time synchronisation. IEEE802.15.4e is the new medium access control (MAC) for the IEEE802.15.4 standard.

The OpenWSN project* offers a free and open-source implementation of this protocol stack and the surrounding debugging and integration tools, thereby contributing to the overall goal of promoting the use of low-power wireless mesh networks. It is, to the best of our knowledge, the first open-source implementation of the IEEE802.15.4e standard. The OpenWSN stack has been ported to four off-the-shelf platforms. It includes the ability to connect the network to the Internet Protocol version 6 (IPv6) Internet and to simulate a complete network on a PC.

---

*http://openwsn.berkeley.edu/

One of the goals of the OpenWSN project is to investigate the use of IEEE802.15.4e in Internet-connected low-power mesh networks. It shows how, contrary to common belief, IEEE802.15.4e (and, more generally, time synchronized channel hopping protocols) can be implemented on off-the-shelf platforms, without the need of dedicated hardware. This article presents implementation results of this protocol on four different platforms, using a range of 16- and 32-bit microcontrollers and radios. Moreover, OpenWSN is a 'pure C' implementation, that is, no extensions to the C language are needed. It is therefore not tied to any specific tool chain. The IEEE802.15.4e implementation runs in interrupt context for timing accuracy and is independent from the operating system running on the mote. This implementation can therefore easily be ported to other operating systems. Finally, it implements, on top of IEEE802.15.4e, Internet of Things standards such as IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN), Routing Protocol for Low Power and Lossy Networks (RPL) and Constrained Application Protocol (CoAP), enabling an OpenWSN network to connect seamlessly to the IPv6 Internet. Figure 1 depicts regular use cases of this protocol stack.

This article highlights its contributions and positions the OpenWSN project within related products and projects. Section 2 gives an overview of the protocols implemented and highlights their applicability in several use cases. Section 3 gives an overview of related products and (open-source) projects in the field of low-power wireless networks. Section 4 presents the hardware platforms used by the OpenWSN project, as well as the tools developed for debugging and Internet integration. Results of a performance evaluation of OpenWSN are presented in Section 5, with a particular focus on synchronisation and power consumption. Section 6 concludes the paper and presents the features to be included in future releases of OpenWSN.

2. PROTOCOL STACK AND USE CASES

Figure 2 depicts the protocol stack implemented in OpenWSN. This protocol stack is based entirely on Internet of Things standards. This section highlights the key aspects

of these protocols and indicates the type of use cases they enable.

2.1. Abstractions

The OpenWSN stack utilises abstraction at two levels. The Berkeley Socket Abstraction was developed as part of the Berkeley Software Distribution operating system development; it has been adopted by all operating systems, and it is at the heart of today's Internet. It considers that applications on two Internet hosts communicate through a socket, which is uniquely identified by the IP addresses of the hosts and the two ports corresponding to each application. The OpenWSN stack respects this abstraction, so that developing an application on top of the OpenWSN stack is very similar to developing an application on a regular Internet host.

The Hardware Abstraction consists in grouping all functions accessing the hardware (i.e. the functions that write to the registers) into a group of files called the 'board support package' (BSP). This allows the vast majority of the code to be shared among all platforms. There is one BSP per supported platform; the remainder of the stack code is shared between all implementations.

2.2. Inside the stack

The underlying radio technology of virtually all low-power wireless solutions is the IEEE802.15.4-2006 [2] standard, a 'double' standard defining both the physical layer (modulation, datarate, transmission power, etc.) and a standard MAC layer (how to arbitrate access to the wireless medium). This historical MAC protocol has suffered from two major flaws [3].

The first flaw is that a wireless device does not know a priori when its neighbors will transmit data, so it must leave its radio on to listen for incoming communications. Readily available, IEEE802.15.4 radios draw 5–30 mA of current while listening. If persistently kept on, such a radio drains a set of AA batteries in about a week. The second flaw of existing MAC layers relates to frequency diversity.
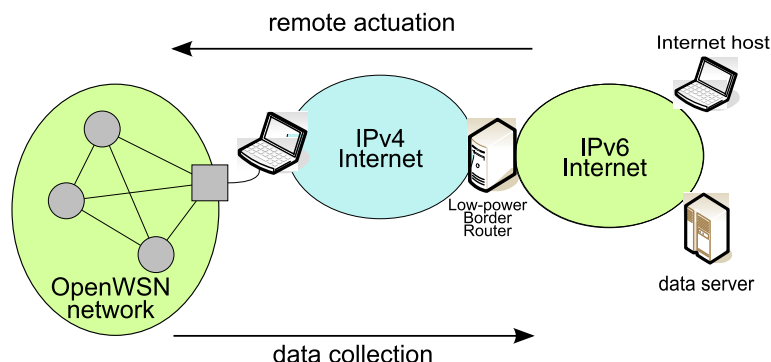
Figure 1. Use cases for an OpenWSN network. IPv4, Internet Protocol version 4; IPv6, Internet Protocol version 6.
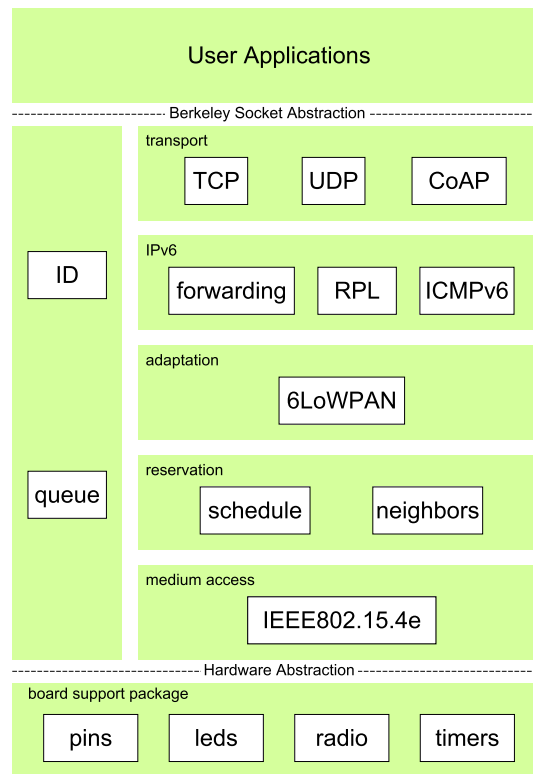
Figure 2. The OpenWSN protocol stack. Identification (ID), TCP, Transmission Control Protocol; UDP, User Datagram Protocol; CoAP, Constrained Application Protocol; IPv6, Internet Protocol version 6; RPL, Routing Protocol for Low Power and Lossy Networks; ICMPv6, Internet Control Message Protocol version 6; 6LowPAN, IPv6 Low-Power Wireless Personal Area Network.

Although IEEE802.15.4-compliant devices can switch frequencies, the historical MAC protocol specifies operating on only one frequency at a time. Unlike systems employing frequency diversity, the historical MAC is prone to external interference and multi-path fading. External interference is especially bothersome in the unlicensed 2.4-GHz band, shared with WiFi and Bluetooth, among others. Multi-path fading is the phenomenon whereby several 'echoes' of the same signal destructively interfere at the receiver. Realistic examples often occur indoors, where reflections from walls, furniture and people interact unpredictably, sometimes to the extent that a receiver cannot receive even strong signals sent from a nearby transmitter.

The IEEE802.15.4e standard replaces the historical MAC protocol, without changing the underlying physical layer. Thus, it can be implemented as a 'software update' in already existing IEEE802.15.4 devices. In an IEEE802.15.4e network, time is sliced up into time slots, and motes synchronise to each other. A superframe consists of a number of slots (typically tens to a few thousand slots) that repeats over time. A schedule indicates to a mote what action to take in each slot of the superframe: transmit, receive or sleep. Modifying this schedule allows for a clean trade-off between latency, network throughput and power consumption. Finally, each slot is assigned a channel offset, translating into a different frequency each time the

superframe repeats. While motes retain the same schedule, each (re)transmission takes place on a different frequency. This technique, known as 'channel hopping', is commonly used to combat external interference and multi-path fading.

Time Synchronized Channel Hopping has not been introduced by IEEE802.15.4e, as it is also the MAC technology underlying to Timing-Sync Protocol for Sensor Networks [4], Time Synchronized Mesh Protocol [5], Bluetooth and WirelessHART [6]. [7] reports experimental results from a 45-mote network deployed for 26 days. This network, running TSMP, yielded 99.999% end-to-end reliability and radio duty cycle well below 1% (i.e. motes have their radio on less than 1% of the time).

6LoWPAN [8] is an adaptation layer that compacts IPv6 headers to minimise the size of wirelessly transmitted packets. Frames exchanged in an IEEE802.15.4 network are, at most, 127-byte long; if the full 40 bytes were used, it would occupy almost a third of each packet. The 6LoWPAN specification consists of a set of rules for analysing the IPv6 header to be included in the packet. It removes the fields that are not needed (e.g. the version field, because it is always the same) and compresses other fields where feasible (e.g. the source and destination address, because parts of it may be inferred from the network's IPv6 prefix). All packets travelling inside the low-power mesh contain only the resulting 6LoWPAN header,

which can be as small as 2 bytes in the most favourable case. Because a full IPv6 header is required to support functionality on the Internet, an OpenWSN network implements a 'low-power border router' (LBR), a device that sits between the mesh and an Internet connection. The LBR inflates 6LoWPAN headers to normal IPv6 header on packets leaving the mesh and compacts the IPv6 headers on incoming packets. The result is that each mote can be assigned a unique IPv6 address and appear on the Internet as a regular Internet host. This permits for client-side applications to be developed easily, especially in cases where users may not have much previous knowledge about low-level WSN technologies.

RPL is used on top of 6LowPAN to maintain a routing topology. In RPL, both collection and source routing mechanisms are implemented. To collect information, a network gradient (named destination-oriented directed acyclic graph) is built. OpenWSN defines different metrics to establish that gradient being the inverse of the probability-of-delivery ratio used by default. Source routing (downstream) is maintained by the LBR nodes keeping a table with a route to each of the possible destinations in the network. This table is updated periodically by destination advertisement objects that are sent upstream by all the nodes in the network [9].

OpenWSN also supports CoAP [10], a protocol that enables RESTful interaction with individual motes, without the overhead of Transmission Control Protocol (TCP) and verbose nature of Hypertext Transfer Protocol (HTTP). It consists of a 4-byte header on top of the User Datagram Protocol (UDP). A CoAP-enabled mote acts both like a Web browser and a Web server.

2.3. Use cases

Figure 1 illustrates a typical use case and shows how an OpenWSN network connects to the Internet. Given the Berkeley Socket Abstraction, it is easy to implement an application on top of the OpenWSN protocol stack to communicate with clients over the Internet, sample sensors and actuate devices. Here, we present three typical use cases, which should cover a broad set of applications.

The most common use case is data collection. A mote is connected to a physical sensor, and an application runs on top of the OpenWSN stack to sample that sensor and initiate a transmission to the CoAP UDP port of a data server on the Internet. Sensor data are passed to the CoAP, which adds a header indicating which 'resource' these data come from. These payloads are prepended with UDP, 6LoWPAN and IEEE802.15.4e headers. The resulting frames are then scheduled for transmission using the IEEE802.15.4e Time Synchronized Channel Hopping MAC layer. When reaching the edge of the network, the packets are forwarded to the LBR, which inflates the 6LoWPAN into a full IPv6 header and transmits into the IPv6 Internet. A data server, which listens to the well-known CoAP UDP port, receives the data and stores them in a database, where the data can be displayed or processed.

A second use case is to send data from the Internet to an individual mote (e.g. a mote equipped with an actuating device to control a light fixture in a smart building). A host on the Internet has a client application that formats CoAP commands and sends them to the `coap://ipv6::addr/light/` resource of the mote. This packet travels over the IPv6 Internet, through the LBR, to the edge of the mesh and is then received by the application on the mote. The CoAP application then parses the command to execute any necessary local commands.

More complex client–server interactions are possible between a mote in the low-power mesh and a host on the Internet. An Internet host can query the mote for its available resources (i.e. the list of applications running on top of the CoAP) by querying its `well-known/` resource at `coap://ipv6::addr/well-known/`. This retrieves a list of available resources, which the client can individually query to obtain the latest sensor readings or to trigger an actuation event. It is also possible for a mote to browse available CoAP resources on the Internet. For example, an individual mote attached to a smart sprinkler can query the weather forecast of a CoAP-enabled weather server on the Internet to optimally irrigate a garden.

3. RELATED PRODUCTS AND PROJECTS

OpenWSN is part of an ecosystem of commercial products and open-source projects, which gravitate around the Internet of Things and machine-to-machine concepts. This section gives a list of the most relevant related products and projects, highlighting their similarities and differences with regard to OpenWSN.

3.1. Related commercial products

Our choice to build OpenWSN was inspired in part by the lack of commercially available platforms, which are simultaneously highly reliable and low cost while still consuming little power. Table I compares OpenWSN to three popular platforms.

A particular platform of interest is Digi's XBee® product line,[†] a module that allows an external microcontroller to send/receive wireless packets by controlling the radio over a simple serial interface. One variant of the modules implements the ZigBee low-power wireless stack, a standard almost ubiquitously adopted among most wireless chip manufacturers.[‡] Figure 3 shows the measured current profile of one such module operating as an end device (the lowest power configuration). Note that the module tested uses a power amplifier, significantly increasing both the

---

[†]http://www.digi.com/, presently priced at around $30

[‡]http://www.zigbee.org/

Table I. Platforms for low-power mesh networking.

Hardware	GINA [11] (OpenWSN)	XBee-PRO ZB S2	MICAz[a] (TinyOS)	SmartMesh IP[b]
Upper stack	CoAP 6LoWPAN RPL	ZigBee	CoAP 6LoWPAN RPL	6LoWPAN SmartMesh
Medium access	IEEE802.15.4e TSCH	IEEE802.15.4 CSMA/CA	IEEE802.15.4 CSMA/CA	IEEE802.15.4e TSCH
Sleep current RX current (sensitivity) TX current (power)	35 μA 11 mA[c] (−101 dBm) 13 mA (0 dBm)	3.5 μA 43 mA[c] (−102 dBm) 250 mA[c] (17 dBm)	15 μA 19.7 mA (−94 dBm) 17.4 mA (0 dBm)	1.2 μA 4.5 mA (−91 dBm) 5.4 mA (0 dBm)
Comments	Channel hopping, open source	Routers cannot sleep	Open source	Channel hopping, industrial

[a] Measured.
[b] http://www.memsic.com/.
[c] http://www.dustnetworks.com/.
GINA, Guidance and Inertial Navigation Assistant; IP, Internet Protocol; CoAP, Constrained Application Protocol; 6LowPAN, IPv6 over Low-Power Wireless Personal Area Network; RPL, Routing Protocol for Low Power and Lossy Networks; TSCH, Time Synchronized Channel Hopping; CSMA/CA, carrier sense multiple access with collision avoidance; RX, reception ; TX, transmission.
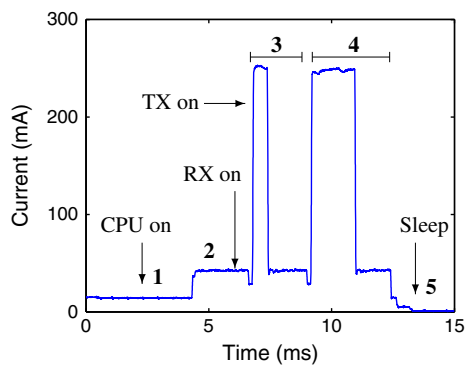
Figure 3. Current consumption trace for one wake cycle of a ZigBee end device (XBee-PRO S2) showing the different power states and steps of data transmission. TX, transmission; RX, reception.

output power and current consumption. This current profile was taken as the device collected one analog data sample and then transmitted it to a central coordinator.

Figure 3 shows the four different levels of power consumption of the XBee device: awake with radio off (approximately 10 mA), radio listening (approximately 40 mA), radio transmitting (approximately 250 mA) and asleep (<1 mA). Over the course of a sample transmission, there are five distinct phases:

(1) the device collects one analog data sample,
(2) the radio listens to verify that the wireless medium is free,
(3) the device sends a short message polling the coordinator for queued messages and then listens for the response,

(4) the radio sends the data sample and then listens for an acknowledgment, and
(5) the radio goes back to sleep.

Figure 3 indicates that each data sample consumes about 3 mJ of energy; therefore, a 2600-mAh alkaline AA primary battery can supply the collection and transmission of 4.7 million samples. A feature of ZigBee networks is that end devices can sleep an arbitrary length of time, consuming very little current while doing so. Thus, by adjusting how often samples are taken, the battery lifetime of an end device can vary in a week (sampling every 600 ms), to a year (sampling every 7 s), to 8 years (sampling every minute). However, a drawback, directly related to the fact that end devices can sleep arbitrarily long intervals, is that routers must always be listening for data from end devices. **A practical ZigBee deployment therefore requires that at least one non-battery-powered router node must be in range of every battery-powered end-device node**. This requirement is highly impractical for a number of potential deployments, such as remote sensing and environmental habitat monitoring. Devices on a ZigBee network also only operate on a single frequency channel and do not benefit from channel diversity. This makes them highly susceptible to external interference and multi-path fading. Lack of synchronisation also increases the risk of inter-network interference. This is especially true in dense network deployments, where nodes interfere with the transmissions of their neighbors because of the lack of an explicitly defined transmission schedule. All of these concerns are however addressed through OpenWSN's use of the IEEE802.15.4e MAC layer.

The IEEE802.15.4e standard synchronises all nodes in the network to within tens of microseconds. This allows two neighbor nodes wishing to talk to wake up at the

same time, thereby avoiding idle listening and significantly reducing the radio duty cycle and improving energy consumption. Router nodes know when their neighbors are scheduled to communicate and therefore do not need to listen all the time. The major benefit relates to achieving a long battery life for all nodes, including routing nodes, in a scenario where providing wired electricity to devices is not an options. This permits every node to run on battery power for extended periods while improving overall reliability.

3.2. Related open-source operating systems

TinyOS is an event-driven operating system for embedded devices developed at University of California Berkeley [12]. It is implemented using a component-oriented programming abstraction that provides code modularity and facilitates component reuse. This comes at the cost of a larger learning curve and code complexity. It features a non-preemptive scheduler and multiple abstractions, including communication interfaces and hardware timer virtualisation. TinyOS has been ported to numerous hardware platforms and, during the last 10 years, served as the main platform for new research development on communications protocols. It provides multiple MAC layer implementations for IEEE802.15.4 networks, including preamble sampling or low-power listening MAC [13]. One of the first well-known implementation of 6LoW-PAN and RPL was also developed on the TinyOS Berkeley Low-Power IP Stack [14]. Nowadays, Berkeley Low-Power IP Stack 2.0 provides a renovated implementation of 6LoWPAN including header compression, Dynamic Host Configuration Protocol for IPv6 for address assignment and RPL routing. CoAP is also part of the support package provided by the TinyOS core distribution, and it is based on the libcoap C library. The large community around TinyOS has created numerous add-ons providing a large amount of tools and functionalities. There is also a TinyOS simulator, TOSSIM, that enables the simulation of large networks using TinyOS native codes.

Contiki is an open-source operating system for WSNs and embedded devices developed at the Swedish Institute of Computer Science [15]. It is based on a multi-tasking non-preemptive scheduler that uses the protothreads abstraction [16]. The use of protothreads is similar to cooperative scheduling, including the caveat that an executing task may starve other waiting tasks. In addition to the operating system, Contiki includes several add-ons and libraries providing communication functionalities. The most relevant is the ContikiMAC [17], a carrier sense multiple access with collision avoidance preamble-sampling MAC using periodical wake-ups to listen for packet transmissions from neighbors. The μIPv6 library provides 6LoWPAN and RPL [18] routing functionality. The transport layer implements both UDP and a lightweight version of TCP [19]. Contiki also implements CoAP [20], similar to OpenWSN. Finally, the Contiki project develops

the Cooja [21] simulator, for simulating large Contiki networks on a PC.

4. OpenWSN PLATFORMS AND TOOLS

This section introduces the main platforms and tools that have been developed around the OpenWSN project.

4.1. Multiple hardware platforms

OpenWSN is curently ported to four, off-the-shelf hardware platforms, listed in Table II. This selection of platforms is intended to be a representative sample of the hardware that can readily be encountered today. TelosB is the oldest and lowest performance platform; the K20 platform is its high-end counterpart. Although TelosB is still very popular in the academic community, more powerful 32-bit platforms microcontrollers are becoming more and more commonplace. Note that all of the platforms presented in Table II use an external radio, communicating with the microcontroller using serial peripheral interface (SPI), a common serial interface.

4.2. Toolchains

Because OpenWSN is 'pure C', the source can be compiled with any toolchain compatible with the target platform. The choice of a toolchain is a complex decision, which trades mainly debugging functionality and resulting code size against cost.

Debugging on all hardware platforms is performed over a Joint Test Action Group interface, that is, it is possible to place breakpoints to freeze the code execution and inspect the value of variables and registers. We have used the MSP–FET430UIF debugger by Texas Instruments (Dallas, TX, USA) for the Guidance and Inertial Navigation Assistant (GINA) and TelosB, the IAR (Uppsala, Sweden)/Segger (Hilden, Germany) J-Link for the K20 and the built-in LPC-Link for the LPC platforms.

We have used IAR workbench[§] for MSP430 for the GINA and TelosB platforms. IAR workbench is currently one of the most commonly used integrated development environments for embedded systems.

We have used the LPCXpresso Integrated Development Environment, developed by Code Red[¶] for NXP (Eindhoven, The Netherlands), for the LPC platform. Its graphical user interface front end is based on Eclipse. The free edition has a target code size limit of 128kB, which is enough for the OpenWSN project.

For the K20, we support the CodeWarrior Integrated Development Environment[‖], developed by Freescale (Austin, TX, USA). Its graphical user interface front end is


[§]http://www.iar.com/.

[¶]http://www.code-red-tech.com/.

[‖]http://www.freescale.com/CodeWarrior/.

Table II. Platforms running OpenWSN.

	GINA	TelosB	LPC	K20
Toolchain	IAR	IAR	Code Red	CodeWarrior
Microcontroller				
Manufacturer	Texas Instruments	Texas Instruments	NXP	Freescale
Part number	MSP430f2618	MSP430f1611	LPC1769	K20DX256VLL7
Architecture	16-bit	16-bit	32-bit ARM Cortex M3	32-bit ARM Cortex M4
Maximum speed	16 MHz	8 MHz	120 MHz	72 MHz
Flash	116 kB	48 kB	512 kB	256 kB
RAM	8 kB	10 kB	64 kB	64 kB
Radio				
Manufacturer	Atmel	Texas Instruments	Atmel	Atmel
Part number	AT86RF231	CC2420	AT86RF231	AT86RF231
Interface	SPI	SPI	SPI	SPI
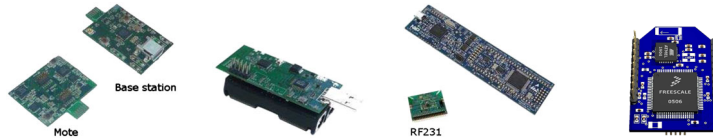
GINA, Guidance and Inertial Navigation Assistant; LPC, ; SPI, serial peripheral interface.

based on Eclipse. The free edition has a target code size limit of 64 kB, which is enough for the OpenWSN project.

For the OpenSim simulator (Section 4.6), the OpenWSN code is compiled to run on a standard PC. We have used the Visual Studio 2010 by Microsoft (Redmond, WA, USA) and the `gcc` compiler for Linux-based systems.

4.3. OpenOS: a simple scheduler

OpenOS is the kernel scheduler developed as part of the OpenWSN project. Hardware and timer interrupts order tasks on the basis of priority and push them onto a task list. As long as there are tasks in the list, the scheduler calls the callback associated with each task and removes (pops) the task from the list. When no more tasks are present, the scheduler switches the microcontroller to a deep sleep state, waiting for an interrupt to push a new task into the list. OpenOS is non-preemptive, that is, tasks do not interrupt one another. The OpenWSN stack is not directly tied to the OpenOS scheduler, and the stack can be run as part of a different operating system.

4.4. 6LoWPAN low-power border router

OpenWSN implements 6LoWPAN, a specification that allows individual motes to have a globally addressable IPv6 address without having to carry the full 40-byte IPv6 header in each short 127-byte IEEE802.15.4 frame. All the packets in the low-power mesh contain a 6LoWPAN header. To communicate with the IPv6 Internet, OpenWSN implements an LBR, which inflates 6LoWPAN headers into IPv6 headers for packets going from the low-power mesh into the Internet and compresses headers coming in. The LBR implementation is carried out in Python and runs on any Linux computer.

4.5. OpenVisualizer debug platform

The OpenVisualizer is a Python-based debugging and visu-alisation program that runs on a PC and interacts with the OpenWSN motes connected to it. It communicates with each connected mote over the serial port and displays relevant network information, such as showing the internal states of each mote on the network (connectivity, neighbor tables, and queue states), displaying the multi-hop connectivity graph, displaying low-level error/debug codes generated by the motes and interacting with the applications running on the mote. Written in Python, it is designed to be operating system-independent and can be set up to run on any computer supporting a serial interface. The software can also be used to a remote manager, by providing its IP address. OpenVisualizer is also used to facilitate IPv6 functionality, by allowing the user to connect to an LBR. Aside from providing a visualisation framework, OpenVisualizer is composed of a modularised Python framework, which can be used easily to write powerful client-side applications that interface with the network.

4.6. OpenSim PC-based simulator

As depicted in Figure 2, functions that interact directly with the hardware are grouped into a platform's BSP. There is one BSP per supported platform; the remainder of the code (the vast majority) is shared among all. OpenWSN comes with a special BSP, which emulates the behaviour of the hardware on a regular PC. That is, **it is possible to build the OpenWSN stack and applications, and emulate a full network on Windows or Linux**.

Running multiple emulated motes is carried out by connecting them to a simulation core that handles concurrency

between the emulated devices and the propagation of packets. This simulation framework is called OpenSim and is shown in Figure 4.

Each emulated mote (compiled C code) runs as a process on the host PC and communicates with the simulation core (written in Python) over a TCP session. When an OpenSim environment is started, the simulation core is initialised, and as many emulated mote processes are started as there are motes in the simulated network. When it boots, an emulated mote connects to the simulation core, which instantiates an object representing that mote's BSP. When the stack in the emulated mote calls a BSP function, this translates into a remote procedure call from the emulated mote to the simulation core, which executes the BSP function.

The simulation core and emulated motes execute code synchronously. That is, as long as the simulation core has not returned from the BSP call, the emulated mote does not continue executing code. This enables the simulation core to 'pause' execution of any emulated mote at any given time, and as a consequence, to coordinate concurrent execution between the different motes.

The simulation core is a discrete-event simulator: it contains a timeline, which consists of a number of events to happen in the future, and the code to execute at each. An event is typically the expiration of a hardware timer on an emulated mote. During a simulation, the execution of a BSP call causes more events to be pushed onto the timeline for execution in the future; the simulation core then consumes the events one after another.

When developing code in OpenWSN, the OpenSim environment is complementary to running code on real platforms. Each emulated BSP contains a model of the crystal used as a clock source, so that drift between motes can be modelled. Because of the architecture of OpenSim, it is possible to 'freeze' the execution of the whole network, at any time, including with the help of complex triggers. Given that all emulated motes are connected via TCP, it is also possible to run OpenSim in tandem with a real WSN, thus allowing real-world motes to communicate with virtual motes over the Internet.
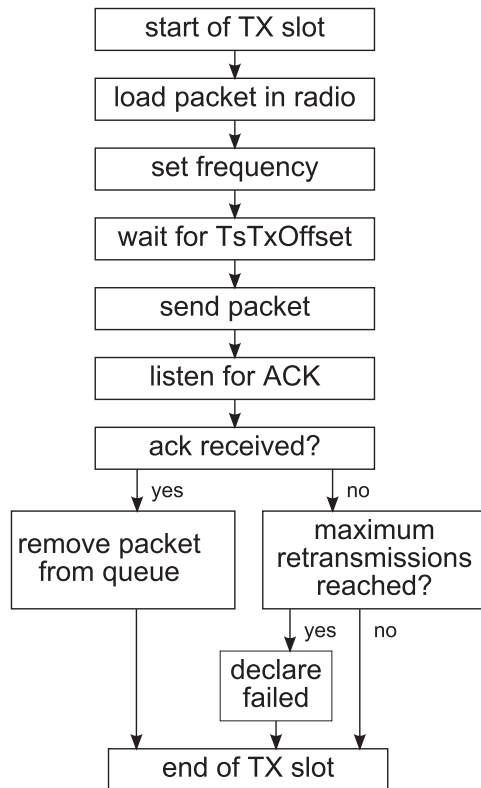
5. STACK EVALUATION
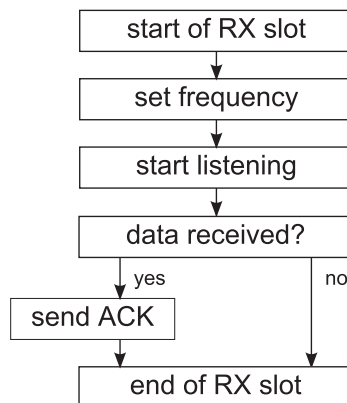
5.1. IEEE802.15.4e state machine

In an IEEE802.15.4e network, time is sliced up into slots. In each slot, the mote transmits, receives or sleeps. When
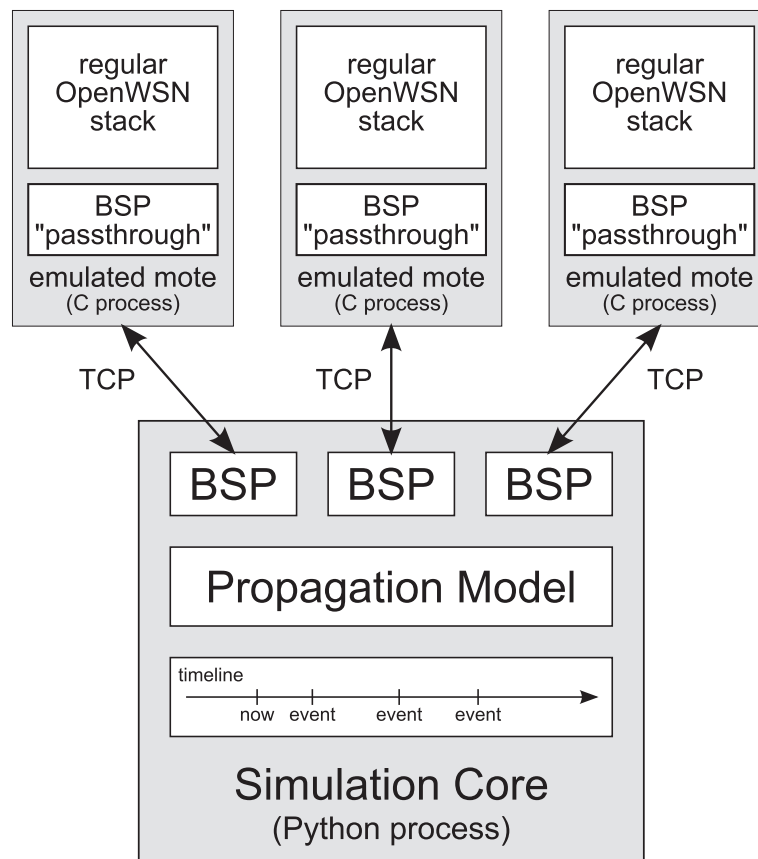
Figure 4. Architecture of the OpenSim simulator. BSP, board support package; TCP, Transmission Control Protocol.

Figure 5. Simplified state machine of a IEEE802.15.4e transmission (TX) slot. ACK, acknowledgement.

Figure 6. Simplified state machine of a IEEE802.15.4e reception (RX) slot. ACK, acknowledgement.

transmitting or receiving, it needs precise time when to transmit and listen for a packet to maintain synchronisation accuracy. Figures 5 and 6 present a simplified state machine** of a transmit and receive slots, respectively.

**As an online addition to this paper, the complete state machine is described at http://openwsn.berkeley.edu/wiki/TschFsm.

A transmitting mote has to send a data packet exactly `TsTxOffset` after the beginning of the slot. As described in Section 5.2, this is used for the receiving mote to be able to evaluate how out of sync it is from the transmitter. `TsTxOffset` is set to 4 ms in OpenWSN.

The slot needs to be long enough for the transmitter to be able to send the longest frame, 127 bytes, and receive an acknowledgment. All OpenWSN platforms feature a radio chip, which is separate from the microcontroller; those chips communicate with one another by using an SPI. The TelosB platform has the slowest SPI and takes 2.5 ms to transfer the 127 bytes of a packet from the microcontroller to the radio. To be compatible with this 'slow' platform, the slot duration for all platforms is set to 15 ms.

Within a single slot, the (full) transmit and receive state machine consists of nine states. At each state, the microcontroller has to perform atomic tasks, such as communicating with the radio and scheduling the expiration of a hardware timer. The associated code of these states executes in interrupts context on the microcontroller, without intervention from the scheduler.

5.2. Achieving synchronisation

Figure 7 shows a screen capture of a logic analyser connected to visualisation pins on a TelosB and a GINA board participating in the same network. The front part is a 'zoomed-in' version of the slot around time 0 in the back portion.

Three types of activity are depicted: the `radio` bar is present when the radio is on, either transmitting or receiving; the `task` and `isr` bars indicate when the microcontroller is executing code, in task and interrupt modes, respectively. Slots are indicated by alternating shading; each is 15 ms long.

The schedule the motes follow consists of nine slots: slots 0 and 1 are used for communicating, and slots 2 and 5–8 are used for serial communication. Each byte exchange over the serial port triggers an interrupt on the microcontroller, yielding a 'train' of `isr` activity.

The front part of Figure 7 is a zoomed-in version of a slot in which the TelosB mote sends a packet to the GINA mote. At the beginning of the slot, the motes are slightly desynchronised because of clock drift. the TelosB mote starts by loading the packet to send into its radio, which it will send `TsTxOffset` into the slot. The GINA mote starts listening 'guard time' early, to account for a possible drift. The GINA timestamps the instant it starts receiving the data packet and sees if it has received it a bit late. After reading the packet out of its radio buffer, it prepares the acknowledgment packet and indicates, in one of the fields of this packet, how late it received the packet. The GINA mote then sends the acknowledgment packet, which the TelosB receives. The TelosB then reads the time update field from the acknowledgment and applies it to its slot length. The end of the slot on the GINA and TelosB platforms happens synchronously: the motes have resynchronised.
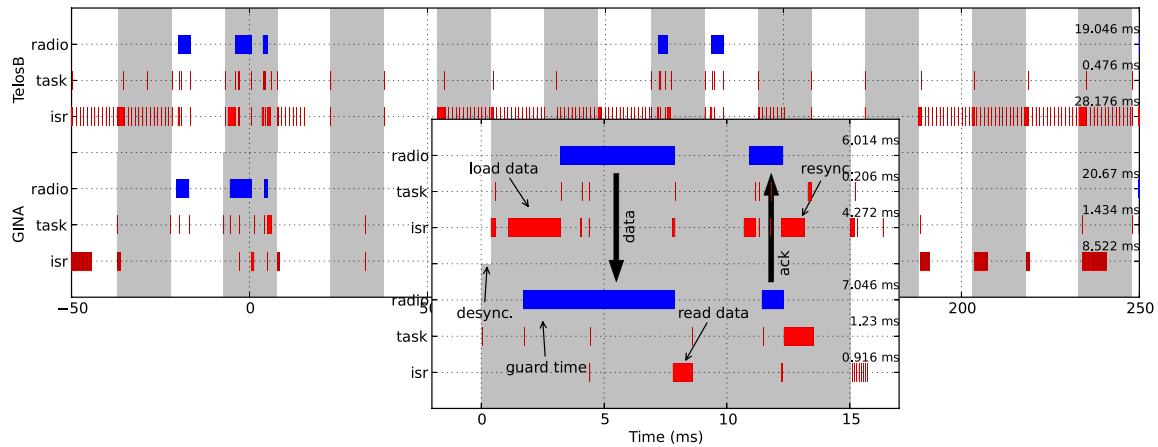
Figure 7. Witnessing a TelosB and a Guidance Inertial and Navigation Assistant mote resynchronise on a logic analyser.

Motes that are already part of the network use the first slot of their superframe to transmit advertisement (ADV) packets. These packets contain enough information to allow a new mote to synchronise to this newfound 'parent' and thereby 'join' the network. When a new mote is switched on, it leaves its receiver listening on a specific channel for ADV packets. When it receives this ADV packet, it aligns its superframe to that of the overall network and thus synchronises to this network. From that moment on, it follows a schedule and only turns its radio on in communicating slots.

Motes send ADV packets with a probability of $1/N$, with N its number of neighbors. This mitigates the probability of two ADV packets being sent simultaneously and colliding. Because the network uses channel hopping, subsequent ADVs are sent on different channels, and eventually, ADVs are sent on all available frequency channels. After turning its receiver on, it takes a mote at most $numChannels \cdot len(superframe)$ to join the network. If using a superframe of 101 slots, slots of 15 ms and communication on 16 channels, it takes a mote at most 24 s to synchronise to the network.

Once they have joined the network, motes need to keep synchronised. At the hardware level, motes use a crystal oscillator to keep track of time, the frequency of which changes slightly over manufacturing conditions, temperature and supply voltage. The result is that motes 'drifts' in time 1 with respect to another. A drift of 10 parts per million (ppm) is typical; that is, 1 s after synchronising, the time on two different motes may differ by up to 20 μs (if one mote is 10-ppm fast and the other 10-ppm slow). To allow for a slight de-synchronisation, motes start listening a bit early to their neighbor; this time buffer is called the 'guard time'.

Every time motes communicate, the receiver evaluates how offset it is from the sender by timestamping the reception of a packet and comparing that with the theoretical `TsTxOffset` (Section 5.1). It then either adjusts its clock or asks the sender to adjust its clock. This resynchronisation needs to happen periodically, because motes continuously drift with respect to each other; resynchronisation results in resetting the time offset between the sender and the receiver. All packets exchanged between two nodes , including data packets, are used to resynchronise. If the link between two motes is used to transmit data frequently, resynchronisation thus comes 'free'.

However, in low-throughput networks, there may occur prolonged periods of silence between two motes. This causes their clocks to offset too much, thus causing de-synchronisation. If this event occurs, the mote will have to attempt to join the network again in order to regain synchronisation. Avoiding this, and in the absence of data packets, consists of motes periodically transmitting *KeepAlive* packets to one another. These packets contain no payload and are used solely for synchronisation. The frequency of transmission of such packets depends on the motes' clock drift and the value of the mote's *guard time*.

For example, if a mote expects to receive a packet 4 ms into its slot, it turns its radio on 1 ms early and turns its radio off 1 ms after the 4-ms mark in case it has not yet started receiving a packet. Motes need to exchange (data or *KeepAlive*) packets often enough that they never drift outside of this 2-ms window. Equation (1) can be used to calculate the maximum resynchronisation period.

$$MaxResyncPeriod = \frac{guard\ time}{drift} \tag{1}$$

With a *guard time* of 1 ms and a 10-ppm clock drift, clocks on the two motes will drift outside of their guard time 50 s after having last communicated. Figure 8 shows a plot of the resynchronisation period (seconds between *KeepAlive* packets) as a function of the clock drift (ppm) assuming a 1-ms *guard time*. In the OpenWSN implementation, with a 10-ppm clock drift, if no data packets are transmitted for prolonged periods, *KeepAlive* packets are generated every 30 s to maintain synchronisation.
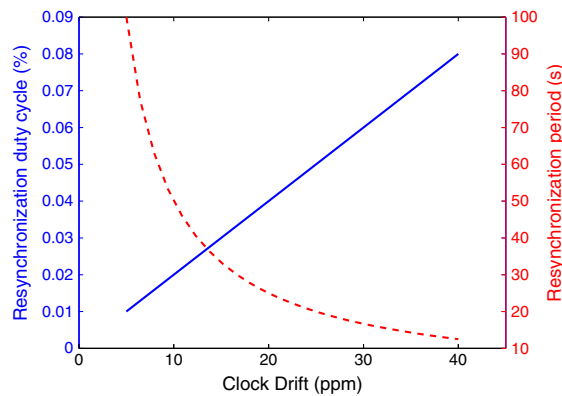
Figure 8. Maximum resynchronisation period (dashed line) and minimum achievable duty cycle (solid line) as functions of clock drift. Using higher quality, lower-drift clock sources reduce the need to resynchronise, thus saving energy through a lower resynchronisation duty cycle.

In the absence of other traffic, motes keep-alive to one another periodically to remain synchronised. This results in an incompressible radio duty cycle, that is, the minimum duty cycle an OpenWSN network can achieve. In an Open-WSN network, to exchange a keep-alive message and its acknowledgement, the transmitter and receiver have their radio on for 5 ms. Because this happens every 30 s, this results in a duty cycle of 5 ms/30 s = 0.02%.

Figure 8 depicts the resynchronisation duty cycle as a function of the clock drift. A mote equipped with a 30-ppm clock source will require a re-synchronisation duty cycle of 0.04%.

5.3. Code footprint

The code footprint is the amount of flash and RAM memory the OpenWSN system occupies. This includes the BSP, the stack and the default sample applications. Table III lists the footprints on the different platforms and indicates how much space is left for custom applications utilising the OpenWSN stack.

5.4. Power consumption

Figure 9 shows the current consumption during two slots for the four OpenWSN platforms, as read from an oscilloscope. A reception slot starts at time 0 ms. The second slot (15 ms later) is a transmission slot. Within a reception slot, the mote keeps the radio listening for the guard time. If nothing is received after the guard time, the radio is turned off. In a transmission slot, the packet is first loaded in the radio's transmit buffer; `TsTxOffset` into the slot, the radio transmits the packet. The radio is turned on at the end of the slot to receive the acknowledgement packet.

The LPC platform cannot be clocked exclusively from an external 32-kHz crystal and requires that its main clock tree remains on to keep an accurate sense of time. Running this clock tree on consumes a significant amount of power, which explains the offset in power consumption of this platform.

The remaining platforms can be clocked from an external crystal and therefore have a very low idle current. The GINA and K20 platforms use the same Atmel AT86RF231 (San Jose, CA, USA) radio chip and therefore consume roughly the same amount of energy (14 mA when listening and 17 mA when transmitting at 0 dBm). The TelosB platform uses the older Texas Instruments CC2420 radio, which consumes slightly more (19 mA receiving and 25 mA transmitting at 0 dBm).

Transmitted packets are of variable size, depending on whether they are (short) keep-alive packets or (long) CoAP messages. The time it takes to send the a packet is therefore variable, as shown in Figure 9.

Figure 10 shows the current consumption of a TelosB platform, as it executes the schedule shown in Figure 7. In particular, radio activity accounts for most of the current drawn; in slots 0 and 1, the mote is listening. The microcontroller wakes up for a short amount of time at each new slot, which explains the associated current draw. Finally, the 'train' of activity in slots 2 and 5–8 is due to the activity on the serial port.

The extremely low duty cycle achievable by IEEE802. 15.4e not only translates in prolonged lifetimes for battery-powered devices but also enables a new range of applications with motes running from energy scavenging power sources [22]. In [23], the authors powered

Table III. Code footprint of the OpenWSN stack and applications.

|	GINA	TelosB	LPC	K20
Toolchain	IAR	IAR	Code Red	CodeWarrior
OpenWSN footprint				
Flash	31 428 bytes	33 185 bytes	70 944 bytes	57 224 bytes
RAM	3 831 bytes	3 696 bytes	4 432 bytes	4 000 bytes
Available space				
Flash	87 356 bytes (74%)	15 967 bytes (32%)	453 344 bytes (86%)	204 920 bytes (78%)
RAM	4 361 bytes (53%)	6 544 bytes (64%)	61 104 bytes (93%)	61 536 bytes (94%)

GINA, Guidance Inertial and Navigation Assistant; LPC.
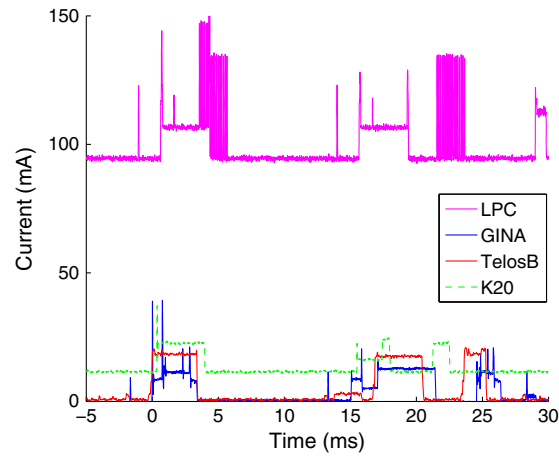
Figure 9. Current draw of the different OpenWSN platforms, as read from an oscilloscope. LPC, GINA, Guidance Inertial Navigation Assistant.
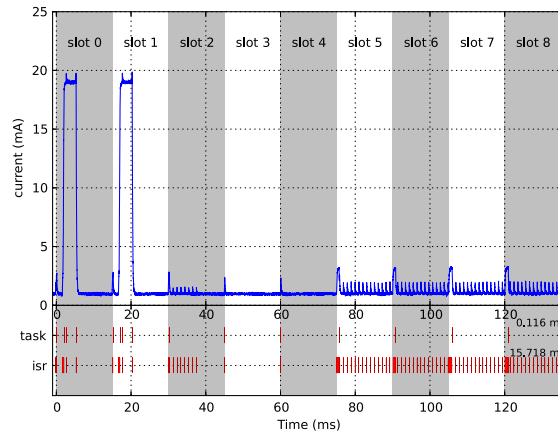
Figure 10. Current draw and CPU activity in a nine-slot frame on the TelosB platform. The mote is listening in slots 0 and 1, and is sending data over its serial port in slots 2, 5, 6, 7 and 8.
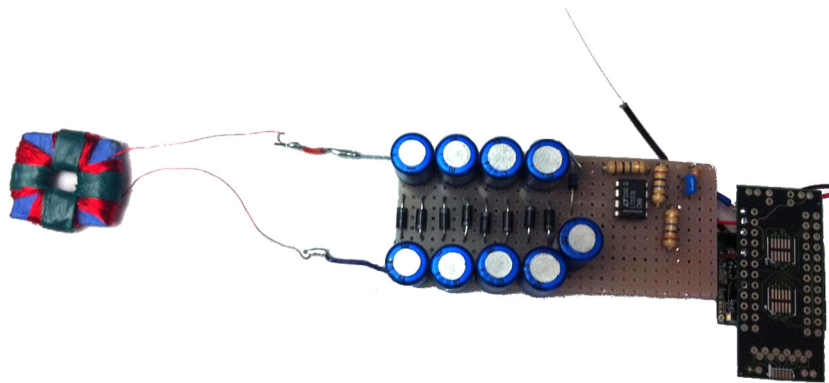
Figure 11. Guidance Inertial and Navigation Assistant platform connected to a contact-less (plug-through) power-line energy scavenger and consuming 68 μA average. It uses OpenWSN and Constrained Application Protocol to report current usage every 2 s to a server on the Internet.

GINA motes running the OpenWSN protocol stack from power-line energy scavengers. These scavengers, depicted in Figure 11, centre around a transformer that picks up the magnetic field emitted by a current-supplying AC line and converts it to a DC voltage supply, which powers the GINA mote. The scavenging device is placed around the primary prong of an appliance's electrical plug. When the appliance is turned on and draws 10 A (at 110 VAC and 60 Hz) or more through the line, the scavenger can supply the $68 - \mu A$ average current needed to operate the GINA mote. The GINA mote runs the OpenWSN stack depicted in Figure 2; each mote generates a measurement every 2 s, which it transmits to a CoAP-enabled data server on the Internet, as shown in Figure 1.

Future directions of OpenWSN aim at optimising the energy consumption through power control of nodes in the overall network [24]. This can be achieved by improving the routing metrics of RPL. Yet mitigating the hot-spot problem in well-connected networks through energy or load balancing techniques needs further exploration [25, 26].

6. CONCLUSION

OpenWSN is an open-source implementation of a fully standards-based protocol stack, with, as foundation, the new IEEE802.15.4e 'Time Synchronized Channel Hopping' standard. Because motes are synchronised, they can wake up only when they need to transmit or receive. And although motes need to periodically communicate to keep synchronised when the network is idle, this overhead is extremely small: about 0.02% radio duty cycle in an OpenWSN network.

On top of IEEE802.15.4e, OpenWSN implements Internet of Things-related standards such as 6LoWPAN (which makes each mote globally addressable on the Internet) and CoAP (which turns each mote into a Web server and a Web browser). The resulting protocol stack, combining ultra-low power, high reliability and Internet connectivity, will be key to the capillary and cellular machine-to-machine revolution [27, 28].

The protocol stack implementation is based entirely on C and can be built with any toolchain that supports the target platform. OpenWSN has been ported to four off-the-shelf platforms, as well as a PC port, which allows an OpenWSN network to be emulated on a computer. OpenWSN is, to the best of our knowledge, the first open-source implementation of the IEEE802.15.4e standard.

ACKNOWLEDGEMENT

REFERENCES

1. 802.15.4e-2012: IEEE Standard for Local and Metro politan Area Networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer 16 April 2012.

2. 802.15.4-2006: IEEE Standard for Information Technology, Local and Metropolitan Area Networks, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs) 2006.

3. Bachir A, Dohler M, Watteyne T, Leung K. MAC essentials for wireless sensor networks. *Communications Surveys Tutorials, IEEE Communications Surveys & Tutorials* 2010; **12**(2): 222–248.

4. Ganeriwal S, Kumar R, Srivastava MB. Timing-sync protocol for sensor networks. In *Conference on Embedded Networked Sensor Systems (SenSys)*. ACM: Los Angeles, California, USA, 2003; 138–149.

5. Pister KSJ, Doherty L. TSMP: time synchronized mesh protocol, In *IASTED International Symposium on Distributed Sensor Networks (DSN)*, Orlando, Florida, USA, 2008.

6. WirelessHART Specification 75: TDMA Data-Link Layer 2008. HCF_SPEC-75.

7. Doherty L, Lindsay W, Simon J. Channel-specific wireless sensor network path data. In *International Conference on Computer Communications and Networks*. IEEE: Turtle Bay Resort, Honolulu, Hawaii, USA, 2007; 89–94.

8. Hui J, Thubert P. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks September 2011.

9. Watteyne T, Molinaro A, Richichi MG, Dohler M. From MANET to IETF roll standardization: a paradigm shift in WSN routing protocols. *IEEE Communications Surveys and Tutorials* 2011; **13**(4): 688–707.

10. Shelby Z, Hartke K, Bormann C, Frank B. Constrained Application Protocol (CoAP) 12 March 2012.

11. Mehta A, Pister K. WARPWING: a complete open-source control platform for miniature robots. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* 2010: 5169–5174. DOI: 10.1109/IROS. 2010.5649382.

12. Hill J, Szewczyk R, Woo A, Hollar S, Culler D, Pister K. System architecture directions for networked sensors. *ACM SIGOPS Operating Systems Review* 2000; **34**(5): 93–104. DOI: 10.1145/384264.379006.

13. Ye W, Heidemann J, Estrin D. An energy-efficient MAC protocol for wireless sensor networks, In *Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, USA, 2002; 1567–1576.

14. Hui J, Culler D. Extending IP to low-power, wireless personal area networks. *Internet Computing, IEEE* 2008; **12**(4): 37–45. DOI: 10.1109/MIC.2008.79.

15. Dunkels A, Gronvall B, Voigt T. Contiki—a lightweight and flexible operating system for tiny networked sensors, In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, 2004; 455–462.

16. Dunkels A, Schmidt O, Voigt T, Ali M. Protothreads: simplifying event-driven programming of memory-constrained embedded systems, In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, 2006; 29–42.

17. Dunkels A. The ContikiMAC radio duty cycling protocol. *Technical Report T2011:13*, Swedish Institute of Computer Science Decembers 2011.

18. Ko J, Eriksson J, Tsiftes N, Dawson-Haggerty S, Terzis A, Dunkels A, Culler D. ContikiRPL and TinyRPL: happy together, In *Proceedings of the Workshop on Extending the Internet to Low Power and Lossy Networks (IPSN 2011)*, Chicago, IL, USA, 2011.

19. Braun T, Voigt T, Dunkels A. TCP support for sensor networks, In *IEEE/IFIP WONS 2007*, Obergurgl, Austria, 2007; 162–169.

20. Kovatsch M, Duquennoy S, Dunkels A. A low-power CoAP for Contiki, In *Proceedings of the IEEE Workshop on Internet of Things Technology and Architectures*, Valencia, Spain, 2011; 855–860.

21. Osterlind F, Dunkels A, Eriksson J, Finne N, Voigt T. Cross-level simulation in COOJA, In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo Session*, Delft, The Netherlands, 2007.

22. Alvarado U, Juanicorena A, Adin I, Sedano B, Gutierrez I, de No J. Energy harvesting technologies for low-power electronics. *Transactions on Emerging Telecommunications Technologies* 2012. DOI: 10.1002/ett.2529.

23. Stanislowski D. Energy harvesting for wireless power monitoring—implementing a low power, low data rate wireless network (OpenWSN), *Master's Thesis*, EECS Department, University of California, Berkeley June 2012.

24. Bravos G, Kanatas AG. Integrating power control with routing to satisfy energy and delay constraints in sensor networks. *European Transactions on Telecommunications* 2009; **20**(2): 233–245. DOI: 10.1002/ett.1248.

25. Ishmanov F, Malik AS, Kim SW. Energy consumption balancing (ECB) issues and mechanisms in wireless sensor networks (WSNs): a comprehensive overview. *European Transactions on Telecommunications* 2011; **22**(4): 151–167. DOI: 10.1002/ett.1466.

26. Vilajosana X, Llosa J, Pacho JC, Vilajosana I, Juan AA, Vicario JL, Morell A. Zero: probabilistic routing for deploy and forget wireless sensor networks. *Sensors* 2010; **10**(10): 8920–8937.

27. Accettura N, Palattella M, Dohler M, Grieco L, Boggia G. Standardized power-efficient & Internet-enabled communication stack for capillary M2M networks, In *IEEE Wireless Communications and Networking Conference*, Paris, France, 2012; 226–231.

28. Palattella M, Accettura N, Dohler M, Grieco L, Boggia G. Traffic aware scheduling algorithm for multi-hop IEEE 802.15.4e networks, In *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, Sydney, Australia, 2012.