2015 IEEE 18th International Conference on Computational Science and Engineering

Towards Out-of-the-Box Programming of Wireless Sensor-Actuator Networks

Gil Ferro Roberto Silva Luís Lopes
CRACS/INESC-TEC
University of Porto
Email: {gil.ferro,roberto.silva,luis.lopes}@dcc.fc.up.pt

***Abstract*—We address the problem of providing users, namely non specialists, with out-of-the-box, programmable, Wireless Sensor-Actuator Networks (WSN). The idea is that users get a package containing a gateway and an undetermined number of nodes, pre-configured to work as a self-organized wireless mesh. Each node comes with two pre-installed components: a small operating system and a virtual machine. The user can then use a simple, domain-specific, programming language to implement periodic tasks that are compiled into byte-code, and can be sent to the nodes for execution. At the nodes, the operating system manages a task table and schedules non-preemptive tasks for execution using the virtual machine. No subtle hardware or software configuration is required from the user as these details are abstracted away by the virtual machine. We developed a full specification for a data-layer that follows the aforementioned guidelines and implemented a complete prototype, integrated in our own Publish/Subscribe middleware called SONAR. In this paper we report the first results of using the prototype as compared to using the low level programming tools provided with the hardware. We measure a small increase in both resource consumption and processing overhead suggesting that this data-layer can be used effectively in WSN, even in cases where nodes have very limited hardware resources.**

I. Introduction

Programming wireless sensor-actuator networks (WSN) is a non-trivial task. The multitude of hardware configurations, highly dependent on the final application, operating systems and programming languages conspires to make WSN a task for the technically inclined or even the specialist [1]. This state of affairs makes the technology unappealing to the masses and therefore precludes its wider dissemination.

With SONAR [2] we have introduced an architecture that aims to solve some of these limitations and, specifically, to make setting up, configuring, programming and monitoring small to medium sized WSN deployments (up to tens of nodes) a task accessible to users with minimal background on computers. We do this by making several simplifying assumptions on the way WSN work, based on what we perceived from the literature [1], [3], [4] and deployment case studies: (a) in most deployments there are one or more gateways and an indeterminate number of nodes with sensors and actuators; (b) gateways are simple forwarders, they receive data from the nodes and forward it to a middleware; in some cases they may receive commands from the middleware and broadcast them to the nodes; (c) nodes read onboard sensors periodically, eventually sending data to the gateway; they may receive and execute commands from the gateway; (d) most applications require only basic local intelligence at the nodes, e.g., periodic checks of sensor readings and eventually of some stored state, to trigger actuation commands, and basic data processing.

The design of a data-layer, based on the aforementioned assumptions, that effectively shields the user from the hardware/software low-level details of the WSN deployments, is critical for the SONAR architecture. We established the following goals: (a) the data-layer must be readily available, i.e., pre-installed in the gateway and nodes, and be easily ported to other hardware/software architectures/configurations; (b) it should introduce only minimal overhead and resource consumption relative to traditional low-level programming of WSN; (c) nodes run periodic tasks written in a compact domain-specific language and compiled to byte-code; (d) the byte-code is radioed to the nodes where tasks are scheduled for execution in EDF-style with the assistance of a tiny operating system, i.e., nodes can be dynamically reprogrammed, and; (e) tasks are executed in a virtual machine in the nodes.

The main contributions of this paper are as follows: (a) the specification of a platform independent data-layer for WSN, including the modules for the gateway and for the nodes; (b) the specification of STL (Sonar Task Language), a domain-specific programming language for implementing periodic tasks, and a virtual machine to execute them in the nodes of WSN deployments, and; (c) a full prototype implementation of the data-layer and an evaluation of overhead and resource consumption relative to equivalent applications implemented with Arduino's C/C++/Wiring.

The remainder of the paper is organized as follows. Section II describes related work and the SONAR Publish/-Subscribe middleware, for managing datastreams from WSN deployments, and the data-layer in that context. Section III describes the STL language and its semantics. Section IV continues with the byte-code format, the compilation function and, the virtual machine. Section V introduces the software components that make up the gateway and the nodes. Sections VI and VII give an overview of the prototype implementation and report the results of our evaluation, in terms of overhead and resource consumption, with respect to applications implemented in Arduino's SDK. The paper ends with Section VIII where we discuss the approach taken, its virtues and limitations, as well as current work.

II. Related Work

SONAR [2] is a 3-layer publish-subscribe architecture (Figure 1) in which clients subscribe to data streams generated by tasks running on the nodes of WSN deployments.

978-1-4673-8297-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CSE.2015.20

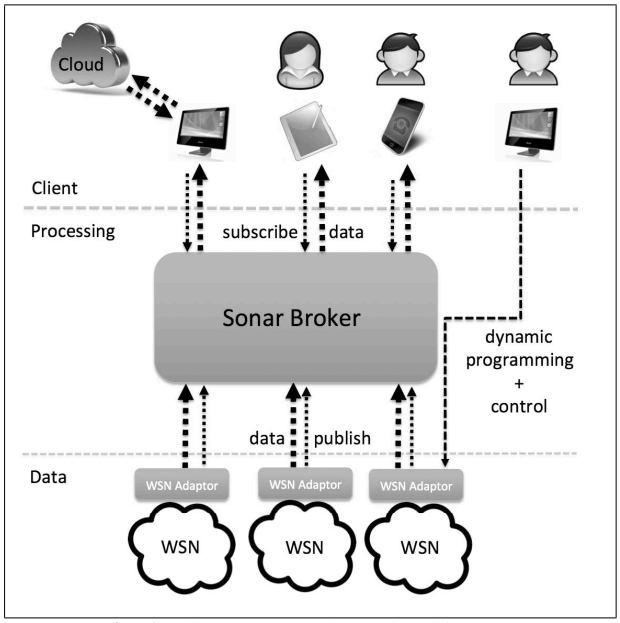

Fig. 1: The SONAR 3-layered architecture.

The data layer abstracts the WSN deployments and publishes data streams in the Sonar Broker. Each deployment is composed of a gateway and an indeterminate number of nodes. The gateway gathers data from the nodes and forwards it to the broker through a component called WSN Adaptor. Once the data reaches the broker, it is forwarded to all clients that subscribed the stream. The gateway also receives commands from an administration client and forwards them to the nodes. These commands allow for the dynamic reprogramming of WSN and include: sending a new task to the nodes, killing or changing the period of a task already running in the nodes, and reseting the nodes.

Sonar tasks are programmed in a domain specific programming language called Sonar Task Language (STL). The tasks are compiled into a byte-code representation that is sent to the nodes and executed *in loco* using the Sonar Virtual Machine (SVM). Multiple tasks, with different periods, can run in a WSN node. The management of memory and resources and the scheduling of the tasks in a node is performed by a tiny operating system, co-located with the virtual machine. Once running, a typical task may generate readings from the sensors in a node and send these periodically to the gateway, that in turn forwards the data to the broker and henceforward until it reaches the clients. Sonar clients are very simple, composed of shell commands that allow the selection of streams for subscription and the connection to the broker to receive the data, one connection per stream. The data received by the client is them sent to the standard output and can be composed (piped) with other programs developed by users to perform the desired processing and/or storage of the data.

As we stated in the previous section, the design, implementation and evaluation of the data layer is the main subject of this paper. Our goal is to provide a platform that: a) supports multiple node architectures; b) allows multiple tasks

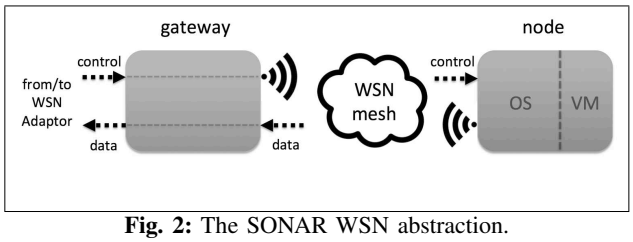

Fig. 2: The SONAR WSN abstraction.

to run on a single node; c) allows the dynamic reprogramming of nodes independent of whether or not OTA programming is available for a platform; d) does all of the above with limited overhead and resource consumption. Thus, to improve portability, we decided to implement two generic modules for nodes: an operating system to manage tasks, and a virtual machine to execute tasks (Figure 2). The two modules would come pre-installed in the nodes and be easily ported to multiple architectures. To simplify the programming of the nodes and to allows for dynamic reprogramming, we adopted a model based on tiny tasks, several of which can run in a node at a given time. Tasks are written in a small domain-specific language, compiled into byte-code and executed by the virtual machine. They are sent to the nodes as control messages from the gateway. In our programming model, tasks never listen for data and they can only send data to the gateway. The operating system schedules the tasks in EDF-style, but otherwise no atempt is made to ensure that deadlines are met. In other words, we assume that the number of tasks in a node is small and that the execution time is a small fraction of the period. Both the operating system and the virtual machine were designed and implemented to minimize extra resource consumption relative to the baseline set by native code implementations. The programming is, however, far simpler with STL, as we shall see in the next section.

Using virtual machines to support programming models for WSN is not a novelty. Mate [5] is a compact virtual machine implemented on top of TinyOS. Programs, called capsules, may be injected in the network at any time to perform specific tasks. They are written in a very simple assembly language and have the capability to move between sensor nodes. The Regiment [6] macro-programming language implements the Distributed Token Machine, based on a event-based programming model. Each token is a typed message with some data or code that triggers a specific handler upon reception. Sun Microsystems (now Oracle) introduced the Squawk [7] virtual machine to support applications for their SunSPOT devices. Squawk is a very compact Java virtual machine, with a simplified bytecode layout that runs without an underlying operating system. Our goal is to develop a compact virtual machine, in the line of Mate, but associated with a user friendly programming language. Unlike Mate, however, we are not interested in the mobility aspects of tasks.

In what concerns operating systems for WSN, TinyOS [8] is perhaps the most widespread. It provides a simple event-based execution-model with non-preemptive tasks. The system is loaded onto the sensor nodes as a set of modules linked with the user application. Contiki [9] is also based in a event-driven execution-model but supports multi-threaded applications, using very lightweight threads, and the dynamic

$$
\begin{array}{lll}
T & ::= & \textbf{sensors}\ \{s_1 : \sigma_1 \ldots s_n : \sigma_n\} \quad \textit{Tasks} \\
 & & \textbf{actuators}\ \{a_1 : \sigma_1 \ldots a_m : \sigma_m\} \\
 & & \textbf{init}\ \{\tilde{q}\} \\
 & & [\tilde{\tau}]\ \textbf{loop}\ \{\tilde{r}\} \\
\sigma & ::= & \tilde{\tau} \mapsto \tau \quad \textit{Types} \\
\tau & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{float} \mid \textbf{void} \\
q & ::= & \tau\ x = v \quad \textit{Initializations} \\
r & ::= & x = e \quad \textit{Instructions} \\
 & \mid & a(\tilde{e}) \\
 & \mid & \textbf{radio}\ [\tilde{e}] \\
 & \mid & \textbf{if}\ e\ \{\tilde{r}\}\ \textbf{else}\ \{\tilde{r}\} \\
 & \mid & \textbf{while}\ e\ \{\tilde{r}\} \\
e & ::= & s(\tilde{e}) \mid e\ op\ e \mid op\ e \mid (e) \mid v \quad \textit{Expressions} \\
v & ::= & x \mid u \quad \textit{Values} \\
u & ::= & \textit{bools} \mid \textit{ints} \mid \textit{floats} \quad \textit{Constants}
\end{array}
$$

Fig. 3: The syntax of STL.

loading of program modules. SOS [10], also event-driven, is built from very small modules these are dynamically loaded, using a clever memory management scheme. MANTIS [11] and Nano-RK [12] diverge from the above systems in that they support preemptive multithreading, required for real-time systems. Our operating system is much simpler than any of the aforementioned siblings. Memory management for tasks is very simple as memory is all allocated statically. The operating system just manages a table of tasks, schedules tasks for execution, and processes incoming commands from the gateway, including new tasks to be executed.

III. Programming Language

In this section we describe the syntax and semantics of the domain-specific programming language used to implement periodic tasks - the SONAR Task Language (STL).

A. Syntax

The syntax for tasks is described in Figure 3. A task T uses two sets of identifiers, $\tilde{s}$ and $\tilde{a}$, to specify the available sensors and actuators in a given platform. The notation $\tilde{\alpha}$ is used to denote a sequence of pairwise distinct elements, α, of a given syntatic cathegory. Each of these identifiers maps to a unique sensor or actuator in the hardware. This declaration is thus similar for all tasks running on the same hardware configuration and in a more concrete syntax would simply be included by the programmer using a compiler directive.

The code that is actually specific for the task starts with the **init** block, used to initialize global task variables. This code is not executed, rather the compiler will copy the initial values for each variable directly to the data segment of the bytecode generated for the program. The **loop** block, on the other hand, is the code executed for every (periodic) activation of the task. It is immediately preceded by the type of message sent back by the task to the gateway using the construct $[\tilde{\tau}]$. The task only sends messages of this type to the gateway and the type is checked against all **radio** statements in the task. The instructions available to the programmer include: assignment, actuation - $a(\tilde{e})$, sending a set of evaluated expressions to the gateway - **radio** $[\tilde{e}]$, a conditional execution construct - **if** e $\{\tilde{r}\}$ **else** $\{\tilde{r}\}$, and a while loop - **while** e $\{\tilde{r}\}$. The expressions are standard except for $s(\tilde{e})$ that is used to read a value from a given sensor.

As we said, for a given platform and configuration, the hardware description provided by the constructs **sensors** and **actuators** is the same. We use a preprocessing directive - **use** - to include this description at the top of all programming examples in this paper (Figure 4).

```
sensors {
  temperature: void -> float,
  humidity   : void -> float
  light      : void -> float
}
actuators {
  led        : bool -> void
}
```

Fig. 4: Hardware description for Arduino 2560 prototype WSN - file "ard2560.hw".

The example in Figure 5 show a STL program that at each activation reads the temperature and humidity and radios the values to the gateway. The example uses two sensors, designated as temperature and humidity, whose types are declared in hardware description file "ard2560.hw". Notice that the periodicity of the task is not included in the code. It is an external attribute set with the administration client when the task is sent to the gateway to be radioed to the nodes. In this way, users with admin access can dinamically change the period of running tasks using simple control messages. Note that, for compactness, some of the examples in this paper use a slightly sugared version of the syntax, e.g., allowing variables to be declared in the loop and initialized with expressions, not just constants.

```
use "ard2560.hw"

[float, float] loop {
  float t = temperature();
  float h = humidity();
  radio [t,h];
}
```

Fig. 5: STL program that reads the temperature and humidity and radio the results to the gateway.

The language specification is complete with both the operational and static semantics that together define how well-formed programs are executed.

IV. Compiler and Virtual Machine

In this section we give the specification for the SONAR Virtual Machine (SVM), one of the modules pre-installed in the nodes. The virtual machine executes STL tasks, translated into byte-code by a compiler. Thus, we begin by defining the byte-code format and then give the translation function for the STL source code.

$$
\begin{array}{lll}
p & ::= & h\ d\ b & \textit{Program} \\
h & ::= & i_1\ i_2 & \textit{Header} \\
d & ::= & \tilde{v} & \textit{Data Segment} \\
v & ::= & \textit{bools} \mid \textit{ints} \mid \textit{floats} & \textit{Values} \\
b & ::= & \tilde{r} & \textit{Text Segment} \\
r & ::= & \textbf{ld}\ i \mid \textbf{st}\ i \mid \textbf{wrt}\ i_1\ i_2 \mid \textbf{rd}\ i_1\ i_2 & \textit{Instructions} \\
 & & \mid \textbf{rad}\ i \mid \textbf{bf}\ i \mid \textbf{jp}\ i \mid \textbf{ret} & \\
 & & \mid \textit{bop} \mid \textit{uop} &
\end{array}
$$

Fig. 6: Byte-code syntax.

The byte-code is composed of 4 segments: header, data, stack and text (Figure 6). The header contains the total size of the bytecode as well as the offset to the beginning of the text segment. The stack segment is allocated between the data and text segment, growing towards the lower addresses. Its size is calculated at compile time since there are no calls to user defined functions. The data segment provides space for all the variables in a STL program. Constants and the initial values of global variables are stored there by the compiler. The data segment can be seen as the only activation record required for the virtual machine since, again, there are no calls to user functions or user functions in tasks. All variables, of types **bool**, **int** and **float**, use 4 bytes in the data segment in this version, but this can and should be optimized to minimize the size of the byte-code. The text segment is composed of instructions that have a 1 byte opcode and eventually 1 or 2 extra bytes for arguments. There are instructions for loading a value to the stack (**ld**), storing a value from the stack (**st**), sending an actuation command (**wrt**), reading a sensor (**rd**), sending a message over the radio (**rad**), the usual control flow (**bf**, **jp**, **ret**) and, the usual integer and floating-point arithmetic and logic and relational operators (**bop**, **uop**). Bytecode instructions map almost one-to-one with reduction rules from the operational semantics. This correspondance is important for proving that the virtual machine correctly executes the bytecode, but this is a problem we will not address here.

The translation function receives a syntactic term and returns a pair of sequences (D, B) (Figures 7 and 8). The first, D, is the contribution of the term to the data segment, the latter, B, is the contribution to the text segment. The top level translation function $[\![\cdot]\!]$, for STL tasks, breaks the translation into a sequence of pairwise concatenations (operator ":") and uses appropriate translation functions for each syntactic category. We use the same notation $[\![\cdot]\!]$ to simplify the notation, but these should be seen as distinct functions. The translation function uses 3 sets which hold integer identifiers for sensors and actuators, S and A, and data segment offsets for variables (set Var) and constants (set Const), V and U, defined as follows:

$$
\begin{aligned}
S &= \{(s_i, i) \mid s_i \in \widetilde{s : \tau}\} \\
A &= \{(a_i, i) \mid a_i \in \widetilde{a : \tau}\} \\
V &= \{(x, i) \mid x \in \mathrm{Var} \wedge i = \mathrm{offset}(x)\} \\
U &= \{(u, i) \mid u \in \mathrm{Const} \wedge i = \mathrm{offset}(u)\}
\end{aligned}
$$

The translation function is quite straightforward. The translation of an actuation command, $a(\tilde{e})$, is simply the translation

$$
\begin{aligned}
[\![T]\!] =& [\![\textbf{sensors}\ \{\widetilde{s : \sigma}\}]\!] : \\
& [\![\textbf{actuators}\ \{\widetilde{a : \sigma}\}]\!] : \\
& [\![\textbf{init}\ \{\tilde{q}\}]\!] : \\
& [\![[\tilde{\tau}]\ \textbf{loop}\ \{\tilde{r}\}]\!] : \\
& (\epsilon, \textbf{ret}) \\
[\![\textbf{sensors}\ \{\widetilde{s : \sigma}\}]\!] =& (\epsilon, \epsilon) \\
[\![\textbf{actuators}\ \{\widetilde{a : \sigma}\}]\!] =& (\epsilon, \epsilon) \\
[\![\textbf{init}\ \{\tilde{q}\}]\!] =& [\![\tilde{q}]\!] \\
[\![\tau\ x = v\ \tilde{q}]\!] =& (v, \epsilon) : [\![\tilde{q}]\!] \\
[\![[\tilde{\tau}]\ \textbf{loop}\ \{\tilde{r}\}]\!] =& [\![\tilde{r}]\!] \\
[\![r\ \tilde{r}]\!] =& [\![r]\!] : [\![\tilde{r}]\!] \\
[\![x = e]\!] =& [\![e]\!] : (\epsilon, \textbf{st} : V(x)) \\
[\![a(\tilde{e})]\!] =& [\![\tilde{e}]\!] : (\epsilon, \textbf{wrt} : A(a) : |\tilde{e}|) \\
[\![\textbf{radio}\ [\tilde{e}]]\!] =& [\![\tilde{e}]\!] : (\epsilon, \textbf{rad} : |\tilde{e}|) \\
[\![\textbf{if}\ e\ \{\tilde{r}_1\}\ \textbf{else}\ \{\tilde{r}_2\}]\!] =& [\![e]\!] : (D', B') \\
& \text{where} \\
& (D_1, B_1) = [\![\tilde{r}_1]\!] \\
& (D_2, B_2) = [\![\tilde{r}_2]\!] \\
& D' = D_1 : D_2 \\
& j_1 = 2 + |B_1| \\
& j_2 = |B_2| \\
& B' = \textbf{bf} : j_1 : B_1 : \textbf{jp} : j_2 : B_2 \\
[\![\textbf{while}\ e\ \{\tilde{r}\}]\!] =& [\![e]\!] : (D, B') \\
& \text{where} \\
& (D, B) = [\![\tilde{r}]\!] \\
& j = 2 + |B| \\
& B' = \textbf{bf} : j : B : \textbf{jp} : -j - 2 \\
[\![\epsilon]\!] =& (\epsilon, \epsilon)
\end{aligned}
$$

Fig. 7: Translation to bytecode (part I).

of the arguments $\tilde{e}$, followed by a **wrt** instruction with the integer identifier for the actuator $A(a)$ and the number of expressions, $|\tilde{e}|$, as the arguments. Similarly, reading a sensor, $s(\tilde{e})$, translates into the translation of the expressions followed by a **rd** instruction with the integer identifier for the sensor $S(s)$ and the number of expressions, $|\tilde{e}|$, as the arguments. Likewise, the translation for **radio** $[\tilde{e}]$ is simply the translation of the expressions to be sent, followed by a **rad** instruction with the number of expressions, $|\tilde{e}|$, as the argument.

The state of the virtual machine is represented as the term $[D|S|B]_j$, where j is the program counter and is used to travel the instructions in the text segment. The halted machine is represented by a special state denoted $\perp$. To run a task T in the virtual machine we use the translation function to get its byte code $[\![T]\!] = (D, B)$ and set its initial state to:

$$[D|\underbrace{0 \ldots 0}_{k}|B]_0$$

where k is the maximum stack size computed by the compiler and included in the bytecode. The computation proceeds according to the reduction rules presented in Figure 9, of the

$$
\begin{aligned}
[\![e_1, \ldots, e_n]\!] &= [\![e_1]\!] : \cdots : [\![e_n]\!] \\
[\![s(\tilde{e})]\!] &= [\![\tilde{e}]\!] : (\epsilon, \textbf{rd} : S(s) : |\tilde{e}|) \\
[\![e_1 \ \textbf{bop} \ e_2]\!] &= [\![e_1]\!] : [\![e_2]\!] : (\epsilon, \textbf{bop}) \\
[\![\textbf{uop} \ e]\!] &= [\![e]\!] : (\epsilon, \textbf{uop}) \\
[\![x]\!] &= (\epsilon, \textbf{ld} : V(x)) \\
[\![u]\!] &= (u, \textbf{ld} : U(u)) \\
[\![\epsilon]\!] &= (\epsilon, \epsilon)
\end{aligned}
$$

Fig. 8: Translation to bytecode (part II).

$$\frac{B[j] = \textbf{ld} \quad B[j+1] = i \quad v \leftarrow D[i]}{[D|S|B]_j \rightarrow [D|v, S|B]_{j+2}}$$

$$\frac{B[j] = \textbf{st} \quad B[j+1] = i \quad D' \leftarrow D + \{i : v\}}{[D|v, S|B]_j \rightarrow [D'|S|B]_{j+2}}$$

$$\frac{\begin{array}{c} B[j] = \textbf{rd} \quad B[j+1] = i \quad B[j+2] = n \\ f \leftarrow sensors[i] \\ v \leftarrow f(v_1, \ldots, v_n) \end{array}}{[D|v_1, \ldots, v_n, S|B]_j \rightarrow [D|v, S|B]_{j+3}}$$

$$\frac{\begin{array}{c} B[j] = \textbf{wrt} \quad B[j+1] = i \quad B[j+2] = n \\ g \leftarrow actuators[i] \\ g(v_1, \ldots, v_n) \end{array}}{[D|v_1, \ldots, v_n, S|B]_j \rightarrow [D|S|B]_{j+3}}$$

$$\frac{B[j] = \textbf{rad} \quad B[j+1] = n \quad send(v_1, \ldots, v_n)}{[D|v_1, \ldots, v_n, S|B]_j \rightarrow [D|S|B]_{j+2}}$$

$$\frac{B[j] = \textbf{bf} \quad B[j+1] = i}{[D|\mathsf{false}, S|B]_j \rightarrow [D|S|B]_{j+2+i}}$$

$$\frac{B[j] = \textbf{bf} \quad B[j+1] = i}{[D|\mathsf{true}, S|B]_j \rightarrow [D|S|B]_{j+2}}$$

$$\frac{B[j] = \textbf{jp} \quad B[j+1] = i}{[D|S|B]_j \rightarrow [D|S|B]_{j+2+i}}$$

$$\frac{B[j] = bop}{[D|v_2, v_1, S|B]_j \rightarrow [D|v_1 \ bop \ v_2, S|B]_{j+1}}$$

$$\frac{B[j] = uop}{[D|v, S|B]_j \rightarrow [D|uop \ v, S|B]_{j+1}}$$

$$\frac{B[j] = \textbf{ret}}{[D|S|B]_j \rightarrow \bot}$$

Fig. 9: Transition rules for SVM.

form:

$$\frac{c_1 \ \ldots \ c_n}{S \rightarrow S'}$$

where the c_i are preconditions or actions that must be fulfilled to make the transition from the current state, S, to a given state, S', possible. For example, when the current instruction (the one the program counter j is indexing) is **ld** (1st rule), the next byte contains i, the offset of a variable or a constant in the data segment, and we use it to access the value (denoted as $v \leftarrow D[i]$). The new state has the same data and text segments but the stack has the value v on top of it, and the program counter was updated to $j + 2$. Similarly for **st** (2nd rule), we have a value v in the stack in the current state and we make a transition to a state where that value has been removed from the stack and copied to position i in the data segment (denoted as $D' = D + \{i : v\}$). The arrays *sensors* (3rd rule) and *actuators* (4th rule) provide the pointers to built-in functions associated with the identifiers. For example, the **rd** instruction (3rd rule) has two arguments: i, the index that identifies the built-in sensor function to be called, and n, the number of arguments that function takes. The latter, $v_1 \ldots v_n$, are all stored at the top of the stack. The rule evolves by calling a built-in function $f \leftarrow sensor[i]$ with the arguments taken from the stack, $f(v_1 \ldots v_n)$ and placing the result of the call, v, at the top of the stack. The function *send* (5th rule) is a built-in that sends data over the radio. Finally, instructions *bop* and *uop* (9th and 10th rules) actually encapsulate a set of rules that include the usual arithmetic, relational and logical binary and unary operators.

Algorithm 1 The gateway program

```
function MAIN()
    ATTACH(RADIO_RCV, HANDLERADIOMSG)
    ATTACH(SERIAL_RCV, HANDLESERIALMSG)
    loop
        MICROSLEEP()
        switch ( src )
            case RADIO:
                msg ← READRADIORCVBUFFER()
                FORWARDTOADAPTER(msg)
            case SERIAL:
                msg ← READSERIALRCVBUFFER()
                FORWARDTONODES(msg)
        end switch
    end loop
end function

function HANDLERADIOMSG()
    src ← RADIO
end function

function HANDLESERIALMSG()
    src ← SERIAL
end function
```

V. Operating System

A node in a SONAR deployment may run multiple periodic tasks that generate data streams. Users with administration access to the deployments can program tasks, compile them and inject them in the deployment via the WSN Adaptor (a Web service) which then forwards the tasks to the gateway to be radioed to the nodes. A simple protocol, implemented on top of the MAC layer, allows tasks, eventually divided into multiple blocks, to be sent over-the-air to the nodes. On arrival, the tasks are reassembled and installed in the nodes. Other control messages are also forwarded from the gateway to the nodes. From the nodes, the gateway receives data messages that it forwards to the deployment's Adaptor to be forwarded

to the SONAR Broker. Thus, the gateway does not run tasks, it acts simply as a message forwarder: it receives data messages from nodes in the deployment and passes them to the Adaptor, and receives control messages (including new tasks) from the Adaptor and radios them to the nodes in the deployment. Algorithm 1 shows this basic component. The gateway is initialized by attaching two handlers for interrupts signaling radio (from the nodes) and serial port (from the Adaptor) data reception. It then sleeps most of the time. When one of the interrupts is detected, the corresponding handler is executed and a flag is set to identify the source. The remainder of the loop then processes the incoming message.

Each node in a SONAR deployment has 2 pre-installed components: a small operating system and the SONAR virtual machine. The operating system is responsible for processing incoming control messages, for managing memory resources for tasks and for scheduling them EDF-style. Nodes keep information about tasks in a table. For each task, an entry in the table stores: a boolean - indicating if the entry is valid; three integers - the identifier, the period and the next activation of the task, respectively, and; an array of bytes - the bytecode for the task. The identifier is attached to messages sent by the task to the gateway so that the latter can distinguish to which stream the data it is receiving belongs to. This information is used to schedule the tasks and to prepare their execution with the SVM. The operating system executes a loop as described in Algorithm 2. The currently active task is identified by its integer index in the task table, denoted $curr$ in the following algorithms.

Algorithm 2 The node main loop

```
function MAIN()
    ATTACH(RADIO_RCV, HANDLERADIOMSG)
    ATTACH(RTC_ALARM, HANDLERTCALARM)
    loop
        RUN()
        SCHEDULE()
        SLEEP()
        LISTEN()
    end loop
end function
```

A brief initialization attaches handlers for radio reception and real-time clock interrupts. The node then enters the loop and executes the following procedures: RUN, that executes the current task; SCHEDULE - that selects the next task to be executed; SLEEP - that sleeps until the next task must be activated, and, finally - LISTEN - that listens for incoming radio commands that may have been received while executing elsewhere in the loop. The first 3 procedures are executed only if there are valid tasks in the table, i.e., the predicate TABLEEMPTY evaluates to false.

Procedure RUN (Algorithm 3) gets the stored state for the current task, its data, stack and text segments, and runs the task in the SVM. Note that changes to variables in a task are made directly in the data segment of the bytecode so that any state is preserved in between successive activations of the task. The virtual machine preserves the invariant that the stack S is empty when a task begins to execute and when it exits. Finally, the procedure adjusts the next activation time for the task by adding its period to the current time given by the Real-Time Clock (RTC).

Algorithm 3 Run current task

```
function RUN
    if ¬TABLEEMPTY() then
        (D, S, B) ← GETBYTES(curr)
        RUNSVM(D, S, B)
        t ← RTCTIME()
        p ← GETPERIOD(curr)
        SETNEXTACTIV(curr, t + p)
    end if
end function
```

Algorithm 4 Select next task

```
function SCHEDULE()
    if ¬TABLEEMPTY() then
        min ← MAX_INT
        for 0 < i < MAX_TASKS do
            if TASKVALID(i) then
                t ← GETNEXTACTIV(i)
                if t ≤ min then
                    min ← t
                    curr ← i
                end if
            end if
        end for
    end if
end function
```

The SCHEDULE procedure (Algorithm 4) computes the index of the (valid) task with the closest activation time. This becomes the next task to be executed by the operating system. Otherwise the predicate TABLEEMPTY will evaluate to true.

Algorithm 5 Sleep until next task activation

```
function SLEEP()
    if ¬TABLEEMPTY() then
        t ← GETNEXTACTIV(curr)
        RTCALARM(t)
    end if
    MICROSLEEP()
end function
```

Procedure SLEEP (Algorithm 5) computes the time until the next task activation and programs an alarm to wake up the node. The node then goes to sleep. This specification builds on the underlying assumption that tasks, being so small, execute in only a tiny fraction of their corresponding periods. In other words, if a task has a period p and an execution time, per activation, of t, then $t \ll p$. Otherwise we make no effort to schedule tasks within their periods. Since t is in the order of milliseconds we find this assumption adequate for practical purposes.

Finally, procedure LISTEN (Algorithm 6) checks for any incoming messages while the main loop was running. We assume that the nodes have the means to receive and to buffer messages asynchronously, by programming an appropriate handler to process the corresponding hardware interrupts. If

Algorithm 6 Handle Incoming Radio Message

```
function HANDLERADIOINTERRUPT()
    interrupted ← TRUE
end function

function LISTEN()
    if interrupted then
        msg ← READRADIORCVBUFFER()
        tag ← GETTAG(msg)
        switch ( tag )
            case TASK :
                i ← GETID(msg)
                p ← GETPERIOD(msg)
                b ← GETBYTES(msg)
                ADDTASK(i, p, b)
            case PERIOD :
                i ← GETID(msg)
                p ← GETPERIOD(msg)
                CHANGEPERIOD(i, p)
            case KILL :
                i ← GETID(msg)
                REMOVETASK(i)
            case RESET :
                for i = 0 ... TABLESIZE − 1 do
                    REMOVETASK(i)
                end for
        end switch
        interrupted ← FALSE
    end if
end function
```

a message is received, its tag is checked to identify its type and it is processed accordingly. At this point, there are 4 types of control messages: TASK - sends the identifier, the period and the bytecode for a new task to be executed in the node; PERIOD - sends the identifier and the new period for a running task in the node; KILL - sends the identifier of a task to be invalidated in the node, and; RESET - that invalidates all tasks running on a node. When a new task is reassembled and copied to the task table, its next activation is set to GETNEXTACTIV($curr$) + δ, where δ is a delay introduced to make sure that the task is schedulable in the next loop run, i.e., its activation time is in the future when the SCHEDULE procedure is called.

VI. PROTOTYPE IMPLEMENTATION

We implemented a full prototype of the specification for the data layer for a WSN composed of Arduino Mega 2560 devices. Each node is equipped with a XBee Series 2 radio, a SHT-15 temperature and humidity sensor, a light-dependent resistor sensor, a red LED and, a Adafruit Chronodot Real-Time Clock. We used the C/C++/Wiring language and the available Arduino libraries whenever possible. One of the devices acts as the gateway and is connected to a desktop computer through a USB port. The computer runs the Adaptor web service that connects the data layer with the SONAR Broker web service and provides a remote interface for the administration of the deployment. The binaries for the gateway and for the nodes (including the OS and the SVM) are loaded into the devices before they are deployed physically. Henceforth, programming is done through clients by injecting tasks into the mesh as described previously in this paper.

TABLE I. MEMORY CONSUMPTION AND CODE SIZE.

	Flash (256 kb)	SRAM (8 kb)	#lines
gateway	15.3 kb (6%)	1.0 kb (13%)	677
node	22.7 kb (9%)	2.7 kb (34%)	1325

Table I shows the memory footprint and total number of code lines for both the gateway and the nodes in this implementation. In this case the SVM in the nodes is configured to support a maximum of 8 tasks, each with a maximum bytecode size of 200 bytes. At this stage no effort was made to optimize the code both in terms of size and energy consumption. This SVM configuration, although quite generous, is actually quite compact and fits easily in the Mega 2560. Even in its current state, with support for 8 tasks, it almost fits in the smallest AVR Atmel micro-controller, the ATMega32 (32 kb Flash, 2 kb SRAM), with 2.7kb used versus 2kb available SRAM. The task management data structures are by far the largest loaded into the SRAM by our code. They take up 1.1kb (56% of used SRAM) and 0.6kb (35% of used SRAM), respectively, for 8 and 4 task configurations. The remainder of the space is used by the Arduino libraries, some of which are overkill for our needs, so there are some opportunities for optimizations. Moreover, of the 1325 lines of code in the nodes, 37% correspond to the OS, 23% to the SVM, and just 40% correspond to hardware specific code, e.g., modules for sensors, actuators, radio, and real-time clock. These numbers give us confidence that porting (and optimizing) this data-layer to more resource constrained devices will not present major problems.

VII. EVALUATION

We implemented a set of tasks to analyze the energy consumption and computational overhead of our prototype. The tasks test radio transmission, access to sensors and actuators and computation. Each test was implemented both in STL, running on top of SVM in each node, and directly in Arduino's native C/C++/Wiring. To measure the timings and the energy consumption, we connected a multimeter in series with one node (Figure 10) and registered the electric current variation associated with the execution of each task. The multimeter we used was a TENMA 72-7732A. A Keysight InfiniiVision MSO-X 2002A oscilloscope was also used for some measurements.

A. Experimental Results

The first tasks test radio transmissions. Figure 11 presents the STL code for a task that radioes 64 bytes, simulating a case where, for example, a task is programmed to radio 16 sensor readings (floating point values) to the gateway. The 64 bytes refer only to the payload of the messages which carry an additional 10 bytes of header information.

Figure 12 (blue line) shows current intensity vs. time when running the task with a period of 10 seconds. The first records the successful reception of the message by the node (our code puts the red LED on for 1.0 second). After the reception, one

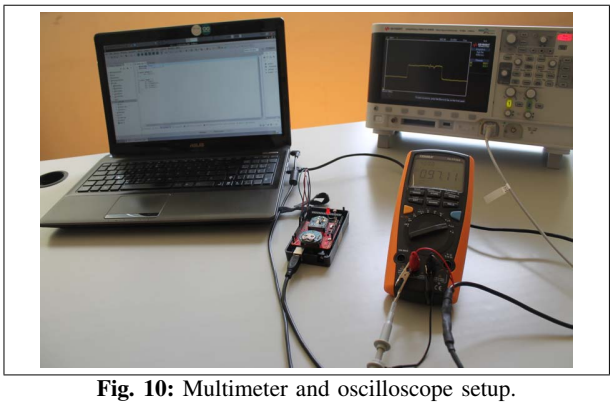

Fig. 10: Multimeter and oscilloscope setup.

```
use "ard2560.hw"

[float, ..., float] loop {
  radio [ 2.0 , 2.0 , 2.0 , 2.0 ,
          2.0 , 2.0 , 2.0 , 2.0 ,
          2.0 , 2.0 , 2.0 , 2.0 ,
          2.0 , 2.0 , 2.0 , 2.0   ];
}
```

Fig. 11: Transmission of data.

more peak (around $t = 20$) is visible, corresponding to the moments where the node radioed the 64 bytes. We would expect five more peaks within that time interval, given the period of the task. Their absence is due to a sampling problem related with the number of measurements the multimeter can execute per second. We ran the task again with a delay of 500 ms inserted and, sure enough, the other peaks became visible (black line). This delay was used only to allow the graphical visualization of the peaks. All the measurements given here were performed *without* the delay. Also, given the limited time resolution of the multimeter, we decided to use the digital oscilloscope to make all timing measurements. In this figure, the last, wide peak is due to a retransmission.

Figure 13 depicts the data obtained in one execution of this task. When the message is sent over the radio, a slight increase in the voltage ($\approx$ 0.1V) is detected, allowing us to time a full execution at 135ms. Similar measurements were done for messages carrying 4, 8, 16, 32 and 64 bytes (see

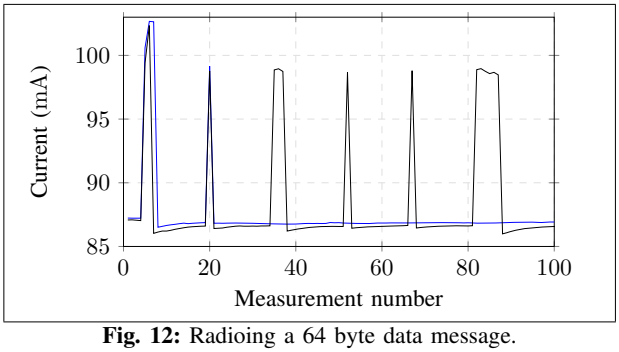

Fig. 12: Radioing a 64 byte data message.

below) to assess how power varies with message size.

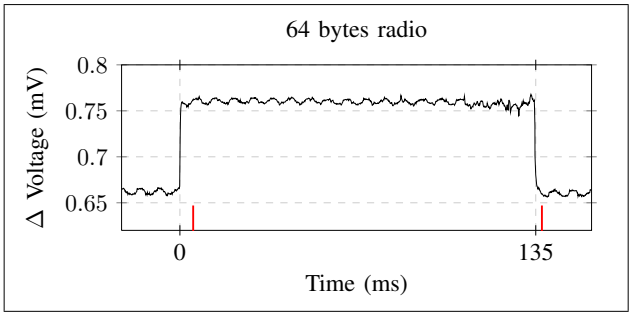

Fig. 13: Oscilloscope data of task execution

To test the access to sensors, we wrote a STL task (Figure 14) that accesses the temperature and humidity sensors in sequence. A similar program was written in C++ for Arduino. Figure 15 shows 6 executions of the STL task (black line) and of the Arduino code (red line). The graph plots current intensity vs. time when running the task with a period of 10 seconds. The fact that the peaks for the red and black lines are out of phase is due to overhead in the reception and initial scheduling of the task, otherwise the approximate periodicity is observed. The first peak is, again, due to the reception of the task in the node. This color code - *black for STL and red for Arduino* - will be used for all figures henceforth.

```
use "ard2560.hw"

[ ] loop {
  float t = temperature();
  float h = humidity();
}
```

Fig. 14: Access to sensors.

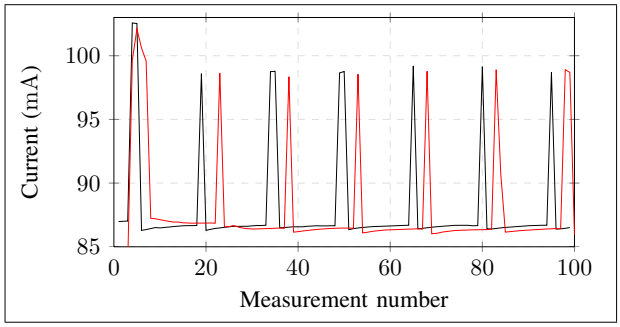

Fig. 15: Temperature and humidity sensors access.

The third task tests computation within the microprocessor. The task computes the 1000th term of the *logistic map* [13], a famous simple map that produces a series of numbers between 0 and 1. Figure 16 shows the STL code and Figure 17 shows the execution of 6 such tasks. A careful measurement with the oscilloscope, for this and the other examples, allowed us to conclude that the peaks are well approximated by a square wave with a maximum current of 98.9mA.

Finally, a task tests the triggering of actuators by alternatively activating and deactivating the external red LED.

```
use "ard2560.hw"

[] loop {
  float x = 0.2;
  float k = 4.0;

  int i = 0;
  while (i < 1000) {
    x = k * x * (1.0 - x);
    i = i + 1;
  }
}
```

Fig. 16: Computation of the logistic map.

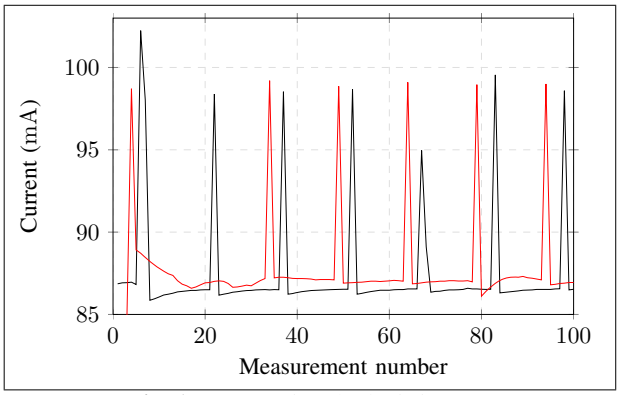

Fig. 17: Computing the logistic map.

Figure 18 presents the code for the task. Figure 19 shows

```
use "ard2560.hw"

init {
  bool state = false;
}

[] loop {
  led(state);
  state = !state;
}
```

Fig. 18: Access to extenal LED (actuators).

the execution of 6 tasks. The task activations correspond to the observed peaks in the graph, except for the first one. After a task is executed the current stabilizes in one of 2 levels, corresponding to LED disconnected and LED connected, with a difference of 4.3mA.

The intensity of the current in the Arduino Mega 2560 board varies between a base value, when the board is in sleep mode, not running a task, and a peak value, when a task is being executed by the virtual machine. The same values are observed for the corresponding Arduino programs. Table II shows the base and peak values for current, voltage and instantaneous power in the experiments. The Arduino 2560 provides a set of sleep modes with different levels of energy savings and hardware components turned off. Our prototype uses the IDLE mode, which is not the most power efficient but allows us to wake up the board in time to properly receive

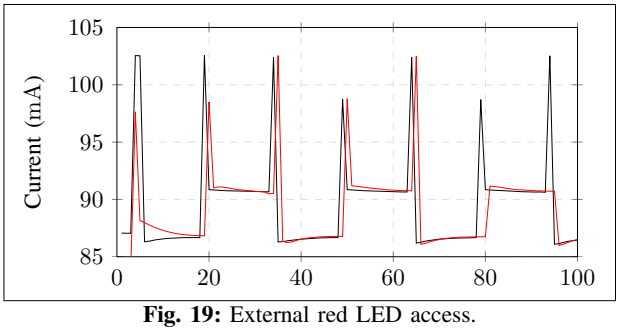

Fig. 19: External red LED access.

asynchronous messages from the gateway. In order to save as much power as possible while in IDLE mode, we disable also: the analog-to-digital converter, the peripheral interface, 3 different timers and the two wire interface.

From these base and peak values, and from the execution times of the tasks, we can compute the total energy spent by a SONAR task and by the corresponding native Arduino program. The values were computed from the measurements using the following equations for the instantaneous power and energy consumption:

$$P = V \times I$$
$$E = P \times \Delta t$$

These equations are adequate as we measured the profile of the tasks to be well approximated by rectangles of height equal to the peak intensity and width equal to their execution time. Table III shows, for each test: the size in byte of the STL task and then, the time and energy consumed to execute both the STL tasks and the equivalent Arduino program.

TABLE II. BASE AND PEAK VALUES

	base value	peak value
current (mA)	86.3	98.9
voltage (V)	5.0	5.0
power (mW)	432	495

TABLE III. ENERGY CONSUMPTION

task	size	time (ms)		energy (mJ)		$\frac{\text{STL}}{\text{Ard}}$
		STL	Ard	STL	Ard	
comp	121	160	27	79.1	13.4	5.9
temp	37	275	247	136.0	122.1	1.1
hum	37	274	80	135.5	39.6	3.4
lum	37	37	11	18.3	5.4	3.4
act	71	38	13	18.8	6.4	2.9
rad-4	36	74	36	36.6	17.8	2.1
rad-8	50	80	40	39.6	19.8	2.0
rad-16	54	89	49	44.0	24.2	1.8
rad-32	78	105	66	51.9	32.6	1.6
rad-64	126	135	99	66.8	49.0	1.4

B. Discussion

The analysis of the measured data allowed us to conclude some interesting facts about the current prototype. The Sonar operating system and the virtual machine in the

nodes introduce the highest overhead for tasks that are purely computational, by a factor of 5.9, for a cycle with 1000 iterations. However, a closer look at the ratio between the execution times in STL and Arduino (Figure 20) shows that part of this overhead includes an initial setup time by the node's operating system. In fact, as the number of iterations grows, the contribution of this initial overhead gets diluted and the real ratio between STL and Arduino (native) operations stabilizes at around 4.5 (the bars represent 95% confidence intervals). This is expected, and is due to the fact that we are running byte-code tasks on top of a virtual machine, rather than native code generated from C++ programs. We believe that optimizations of the byte-code generator and of the virtual machine implementation will diminish this gap but it will otherwise be always present. It is the price of portability and dynamic reprogramming. The difference for other tests is far more modest. with access to sensors and actuators around 3 times slower and radio transmission of any size around 2. Though we throughly analysed the code that accesses the temperature sensor, we cannot yet explain the lower overhead (only a factor of 1.1 slower) relative to the other sensors and actuators (globally around 3 times slower). As the energy consumed is proportional to the time the task takes to execute, clearly the optimizations must focus on this aspect, all other being the same for STL tasks and for Arduino programs. We believe, however, that even in this unoptimized state our prototype compares well with Arduino native code with the added benefits of simplified programming and dynamic reprogramming.

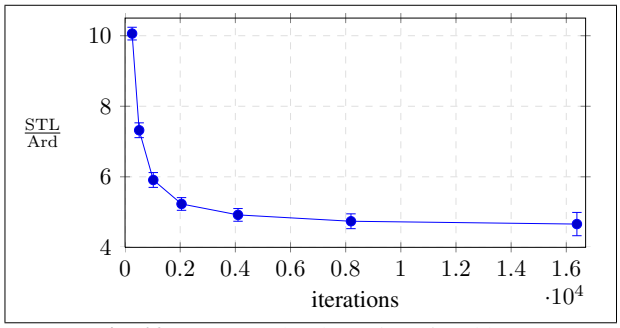

Fig. 20: SVM overhead vs. size of problem.

VIII. Conclusions

In this paper we present a specification for a compact, portable, data-layer that can be used to suport seamless dynamic reprogramming of WSN, based on the notion of periodic, non-preemptive, tasks. The idea is that users get a package containing a gateway and an undetermined number of nodes, pre-configured to work as a self-organized wireless mesh. The gateway is a simple forwarder of data and control messages. The nodes come with two pre-installed components: a small operating system and a virtual machine to run the tasks. Tasks are injected in the network by clients, through the gateway, with the mediation of an Adaptor web service. The current implementation has both low memory footprint, and is highly modular and portable, with only 40% of the code of the nodes hardware specific.

We report measurements of the resource consumption for our system as compared to traditional C++ programming for Arduino. We show that, despite our code not being optimized, we compare favorably with native code performance (a factor of 2 or 3), except for pure computational tasks were the overhead is more noticeable (a factor of 4.5). On the other hand, our system is portable across distinct WSN architectures, simplifies programming greatly and allows the dynamic programming of WSN.

Acknowledgment

We would like to thank our colleague Carlos Machado, for his advice and expertise. This work is sponsored by projects SENSING (contract: NORTE-07-0124-FEDER-000058) and RTS (contract: NORTE-07-0124-FEDER-000062).

References

[1] L. Lopes, F. Martins, and J. Barros, *Middleware for Network Eccentric and Mobile Applications*. Springer-Verlag, 2009, ch. 2, pp. 25–41.

[2] E. Neto, R. Mendes, and L. Lopes, "An Architecture for Seamless Configuration, Deployment, and Management of Wireless Sensor-Actuator Networks," in *3rd International Conference on Sensor Networks (SENSORNETS 2014)*, Lisbon, Portugal, 2014, pp. 73–80.

[3] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, 2002.

[4] J. Yick and B. Mukherjee and D. Ghosal, "Wireless Sensor Network Survey," *Computer Networks*, vol. 52, pp. 2292–2330, August 2008.

[5] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM Press, October 2002, pp. 85–95.

[6] R. Newton and M. Welsh, "Region Streams: Functional Macroprogramming for Sensor Networks," in *First International Workshop on Data Management for Sensor Networks (DMSN'04), Toronto, Canada*, 2004.

[7] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine," in *Virtual Execution Environments (VEE'06)*, June 2006.

[8] TinyOS, "The TinyOS Documentation Project," available at `http://www.tinyos.org`.

[9] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (EmNets'04)*, Tampa, Florida, USA, November 2004.

[10] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A Dynamic Operating System for Sensor Nodes," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys'05)*. New York, NY, USA: ACM Press, 2005, pp. 163–176.

[11] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, vol. 10, no. 4, pp. 563–579, August 2005.

[12] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'05)*, December 2005.

[13] J. C. Sprott, *Chaos and Time-Series Analysis*. Oxford University Press, 2003.