PUC

ISSN 0103-9741

Monografias em Ciência da Computação

nº 04/13

Determining the Boundary Cost and Flexibility in Wireless Sensor Networks

Adriano Francisco Branco

Noemi de La Rocque Rodriguez

Silvana Rossetto

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 04/13 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena April, 2013

Determining the Boundary Cost and Flexibility in Wireless Sensor Networks

Adriano Francisco Branco Noemi de La Rocque Rodriguez
Silvana Rossetto

{abranco , noemi}@inf.puc-rio.br , silvana@dcc.ufrj.br

Abstract. Several authors have proposed solutions for remotely updating wireless sensor network applications. These proposals usually make a trade-off in flexibility versus update cost. At one extreme, full-scale binary upgrades provide full flexibility at unacceptable communication cost, while at the other end parameter tuning typically provides the least expendable but also the least flexible form of updates. In this work, we describe WDvm, a platform that allows the programmer to experiment with different combinations of flexibility and cost, choosing the best fit for each application. WDvm provides a virtual machine which runs over TinyOS and which can be installed with different sets of ready-made components, facilitating tuning of the abstraction boundary that can be used for reconfiguration. A simple intermediate language runs over this virtual machine. Parameter-based configuration receives special attention in WDvm. Parameters are directly integrated into the intermediate language, allowing scripts to act both as simple parameter redefinitions and to determine new parameter values as a result of arbitrary operations. In this report, we describe the structure of WDvm and describe some experiments, in which we evaluate the overhead imposed by the virtual machine by comparing the execution of simple scripts with a TinyOS similar application. We also illustrate, through an example, the possibility of defining different abstraction boundaries for the virtual machine.

Keywords: Virtual Machine, Sensor Network, Reprogramming, Reconfiguration

Resumo. Existem vária propostas para atualização remota em redes de sensores sem fio. Essas propostas, normalmente, trabalham a relação entre flexibilidade versus custo de atualização. Em um extremo a carga completa do arquivo binário possibilita grande flexibilidade com alto custo de comunicação. Por outro lado a reconfiguração via parâmetros permite um baixo custo de comunicação com pouca flexibilidade. Neste trabalho descrevemos WDvm, uma plataforma que permite experimentar diferentes combinações de flexibilidade e custo. WDvm executa uma máquina virtual em cima do TinyOS e pode ser instalada com diferentes conjuntos de componentes. Esses componentes são parametrizados e integrados diretamente com uma linguagem intermediária da VM. Isso permite que a aplicação tenha um maior controle das funcionalidades ajustando os parâmetros automaticamente. Nesse relatório descrevemos a estrutura de WDvm e alguns experimentos. Avaliamos o "overhead" imposto pelo uso de VM comparando com a execução de aplicações similares em TinyOS. Também apresentamos, através de exemplos, a possibilidade de definição de diferentes fronteiras de abstração para a VM.

Palavras-chave: Máquina Virtual, Redes de Sensores, Reprogramação, Reconfiguração

In charge of publications :

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Contents

1 Introduction

Due to the number of devices involved in a wireless sensor network in applications such as environmental monitoring or production control, such networks are often built using devices (*motes*) with very limited resources. On the other hand, it is also frequently difficult to recover motes from the position in which they are installed for software corrections or reconfigurations. Thus, several authors have proposed solutions for remotely updating wireless sensor network applications. As discussed in [1], these proposals usually make trade-offs in flexibility versus update cost. At one extreme, full-scale binary upgrades provide full flexibility at unacceptable communication cost, while at the other end parameter tuning typically provides the least expendable but also least flexible form of updates.

In this work, we describe WDvm, a platform which allows the programmer to experiment with different combinations of flexibility and cost, choosing the best fit for each application. WDvm provides a virtual machine which runs over TinyOS and which can be installed with different sets of ready-made components, facilitating tuning of the abstraction boundary that can be used for reconfiguration. A simple intermediate language runs over this virtual machine.

The idea of allowing the user to fit the abstraction level of the virtual machine to the set of application domains that the WSN is expected to handle was first proposed by Levis et al. [2]. In WDvm, we explore the possibility of defining different abstraction boundaries for the virtual machine but focus on extending the very basic parameter-based configuration model into simple scripts. Parameters are directly integrated into the intermediate language, allowing scripts to act both as simple parameter redefinitions or to incorporate these redefinition in programs.

The remainder of this report is organized as follows. In Section 2, we discuss different alternatives for dynamic update of WSN applications and the impact of their choices on cost and performance. In Section 3, we describe the architecture of WDvm and the components that are part of its basic structure. Next, in Section 4, we describe some practical results of the implementation of WDvm over TinyOS2. Finally, Section 5 contains some final remarks.

2 Dynamic updates in WSN applications

Software updates and corrections are common across all areas of application, but in wireless sensor networks, the number of devices involved in applications — and often also their physical location — makes remote updating specially attractive.

Remote software update necessarily involves the communication cost of sending the updates over the network. Thus, smaller or less update messages imply less energy spent by radios. Besides, fault tolerance and version stabilization are also easier to achieve with smaller messages. Proposed mechanisms for remote reconfiguration and reprogramming represent different trade-offs between flexibility and cost.

The simplest form of remote reconfiguration is through the redefinition of parameters [3, 4]. Small-sized messages can be sent over the network containing (name, value) pairs. This approach has very low communication cost but has limited power, allowing running applications to be modified to specific, foreseen, behaviors. This is adequate for particular scenarios: for instance, Greenstein and others [5] show the benefits of this approach in tuning the filters to be applied to collected raw data.

At the other end of the spectrum, one can dynamically update the binary code running on a mote. Because wireless sensor network applications often follow an SPMD (single program, multiple data) pattern, it is possible to broadcast the binary code without too much difficulty on the programming aspects. Deluge [6] follows this approach to the extreme, allowing for updates of a node's complete binary image. This is of course the most flexible approach, but incurs in prohibitive costs, as a typical binary image will be of the order of tens of kilobytes. The SOS [7] operating system allows for partial binary updates, diminishing costs somewhat.

Several authors have proposed approaches based on virtual machines, defining intermediate combinations of cost and flexibility. One of the first such proposals was that of Maté [8], which uses a very simple virtual machine running over TinyOS-1. The authors of Maté themselves later concluded that the virtual machine was too limited to support higher level programming, and proposed the concept of ASVMs [2] (application specific virtual machines), which provides a way for the user to explore the boundary between virtual code and the VM engine. ASVMs are customized for specific domains of applications, and because of this customization, allow complex operations to be described in high-density code, reducing interpretation overhead and communication costs. This idea is further investigated by Balani and others [1] in their proposal of DVM. The authors combine the concept of ASVM with SOS's capability of updating binary models, allowing the *abstraction boundary* provided by the VM to be dynamically modified. Thus the DVM model supports frequent updates with high-density scripts but also other, less frequent ones, when major changes in applications are needed, with binary modules.

Our work fits into the ASVM and DSM concepts of providing support for different abstraction boundaries. In this work, we concentrate on investigating the range of cost/flexibility combinations that goes from the parameter-based model to full-fledged virtual machines. We propose a virtual machine which, like ASVM, can be tailored to different application areas in order to allow for high-density domain-specific code. However, all parameters of the binary modules that compose an image are exported to the interpreted language and can be manipulated in the virtual program as predefined variables. Thus, a parameter-based configuration script can consist of a simple program containing only assignments to these variables or of programs of arbitrary complexity with accesses to predefined variables and parameters provided by the runtime image.

3 System Description

The main components of the WDvm system are shown in Figure 1. Module "VM Engine"

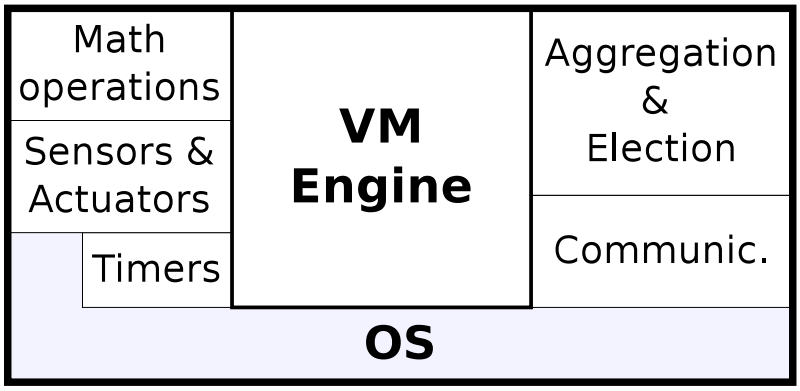

Figure 1: WDvm system modules

is responsible for the operation of the virtual machine. Modules 'Aggregation & Election" and 'Communication" implement frequent interaction patterns and can be parameterized to provide different variants of group operations, leader election, and routing to the base station. Module "Communication" also implements the support for remote updates. Modules "Math operations", "Sensor & Actuator" and "Timers" implement frequently needed functions for math operations, access to sensors and actuators, and timer control, respectively.

3.1 WDvm programming model

WDvm is based on an event-driven programming model where each event is treated by a procedure written in a statically typed script language. In order to simplify the program control flow, events are queued to be processed after the previous one has finished. Operating system events are treated regardless of the WDvm processing, but in some cases operating system events may generate new events for the virtual machine. In order to deal with the limited memory space, we use static memory allocation which allows us to check all memory allocation during the compilation phase. Besides, the subroutine scope is restricted (only global entities can be created) and the instructions' parameters are passed by reference, instead of using a stack.

The engine shown in Figure 2 has a memory area, a Program Counter (PC) registers stack and a queue of events. The PC on the top of the stack always points to the next instruction to be performed. Each instruction is handled as an independent task which must fetch the parameters of the instruction, execute it and increment the current PC at the end of its execution. The use of independent tasks to process each instruction ensures that the WDvm script will not block the rest of the system. Each task is scheduled to run only after the previous task has finished. When the PC stack is empty, the engine must process the next event from the queue of events. All event is associated to a type and a numeric identifier. By using this tuple (event type-identifier), the engine finds the address of the event handler in the event configuration table. This address is pushed as a new PC register into the PC stack and then a new task is scheduled to process it.

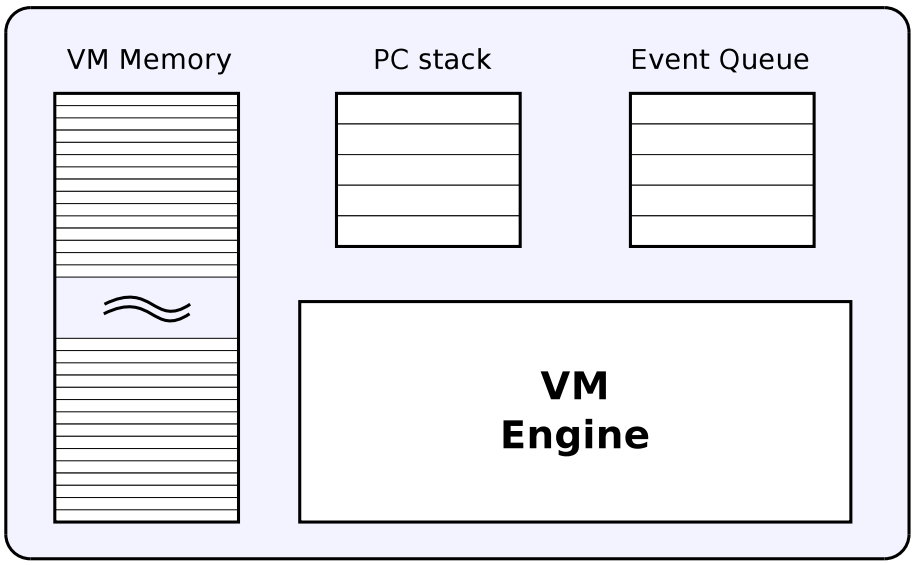

Figure 2: WDvm Engine

3.2 WDvm built-in functionalities

WDvm offers a set of configurable features in order to facilitate the development of new applications. These features were identified from the study of the main characteristics of typical WSN applications as well as WSN macroprogramming models. As examples of typical applications, we analyzed a simple case of environmental and building monitoring and an urban parking application. The programming models studied were: Regiment [9, 10], Pleiades [11], Cosmos [12], TinyDB [13], WADL [14] and ATaG [15].

We divide the set of selected features into "local operations" and "distributed operations". Local operations include all operations that are internal to a node (mote), such as timers and sensor reading. Distributed operations include all operations that involve communication/interaction between nodes (motes) and are typically the most difficult part of the WSN program implementation. In the next subsections we describe the main local and distributed operations. The main parameters of all operations are accessible locally through predefined variables and remotely through the reconfiguration tool.

3.2.1 Local Operations

In addition to the typical system operations, such as arithmetic and flow control, WDvm offers four types of local entities with particular operations: `Timer`, `LocalEvent`, `Sensor`, `Actuator`.

The current version of WDvm allows the creation of a maximum of 8 `Timer` and 32 `LocalEvent`. Each `Timer` has a predefined value (in seconds) for the frequency of the timer activation and the memory address of the procedure that must be called when the timer fired event is signalized. The instruction `setTimer` allows us to start a timer in the "periodic" or "one-shot" mode, or to stop an active timer. A *LocalEvent* is a procedure that is scheduled to run following the "first in first out" policy (each event is completely processed before start processing the next event from the queue).

Sensor and *Actuator* are entities that are dependent on the hardware used. Sensors are devices used to collect data from the monitored environment, while actuators are devices used to perform actions on that environment, typically in response to the data analysis. The current version of WDvm includes temperature and light sensors and battery voltage. The configuration of a sensor defines the address of the procedure to be performed when a reading is completed and the variable that will receive the value read. The instruction `readSensor` triggers the reading of specific sensors. The current version of WDvm uses leds to simulate actuators.

3.2.2 Distributed Operations

WDvm offers the following distributed operations: Group Builder, Leader Election, Aggregation, Routing to Base Station, and Group Communication;

Communication between nodes/motes is based on the concept of *Group Builder*. A node group is formed by all nodes that have the same group parameter value (type identifier) and are within the area bounded by a maximum number of hops from a message sender. WDvm allows the creation of a maximum of 32 different types of groups (the groups may overlap each other). Each group is identified by an 8-bit parameter and by the number of hops from the sender node of a message. When a node sends a message to a particular group, the algorithm automatically forwards this message to all neighboring nodes that also belong to the same group type and group parameter, up to the maximum

number of hops. The `group` functionality provides the following predefined variables as parameters: `Max Hops` - Maximum distance in hops inside the group, `Group Parameter` - Local Group Type partition ID, `Current Leader` - If elections is ON, it holds the Current Leader ID, and `Group Flags` - Group control flags: Group activity and election mode. The user can dynamically reconfigure the behavior of the group by modifying the parameters of interest. As an example, one can change the scope of a group by adjusting the value of `maxHops` or reorganize groups changing the value of the `group` parameter.

The leader election algorithm works over the Group Builder mechanism. Leader flag defines whether or not the group needs to elect a leader. The default version of the leader election algorithm is based on the battery power of each node (the node with higher remaining power is chosen, and node's identifiers are used as the tie-breaking criterion when two or more nodes have the same remaining power). A special flag, called *participation flag*, can be used to dynamically change the status of a node with respect to a specific group (whether or not the node is member of the group).

Aggregation operations also work over Group Builder. The application sets the type of sensor that will be used (physical quantity to be collected) and which kind of aggregation operation will be applied for the collected values. Optionally, one can define a comparison operation and a upper/lower bound value. The operations currently available are: *average*, *sum*, *maximum value* and *minimum value*. Comparison operations include: $<, \leq, >, \geq, =, \neq$. Instruction `aggreg` triggers a specific aggregation operation. At the end of an aggregation operation, an event is queued in order to execute the procedure specified by the user. The `aggregation` functionality provides the following predefined variables as parameters: `SensorId` - Sensor value to be aggregated, `Function` - Aggregation function, `CompOper` - Comparison operator, `RefValue` - Reference value used in comparisons. It also provides the following predefined variables that can be returned by the operation: `Total nodes` - Number of participant nodes in last aggregation, `True answer counter` - Number of true results for comparison operation, `False answer counter` - Number of false results for comparison operation, and `Calculated value` - Aggregation result.

To send and receive messages one must to configure a Message entity containing a data structure (used to receive the payload of the message) and the address of the procedure to handle the messages. Every Message have a unique identifier on the network. When a node receives a valid message, the VM is notified by the operating system and generates a local event to handle the incoming message, using the procedure address defined.

The instruction `sendMsg` allows send a message to the nodes in the same group or to the base station. It is also possible to reply a message to a requester node in the same group. Routing to base station is implemented by using a particular routing component which routes a message to the base station.

3.3 Program structure

A WDvm program consists of a sequence of instances declarations. The current version of WDvm implements the following entities: `Variable`, `Sensor`, `Message`, `Timer`, `Grouping`, `Aggregation`, `LocalEvent`. `Variable` is used to represent the variables of the program. Variables can be integer types with or without signal and with 8, 16 or 32 bits. It is also possible to define simple data structures containing a set of variables. Remaining entities are related to the other facilities offered by the system. Each one is defined based on a set of parameters whose values are provided when the instance is created.

A WDvm program is organized in two main areas: Data/Configuration and Procedures. The Data/Configuration area is divided into sections for entity instantiation. Global variables and some of entities parameters can be referenced directly by the user program. The Procedure area contains the code (procedures) of the user program. Each procedure is identified by its start address.

We define a set of mnemonics and a compiler that converts the user program to the VM bytecodes. The set of instructions/mnemonics of the current version of WDvm is presented in Table 1.

Table 1: WDvm Instruction Set

Mnemonic	Definition
add, sub	+, -
mult, div	*, /
inc, dec	increment-of-1, decrement-of-1
compare	Comparison operation
cast	Type conversion
call	Unconditional jump
if, ifelse	Conditional jump
where	Conditional jump with repetition
ret, end	Return to jump origins
setGroup	Set group flag
setTimer	Set timer to start or stop
genEvent	Generate a local event
aggreg	Starts an aggregation
setVar	Set a value into a variable
readSensor	Start a sensor reads
setActuator	Writes a value into an actuator
sendMesg	Sends a message to a Group or to the Base-Station

3.4 Reconfiguration tool

To allow remote reconfiguration and reprogramming, we built a tool that can load a new program (VM bytecode) or update specific sections of the VM memory. The protocol for bytecode reprogramming is based on flood dissemination. Each new program version is uniquely identified and it is automatically disseminated to the entire network. Versioning allows nodes to retrieve the latest version of the program from their neighbors, in case they are not active during the dissemination phase.

The memory update protocol allows us to disseminate selective parameters for certain types of nodes or groups. To minimize the cost of the dissemination protocol, we build small routing tables that are updated with the information loaded by the latest messages routed by the node to the base station. Messages are also associated with a unique version identifier and each node maintains a small history of the last messages. Thus outdated nodes can retrieve the latest messages from its neighbors. The tool trans-

mits a reconfiguration message with up to 18 bytes. These bytes may contain continuous or not continuous VM memory region sections. Each section is represented by $2 + n$ bytes containing the starting address, the value n and the n data bytes.

3.5 Parameters integration

WDvm combines three principles to explore the use of parameters and increase flexibility in the reconfiguration: (i) the features of the runtime are generic and their parameters are exposed as program variables; (ii) any user program variable or constant can be treated as a parameter; (iii) the reconfiguration tool has indexed access to the VM memory, therefore it is possible to override the values of program variables and constants.

The item (i) allows the user application or the reconfiguration tool to have remote access to the parameters associated with the runtime functionality, making functional changes even more flexible. The item (ii) facilitates the creation of configurable applications, increasing the possibilities for future adjustments. With the item (iii) the user has remote access to manipulate any program value as a reconfiguration parameter. Items (i) and (ii) enable the implementation of scenarios where a simple configuration parameter can triggers the reconfiguration of a set of parameters.

4 Practical results

We implemented WDvm in TinyOS-2. Besides the basic components of TinyOS, the basic WDvm image includes the CTP (Collection Tree Protocol) component [16]) for routing messages from motes to the base station. The protocols for group communication and data dissemination were built over the primitives for communication between nodes. Control of procedures and events in the virtual machine were easily developed over nesC/TinyOS programming model. WDvm currently runs on the MicaZ mote [17] coupled with MDA100.

In this section, we conduct evaluations of WDvm from three different points of view. In Section 4.1, we try to estimate the overhead incurred by interpretation. To this end, we compare computing-intensive code written in WDvm and in the native nesC programming language. Next, in Section 4.2, we measure the cost of updating an application. Finally, in Section 4.3, we illustrate the possibility of working at different abstraction boundaries, and measure the cost of writing a given application working at different abstraction levels.

We used the Avrora instruction-level simulator [18] to simulate the MicaZ hardware in the controlled tests.

4.1 VM Overhead Benchmarking

We use two different tests to evaluate the overhead incurred by the VM as compared with direct execution over TinyOS. In the first test, we run a simple CPU-bound application: a loop that continuously increments a value. This would be an extremely uncharacteristic pattern for sensor network applications, which typically pass through relatively long intervals of quiescence, followed by short periods of activity, triggered by external events. The idea of this test is to stress the processing capacity of WDvm to the limit. In the second test, we measure the overhead of the VM in a more typical scenario, in which the application repeatedly reads data from a sensor in a loop.

In each test, we run both variants of the application for five minutes. At interval of ten seconds, the applications send the value of the loop counter to the base station.

In both tests, programs are coded with event-based loops. In the nesC/TinyOS versions, each iteration posts a task representing the following one. In WDvm, each iteration generates a local event with the same intent. The CTP component is used both in the implementation of the VM runtime and in the nesC/TinyOS versions.

To compare the results, we use two metrics. The first one is the total number of iterations executed along the five minutes that the applications are left running. This number is the value of the counter sent to the base station at time 300s. One of the advantages of this metric is that it could be used on real motes as well as on the simulator. The second metric we use is the total number of cycles in *Active* and *Idle* state [1]. The values for this metric were obtained through the simulations on Avrora.

4.1.1 Scenario 1 - CPU-bound Application

Table 2 presents the results obtained with Avrora for our first test scenario. In the nesC version, the main loop is executed in a TinyOS task that contains only two commands: the loop counter increment(++ in C) and the the (re)post of the task itself. In WDvm, the loop is the main procedure executed by the VM, with three commands: the loop counter increment, the instruction that generates a local event, and the end of procedure instruction. This local event makes the VM post a task that (re)initiates the main procedure.

Table 2: CPU-bound Test

Metric	Program Version		
	WDvm(a)	**nesC**(b)	b/a
loop counter	1,573,794	9,902,517	6.29
active cycles	2,177,237,555	2,211,835,350	1.02
idle cycles	34,602,445	4,650	0.0

As expected in loops with no blocking operations, the CPU was kept busy almost 100% of the execution time. The cost of interpretation becomes explicit in the value of the loop counter obtained at time 300s. The TinyOS version ran 6.29 times the iterations executed by the VM version.

We also executed this same test directly on a MicaZ mote. The relation between the values obtained for the loop counter were quite close to the ones from the simulation. (Values were respectively 1,573,729 and 9,902,222.)

We can compute the number of cycles per instruction in WDvm if we consider that the main loop of our test script involves three instructions and divide the total number of CPU cycles by the final value of the counter (number of times that the loop was executed) multiplied by three. The result is 461 cycles, which is close to the 400-cycles value obtained in the micro-benchmark of ASVM (section 4.5 §2 of [2]) and to the value of 550 cycles reported for DVM (section 4.1 §2 for [1]).

[1] TinyOS keeps the CPU in idle state when the task queue is empty. The CPU goes into active state when it receives an interruption.

4.1.2 Scenario 2 - IO-bound application

In this test, the application repeatedly reads the sensor and increments the loop value when the sensor returns a value. In the nesC version, the main loop is a TinyOS event handler that again contains two commands: the loop counter increment (++ in C) and the *call sensor.read()* call, which initiates a new sensor reading. At this point, TinyOS places the CPU in *Idle* state. When an interruption occurs, TinyOS generates a new task to (re)execute the event handler. In the WDvm version, the loop is the main procedure for the VM, and also contains two commands: the loop counter increment and the instruction for requesting a value from the sensor. After this request, the VM becomes idle awaiting new events, and again TinyOS puts the CPU in *Idle* state. When an interruption occurs, TinyOS generates a task to execute the event handler for the sensor, and this in turn generates en event for the VM. The VM then posts a task to (re)initiate the main procedure.

Table 3 presents the results for this scenario.

Table 3: IO-bound Test

Metric	Program Version		
	WDvm(a)	**nesC**(b)	b/a
loop counter	29,977	29,948	1.00
active cycles	235,063,740	116,793,767	0.50
idle cycles	1,976,776,260	2,095,046,233	1.06

In this case, predictably, CPU active time was much less than in the first scenario. CPU was idle around 90%-95% of the time. The two variants executed approximately the same number of interactions in the 300 seconds of execution time. As regards CPU cycles, however, the WDvm version needed around double the cycles used by nesC. In WDvm, CPU was active 10% of the time, while in nesC only 5%.

Direct execution on the MicaZ mote again produced results close to the simulator's: the value of the counter was 29,994 for the WDvm version and 29,997 for the nesC one.

In WDvm, approximately 100 iterations were executed per second. In ASVM, in a similar test, the ratio of 312.5 iterations per second was obtained (5000 loops per 16.0 sec in section 4.5 §4 of [2]). The difference in values was apparently due to the analog-digital conversion in sensor readings, as in our case the number of iterations was the same as that of direct execution over nesC/TinyOS.

The results for this second scenario give us an important insight about the real costs incurred by interpretation. Although the execution of interpreted code is more expensive than that of the native, nesC, code, this difference practically disappears in an I/O bound pattern, which although extreme in this case, is closer to the typical pattern for wireless sensor network applications.

4.2 Reconfiguration cost

In this section, we try to estimate the cost of disseminating new code for WDvm. This cost depends on several factors, such as the number of nodes, the topology of the network, the dissemination algorithm, and the noise/failure conditions that can lead to retransmissions. In this work, we stick to measures that are independent from the network, and use the number of bytes (and consequent number of messages) as our metrics for reconfiguration cost.

In section 4.2.1, we compare reprogramming costs in WDvm and nesC by measuring the size, in bytes, of complete applications written for the WDvm default image and in nesC. In the next Section, we measure reconfiguration costs by looking at a parameterized alarm-oriented application and analyzing different reconfiguration alternatives.

4.2.1 Reprogramming Cost

In this Section, we compare the size, in bytes, of applications written in WDvm with their counterparts coded directly over TinyOS. This is of course the advantageous situation for a virtual machine, as in WDvm only the script needs to be transferred to motes, while in TinyOS one must transfer the whole binary image. Nevertheless, it is useful to have a more exact idea of the difference between the two approaches.

For the nesC/TinyOS applications, application size is obtained from the compiler. In the case of WDvm applications, we measure the size of the code in the files generated by the compilation procedure. We consider that messages can hold up to 24 bytes to estimate the number of messages necessary for reprogramming the network with these applications, assuming the default 28-byte message size of TinyOS with 4 bytes used by the control protocol.

Table 4 presents the values we obtained for three applications. The first value in each cell indicates the number of bytes and the second one, in square brackets, the number of required messages. The first application is the famous *Blink* example from the TinyOS tutorial. This is a good example because it uses no special components, only timers and leds. In the WDvm version we kept the same structure of `Blink.nc`, using three timer entities. Application *RdLoop1* is the same application used in Section 4.1.2 in its WDvm and nesC versions. Application *RdLoop2* is a version of*RdLoop1* for nesC without the CTP component.

Table 4: Reprogramming Cost

App	Program Version		
	WDvm(a)	**nesC**(b)	b/a
Blink	63 [3]	2048 [86]	32.5
rdLoop1	80 [4]	18188 [758]	254.8
rdLoop2		13022 [543]	162.8

Units: Bytes [Messages]

The *Blink* application illustrates the cases in which the nesC application requires no auxiliary components for communication. The large difference to the value in *rdLoop1* is due to the latter's use of components for radio communication and message routing (CTP). In the nesC version, the CTP component is included in the generated code, while in WDvm it is pre-loaded in the motes. Application *rdLoop2* attains an intermediate value because it does not use CTP, but still relies on basic communication components (With *rdLoop2*, we are simulating a situation in which the programmer knows he will not need a given module. In the specific case of *rdLoop2*, the application runs without routing — each node involved is in the direct range of the base station.).

4.2.2 Reconfiguration Cost

We now try to evaluate the cost of reconfiguration using WDvm's resources for parameter manipulation, through an application that explores the flexibility of parameter-based

reconfiguration in WDvm. As we did for reprogramming, we measure the cost of this reconfiguration by measuring the number of bytes to be disseminated.

Listings 1 and 2 present an application that involves remote configuration and auto-tuning. The application monitors the average temperature as measured by motes in well-lighted points. The user must remotely define the threshold for a place to be regarded as "well-lighted".

The basis of the application is the algorithm for group creation available in the default WDvm runtime. The average value is computed by the aggregation modules, at all nodes participating in a single group (corresponding to nodes at well-lighted spots). The coordinator node is defined statically, and in our example is node with ID 2. Each node decides whether it participates or not in the group, according to its luminosity reading and to the current threshold.

The application uses two timers, one for monitoring the PHOTO sensor and another one for triggering value aggregation. Initially, the application sets both timers.

When the monitoring timer is fired (event `PhotoFired`), the script requests a reading from the luminosity sensor (line 20 in Listing 2). When the light reading is ready, event `PhotoDone` is signaled (line 23) and the resulting value is compared to parameter `Photolevel` and the `setGroup` instruction is used to include or remove the node from the group (lines 27 and 30).

When the aggregation timer is fired (event `TempFired`), the node verifies whether it is the group coordinator, and if so, starts aggregation (line 6). When aggregation is completed (event `AggDone`), the coordinator sends the result (configured to be the average value in the declarations section) to the base station (line 13).

Listing 1: Parametrized App. - Declarations

```
1  // IPSNAppC1 – Dynamic aggregation
2  #StructsDef
3    msg1Def:                  // Msg struct def
4      U16    count
5      U16    result
6  #VarsSpace
7    msg1Def msg1              // Msg struct data
8    u16     PhotoValue        // Sensor data
9    u16     PhotoLevel        // Level Param
10 #SensorsSpace
11   PHOTO PhotoDone PhotoValue
12 #MsgsSpace
13   1 dummy msg1              // Message def
14 #TimersSpace
15   1 PhotoFired 20           // Periodic Timer
16   2 TempFired 60            // Periodic Timer
17 #GroupSpace
18   1 0 3 true off 0          // Group def
19 #AggregSpace
20                             // Aggregation def
21   1 AggDone 1 TEMP u16 avg gte 0
```

This application requires two parameters from the user: variable `Photolevel` indicates the threshold level for a point to be considered well-lighted and the `CoordId` variable indicates the ID of the coordinator node.

Listing 2: Parametrized App. - Procedures

```
#FunctionsArea
main:                       // Init proc
  setTimer PERIODIC 1       // Start timer 1
  setTimer PERIODIC 2       // Start timer 2
  end
TempFired:                  // Timer fired
  compare EQ MOTEID 2
  if U16 StartAgg // If ID=10,call startAgg
  end
StartAgg:                   // Start aggregation
  aggreg 1
  end
AggDone:                    // Aggregation ended
  set msg1.count ag1.count
  set msg1.result ag1.value.U16
  sendMsg BS 1 msg1         // Send value to BS
  end
dummy:
  end
PhotoFired:                 // Timer fired
  readSensor PHOTO
  end
PhotoDone:                  // Photo sensir done
  compare GT  PhotoValue PhotoLevel
  ifelse U16 InGroup OutGroup
  end
InGroup:                    // Group IN
  setGroup IN_GROUP 1
  end
OutGroup:                   // Group OUT
  setGroup OUT_GROUP 1
  end
#EndPoint
```

The declaration section uses the following parameters from binary entities: timer period (Timer), maximum hops on group definition (Group), sensortype and aggregation function (Aggregation).

A typical use of reconfiguration in this application would be to redefine threshold and timer periodicity. Using 16-bit values for each of these, it would be possible to disseminate their reconfiguration with a single message.

We can derive the maximum capacity for a reconfiguration message used in the reconfiguration tool presented in Section 3.4. Table 5 presents some examples of the number of parameters that is supported by a reconfiguration message as a function of the size of parameters and of their distribution in memory. We consider that the size and distribution are homogeneous among all the parameters in one same message (It would, however, be possible to combine different sizes and memory distributions, providing the 24-byte limit is respected).

The largest possible number of parameters is 22, obtained when using a sequential distribution with 8-bit parameters. Greater capacities occur when parameters are allocated sequentially in memory. So that the user can take advantage of this, WDvm allocates parameters in the same order that the user defines them in the program. The only excep-

Table 5: Total parameters per message

Distribution	Parameter Size		
	8 bits	16 bits	32 bits
Sequential	22	11	5
Non-sequential	8	6	4

Unit: Parameters

tion is for parameters of runtime components, because these values are allocated in the configuration section and are interleaved with other configuration data.

If necessary, more than one message can be used to update parameters. However, because the reconfiguration message need contain only the parameters that undergo modifications, it is often possible to work with a single message.

This example intends to illustrate some of the flexibility of parameters in WDvm. The programmer can use reconfiguration messages to modify both parameters that he defined in his own application and that are predefined in WDvm's binary modules.

4.3 Abstraction Boundary

In this section we illustrate the trade-offs of choosing different abstraction boundaries for the virtual machine. We compare two versions of an application that aggregates values, written over different sets of components.

Our example application periodically calculates the average value read by sensors in a group of nodes. A node acting as *group leader* requests values from its neighbors, and when it receives their answers calculates the average and sends the result to the base station. Both variants shown here use WDvm runtime components for group communication and leader election.

The first version is shorter because it uses a runtime component for aggregation. The second, longer, version, implements aggregation in WDvm's scripting language. This version uses a different WDvm image, compiled without the aggregation component. We can thus compare the gains and losses in memory usage and in application code.

4.3.1 Short version

Listing 3 presents the declaration section and listing 4 presents the code for the short variant of the aggregation application. The program maintains a periodic timer. When this timer generates an event (`PeriodFired`), the program verifies whether it is the group leader. Leader election is parameterized in the group definition and executed by a runtime component. If the node is the group leader, it executes instruction *aggreg 1* to trigger aggregation. The aggregation procedure is performed by another runtime component. When aggregation is complete and the result is available, this component generates an event (`AggDone`). The main program then prepares a message and sends it to the base station.

4.3.2 Long version

Listing 5 presents the declaration section and listing 6 presents the code for the long version of our application. The program structure remain the same, but now aggregation

Listing 3: Short Version (Declarations)

```
// IPSNAppB1 - Periodic aggregation
// Using runtime aggregation component
#StructsDef
  msg1Def:                  // Msg struct def
    U16   count
    U16   result
#VarsSpace
  msg1Def msg1              // Msg struct data
  u16     PhotoValue        // Sensor data
#MsgsSpace
  1 dummy msg1              // Message def
#TimersSpace
  1 PeriodFired 10          // Periodic Timer
#GroupSpace
  1 0 2 true active 0       // Group def
#AggregSpace
                            // Aggregation def
  1 AggDone 1 PHOTO u16 avg gte 0
```

Listing 4: Short Version - (Procedures)

```
#FunctionsArea
main:                       // Init proc
  div MOTEID 10             // Find Group ID
  cast U8 RESULT.U16
  set gr1.param RESULT.U8
  setTimer PERIODIC 1       // Start timer 1
  end
PeriodFired:                // Timer fired
  compare EQ MOTEID gr1.leader
  if U16 StartAgg // If Leader,call startAgg
  end
StartAgg:                   //Starts aggregation
  aggreg 1
  end
AggDone:                    // Aggregation ended
  set msg1.count ag1.count
  set msg1.result ag1.value.U16
  sendMsg BS 1 msg1         // Send value to BS
  end
dummy:
  end
#EndPoint
```

is performed by the script. The aggregation protocol uses two types of messages. The first is used by the group leader to request sensor readings from the other group members. The second type of message is used by each non-leader node to answer the leader's request.

When the periodic timer fires, the node checks whether it is the leader, and if so, it now generates internal event `StartAgg`; the scripts reacts by initializing the total of received answers, activating a timeout Timer, and sending the request message to group members. When an answer message is received, event `RecRetGr1` is generated. The program reacts

by incrementing the total of received answers and adding the received value to variable `Total`. When the aggregation timeout triggers event `TimeoutFired`, the program computes the average and sends it in a message to the base station.

The listing also includes the program for group members that are not leaders. Event `RecGr1` indicates that a request message has been received. At this point, the program request a reading from the sensor. When this is done (event `SensorDone`), the node sends the new value to the group leader.

Listing 5: Complex Version (Declarations)

```
1  // IPSNAppB2 - Periodic aggregation
2  // Using pure script
3  #StructsDef
4    msg1Def:                  // Msg struct def
5      U16    count
6      U16    value
7  #VarsSpace
8    u16      Count
9    u32      Total
10   msg1Def  msg1              // Msg struct data
11   u16      PhotoValue        // Sensor data
12 #SensorsSpace
13   PHOTO SensorDone  PhotoValue
14 #MsgsSpace
15   1 RecGr1   msg1            // Message 1 def
16   2 RecRetGr1 msg1           // Message 2 def
17 #TimersSpace
18   1 PeriodFired 10           // Periodic Timer
19   2 TimeoutFired 2           // Time-Out Timer
20 #GroupSpace
21   1 0 2 true active 0        // Group def
```

4.4 Results

Table 6 presents the size, in bytes, and the number of messages necessary for the dissemination of the short and long versions. Table 7 presents the amount of memory used for code (ROM) and data (RAM), also for both versions. As a basis for comparison, we also show in this table the number of messages that would be necessary to disseminate the binary code.

Table 6: Applications byte size

	App version		
	Simple(a)	**Complex**(b)	**($b-a$)**
Code size	130 [6]	232 [10]	102 [5]

Units: Bytes [Messages]

The long version of our aggregation application occupies 232 bytes, 102 more than the short one. This represents an increase of 66% in the number of messages needed. If we were to transfer remotely the 5,020 bytes of binary code for the components we removed, we would need 210 messages.

The removal of the aggregation code from the WDvm binary image freed 157 bytes of

Listing 6: Complex Version (Procedures)

```
#FunctionsArea
main:                    // Init proc
  div MOTEID 10          // Find Group ID
  cast U8 RESULT.U16
  set gr1.param RESULT.U8
  setTimer PERIODIC 1    // Start timer 1
  end
PeriodFired:             // Timer fired
  compare EQ MOTEID gr1.leader
  if U16 StartAgg // If Leader, call startAgg
  end
StartAgg:                // Start aggregation
  set Count 0
  set Total 0
  set msg1.value MOTEID
  setTimer ONESHOT 2     // Start timeout
  sendMsg GR 1 msg1 1    // Request values
  readSensor PHOTO       // Read local sensor
  end
RecRetGr1:               // Save reads
  inc Count
  cast U32 msg1.value
  add Total RESULT.U32
  set Total RESULT.U32
  end
TimeoutFired:            // Final Aggreg
  set msg1.count Count
  cast U32 Count
  div Total RESULT.U32   // avg()
  cast u16 RESULT.U32
  set msg1.value RESULT.U16
  sendMsg BS 1 msg1      // Send value to BS
  end
RecGr1:                  // Receive request
  readSensor PHOTO       // Read local sensor
  end
SensorDone:              //Proc. local sensor
  compare EQ MOTEID gr1.leader
  ifelse U8 SensorLeader SensorOthers
  end
SensorLeader:            // Leader sensor
  inc Count
  cast U32 PhotoValue
  add Total RESULT.U32
  set Total RESULT.U32
  end
SensorOthers:            // Others sensor
  set RESULT.U16 msg1.value
  set msg1.value PhotoValue
  sendMsg GRND 1 msg1 1 RESULT.U16
  end
#EndPoint
```

RAM. This released space could be used to allow more complex programs to be written

Table 7: WDvm runtime memory size

	Embedded Aggregation		
	with(a)	**without**(b)	**($a - b$)**
ROM	53.2198 [2217]	48178 [2008]	5020 [210]
RAM	3560	3403	157

Units: Bytes [Messages] or Bytes

for the VM. The support for aggregation included in the default binary image of WDvm is obviously much more complex than the aggregation performed by our script. It allows different aggregation operations to take place simultaneously and provides a choice of reduction operations. However, it can be the case that the programmer knows that this richer support will not be needed in a given application domain.

This example, although simple, shows concretely the trade-off between functionality in the VM runtime and memory usage. A richer set of components in the runtime means that script can be simpler to write and smaller to send over the network, but on the other hand leaves less memory available for unforeseen needs. Our goal with WDvm is to allow the programmer to explore a range of abstraction levels. For mature projects, possibly most of the functionality can be installed as binary runtime components, leaving only fine adjustments to be done through parameters. For more experimental settings, the programmer could initially install only a very basic set of components on the motes and remotely program them with more complex scripts.

5 Final Remarks

In this report, we discussed WDvm, a virtual machine for mote devices used in wireless sensor networks. WDvm's goal is to allow the programmer to explore different abstraction boundaries between the binary image, containing the set of runtime components initially installed on the mote, and the virtual machine script, which can be remotely updated. This goal is in line with that of ASVM [2] and DVM [1], but in WDvm we investigate a seamless transition between parameter-based and script-based reconfiguration. The promotion of parameters and variables to language-level elements allowed us to integrate the VM programs with the runtime components. A script can vary from simple assignments to parameters to quite sophisticated code containing decisions based on values made available by the runtime. This allows different classes of users to rely on the same set of tools.

Our practical experiments showed that, as expected, the virtual machine extracts a significant overhead for processing. However, we were also able to put his overhead in some perspective when analyzing an I/O-intensive application, showing that the cost of external actions and events can dwarf the differences in processing costs. A systems such as WDvm can be used to prototype applications in laboratories or testbeds, tuning the amount of flexibility and desired performance needed for a specific applications domain.

As future work, we intend to revisit the original classes of applications that inspired our set of runtime components and evaluate WDvm's use on them.

We understand that the WDvm scripting language is not at this point an attractive language for programmer use. The operation codes and parameter definition conventions are designed to be mapped directly into the intermediary byte language used by the VM and do not make easy reading or writing. There are two directions on which we intend

to work to enhance usability. In the first place, this first version of the scripting language had the main intention of investigating whether the virtual machine approach would be worthwhile for limited platforms such as the MicaZ. Because we think our results are quite encouraging, we now intend to enhance our compiler, allowing the programmer to write tractable scripts. In the second place, this work is part of a larger project in which we are interested in alternatives for *macroprogramming* [9,12]. Our goal in the future is to use the language we describe in this report as the target language for a macroprogramming compiler.

6 Acknowledgments

This work has been partially supported by CNPq - Conselho Nacional de Desenvolvimento e Pesquisa and by FAPERJ - Fundação de Amparo a Pesquisa do Estado do Rio de Janeiro (grant E-26/110.320/2011).

References

[1] BALANI, R.; HAN, C.-C.; RENGASWAMY, R. K.; TSIGKOGIANNIS, I. ; SRIVASTAVA, M.. **Multi-level software reconfiguration for sensor networks**. In: PROCEEDINGS OF THE 6TH ACM & IEEE INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, EMSOFT '06, p. 112–121, New York, NY, USA, 2006. ACM.

[2] LEVIS, P.; GAY, D. ; CULLER, D.. **Active sensor networks**. In: PROCEEDINGS OF THE 2ND CONFERENCE ON SYMPOSIUM ON NETWORKED SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 2, NSDI'05, p. 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[3] STEINE, M.; VIET NGO, C.; SERNA OLIVER, R.; GEILEN, M.; BASTEN, T.; FOHLER, G. ; DECOTIGNIE, J.-D.. **Proactive reconfiguration of wireless sensor networks**. In: PROCEEDINGS OF THE 14TH ACM INTERNATIONAL CONFERENCE ON MODELING, ANALYSIS AND SIMULATION OF WIRELESS AND MOBILE SYSTEMS, MSWiM '11, p. 31–40, New York, NY, USA, 2011. ACM.

[4] KOGEKAR, S.; NEEMA, S.; EAMES, B.; KOUTSOUKOS, X.; LEDECZI, A. ; MAROTI, M.. **Constraint-guided dynamic reconfiguration in sensor networks**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL SYMPOSIUM ON INFORMATION PROCESSING IN SENSOR NETWORKS, IPSN '04, p. 379–387, New York, NY, USA, 2004. ACM.

[5] GREENSTEIN, B.; PESTEREV, A.; MAR, C.; KOHLER, E.; JUDY, J.; FARSHCHI, S. ; ESTRIN, D.. **Collecting high-rate data over low-rate sensor network radios**. Technical report, University of California eScholarship Repository [http://repositories.cdlib.org/cgi/oai2.cgi] (United States), 2007.

[6] HUI, J. W.; CULLER, D.. **The dynamic behavior of a data dissemination protocol for network programming at scale**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, SenSys '04, p. 81–94, New York, NY, USA, 2004. ACM.

[7] HAN, C.-C.; KUMAR, R.; SHEA, R.; KOHLER, E. ; SRIVASTAVA, M.. **A dynamic operating system for sensor nodes**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, MobiSys '05, p. 163–176, New York, NY, USA, 2005. ACM.

[8] LEVIS, P.; CULLER, D.. **Maté: a tiny virtual machine for sensor networks**. In: ASPLOS-X: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, p. 85–95, New York, NY, USA, 2002. ACM.

[9] NEWTON, R.; MORRISETT, G. ; WELSH, M.. **The regiment macroprogramming system**. In: IPSN '07: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING IN SENSOR NETWORKS, p. 489–498, New York, NY, USA, 2007. ACM.

[10] NEWTON, R.; WELSH, M.. **Region streams: functional macroprogramming for sensor networks**. In: DMSN '04: PROCEEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON DATA MANAGEMENT FOR SENSOR NETWORKS, p. 78–87, New York, NY, USA, 2004. ACM.

[11] KOTHARI, N.; GUMMADI, R.; MILLSTEIN, T. ; GOVINDAN, R.. **Reliable and efficient programming abstractions for wireless sensor networks**. PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 200–210, 2007.

[12] AWAN, A.; JAGANNATHAN, S. ; GRAMA, A.. **Macroprogramming heterogeneous sensor networks using cosmos**. In: PROCEEDINGS OF THE 2ND ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2007, EuroSys '07, p. 159–172, New York, NY, USA, 2007. ACM.

[13] MADDEN, S. R.; FRANKLIN, M. J.; HELLERSTEIN, J. M. ; HONG, W.. **Tinydb: an acquisitional query processing system for sensor networks**. ACM Transactions on Database Systems, 30(1):122–173, 2005.

[14] CERVANTES, H.; DONSEZ, D. ; TOUSEAU, L.. **An architecture description language for dynamic sensor-based applications**. In: CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE, 2008. CCNC 2008. 5TH IEEE, p. 147–151, Jan. 2008.

[15] BAKSHI, A.; PRASANNA, V. K.; REICH, J. ; LARNER, D.. **The abstract task graph: a methodology for architecture-independent programming of networked sensor systems**. In: PROCEEDINGS OF THE 2005 WORKSHOP ON END-TO-END, SENSE-AND-RESPOND SYSTEMS, APPLICATIONS AND SERVICES, EESR '05, p. 19–24, Berkeley, CA, USA, 2005. USENIX Association.

[16] GNAWALI, O.; FONSECA, R.; JAMIESON, K.; MOSS, D. ; LEVIS, P.. **Collection tree protocol**. In: PROCEEDINGS OF THE 7TH ACM CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, SenSys '09, p. 1–14, New York, NY, USA, 2009. ACM.

[17] CROSSBOW. **Micaz datasheet**. Product folder, 2004.

[18] TITZER, B. L.; LEE, D. K. ; PALSBERG, J.. **Avrora: scalable sensor network simulation with precise timing**. In: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM ON INFORMATION PROCESSING IN SENSOR NETWORKS, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.