Terra: Flexibility and safety in Wireless Sensor Networks

ADRIANO BRANCO, Pontifícia Universidade Católica do Rio de Janeiro
FRANCISCO SANT'ANNA, Pontifícia Universidade Católica do Rio de Janeiro
ROBERTO IERUSALIMSCHY, Pontifícia Universidade Católica do Rio de Janeiro
NOEMI RODRIGUEZ, Pontifícia Universidade Católica do Rio de Janeiro
SILVANA ROSSETTO, Universidade Federal do Rio de Janeiro

Terra is a system for programming wireless sensor network (WSN) applications. It combines the use of configurable virtual machines with a reactive scripting language which can be statically analysed to avoid unbounded execution and memory conflicts. This approach allows the flexibility of remotely uploading code on motes to be combined with a set of guarantees for the programmer. The choice of the specific set of components in a virtual machine configuration defines the abstraction level seen by the application script. We describe a specific component library built for Terra, which we designed taking into account the functionality commonly needed in WSN applications — typically for sense and control. We also discuss the programming environment resulting from the combination of a statically analyzable scripting language with this library of components. Finally, we evaluate Terra by measuring its overhead in a basic application and discussing its use and cost in a typical monitoring WSN scenario.

Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization-Computer-Communication Networks**]: Distributed Systems; D.3.4 [**Software-Programming Languages**]: Processors

General Terms: Design, Languages, Verification, Experimentation

Additional Key Words and Phrases: WSN Wireless Sensor Networks, Virtual Machine, Reactive Programming, Safety

ACM Reference Format:
Adriano Branco, Francisco Sant'Anna, Roberto Ierusalimschy, Noemi Rodriguez, and Silvana Rossetto, 2015. Terra: Flexibility and safety in Wireless Sensor Networks. *ACM Trans. Sensor Netw.* 1, 1, Article 11 (January 2015), 28 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Programming a wireless sensor network (WSN) remains a challenge. WSNs are typically composed by computing devices (*motes*) that communicate via radio and rely on batteries for energy. Although a whole range of microcontrollers can be used in this setting, it is very common, due to cost restrictions and scale of usage, to employ units with very limited memory and computing resources. This scarcity of resources, along with the event-oriented nature of applications and the need for coordination among large numbers of nodes, make programming applications a difficult and error-prone task [Awan et al. 2007; Kothari et al. 2007; Mottola and Picco 2011].

Author's addresses: A. Branco and F. Sant'Anna and R. Ierusalimschy and N. Rodriguez, Department of Informatics, Pontifícia Universidade Católica do Rio de Janeiro; S. Rossetto, Department of Computer Science, Universidade Federal do Rio de Janeiro.

DOI:http://dx.doi.org/10.1145/0000000.0000000

It is also often the case that the user must reprogram sensor network nodes after they are in place. This is hard to do physically, because in most cases it is difficult to recover the motes from the position in which they are installed. The obvious solution is to do the updates through radio messages; however, transferring complete binaries over radio can lead to high energy consumption, and is thus undesirable.

On the other hand, because of their restricted resources and deployment characteristics, a given sensor network is normally used for a single category of application, such as environment control or building security, even if the application itself evolves over time. This indicates that a small set of coordination and processing patterns can support all of the applications that a sensor network must run along its lifetime.

Taking this into consideration, researchers have in the past proposed the idea of *application-specific virtual machines (ASVMs)* [Levis et al. 2005], combining domain-specific languages with virtual machines. A convenient ASVM can make programs very concise, imposing very little overhead on their interpretation and on their transmission over the network. In his work with the virtual machine Maté, Levis experimented with three different domain-specific languages, each of them translated to code interpretable by the virtual machine using specific compilers. However, developing a separate language for each new application niche is costly. Areas with small user bases will seldom be able to harness the resources for designing and implementing a new programming language. Besides, even in different application areas, WSN applications tend to follow restricted patterns of behavior and interaction, such as creating and maintaining groups of nodes to collect data and make local decisions, or forwarding the collected data to a sink node.

We believe that WSN programming environments can benefit from commonality not only inside a single application area. Programming patterns such as collecting values to a base station or broadcasting them to the whole network are recurrent in different application areas, with variations regarding issues of reliability or security. So we discuss an alternative approach to ASVMs, in which common programming patterns are designed and implemented separately as libraries of components. These components may be combined as needed, creating customized virtual machines with abstractions provided by the component interfaces. Instead of using application-specific programming languages, we propose the use of a scripting language with general support for the problems encountered in WSNs. Scripting languages enforce a programming model that glues components together to create powerful applications in a few lines of code [Ousterhout 1998]. This makes them suitable for creating programs that benefit from pre-defined sets of components and that can be easily sent over the network.

In this paper, we describe the Terra programming system, which combines customized virtual machines with tCéu, a reimplementation of the Céu [Sant'Anna et al. 2013] programming language. Figure 1 presents the three basic elements of Terra – the language tCéu, a set of customised pre-built components, and an embedded virtual-machine engine tVM.

Céu is a scripting language that addresses general issues related to programming WSN motes. Its synchronous and reactive nature facilitates the handling of events, and it provides concurrency guarantees that simplify interactions with multiple external sources. We believe it is fundamental to push as many safety guarantees to compile time as possible, specially in what relates to shared-memory issues and to responsiveness. In Terra, applications are written in tCéu and are compiled to tVM virtual-machine code, which interacts with the environment through Terra's virtual machine components. tVM is specifically designed for the tCéu execution model, while Terra components represent a second level of programming and provide abstractions for specific interaction patterns, such as routing and data collection. Terra provides a library of components that implement typical programming patterns. When building

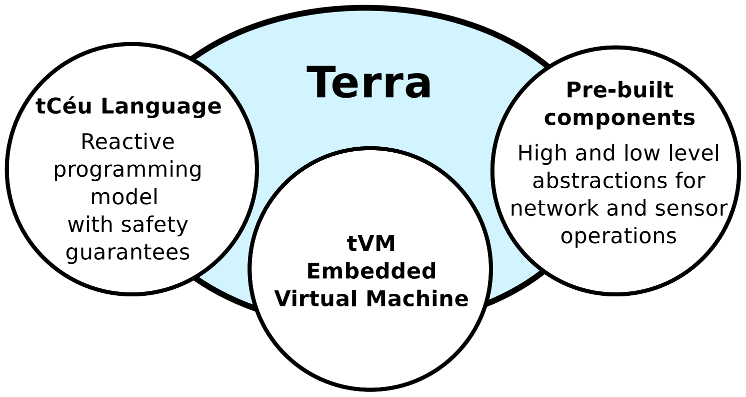
Terra
tCéu Language
Reactive programming model with safety guarantees
tVM Embedded Virtual Machine
Pre-built components
High and low level abstractions for network and sensor operations

Fig. 1. Terra system basic elements.

a tVM for a given wireless sensor network, the programmer can choose whether or not to include each Terra component in the configuration, setting different boundaries for the provided abstractions. New components can also be added.

Figure 2 shows the Terra application lifecycle. The tCéu program is compiled and checked statically in a conventional computer, generating code for the virtual machine (the bytecode). This bytecode must then be transferred (possibly over a wireless connection) to the sensor node on which it will run. The current tVM implementation and its customized components are built over TinyOS [Levis et al. 2004]. The only part of tCéu scripts that escape static analysis are calls to components provided by tVM, which are encapsulated in modules and have been extensively tested beforehand.

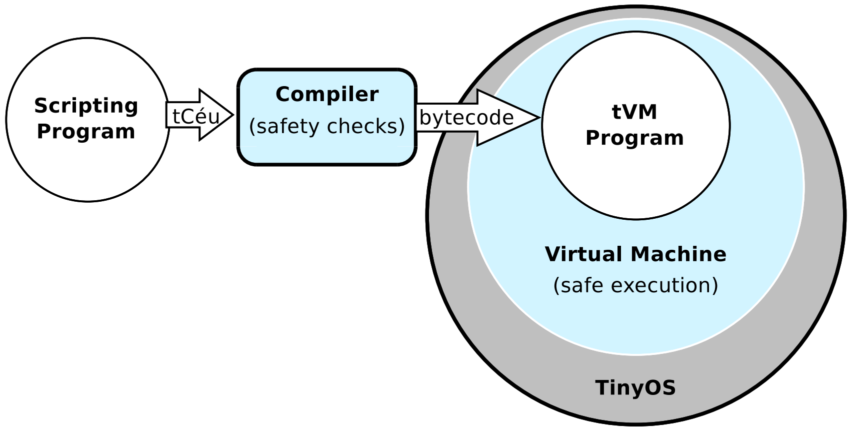

Fig. 2. Terra application lifecycle: compilation and execution.

The remainder of the paper is organized as follows. Section 2 presents the Terra programming environment, discussing its component library as well as its scripting language and their interaction. Section 3 describes the implementation of the Terra Virtual Machine tVM. In Section 4, we discuss some example applications and evaluate their memory and energy usage. Section 5 presents related work. Finally, Section 6 contains some closing remarks.

2. THE TERRA SYSTEM

The Terra system can be built in different coonfigurations. A basic Terra deployment consists of a specific version of the tVM virtual machine and a corresponding definition file. The definition file contains the interfaces of the components included in the virtual machine and must be included in the tCéu application built for this VM. The binary code of the VM must be loaded (over a serial interface, by USB) to all network nodes before they are physically distributed over the working area.

The Terra compiler converts the tCéu code into virtual machine bytecode taking the definition file as part of its input, so as to validate and appropriately translate component usage. The Terra system also includes a tool that disseminates the bytecode to WSN nodes over the radio.

Application scripts are written as a series of *reactions* to events. tCéu supports multiple lines of execution called *trails*. However, the structure of tCéu programs is not that of typical event handlers: trails can contain blocking await statements. Input events are broadcast to all awaiting trails, i.e., at any point in execution, each of a program's trails is either reacting to the current event or awaiting another event; in other words, trails are always *synchronized* at the current (and single) event.

Both Terra components and the tCéu script may trigger and handle events. However, a single event must be handled either by tCéu or by a component, i.e., there can not be handlers for one event defined in both environments. Events are triggered in the tCéu code by using the `emit` keyword, and handled through `await`. Events thus integrate the scripting language with the components that are made available by tVM. *Output events* are emmitted from the tCéu script to be handled by Terra components, while *input events* are triggered by Terra components and awaited by the tCéu script. The tCéu script may also interact with Terra's components through *system calls*, which are used for initialization and configuration of components.

As an introductory example, the Terra program in Figure 3 repeatedly reads the light and temperature sensors and tests whether either reading has risen above a predefined limit. If this happens, the program turns on a led to indicate the anomalous condition.

The `par/and` construct in line 3 spawns two trails in parallel (lines 4–5 and 7–8), and will finish execution only when both of them terminate. Because both trails contain `await` statements, this will happen only when both the PHOTO and TEMP events occur. The events triggered by the program (REQ_PHOTO, REQ_TEMP, and LED0), and the events awaited by it (PHOTO and TEMP) are part of the interface provided by the virtual machine. The program also illustrates the provision of timers as first-class elements: in line 13, it uses a timer to ensure the led is kept on for one minute, before turning it off and proceeding to the next step in the loop.

This program uses output events REQ_PHOTO, REQ_TEMP, and LED0, which are handled by Terra components. REQ_PHOTO and REQ_TEMP trigger readings that return their results by signalling input events PHOTO and TEMP, which also carry the values that were read. Output event LED0 takes one argument (constants ON or OFF), and does not trigger any input event.

The par/and composition rejoins when all of its trails terminates. tCéu also supports par/or compositions, which rejoin when any of the spawned trails terminates (see Section 2.3).

2.1. Terra Components

The components included in a specific virtual machine define the interface between the application script and the environment, and thus determine the abstraction level at which the script programmer will work. Terra offers a library of components that

```
var ushort tValue,pValue;
loop do
  par/and do
    emit REQ_PHOTO();
    pValue=await PHOTO;
  with
    emit REQ_TEMP();
    tValue=await TEMP;
  end
  if pValue > 200 or tValue > 300 then
     emit LED0(ON);
  end
  await 1min;
  emit LED0(OFF);
end
```

Fig. 3. A simple example in Terra.

can be included or not in a specific virtual machine. As far as possible, these components are parameterized for genericity. New components can also be included by programmer-savvy users to create abstractions for new programming patterns, but our goal is to offer a set of components that is sufficient for a range of common applications. This is feasible because most applications for sensor networks are variations of a basic monitoring and control pattern. Because processing resources are limited, these variations typically involve only basic operations for accessing sensors and actuators and coordination among motes.

In order to choose a set of general-use and parameterized components that can be useful in a wide range of different applications, we analysed some WSN applications described in the literature. [Newton et al. 2007; Cervantes et al. 2008; Kothari et al. 2007]. Another important source was the research on macroprogramming (network-wide programming), which in fact was one of our early motivations for this work. We considered proposals for facilitating programming in WSNs with restricted resources [Newton et al. 2007; Newton and Welsh 2004; Kothari et al. 2007; Awan et al. 2007; Madden et al. 2005; Cervantes et al. 2008; Bakshi et al. 2005]. A set of WSN applications was used to validate the functionalities and parametrizations chosen for each component. As the result of this work [Branco 2011], we developed a set of components organized in four areas:

(1) communication — support for radio communication among sensor nodes;
(2) group management — support for group creation and other control operations;
(3) aggregation — support for information collection and synthesis inside a group;
(4) local operations – support for accessing sensors and actuators.

The next sections discuss the components in each of these areas.

2.1.1. Communication. The *Communication* component provides the basic send/receive primitives to exchange radio messages among sensor nodes. Furthermore, it provides specific protocols for message routing from sensor nodes to a base station (the WSN root node) and for dissemination of parameters or new applications from a base station to sensor nodes.

Two types of messages can be used by the application developer. The first one allows exchanging messages among nodes in the same group by using broadcast or unicast dissemination modes. Output event SEND_GR sends a message either to all nodes in a group or to a specific node. When a message is received by a node, this is signalled to the script code through a new REC_GR input event. In the current implementation of the Communication component, messages are routed only within the spatial limits of their group. Unicast messages are typically used to reply to a request made by another node of the same group. The second type of message is used for the specific case in which a node needs to send a message directly to the base station. In this case, the script emits output event SEND_BS. Terra currently implements the SEND_BS event using the TinyOS CTP-Collection Tree protocol [Gnawali et al. 2009].

To provide delivery guarantees (no loss or duplication) for unicast messages, the Communication component implements a confirmation mechanism. When the component is configured with this option, the application does not need to deal explicitly with message retransmissions and duplications.

2.1.2. Grouping. Because WSN applications frequently involve large numbers of node, organizing nodes into groups is one of the basic tasks in programming these applications. The *Group Management* component allows for the simultaneous existence of different network partitions, and is based on identifiers maintained at each node. These identifiers may be initialized statically or dynamically. Messages sent inside a group carry the group identifier and are sent using a flooding protocol with a maximum number of hops (which can also be configured). At each node, such a message is delivered to the application only if the node is currently in the destination group.

This basic module allows the program developer to implement several alternative group structures. As an example, in order to broadcast a message one can initialize all nodes with the same group identifier. In this case, the maximum number of hops can be used to define the reachable range of the group. Another example would be the creation of dynamic groups: the group identifier can be defined by the current node' state, for instance based on the value read from a sensor.

The *Group Management* component also provides leader election. When this option is selected (through a parameter), nodes in a group transparently send queries to locate the leader. In the case when a leader has not yet been defined, a new election is started. The current implementation of this component always chooses the node with the largest remaining battery charge in each group. The script running on each node can also define the node's behavior during the election process, for instance declining to participate in the procedure.

Figure 4 presents an example program that uses group communication facilities.

Line 2 invokes system call `grNew()` which allows the current node to join a new group. In this simple case, all nodes will be included in a single group. The second and third arguments of `grNew()` define constant group identifiers.The fourth argument defines the maximum range in hops. The fifth argument initializes the "active" flag as `TRUE`. The next arguments define an "election off" (OFF) state with node zero as the leader (not used in this case).

Terra maintains all the configuration parameters of a group in a data structure that can be accessed by the application script. In Figure 4, the `gr1` variable (defined in line 1) is used for that. The script can modify these values at any time.

In the program of Figure 4, node 2 periodically sends a counter value to its neighbours (lines 7–13). Each neighbouring node shows the three less significant bits of the received counter on its leds (lines 15–18). As discussed before, the system call `grNew()` (lines 1–2) makes each node join a group described in the structure assigned to `gr1`. A message structure — `countMsg` — is defined in line 4. In this case, we use the first message field value `val1` to send the counter value. The send command at line 11 sends the `countMsg` message structure to all neighbours defined in the `gr1` group. The `await`

```
 1: var group gr1;
 2: grNew(&gr1,1,1,3,TRUE,OFF,0);
 3:
 4: var msg countMsg;
 5:
 6: if NODE_ID == 2 then
 7:   countMsg.val1 = 0;
 8:   emit LEDS(0xff);
 9:   loop do
10:     countMsg.val1 = countMsg.val1 + 1;
11:     emit SEND_GR(&gr1,&countMsg);
12:     await 1s;
13:   end
14: else
15:   loop do
16:     countMsg = await REC_GR;
17:     emit LEDS(countMsg.val1);
18:   end
19: end
```

Fig. 4. Grouping and communication example in Terra.

command at line 16 waits for a group message and updates its `countMsg` structure with the received data.

2.1.3. Aggregation. The aggregation component provides abstractions for collection and synthesis of data within a group of sensor nodes. Because data aggregation requires the implementation of distributed algorithms for group communication and processing of the values collected by different nodes, higher-level abstractions for this pattern can simplify the development of applications for WSNs.

In the related work, we found different approaches to aggregation. Most of them provide facilities to collect values (group communication algorithm) but leave the task of coding the aggregation operation to the developer.

The *Aggregation* component provided by Terra takes as input the group identifier, the physical quantity to be measured by each sensor node (e.g., temperature, photo) and the aggregation/reduction function to be used. The current implementation of this component provides the following built-in functions: SUM (sum of values), AVG (average values), MAX (maximum value) and MIN (minimum value). In addition, each aggregation operation is also associated with a relational operator ($>, <, <=, >=, ==, ! =$) and a reference value. Besides the resulting value, the aggregation operation also accumulates the number of partial values that tested as true in this criterium. Aggregation operations are performed per group, that is, one aggregate value is produced for each group. The script starts an aggregation operation by emitting the ouput event AGGREG with the specific aggregation identifier as parameter. When the aggregation is completed, this is signalled by the AGGREG_DONE event. As in the Group Management component, Terra maintains all the configuration parameters of aggregation in a data structure that can be modified at any moment.

The program in Figure 5 illustrates the use of the aggregation facilities. In lines 1–2, a new group (`gr1`) is created. A single leader will be automatically elected for that group. At each node, the id of the group's leader will be stored in `gr1.leader`.

In lines 3–4, a new agregation (`agA`) is created by invoking the system call `agNew()`. This agregation will be associated with the `gr1` group (the second argument). The third

```
 1: var group gr1;
 2: grNew(&gr1,1,0,2,TRUE,ACTIVE,0);
 3: var aggreg agA;
 4: agNew(&agA,&gr1,TEMP,AVG,GTE,0);
 5:
 6: var agResult data;
 7: var msg dataMsg;
 8:
 9: loop do
10:   await 10s;
11:   if (NODE_ID == gr1.leader) then
12:     emit AGGREG(&agA);
13:     data = await AGGREG_DONE;
14:     dataMsg.value4 = data.value;
15:     emit SEND_BS(&dataMsg);
16:   end
17: end
```

Fig. 5. Aggregation and communication example in Terra.

and fourth arguments to `agNew()` define the sensor to be read (temperature in this case) and the aggregation operation to be applied (average). The next arguments define a relational operator (`GTE`, for greater then or equal) and the reference value (not used in this case). Line 6 creates the predefined data structure that holds the aggregate value. In lines 11–12, the leader node starts the aggregation operation by triggering the AGGREG output event (`emit AGGREG()`). (Non-leader nodes will transparently react to the messages triggered by the aggregation.) In line 13, the leader node waits for the end of the aggregation and assigns the result to `data`. Next, it assigns this value to the data field in `dataMsg` and, in line 15, sends the message to the base station, illustrating the use of the output event SEND_BS. The use of this event is similar to that of the `sendgroup` command, but in this case it is not necessary to indicate a group id.

2.1.4. Local Operations. This set of operations comprises operations to read sensors and residual energy battery, define led's configuration and access input and output devices of the microcontroller. Terra encapsulates all these operations in a component called *Local Operations* providing them as output events. Timers, on the other hand, are handled directly by the tCéu language with the `await <time>` command. Our first example (Figure 3) illustrated the use of sensors, leds, and timers.

2.2. Customizing new VMs

Although Terra has a set of ready-made components which support the basic interaction patterns in typical sensing applications, the idea is that expert programmers can create components for new programming patterns so that the combination of tCéu script with available events and system calls create specific flavors of Terra.

In the current implementation, new components must be programmed in nesC. Terra programs include a configuration file that defines the script/component interface. This file must define all events that can be either triggered or handled by tVM components, as well as commands to create the constants and data structures used by these components. Any modification to tVM components requires that a new runtime be loaded on the motes. This is, in general, not easily done by radio, so the idea is that motes should be installed with runtime support for current and foreseen needs. This isn't a big issue considering that possible functionalities depend on the characteristics of hardware and on the network topology already defined in the deployment phase. For example, there will never be the need for new sensor components, because it is not feasible to install new hardware after the network is deployed. The same idea is roughly true for routing protocols: the protocols selected for the installed tVM will be the ones that are adequate for the network topology and density, which will typically not change. We believe that most changes unrelated to hardware will be possible to implement either with the scripting language or using a different parameter in an existing component.

The creation of new components allows expert programmers to design VMs that offer event interfaces at higher abstraction levels than the basic Terra interface. In another direction, it is also possible to lower the abstraction level of the tVM, removing components, and leave more decisions to the script. This is useful, for instance, to allow specific applications to decide how they will handle faults or even routing. In the constrained-resources environment of WSN, it is often the case that applications must code their own routing protocols, carrying the performance onus only of the specific features needed for the application. The script can implement a specific communication protocol using only very basic communication primitives from the Terra components. The macro system can again be used to allow other parts of the application to use the high-level protocol as if it were defined by components. This allows the programmer more flexibility in experimenting an application with different communication and fault-handling services, which may possibly later become tVM components.

2.3. Programming with Céu in Terra

Céu was originally developed at PUC-Rio as a compiled language, and runs on Arduino, TinyOS, and SDL [1]. We chose Céu as Terra's scripting language because of its reactive nature and its support for high-level control primitives and compile-time safety guarantees. Céu provides a parallel construct and a blocking `await` statement that allows programs to handle multiple events at the same time. In contrast with standard split-phase event-based systems, such as *nesC* and *Contiki* [Gay et al. 2003; Dunkels et al. 2004], Céu can keep sequential and separate lines of execution (trails) for each activity in the program. Furthermore, the extra support for parallelism provides precise information about the program control flow to the Céu compiler, enabling a number of static safety guarantees, such as race-free shared-memory [Sant'Anna et al. 2013].

Trails in Céu are guided by reactions to the environment, which, in the case of Terra, is represented by the tVM components introduced in Section 2.1. tCéu and components in the VM communicate through *system calls*, *output events* and *input events*. System calls and output events cross the script boundary towards the VM components, while input events go in the opposite direction, crossing the VM boundary towards the script. System calls behave like normal function calls and are used to initialize and configure the components (e.g. `grNew()`). The invocation of a system call is synchronous: control returns to the script when the system call finsihes execution. Programmers writing new system calls should make sure their implementation does not block (this restriction is compatible with the intended use of system calls). Output events are used to request asynchronous operations to run in the components (e.g. `emit REQ_TEMP();`). Signalling an output event is an asynchronous operation, and returns immediately without blocking the script. Input events, in contrast, cross the VM boundary towards the script and guide its execution through successive reactions, one for each new event. An event occurrence starts a new reaction in the script, awaking all trails awaiting that event (e.g, `await TEMP`).

[1] http://ceu-lang.org/

Programs in Céu are designed by composing blocks of code through sequences, conditionals, loops, and parallelism. The combination of parallelism with standard control flow enables hierarchical compositions, in which self-contained blocks of code can be deployed independently. To illustrate the expressiveness of compositions in Céu, consider the two variations of the structure in Figure 6.

```
loop do                          loop do
  par/and do                       par/or do
    <...>                            <...>
  with                             with
    await 1s;                        await 1s;
  end                              end
end                              end
```

Fig. 6. Compositions in Céu.

In the `par/and` loop variation, the code block in the first trail (represented as `<...>`) is repeated every second at minimum, as the second trail must also terminate to rejoin the `par/and` primitive and restart the loop. In the `par/or` loop variation, if the code block does not terminate within one second, the second trail rejoins the composition (cancelling the first trail) and restarts the loop. These structures represent, respectively, sampling and timeout patterns, which are typically found in WSN applications.

Scripts in Céu follow the synchronous concurrency model, that is, reactions to input events run to completion and never overlap: in order to proceed to the next event, the current event must be completely handled by the script. To ensure that scripts are always reactive to incoming events, the synchronous model relies on the guarantee that a reaction always executes in bounded time. The original Céu compiler statically verifies that programs contain only bounded loops (i.e., loops that contain an `await` statement in every possible execution path [Sant'Anna et al. 2013]). Even though Céu supports multiple lines of execution, accesses to shared memory are safe. Because programs can react to only one component-triggered event at a time, the Céu compiler also performs a flow analysis to detect concurrent accesses [Sant'Anna et al. 2013]: if two accesses to a variable can occur in reactions to the same event and are in parallel trails, then the compiler issues an error message.

For the use of Céu in Terra, we created a new implementation of the language that generates code for tVM. The tCéu language inherits all of the characteristics of Céu discusse above, and its implementation inherited all the safety checks from the original compiler. In the original language, however, any call to C is exempt of verification. In the new implementation, the system calls provided by Terra are the only way to escape this verification. Because only the system calls that are part of component interfaces are available, it is feasible to ensure that these run in bounded time (e.g., do not contain recursive calls and infinite loops).

As a trade-off for safety, the Céu design imposes limitations on language expressiveness; it is not possible to program computationally-intensive operations and hard real-time responsiveness, possibly making it hard to program low level code such as radio protocols [Sant'Anna et al. 2013]. In Terra, the tCéu language is used only as scripting language to glue components written in em nesC/TinyOS. All virtual machine code and low level components rely on the TinyOS architecture.

3. IMPLEMENTATION

The Céu programming language is originally compiled to C and Céu scripts can include chunks of C code. In Terra, we want only the VM components to escape the safety analysis, so we took out the facility to include arbitrary C code, but we did maintain all of Céu's original control structures.

A Terra program is compiled to a bytecode file that can then be disseminated to the network nodes, where it is interpreted by the Terra virtual machine (tVM), which implements the bytecode interpreter, the execution model, the code dissemination service, and some specific customized components.

Our use of Céu required some extensions to the language itself and a number of changes to the compiler in order to support the concept of a cusomizable virtual machine. The main changes, described in next subsections, are the new type system for the scripting language, the integration between scripting language and the customized components, the adaptation of Céu execution model inside the virtual machine, the new tCéu compiler to generate virtual machine bytecode, the virtual machine itself including the bytecode dissemination algorithm. At the end of the section we present the operation process.

3.1. Types

In Céu, data definition and manipulation relies on the use of C. For Terra, we defined a type system based only on integer values with 8, 16, and 32 bits, and array of integers. Pointer types are not allowed for safety reasons. Floating point types are also not included, as they as not typically used in the resource-restricted context of WSNs. The new Terra types are: byte, short, long, ubyte, ushort, and ulong, respectively 8, 16, and 32 bits signed and unsigned types.

We included support for the definition of structures that maintain information used in the interface between the user program and the tVM components. A `regtype` declaration creates new *registers* (again, originally Céu uses embedded C structures.) A register can only have fields that are values of basic types or arrays of basic type. Figure 7 (lines 4–10) shows an example of register declaration and use. We also defined declarations for creating message types. A `packet` declares a new abstract register type which must contain a field of a special type called `payload`. Subsequently, this abstract register type may be used in a register type definition, using a `pktype` declaration. In this declaration the user can specify sub-register fields for the abstract packet register's payload. Figure 7 (lines 12–26) shows an example.

The type system of Terra has simple rules. Assignments of integer values to any integer variable are allowed and, if necessary, automatic type casting occurs. Each automatic type casting generates a compile-time warning. A register value can be assigned only to another identically-typed variable.

The integer assignment rules and warnings are also applied in passing arguments of functions and events. Register arguments are always passed by reference and an additional rule verifies the compatibility between a packet type and a register type. An integer-type variable may be passed by reference using the `&` operand and, in this case, the argument type must be defined using attribute `*`. An argument is the only place where the address operator is allowed. Figure 7 (lines 29–32) shows valid attribution examples of Terra types/variables.

3.2. Integration between script and components

In Céu, the user program may indicate any external (C) events and functions it will access. In the virtual machine approach, we must allow the script to access only events and functions that have been previously embedded in the virtual machine in which it will execute. We have thus decided that, besides input and output events, the tVM components would also provide functions, to allow some interactions to occur in a more natural way than would be the case if the script had always to resort to emitting and requesting events for all of its interactions with the environment.

```
var ushort nodeId; // Simple var
var ushort[5] sensorReads; // Array var

regtype myData with // Register type
  var ubyte sequence;
  var ushort nodeID;
  var ushort sensorValue;
end

var myData sensorData; // Register var

// Abstract register type
packet radioMsg with
  var ubyte msgId;
  var ushort target;
  var payload[20] data; // 20 bytes
end

// Register/packet type
pktype userMsg of radioMsg with
  var ubyte seq;
  var ushort sensorVal;
end

// Register/packet var
var userMsg sendMsg;

// Valid attribution examples
nodeId = 5;
sensorData.sensorValue = sensorReads[0];
sendMsg.target=1;
sendMsg.sensorVal=sensorData.sensorValue;
```

Fig. 7. New Terra type system examples

The virtual machine developer must describe the custom data structures, external events, and functions that the tVM provides in a configuration block to be included file in the user application program. The customized virtual machine and the configuration file must be distributed together to ensure the correct compatibility. Figure 8 shows an example configuration block.

3.3. Execution model

The interpreter is controlled by the tVM Engine. Each task runs to completion in a single-thread model, guaranteeing race-free conditions over application trails and embedded operations. The only exceptions are the interrupt-handlers, which must be isolated in low-level functions.

Terra uses the same execution model of Céu to provide execution guarantees. Basically the application program is broken in execution trails, and each trail has an address as entry point and a `end` opcode at the end. For example, a simple block with a command `await` is broken in two trails. The beginning of the block is the first entry point and the position after the `await` command is the second entry point. The run-

```
config do
  regtype radioMsg with
    var ubyte sequence;
    var ushort value;
  end

  output void   REQ_TEMP     void     1;
  output void   SEND_SENSOR  radioMsg 2;

  input  ushort TEMP         void     1;

  function ubyte getNodeId();
  function ubyte queuePut(radioMsg);
end
```

Fig. 8. A simple configuration block example.

time maintains a set of slots for execution entry points. When an event is received, the engine scans all slots to execute, one by one, all trails that were awaiting this event.

3.4. The Compiler

The Terra compiler inherits all the static checking and basic structure of the Céu compiler. The compiler checks scripts for non-deterministic memory accesses, tight loops, and other properties, such as whether all possible block cancellations are captured by a `finally` statement.

The main modifications for Terra are the types described in section 3.1 and the new bytecode generation. Other modifications included the addition of expression operations, as Céu relies on the C compiler for expressions, and some checks and code optimizations. The absence of pointers in Terra's type system pointers avoids all kind of references to external variables and also avoids memory leaking. Checking types on assignments further enhances safety.

Terra has a hybrid set of instructions with some opcodes using a stack and other opcodes using arguments. Most opcodes accept variable-sized arguments. This is important to reduce the bytecode program size. The user program is converted to a bytecode representation to be interpreted by the virtual machine. During code generation the compiler checks for optimization opportunities. Whenever possible, code generation avoids the use of stack operations. Expressions with binary operations like sum or minus must use the stack.

An example of optimization is a simple assignment like `v1 = v2;` represented by the bytecode in Figure 9. Considering both as `short` type and in a lower memory locations (i.e needing 1-byte addresses), we will have seven bytes of non-optimized code against three bytes of optimized code.

3.5. Virtual machine

tVM is composed by three modules as shown in Figure 10.

The *VM* module is the main module. It provides an interface for receiving new application code from the *Basic Services* module and three interfaces for customized events and functions. The *Engine* submodule controls the execution of code interpreted by the *Decoder* submodule and of the external events received from the *Event Queue* submodule. We opted for a stack-based architecture because of its smaller code size in comparison to register-based architectures [Gregg et al. 2005].

```
/*** Not optimized + stack ***/
01 : push &v2         : opcode
02 :                  : addr2Low
03 :                  : addr2High
04 : push &v1         : opcode
05 :                  : addr1Low
06 :                  : addr1High
07 : setshort         : opcode
total of 7 bytes

/*** Optimized ***/
01: setshort &v1, &v2 : opcode
02:                   : addr2Low
03:                   : addr1Low
total of 3 bytes
```

Fig. 9. A code optimization example.

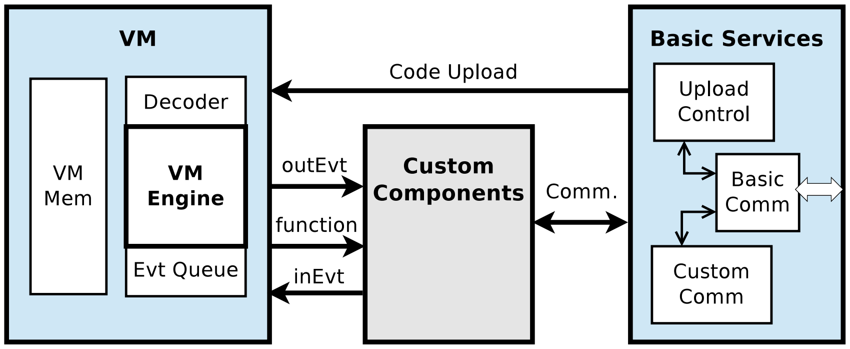

Fig. 10. tVM modules

The *Basic Services* module controls the communication primitives to give support to code dissemination and to the custom communication interface. The *Upload Control* submodule controls the dissemination protocol and loads code into local memory. The *Custom Comm* submodule has a generic interface to support new communication protocols defined at the *Custom Components* module level. The *Custom Components* module implements specific flavors of Terra.

An output event is always defined with no return value. These events may have one argument of any type, including address values of integer types or register/packet types. Each custom operation must know how to deal with its argument. An address value is used by custom operations to directly access data in the virtual machine memory.

Custom functions may have zero or more arguments of integer type or addresses of integers or registers. All arguments are passed via stack and the custom operation must pop from the stack exactly the number of arguments defined in the configuration block. A custom function must always return an integer value by pushing it to the stack. The use of stack is important to enable the use of functions inside expressions.

An input event may be defined to return an integer value or an address. In all cases, the returned data is copied to the memory location defined in the assignment operation. In the case of an address value, the custom operation must pass the internal buffer address that holds the data.

3.6. Terra operation

After tVM is loaded at all network nodes, the user can upload his script to be disseminated via radio to the network nodes.

The VM is typically loaded over a wired interface for each node as for any TinyOS program. It is also possible to run a simulated version in the TOSSIM TinyOS Simulator or in the AVRORA Emulator. Currently, a basic Terra implementation may be configured to run the same script in a hybrid network with MicaZ and TelosB. As both radios are compatible, intercommunication is guaranteed.

To execute a script, the programmer writes a tCéu program, compiles it, and uploads its bytecode to the network. The compilation process must include a specific Terra definition file for the chosen virtual machine. The script can use only events and functions defined in the included file. The generated bytecode must then be disseminated over the WSN using Terra's upload tool. This tool transfers the bytecode to the basestation node connected to the computer via wired interface. The base station node then starts the dissemination algorithm to send the bytecode program to all nodes. This algorithm uses the flood concept where all nodes forward each radio message until all nodes are reached. The current Terra tool and dissemination algorithm distribute the same program to all nodes.

3.7. Bytecode dissemination algorithm

The current Terra version disseminates the same bytecode to all nodes in the network. We assume that bytecode dissemination starts on a computer connected to a basestation node via wired interface. The virtual-machine code includes a dissemination algorithm that floods code blocks into the network. Each block goes as a wave. The basestation starts the flooding process with a *newProgramVersion* message and next sends the bytecode blocks. Each node forwards each new received message to its neighbors (all nodes at 1-hop radio range). All messages carry a version number and a sequence number to allow individual nodes to decide when it is a new program version. A periodic timeout forces each node, if necessary, to request any missing block from its neighbors. For the case when one node is switched on after the others, the algorithm also includes a startup request to neighbor nodes.

Table IV, in Section 4.2, presents some examples of code size. In the same section, Table V presents the dissemination duration.

4. EXPERIMENTS

We implemented Terra in TinyOS-2. Besides the basic components of TinyOS, the basic tVM image includes the CTP (Collection Tree Protocol) component [Gnawali et al. 2009]) 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. tVM currently runs on the MicaZ mote [Memsic 2009a] coupled with MDA100/MTS300 sensor board and on TelosB mote [Memsic 2009b].

In this section, we conduct evaluations of tVM from five 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 Terra and in the native nesC programming language. Next, in Section 4.2, we measure the cost of updating an application and, in Section 4.3, we measure the time of bytecode dissemination over networks of different sizes. In Section 4.4, we compare code size and memory usage for two different tVM customizations. Finally, in Section 4.5, we illustrate how simple it is to write a real parameterized application using Terra.

We used the Avrora instruction-level simulator [Titzer et al. 2005] to simulate the MicaZ hardware in the controlled tests.

4.1. tVM 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 tVM to the limit. In the second test, we measure the overhead of the tVM 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 systems, programs are coded with event-based loops. In Terra, as a tight loop is forbidden, we use an I/O pin reading to break the loop with an await. The I/O pin return event is generated immediately from the request. In the nesC/TinyOS version, each iteration posts a task representing the following one. The CTP component is used for sending messages to the base station both in the implementation of the tVM runtime and in the nesC/TinyOS version.

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. The goal of using this metric — which can be measured both in real motes and in the simulator — is to have a rough idea of the relative processing speeds of the two platforms. The second metric we use is the total number of cycles in *Active* and *Idle* state[2]. The values for this metric were obtained through the simulations on Avrora.

4.1.1. Scenario 1 - CPU-bound Application. Table I presents the results obtained with Avrora for our first test scenario. Figures 11 and 12 shows the code we used for this experiment. In the nesC version, the main loop is executed in a TinyOS task that contains only two commands: the loop counter increment and the the (re)post of the task itself. A periodic timer sends the counter value to BaseStation each 10s. In Terra we have a "par" with two sections. The first section controls the loop and increment the counter variable and the second section sends the counter value to the BaseStation for each 10s.

Table I. CPU-bound Test

Metric	**Program Version**		
	Terra(a)	**nesC**(b)	b/a
loop counter	516,291	9,902,517	19.18
active cycles	2,188,784,911	2,211,835,350	1.01
idle cycles	30,427,889	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 19.18 times the iterations executed by the VM version.

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

```
var msg msg1;
msg1.value4 = 0;
par do
   loop do
       emit REQ_CUSTOM_A(0);
       await CUSTOM_A;
       msg1.value4 =
         msg1.value4 + 1;
   end
with
   loop do
       await 10s;
       emit SEND_BS(msg1);
   end
end
```

Fig. 11. Code for CPU-bound experiment in Terra.

```
task void incTask(){
   counter++;
   post incTask();
}
```

Fig. 12. Code for CPU-bound experiment in TinyOS/NesC.

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 533.353 and 9,902,443.)

We now estimate the number of cycles per instruction in tVM. The main loop of our test script translates to eight instructions in the virtual machine. We can divide the total number of CPU cycles by the final value of the counter (number of times that the loop was executed) to obtain the number of CPU cycles per loop iteration, and then divide this result by 8 to estimate the number of cycles per instruction. The result is 513 cycles, which is close to the 400-cycles value obtained in the micro-benchmark of ASVM (section 4.5 §2 of [Levis et al. 2005]) and to the value of 550 cycles reported for DVM (section 4.1 §2 of [Balani et al. 2006]).

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. Figures 13 and 14 shows the code we used for this experiment. In the nesC version, the main loop is a TinyOS event handler that again contains two commands: the loop counter increment 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 Terra 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 an event for the VM. The VM then posts a task to (re)initiate the main procedure.

Table II presents the results for this scenario.

In this case, predictably, CPU active time was much less than in the first scenario. CPU was idle around 87%-95% of the time. The nesC variant executed approximately

```
var msg msg1;
msg1.value4 = 0;
var u16 value;
par do
    loop do
        emit REQ_PHOTO();
        value = await PHOTO;
        msg1.value4 = msg1.value4 + 1;
    end
with
    loop do
        await 10s;
        emit SEND_BS(msg1);
    end
end
```

Fig. 13. Code for IO-bound experiment in Terra.

```
event void s.readDone(error_t result, uint16_t val){
    counter++;
    call sensor.read();
}
```

Fig. 14. Code for IO-bound experiment in Terra (top) and in TinyOS/NesC (bottom).

Table II. IO-bound Test

Metric	**Program Version**		
	Terra(a)	**nesC**(b)	b/a
loop counter	27,215	29,949	1.10
active cycles	284,515,620	116,793,722	0.50
idle cycles	1,934,697,180	2,095,046,278	1.08

10% more iterations then the Terra variant. As regards CPU cycles, however, the Terra version needed around double the cycles used by nesC. In Terra, CPU was active 12.8% of the time, while in nesC only 5.3%.

Direct execution on the MicaZ mote again produced results close to the simulator's: the value of the counter was 27,129 for the Terra version and 29,997 for the nesC one.

In Terra, approximately 90 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 [Levis et al. 2005]). 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.

4.1.3. Energy consumption analysis. Table III shows the values of energy consumption that are reported at the end of execution of the second test scenario with the Avrora simulator. The values are shown in Joules and represent consumption for a 300 second execution in Terra and in nesC. We analyse only values for the two major energy consumers, radio and CPU.

Table III. IO-Bound Energy consumption results

	Terra	**nesC**
Radio	16.86J	16.86J
CPU	3.50J	3.21J

As expected, because radio utilization was similar in the two implementations, the values are the same for this item. The difference in energy consumed by the CPU is due to the difference in the periods of activity: in Terra we had two times the number of active cycles used by nesC. An active cycle consumes roughly 2.3 times the energy consumed by an idle one. However, because the total number of active cycles still remains small in proportion to the number of idle cycles, energy consumption was only 9% higher. This overhead would typically diminish, possibly to negligible rates, in real applications, in most of which the active/idle ratio is very small.

4.2. Reconfiguration cost

In this section, we try to estimate the cost of disseminating new code for Terra. 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.

Table IV presents the number of bytes we obtained for three simple applications written in Terra. As a basis for reference, we compare these sizes with their counterparts coded directly over TinyOS. This is of course the advantageous situation for a virtual machine, as in Terra 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.

The Terra configuration tool runs in a server computer connected to the *BaseStation* mote and is used to send new tVM programs to nodes. The current implementation stops and resets all nodes at the beginning of program dissemination. Each node starts automatically when the transfer is completed.

For all applications, application size is obtained from the compilers. 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.

The first value in each cell in Table IV indicates the number of bytes and the second one, in square brackets, the number of required messages. The first application is the classic *Blink* example from the TinyOS tutorial. This is a good example because it uses no special components, only timers and leds. In the Terra 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 Terra and nesC versions. Application *rdLoop2* is a version of *rdLoop1* for nesC without the CTP component.

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 Terra it is pre-loaded in the motes. Application *rdLoop2* attains an intermediate

Table IV. Reprogramming Cost

App	Program Version	
	Terra(a)	**nesC**(b)
Blink	93 [4]	2048 [86]
rdLoop1	88 [4]	18188 [758]
rdLoop2		13022 [543]

Units: Bytes [Messages]

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.).

4.3. Bytecode dissemination

Although dissemination itself is not part of our research goals, we report in this Section some measurements of the dissemination time. The dissemination algorithm must include a delay after each disseminated packet/message. This avoids radio messages collision during the flood process, but we would naturally like dissemination time to be short and to scale well. The full upload process comprises the upload to the basestation via wired interface and the dissemination over the network via radio messages. Here we are considering only the time for radio dissemination. To obtain it, we need to get the start time and the end time of the radio dissemination process. Because the process starts with a message sent by the root node and ends at an arbitrary node (the last one to receive the code), we need a global clock to synchronize the local clock in each node, as we only have the local nodes time. Our solution was to use the TinyOS simulator (TOSSIM) as it provides a global simulated clock in all radio messages logs. Additionally, TOSSIM uses a noise model to simulate message collisions and losses. Using this facility, we forced the dissemination algorithm to spend some time in its recovery stage, which would be a probable scenario in a real-world use.

We ran our test using a script for a real monitoring application that includes routing to the basestation. The program bytecode has 24 message blocks to be disseminated. Table V presents the dissemination times for three scenarios. The first scenario is a very basic case with only one node. The second scenario considers a grid with 9 nodes (3 x 3) and the third scenario considers a grid with 49 nodes (7 x 7). In our grid network, each radio node ranges only to its 1-hop neighbor node, i.e. all nodes range up to 8 nodes. Only one corner node exchanges messages with the basestation. This configuration forces the use of the flooding mechanism. Figure 15 shows an example for the 7 x 7 grid with the radio range highlighted for the nodes 11, 32, and 44. We also measured the time it took to load the program in each node. Table V includes the minimum time, the maximum time and the average time for each scenario.

All dissemination tests were done considering that all radios were switched on. We have some premature work in duty cycling the radio on and off to reduce power consumption but, depending of duty cycle configuration, the dissemination time can be 10 times larger. We plan to build a hybrid radio control module which cancels the duty cycle mode during the bytecode dissemination.

The dissemination for the single-hop scenario took 7.17 seconds for 24 messages, that is, approximately 300ms per message. The 300ms step delay time is exactly the configuration parameter used in our algorithm. Using one real node it is possible to measure a similar duration, but the reference is a message sent back to the computer. In our real-node test we got 7.2 seconds. Although it is possible to use lower values for the step delay time to reduce the total dissemination duration, we chose to be more

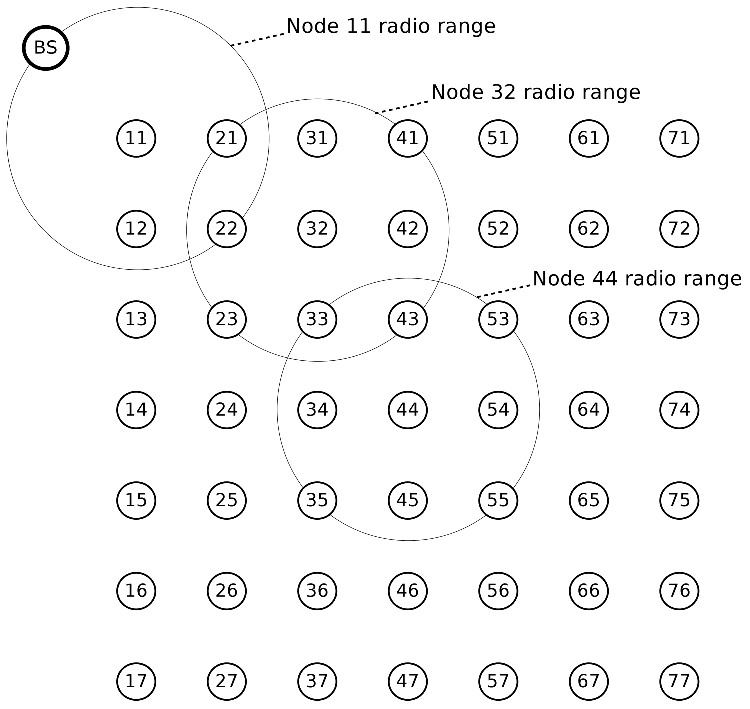

Fig. 15. Simulated 7x7 grid - node 11, 32, and 44 with radio range highlighted.

Table V. Dissemination time

Scenario	Total Nodes	Total Duration	Average time	min time	max time
#1	1	7.17	7.17	7.17	7.17
#2	9	7.25	6.76	6.70	7.17
#3	49	7.50	6.58	6.40	7.17

all durations in seconds

conservative to minimize radio collision in dense networks. The tVM customizer may choose different values for this parameter. This subject is matter of more investigation and depends on network topology. The differences beween minimum, maximum, and average times are of fractions of seconds. Considering the nature of WSN applications, in general these differences don't affect system operation.

Comparing results for the different scenarios we get the time for each additional hop in the network. In general, as our dissemination algorithm floods message by message in sequential waves, the total time doesn't increase much as the network grows. In our case this time varied from 40ms up to 55ms. These values are consistent with our radio-send policy, where the sending message is delayed randomly from 20ms up to 95ms. Based on the scenarios #2 and #3, respectively with 9 and 49 nodes, the dissemination time has increased only 250ms (3.45%) for a an increment of 40 nodes (444%). This shows that the system is easily scalable.

4.4. Code size and memory usage

To measure code size and memory usage, we prepared a minimal Terra configuration called *TerraNet* as defined in section 2.2. This Terra variant provides only very basic "send" and "receive" events. Table VI shows memory usage for two Terra configurations (running on micaZ [Memsic 2009a]): the default configuration, with all components described in Section 2.1 (first column), and *TerraNet* (second column).

Table VI. Terra Memory ocupation

	Terra	**TerraNet**
Binary code size	48286	32162
Total RAM usage	3580	3570
VM Memory size	528	1968

The table shows that the mandatory components take up 32KB of binary code (ROM). The amount of RAM to be used must be adjusted to the specific project at hand. The main RAM consumers are the data buffers used in communication protocols and the memory allocated to variables and code in the application script (VM memory). In our case, we adjust the VM memory to reach 3,5KB of maximum RAM usage because we need some room for nesC stack usage. The MicaZ mote has only 4KB of RAM. In the original tVM, in which the more complex components also need RAM space, we leave up to 528 bytes for the application script. In the TerraNet version, where we took out the complex components, we leave up to 1968 bytes for the application script.

4.5. A monitoring application

In this Section, we present a complete Terra monitoring application and discuss how it can be tuned remotely by sending new values for parameters used in its configuration.

Figure 16 presents our application, which monitors the average temperature as measured by motes in well-lighted points. The user can remotely define the threshold for a place to be regarded as "well-lighted".

The basis for this application is the algorithm for group creation available in the default Terra runtime. The average value is computed by the aggregation modules at all nodes participating in the group composed of 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 currently defined threshold.

The program is formed by a parallel block containing a loop in each of its arms: in the first one, nodes decide, every 20 seconds, whether or not they belong to the group of well-lighted positions, while in the second loop, at every 60 seconds, the coordinator node collects the average temperature reading and sends it back to the base station. In the first loop, after awaiting for 20 seconds (line 12), each node emits an event requesting a reading from the luminosity sensor (line 13) and awaits for its reply (line 14). Next, the result is compared to parameter `phLevel`, the threshold value, and the result is assigned to variable (`grA.isMember`), which determines participation in the group (line 15). In the second loop, after awaiting for 60 seconds (line 19), the node checks whether it is the coordinator (line 20), and if so, it emits an event which triggers aggregation of the temperature readings in the group (line 21). This event is handled by Terra's aggregation component, which was initialized with the convenient parameters in line 2. Next, the coordinator awaits for the aggregation to be completed (line 22), and sends the result to the base station (lines 23-24).

This application uses two values that can be regarded as its parameters: variable `phLevel` indicates the threshold level for a point to be considered well-lighted and variable `gr1.leader` contains the ID of the coordinator node. (`gr1.leader` field is automatically defined inside `var group gr1;` command at lines 1 and 2.) It would be trivial to change these values after the application is installed, causing a simple reconfiguration. Using 16-bit values, only one message would be necessary to carry the new values. The

```
var group gr1;
grNew(&gr1,1,0,3,TRUE,OFF,2);
var aggreg agA;
agNew(&agA,&gr1,TEMP,AVG,GTE,0);

var ushort phLevel=100; // Photo param
var msg dataMsg;   // Message struc
var agResult data; // Agg result struc

par do
  loop do      // Group loop
    await 20s;
    emit REQ_PHOTO();
    var ushort phVal = await PHOTO;
    grA.isMember = (phVal > phLevel);
  end
with
  loop do  // Aggregation/sendBS loop
    await 60s;
    if (NODE_ID == gr1.leader) then
      emit AGGREG(&agA);
      data = await AGGREG_DONE;
      dataMsg.value4 = data.value;
      emit SEND_BS(&dataMsg);
    end
  end
end
```

Fig. 16. Monitoring Application in Terra.

output file from Terra compiler shows the memory address for all program variables, enabling reconfigurations to be carried out by updating the specific addresses corresponding to the variables that must be modified. If necessary, more than one message can be used to update such parameters. However, because the reconfiguration message need contain only the parameters that undergo modifications, it is often possible to work with a single message.

More complex updates to the application could be carried out either by programming it in the first place with more variables for remote configuration, or by sending a whole new script to the motes. The application shown in Figure 16, for instance, occupies 213 bytes, and could thus be sent to a mote with nine messages.

5. RELATED WORK

Terra's basic proposal is to combine the advantages of using application-specific, or high-level, virtual machines with a scripting language that provides a set of facilities and guarantees. In this section we report on work that is related to each of these approaches and discuss how Terra relates to it.

To our knowledge, the first work proposing the use of virtual machines in WSN is Maté [Levis and Culler 2002]. The Maté VM is built on TinyOS and has a very simple instruction set. The code propagation and execution is broken up into 24 instructions called capsules. A capsule fits into a single message packet. Maté limits its context execution to only three concurrent paths, one for sending messages, another one for receiving messages, and a third one for a timer. Maté has up to 8 user-defined instructions that enable additional virtual machine customization and its operand stack has a maximum depth of 16. To address some of Maté's limitations the Maté team built ASVM [Levis et al. 2005]. ASVM is an application-specific virtual machine. The authors proposed a custom runtime machine to support different application-specific high-level languages, but each language needs its own compiler. ASVM implements a central concurrency manager to support the sequential execution on concurrent handlers. This is an optional service to help user applications avoid race conditions. This solution assumes that handlers are short-running routines that do not hold on to resources for very long.

DAViM [Michiels et al. 2006] is very similar to ASVM but adds the possibility of parallel execution. DVM [Balani et al. 2006] is based on the application-specific VM concept from ASVM, but it uses SOS [Han et al. 2005] as its operating system. SOS allows dynamic loading of system modules. In DVM, it is possible to load different combinations of high-level scripting languages and low-level runtime modules. DVM [Balani et al. 2006] and DAViM [Michiels et al. 2006] also use a concurrency manager like ASVM's.

Several groups have worked on VMs for Java. VMStar [Koshy and Pandey 2005] uses the Java as high-level language for customized VMs. The VMStar toolset helps to build a new VM runtime from the device characteristics and the component library. VMStar uses a "select" concept to register event-wait points in a sequential program. The select interface executes event handlers sequentially to avoid race conditions, in a single-thread implementation. VMStar inherits type-safety from Java, like the others Java VMs. NanoVM [Harbaum 2005], ParticleVM [Riedel et al. 2007], TakaTuka [Aslam et al. 2008], and Darjeeling [Brouwers et al. 2009] also use Java as their programming language. Inspired on TinyDB [Madden et al. 2005], SwissQM [Mueller et al. 2007] has a query-specific instruction set and a high-level language similar to SQL.

Cosmos and Regiment implement customizable VMs with high-level languages that are specifically designed for WSNs. Cosmos [Awan et al. 2007] uses mPL as high-level language and mOS as operating system. mPL supports intra-network operation programming, that is, network-wide operations. A Cosmos application is defined by a data-flow graph and some custom C functions loaded within mOS. The mOS system executes the application graph as script. The scripting language is limited to the data flow control using the custom mOS functions. Cosmos also allows dynamic loading of new C functions. The graph approach also limits the application types. In Cosmos, an event handler is represented as a Functional Component (FC). A FC uses only local variables and its data are exchanged by input/output interface queues. These characteristics avoid race conditions. Regiment [Newton and Welsh 2004; Newton et al. 2007] uses a reactive functional language with a special semantic for intra-network operations. The runtime implements the basic operations and access to devices. A Regiment application is compiled to a intermediate language called *Token Machine*(TM). A TM segment propagates into the network and it is interpreted to execute local operations or intra-network operations like group formation and aggregation. In Regiment, an event handler task run to completion and cannot be blocked. This also avoids race conditions.

Discussion

tVM architecture combines small size with a model that is less restrictive than Maté's. Because Terra implements the Céu concurrency model, it is possible to have several concurrent execution paths. Terra also enables up to 255 ids for each group of input events, output events, and functions. tVM stack is defined at compile time and is limited only by memory space shared with application script.

Differently from DVM and Cosmos, Terra doesn't allow low-level code loading, but Terra natively supports remote parameterization of runtime components. We believe Terra's reactive programming model, similarly to Regiment's, is more suitable to event-driven application then the traditional program models. In a tCéu program is possible to suspend a execution of one program block and wait for an event without suspending all others program blocks. Terra inherits the Céu execution control in which a trail (a Céu handler) is serialized to execute to completion. Céu trails are similar to Protothreads coroutines [Dunkels et al. 2006], because they both offer multiple sequential lines of execution to handle concurrent activities. This execution mode minimizes race conditions and doesn't burden the user with synchronization mechanisms (centralized controls, interface queues, or semaphores and mutexes). It still may get race conditions from multiples trails waiting for the same event and writing the same memory address. The compiler has an analysis mode which find these race conditions. This analysis mode is similar to the safe annotations from TinyOS but it is checked at compile time. Well tested built-in components extend the safety guarantees to runtime. tCéu avoids tight loops which not recommended but allowed by most of the related work. By itself, tCéu doesn't give execution guarantees in intra-network operations. In Terra, these guarantees may be given by built-in runtime intra-network operations. Terra doesn't support network-wide programming. The user must think about the application as a whole but write the code that each node will run. However, the provision of components inspired by macro-programming alleviates this problem in some measure, by abstracting some typical collective operations.

6. CLOSING REMARKS

In this work, we explored the combination of two complementary technologies for programming WSNs: a language (tCéu) with static execution guarantees and a virtual machine (tVM) with a component library built to support typical application patterns.

On one hand, we used concepts from synchronous and reactive programming to provide static guarantees of bounded and race-free execution while maintaining sufficient expressiveness. Although a language with these characteristics brings benefits to the programmer, it will always depend on external code to interact with the environment. In WSN applications, this interaction is restricted to a number of patterns for communication, sensing, and control. The use of a virtual machine with pre-packaged components for these tasks complements the guarantees of the language, enabling the program to interact with the environment while maintaining its safety. In general, we don't need to change these pre-packaged components after deployment. Pre-packaged components depend on the characteristics of hardware and on the network topology already defined in the deployment phase.

Terra allows the programmer to combine events with a sequential programming style while ensuring the absence of data races and of out-of-bound errors. It also guarantees that there are no unbounded loops. We know of no other WSN system with this guarantee.

Although this guarantees safety in the tCéu part of the application, the script must have access to the external world to communicate with other motes and to interact with sensors and actuators. In Terra, the only available external calls are those provided by the application-specific VM in use. The operations discussed in sections 2.1.2 and 2.1.1, which were inspired by *macroprogramming* proposals [Awan et al. 2007; Gummadi et al. 2005; Kothari et al. 2007], extend the safe execution guarantees to the Terra runtime.

The use of a virtual machine allows for applications with small code size. This, together to the dissemination algorithm of Terra, allows dissemination of a program in a few seconds with low energy consuption and in a scalable fashion. Also our experiments give us promising insights 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.

Terra is currently being used at PUC-Rio for hands-on classes in WSN as part of two different courses (final undergraduate/graduate): Reactive Programming and Distributed Systems. In our experience (approximately four semesters), students have better learning curve experience with Terra than with the traditional nesc/TinyOS environment, and manage to develop relatively complex projects in a few weeks.

As future work, we are planning to evaluate the resources of Terra for different programming abstraction levels. Each Terra customization will represent a different abstraction flavor that works at a different level of programming abstraction. Also, as future work, we are planning to test Terra limits trying to build a heavy CPU consumer application like Volcano [Tan et al. 2013]. As regards energy consumption, we are planning to investigate the behavior of a radio low power listening protocol during script dissemination. All these experiments and investigations may give us insights and optimizations to be included in the next Terra version. Finally, because the work on Terra was born from our interest in WSN macroprogramming and from thinking that we needed node-level support before moving to the network level, we might now be able to move on to this investigation, using Terra as base system for a new macroprogramming language.

ACKNOWLEDGMENTS

The authors would like to thank the partial support from CNPq – Brazilian National Counsel of Technological and Scientific Development and FAPERJ – Rio de Janeiro Research Foundation.

REFERENCES

Faisal Aslam, Christian Schindelhauer, Gidon Ernst, Damian Spyra, Jan Meyer, and Mohannad Zalloom. 2008. Introducing TakaTuka: a Java virtualmachine for motes. In *Proceedings of the 6th ACM conference on Embedded network sensor systems (SenSys '08)*. ACM, New York, NY, USA, 399–400. DOI:http://dx.doi.org/10.1145/1460412.1460472

Asad Awan, Suresh Jagannathan, and Ananth Grama. 2007. Macroprogramming heterogeneous sensor networks using cosmos. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 159–172. DOI:http://dx.doi.org/10.1145/1272996.1273014

Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. 2005. 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)*. USENIX Association, Berkeley, CA, USA, 19–24.

Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. 2006. Multi-level software reconfiguration for sensor networks. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT '06)*. ACM, New York, NY, USA, 112–121. DOI:http://dx.doi.org/10.1145/1176887.1176904

Adriano Branco. 2011. *A WSN programming model with a dynamic reconfiguration support*. Master's thesis. PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO - PUC-RIO. Text in portuguese.

Niels Brouwers, Koen Langendoen, and Peter Corke. 2009. Darjeeling, a feature-rich VM for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*. ACM, New York, NY, USA, 169–182. DOI:http://dx.doi.org/10.1145/1644038.1644056

H. Cervantes, D. Donsez, and L. Touseau. 2008. An Architecture Description Language for Dynamic Sensor-Based Applications. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*. 147–151. DOI:http://dx.doi.org/10.1109/ccnc08.2007.40

A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. 455–462. DOI:http://dx.doi.org/10.1109/LCN.2004.38

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th* *international conference on Embedded networked sensor systems*. ACM, New York, NY, USA, 29–42. DOI:http://dx.doi.org/10.1145/1182807.1182811

David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC language: A holistic approach to networked embedded systems. (2003), 1–11. DOI:http://dx.doi.org/10.1145/781131.781133

Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. 2009. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*. ACM, New York, NY, USA, 1–14. DOI:http://dx.doi.org/10.1145/1644038.1644040

David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. 2005. The case for virtual register machines. *Science of Computer Programming* 57, 3 (2005), 319 – 338. DOI:http://dx.doi.org/10.1016/j.scico.2004.08.005 Advances in Interpreters, Virtual Machines and Emulators IVME'03.

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming Wireless Sensor Networks Using Kairos. *Distributed Computing in Sensor Systems* (2005), 126–140. DOI:http://dx.doi.org/10.1007/11502593_12

Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international Conference on Mobile Systems, Applications, and Services (MobiSys '05)*. ACM, New York, NY, USA, 163–176. DOI:http://dx.doi.org/10.1145/1067170.1067188

Till Harbaum. 2005. The NanoVM - Java for the AVR. (2005). Retrieved August, 2014 from http://www.harbaum.org/till/nanovm/index.shtml

Joel Koshy and Raju Pandey. 2005. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys '05)*. ACM, New York, NY, USA, 243–254. DOI:http://dx.doi.org/10.1145/1098918.1098945

Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. 2007. Reliable and efficient programming abstractions for wireless sensor networks. *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), 200–210. DOI:http://dx.doi.org/10.1145/1250734.1250757

Philip Levis and David Culler. 2002. 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*. ACM, New York, NY, USA, 85–95. DOI:http://dx.doi.org/10.1145/605397.605407

Philip Levis, David Gay, and David Culler. 2005. Active sensor networks. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 343–356.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. 2004. TinyOS: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag.

Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems* 30, 1 (2005), 122–173. DOI:http://dx.doi.org/10.1145/1061318.1061322

Memsic. 2009a. MicaZ datasheet. Product folder. (2009). Retrieved August, 2014 from http://www.memsic.com/wireless-sensor-networks/MPR2400CB

Memsic. 2009b. TelosB datasheet. Product folder. (2009). Retrieved August, 2014 from http://www.memsic.com/wireless-sensor-networks/TPR2420

Sam Michiels, Wouter Horré, Wouter Joosen, and Pierre Verbaeten. 2006. DAViM: a dynamically adaptable virtual machine for sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks (MidSens '06)*. ACM, New York, NY, USA, 7–12. DOI:http://dx.doi.org/10.1145/1176866.1176868

Luca Mottola and Gian Pietro Picco. 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.* 43, 3, Article 19 (April 2011), 51 pages. DOI:http://dx.doi.org/10.1145/1922649.1922656

Rene Mueller, Gustavo Alonso, and Donald Kossmann. 2007. SwissQM: Next generation data processing in sensor networks. In *Third Biennial Conference on Innovative Data Systems Research*.

Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The Regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. ACM, New York, NY, USA, 489–498. DOI:http://dx.doi.org/10.1145/1236360.1236422

Ryan Newton and Matt Welsh. 2004. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st International Workshop on Data Management for Sensor Networks*. ACM, New York, NY, USA, 78–87. DOI:http://dx.doi.org/10.1145/1052199.1052213

J. Ousterhout. 1998. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer* 31, 3 (1998), 23–30.

Till Riedel, Andreas Arnold, and Christian Decker. 2007. Poster Abstract: An OO Approach to sensor programming. In *European conference on Wireless Sensor Networks (EWSN)*.

Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. 2013. Safe System-level Concurrency on Resource-constrained Nodes. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys '13)*. ACM, New York, NY, USA, Article 11, 14 pages. DOI: http://dx.doi.org/10.1145/2517351.2517360

Rui Tan, Guoliang Xing, Jinzhu Chen, Wen-Zhan Song, and Renjie Huang. 2013. Fusion-based Volcanic Earthquake Detection and Timing in Wireless Sensor Networks. *ACM Trans. Sen. Netw.* 9, 2, Article 17 (April 2013), 25 pages. DOI: http://dx.doi.org/10.1145/2422966.2422974

Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. 2005. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*. IEEE Press, Piscataway, NJ, USA, Article 67.

Received October 2014; revised Xxxx 2015; accepted Yyyy 2015