Pontifícia Universidade Católica
do Rio de Janeiro

Adriano Francisco Branco

Scripting customized components for Wireless Sensor Networks

Tese de Doutorado

Thesis presented to the Programa de Pós–Graduação em Informática of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor : Prof. Noemi de La Rocque Rodriguez
Co–advisor: Prof. Silvana Rossetto

Rio de Janeiro
September 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Adriano Francisco Branco

Scripting customized components for Wireless Sensor Networks

Thesis presented to the Programa de Pós–Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC–Rio, as partial fulfillment of the requirements for the degree of Doutor.

Prof. Noemi de La Rocque Rodriguez
Advisor
Departamento de Informática — PUC–Rio

Prof. Silvana Rossetto
Co–advisor
UFRJ

Prof. Roberto Ierusalimschy
Departamento de Informática — PUC-Rio

Prof. Markus Endler
Departamento de Informática — PUC-Rio

Prof. Claudio Luis de Amorim
UFRJ

Prof. Bruno Oliveira Silvestre
UFG

Prof. José Eugenio Leal
Coordinator of the Centro Técnico Científico — PUC–Rio

Rio de Janeiro, September 10th, 2015

Adriano Francisco Branco

Adriano Branco currently is a PhD candidate of Computer Science Department at PUC-Rio. His research focus on Wireless Sensor Network (WSN) in distributed system area. He also got his master in computer science at PUC-Rio in 2011 working with WSN. He undergraduates in Electronic Engineering at CEFET/RJ in 1992. From the undergraduate course he worked as electronic engineer and system developer at CBPF/CNPq (Brazil) and CERN (Switzerland). At CPBF, in the LAFEX laboratory, he worked on the parallel computer program from Fermilab collaboration group. At CERN he spent two years working in the New Trigger Project for LEP Delphi Experiment. After that he had worked more than 12 years in system integration consulting projects (as developer and project manager) for large companies. Mainly for the Industrial Automation and Telecommunications industries, including an international project in Manila/Philippines.

Bibliographic data

Branco, Adriano Francisco

Scripting customized components for Wireless Sensor Networks / Adriano Francisco Branco; advisor: Noemi de La Rocque Rodriguez; co–advisor: Silvana Rossetto. — 2015.

100 f. : il. (color.); 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Rede de Sensores sem Fio (RSSF). 3. Sistemas Distribuídos. 4. Modelo de Programação. 5. Linguagem Reativa. 6. Máquina Virtual. I. Rodriguez, Noemi de La Rocque. II. Rossetto, Silvana. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgement

Thank to my advisors Prof. Noemi Rodriguez and Prof. Silvana Rossetto for their support and encouragement for this work.

Thank to CNPq, PUC-Rio, and FAPERJ, for the financial support which allowed this work to be done.

To my wife, who accompanied me all this time with direct and indirect support.

To my parents, family and friends who supported me even with my absence in family life.

To all colleagues, faculty and staff of the Department of PUC-Rio, for the fellowship, learning and support.

Abstract

Branco, Adriano Francisco; Rodriguez, Noemi de La Rocque (Advisor); Rossetto, Silvana (Co-Advisor). **Scripting customized components for Wireless Sensor Networks**. Rio de Janeiro, 2015. 100p. D.Sc. Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Programming wireless sensors networks (WSN) is a difficult task. The programmer must deal with several concurrent activities in an environment with severely limited resources. In this work we propose a programming model to facilitate this task. The model we propose combines the use of configurable component-based virtual machines with a reactive scripting language which can be statically analyzed 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. To evaluate this model, we built Terra, a system combining the scripting language Céu-T with the Terra virtual machine and a library of components. We designed this library taking into account the functionalities commonly needed in WSN applications — typically for sense and control. We implemented different applications using Terra and using an event-driven language based on C and we discuss the advantages and disadvantages of the alternative implementations. Finally, we also evaluate Terra by measuring its overhead in a basic application and discussing its use and cost in different WSN scenarios.

Keywords

Wireless Sensor Network (WSN); Distributed Systems; Programming Model; Reactive Language; Virtual Machine.

Resumo

Branco, Adriano Francisco; Rodriguez, Noemi de La Rocque; Rossetto, Silvana. **Programando redes de sensores sem fio com scripts sobre componentes customizados**. Rio de Janeiro, 2015. 100p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Programar redes de sensores sem fio (RSSF) é uma tarefa difícil. O programador tem que lidar com várias atividades simultâneas em um ambiente com recursos extremamente limitados. Neste trabalho propomos um modelo de programação para facilitar essa tarefa. O modelo que propomos combina o uso de máquinas virtuais configuráveis baseadas em componentes com uma linguagem de script reativa que pode ser analisada estaticamente para evitar conflitos de memória e execução de laços infinitos. Essa abordagem permite a flexibilidade de carregamento remoto de código nos nós da rede combinado com um conjunto de garantias para o programador. A escolha de um conjunto específico de componentes numa configuração de máquina virtual define o nível de abstração visto pelo script da aplicação. Para avaliar esse modelo, construímos Terra, um sistema que combina a linguagem de script Céu-T com uma máquina virtual e uma biblioteca de componentes. Nós projetamos esta biblioteca considerando as funcionalidades comumente necessárias em aplicações de RSSF — tipicamente para sensoreamento e controle. Implementamos diferentes aplicações utilizando Terra e uma linguagem orientada a eventos baseados em C. Além disso discutimos as vantagens e desvantagens dessas implementações alternativas. Finalmente, também avaliamos Terra medindo o custo adicional em uma aplicação básica e discutimos sua utilização e custo em diferentes cenários de aplicações WSNs.

Palavras–chave

Rede de Sensores sem Fio (RSSF); Sistemas Distribuídos; Modelo de Programação; Linguagem Reativa; Máquina Virtual.

Contents

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, makes programming applications a difficult and error prone task (Awan et al., 2007; Kothari et al., 2007; Mottola and Picco, 2011).

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.

1.1
Research Question

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 approach in which common programming patterns are designed and implemented separately as a component-based virtual machine. These components may be combined as needed, creating customized virtual machines with abstractions provided by the component interfaces. As discussed by Ousterhout (Ousterhout, 1998), scripting languages enforce a programming model that glues components

together to create powerful applications in a few lines of code, We thus propose the use of a scripting language with support for several of the problems encountered in WSNs. This makes them suitable for creating programs that benefit from the pre-defined and pre-installed set of components and that can be easily sent over the network. We argue that this model, based on virtual machine and combining a reactive scripting language with a set of customized components, is highly convenient for use in WSN.

We formulated the following as research question for this thesis: *To what extent can a programming environment based on the combination of a reactive high-level scripting language with safety guarantees with a virtual machine that encapsulates customized components facilitate the task of programming WSNs, providing abstractions to simplify programming, reducing the possibility of errors, and allowing reprogramming?*

To investigate this idea we built Terra, a flexible system that targets both WSN programmer experts and application programmer. The application programmer benefits from a high-level programming environment where the WSN programmer expert may easily integrate new operations as needed. The system uses virtual machines which embed these new operations as components and facilitate remote distribution of scripts with low energy consumption. Our scripting language is based on the reactive programming language Céu (Sant'Anna et al., 2013).

In the next section we describe some typical difficulties in building distributed systems and event-driven programming in WSNs projects.

1.2 Major problems in programming WSNs

Probably, the major concern in WSN systems is with energy consumption. An approach to reduce this consumption is to put the CPU in sleep mode during idle state and waits for a hardware interruption to wake up the CPU. For example, a sensor converts some physical unit to a voltage value and the microcontroller uses its analog to digital converter (A/D) to read this voltage value. In general, this operation interacts with the CPU in two points, first the CPU starts the conversion and second the converter signals an interruption to indicate a valid value to be read. During this two points, probably, the CPU may be idle and may stay in sleep mode to save energy. The interruptions are also used in timers, radio interface and data memory chip. Depending of the application, a WSN node may be in idle mode during long time. For example, a periodic monitoring application may wake-up the CPU each hour.

The event-driven programming model is very suitable to this execution

regime that alternates sleep, interruption, and processing. In general, a WSN programming system supports the application program with a way to register event handlers and function calls. These systems also implement a control of idle state that puts the CPU in sleep mode. Traditional programing systems for WSNs, like TinyOS (Levis et al., 2004) and Contiki (Dunkels et al., 2004), extend the C language with support for event-driven programming.

Programming distributed systems is not so an easy task even in conventional computer networks. In WSN projects, this task get more difficult because of the combination of resource scarcity with the event-driven programming model. Terra presents an alternative to this programming model, combining a reactive programming model with a set of specialized operations and a remote load support for scripts.

In the next two subsections, about programming issues and cost, we present in detail the problems we want to target with our work.

1.2.1 Programming issues

We separate the programming issues in two main groups. One group is highly related to the programming environment based on event-driven models that extend a procedural language like C. In the other group, we include more general difficulties found in programming a WSN application. These programming issues were also identified in our experience developing WSN applications.

A. Programming complexity

- A.1. Sequential and event-driven programming
  - A1.i. Learning curve
  - A1.ii. Split phase
  - A1.iii. Global variables
- A.2. Local starvation
- A.3. Invalid pointers

B. Networking complexity

- B.1. Radio operations
- B.2. Communication protocols

Programming complexity

The main concern in programming event-driven systems like WSN applications is the opposition between sequential and event-driven coding styles. The majority of programmers learned programming using a procedural language and coding program operations in a sequential manner. In an event-driven environment, the programmer needs to split operations into different event handlers. This complicates context visualization and also is an error prone activity.

Typical WSN development environments like TinyOS and Contiki extend the C language including new constructors to support event-driven operations. The user has a learning curve that affects code productivity and quality (Levis, 2012). Other concerns are related to some weak points of the C programming language, basically the possibility of infinite loops and invalid memory access. These two weak points are more difficult to debug in embedded systems than in standard operating systems in a typical computer.

A typical pattern in event-driven programming is that of split-phase operations, where the programmer needs to split an operation in two parts. For example, when working with timers, one part will contain the command to start the timer and another part a function to handle the elapsed time (function callback). This same structure applies to sensor operations and radio operations.

To maintain the application state in this model, the programmer must typically resort to global variables. But to maintain these states in an event-driven environment is error prone, mainly because the state variables are accessed independently in different locations of the code.

Another recurring problem is local starvation. This happens, in general, when a process enters an infinite loop and does not release the CPU for other processes. This kind of operation prevents the system to handle new events and blocks all application

Related to invalid pointers, we have another issue that are the programming errors enabled by some programming languages that allow pointer manipulation, like C. Also a simple array malformed index may generate an invalid access to memory, causing program malfunction.

Networking complexity

Typically, WSN platforms are limited in memory and CPU resources and, also, have simple radio communication interfaces. These characteristics increase some of regular difficulties of programming distributed systems. Merging the application layer with the communication layer is a common

strategy to reduce code size and work around these resource limitations. However, this merging strategy brings more difficulties in developing new applications.

Basic radio operations, when needing some level of safety or guarantee, may incur in more complexity to manage queue/buffer operations and acknowledgement controls. For example, a simple combination of acknowledgements and message retries must be complemented with a duplicated-message check to prevent duplicated messages when an acknowledgement is lost. Another example is a program that may send messages from different points in the code. It must implement a control component to manage an output message queue and avoid conflict in the use of the radio.

Programming communication protocols is an additional burden that pushes the programmer from the application domain to the distributed communication system domain. In general, the application programmer is not a distributed system programming expert. In WSN applications, the distributed communication programming activity is complicated by the typical resource constraints, like radio energy consumption and small memory size. The program may not be able to build even a simple routing table in memory because of the size restrictions.

1.2.2 Cost

This section identifies two of main costs from the benefits provided by the proposed model – the costs related to virtual machine architecture and to code dissemination.

Typically, the use of virtual machine architecture incurs an additional use of CPU and memory. We evaluate the impact of this overhead on CPU processing, memory usage and energy consumption for typical WSN applications.

In this work we target motes with memory limitations. One of the motes we use has 128k bytes of ROM and only 4k bytes of RAM. Another mote has only 48k bytes of ROM and 10k bytes of RAM. The size of ROM limits the size of the virtual machine and as a consequence the size of the embedded custom components. RAM is shared by the built-in customized components and the memory space available for the user script: the more memory the embedded components use, less is left for the script.

Nodes in a WSN project typically form an ad-hoc network that may cover a large area. In practical term, it is in most cases difficult, expensive, or impossible to recover the nodes to reprogram them over a serial cable.

Reprogramming the nodes via radio interface is an alternative, but it incurs in an undesirable code dissemination cost, in special for battery energy consumption.

1.3 Contributions

We propose a programming model based on configurable virtual machine and combining a reactive scripting language with a set of customized components. The main contribution of this work is to show that this proposed model is feasible to programming WSN applications. We also show that is possible to apply this model even in nodes with very limited resources.

A second contribution is the Terra system, a system that implements the proposed programming model. Terra allows expert programmers to build customized environments for safer and easier application developments. We also built a set of high-level ready-to-use generic components that are useful for several WSN application domains.

1.4 Document structure

In the next chapter we present the Terra system. Chapter 3 presents the evaluation of programming aspects and the Chapter 4 presents the evaluation of aspects related to cost. Chapter 5 presents related works and, finally, Chapter 6 presents our final remarks.

2
Terra programming System

In this chapter we present the Terra programming system. In the next sections we introduce the Terra model and the used technologies, then we present the Terra system in more details and finally we present the three Terra customizations used in the evaluation.

2.1
Terra basics

We propose the Terra programming System, aiming to reduce the WSN programming difficulties of using the event-driven model and programming network protocols. The Terra model is based on a virtual machine and combines a reactive scripting language with a set of customized components. These components may be selected as needed, creating customized virtual machines with abstractions provided by the component interfaces. A scripting language enforces a programming model that glues components together to create powerful applications in a few lines of code. This makes them suitable for creating programs that benefit from the pre-defined and pre-installed set of components and that can be easily sent over the network.

Terra uses Céu-T as its scripting language and implements a component-based virtual machine VM-T to be customized for different application domains. Céu-T implements a variation of Céu programming language (Sant'Anna et al., 2013). This variation, on one hand, excludes some of Céu's characteristics and, on the other hand, includes some language extensions. We use Céu as Terra's scripting language because of its reactive nature and its high-level control primitives and compile-time safety guarantees.

The Terra Virtual machine VM-T, with its customizations, runs on WSN nodes with limited resources. We built VM-T using the nesC programming language (Gay et al., 2003) and the TinyOS operating system (Levis et al., 2004). We used TinyOS, basically, because it is the operating system for WSN most ported to different mote types.

Figure 2.1 presents the three basic elements of Terra – a scripting language (Céu-T), a set of customized pre-built components, and an embedded virtual-machine engine. The VM-T runtime environment is the virtual machine combined with a set of customized components. In the next two subsections we present the basic technologies related to Terra – the nesC/TinyOS environment

used to built the VM-T and the Céu language that we used as the basis for Céu-T.

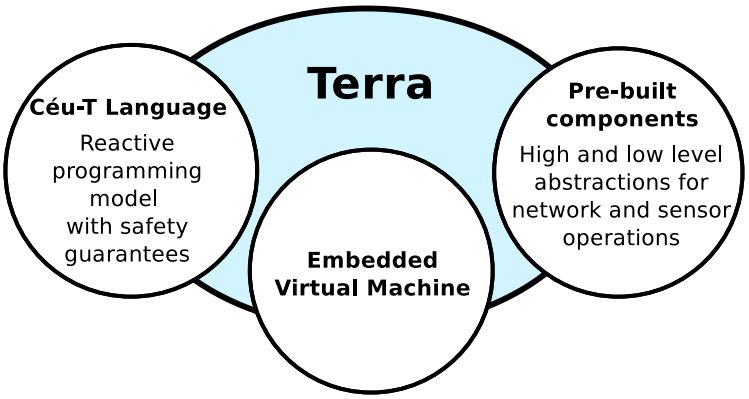


Figure 2.1: Terra programming system basic elements.

2.1.1 nesC and TinyOS

TinyOS (Levis et al., 2004) is an operating system designed for resource-constrained WSN devices. It was built using the nesC programming language (Gay et al., 2003). The nesC language is also used as the application programming language. Its official revision is the 2.1.2 version on August,2012, but the code repository still has some activity. Despite the age of its last revision, TinyOS currently is the WSN operating system ported to more different types of motes. The list of WSN motes maintained in Wikipedia (Wikipedia, 2015) has two predominant operating systems for WSN devices – TinyOS and Contiki (Dunkels et al., 2004). TinyOS covers 49% of total mote types and Contiki covers 12%. Most of the other operating systems in the list cover only one type of mote. These other systems, in general, are proprietary systems or systems created for single experiments. If we count only our target type of constrained motes, with RAM ranging from 4k bytes to 10k bytes, TinyOS covers 66% and Contiki covers 16% of these type of motes.

nesC is a component-based programming language that extends the C language with an event-driven programming model. It includes resources like task management interface and atomic sessions. nesC is a static language that avoids dynamic memory allocation and that determines the program call graph at compile time.

A nesC program consists of a set of components that implement services defined by *interfaces*. An interface may define *commands* and *events*. The component that uses an interface may call its commands and must implement its event handlers. On the other side, the component that provides an

interface must implement its commands and may signal its events. This model allows for different implementations of the same interface, facilitating system configuration for different platforms.

TinyOS provides a library of components and some tools that simplify the task of building new applications. TinyOS does not work as a conventional operating system which runs user applications, but rather as a library that must be linked to these applications to build a single executable program. This executable must be loaded into the WSN mote.

TinyOS implements a task queue to support nesC task management. In TinyOS, each task runs to completion, one at a time. In that way, only an interruption handler may run concurrently with a task. Typically the interruption handler posts a task to the scheduler as soon as possible, to avoid conflicts. When the task queue is empty, TinyOS keeps the CPU in sleep mode to save energy.

The use of the nesC component model facilitate modularity, allowing TinyOS to have equivalent components to access different types of hardware. The selection of suitable components for each hardware is done during the build process and is transparent to the user.

In addition to the tools to compile and load programs, TinyOS provides the TOSSIM tool (Levis et al., 2003) for network simulations.

2.1.2 The Céu programming language

Céu (Sant'Anna et al., 2013) was originally developed as a compiled language, and has bindings[1] to Arduino[2], to the TinyOS environment, and to SDL[3] running in conventional computers (Linux, Windows, and Mac OS X). Céu is a reactive language strongly influenced by Esterel (Boussinot and Simone, 1991). 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* (Gay et al., 2003) and *Contiki* (Dunkels et al., 2004), Céu can keep sequential and separate lines of execution (trails) for each activity in the program. Trails in Céu are guided by reactions to the environments. 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).

[1] `http://ceu-lang.org/`
[2] Arduino open-source microcontroller platform (https://www.arduino.cc/)
[3] SDL - Simple DirectMedia Layer (http://www.libsdl.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 2.2.

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

Figure 2.2: 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 (canceling 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 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.

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 the original

language, the programmer can resort to C for this tasks, but this means loosing the safety guarantees.

2.2 Terra in details

In this section we describe in more details the Terra programming system. Branco and others (Branco et al., 2015) describe a previous version quite close to the this one.

Figure 2.3 shows the Terra application life cycle. The Céu-T program is compiled and checked statically in the user computer to generate the virtual machine bytecode. The user then transfers the generated bytecode to the motes in the network and the virtual machine runtime, previously installed, executes the bytecode. In this version of Terra the same bytecode is loaded in all nodes. The only part of scripts that escapes static analysis are calls to components provided by Terra's VM, which are encapsulated in modules and have been extensively tested beforehand. In this way, Terra strives to provide adequate abstractions while providing a safe execution environment and allowing remote program update.

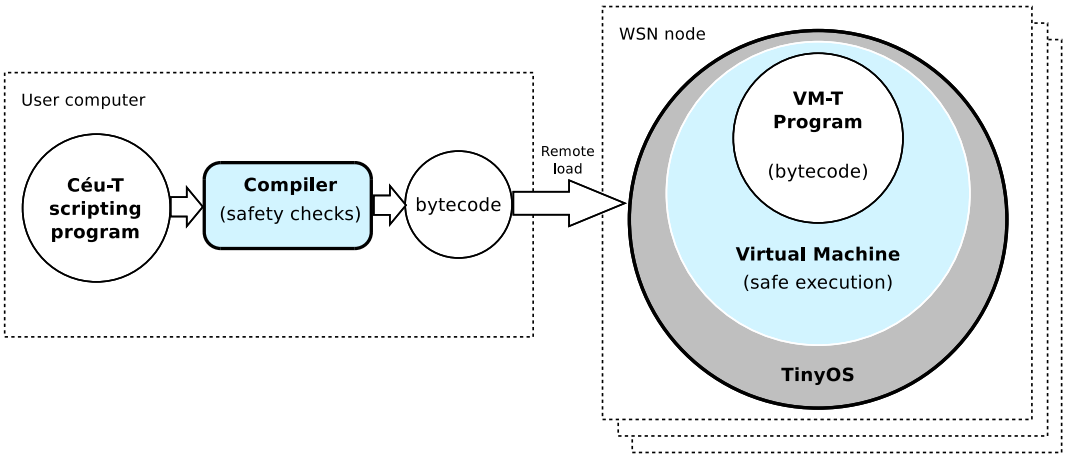


Figure 2.3: Terra application life cycle: compilation and execution.

2.2.1 Céu-T scripting language

For the use of Céu in Terra, we implemented a new variation of the language that generates code for VM-T. Céu is originally compiled to C and Céu scripts can include chunks of C code, however, any call to C is exempt of verification. 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. The Céu-T language inherits almost all characteristics of Céu 0.3 version[4] discussed in section 2.1.2, and its implementation inherited all the safety checks from the original compiler. Because Céu relies on C for typing, function calls, event operations, and expressions, we had to extend Céu-T to include these language elements. Figure 2.4 shows these compilation differences.

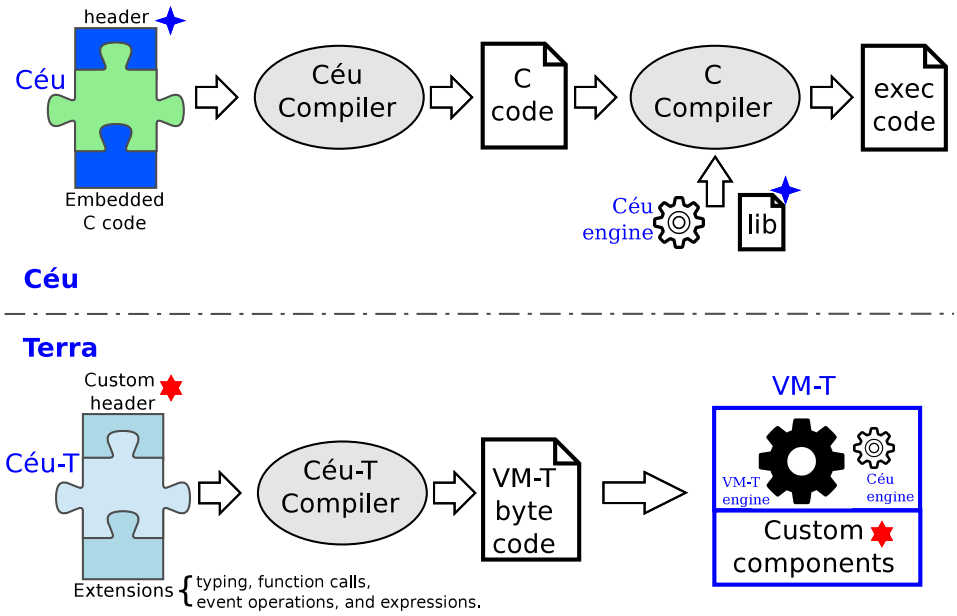


Figure 2.4: Ceu x Terra

In Terra, the Céu-T language is used only to glue components written in *nesC/TinyOS*. All virtual-machine code and low-level components rely on the TinyOS architecture. Céu-T and components in the VM-T 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. In the Céu-T 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). To allow the configuration of these events and system calls we extended the Céu-T language with a special syntax for a configuration block, as detailed in the Appendix section A.3 *Integration between script and components*.

The type system for WSN applications is, in general, very simple. Besides the basic integer types, we need some kind of data structures to exchange data with the customized components. For example, to send a radio message we need to populate the data message, and this data structure may be different

[4]Terra is based on the previous version 0.3 of Céu (Sant'Anna et al., 2013).

depending on the application. The type system we developed and the facilities for defining data structures are explained in the subsection *Types and data structures*.

Types and data structures

In Céu, data definition and manipulation relies on the use of C. For Céu-T, we defined a basic type system that includes integer values with 8, 16, and 32 bits and float values of 32 bits. Pointer types are not allowed for safety reasons. The basic types supported by Céu-T are: byte, short, long, ubyte, ushort, and ulong – respectively 8, 16, and 32-bit signed and unsigned integers and float.

But we need more complex data structures to use in the interfaces between the user Céu-T script and the VM components. For that, we define three types of data structures – one-dimension arrays, registers and packets. One-dimension arrays are defined as a basic type within a dimension. Listing 2.1, in line 3, shows an example of one-dimension array with five ubyte elements. A regtype declaration creates new register type. A register can only have fields that are values of basic types or arrays of basic types. Listing 2.1 (lines 5–11) shows an example of register declaration and use. We also defined a packet declarations for partial predefined structures. This kind of data structure is useful when a component interface needs to specify some fields and the user can define other fields as needed. A typical example of packet use is in the radio message interface, where some fields are mandatory, such as message type and target node, and other fields depend on the application needs. The packet command declares a new abstract register type which must contain, at least, a field of a special type called payload and its length in bytes. Later on, during application writing, the pktype declaration may be used to create a new register type based on the abstract register. In pktype declaration the user must specify at least one basic type field or array field for the abstract packet register's payload. The only restriction is that the sum of bytes of all user-defined fields can not exceed the payload length defined by the packet command. Listing 2.1 (lines 13–27) shows an example. The packet declarations can be used only in the configuration block as its use is intended to the developer of the customization.

Céu-T type system has simple rules for expressions. Assignments of integer values to any integer variable are allowed and, if necessary, automatic type casting occurs. Assignments of integer to float or float to integer are also allowed and, if necessary, automatic type casting occurs. In expressions, math operations with at least one float operand will be evaluated converting all

operands to float. In all other cases, the operation will be done with integers. Each automatic type casting generates a compile-time warning. A register value can be assigned only to another identically-typed variable.

The integer and float assignment rules are also applied to arguments of functions and events. A register argument is always passed by reference and an additional rule verifies the compatibility between the packet type and the register type.

The end of Listing 2.1, lines 30–34 show examples of valid assignment for the variables, array, and register defined in the previous rows.

Listing 2.1: Examples of the Terra type system implementation

```
1  var ushort nodeId;           // Simple integer var
2  var float average;           // Simple float point var
3  var ushort[5] sensorReads;   // Array var
4
5  regtype myData with          // Register type
6      var ubyte sequence;
7      var ushort nodeID;
8      var ushort sensorValue;
9  end
10
11 var myData sensorData;       // Register var
12
13 // Abstract register type
14 packet radioMsg with
15     var ubyte msgId;
16     var ushort target;
17     var payload[20] data;    // 20 bytes
18 end
19
20 // Register/packet type
21 pktype userMsg of radioMsg with
22     var ubyte seq;
23     var ushort sensorVal;
24 end
25
26 // Register/packet var
27 var userMsg sendMsg;
28
29 // Valid attribution examples
30 nodeId = 5;
31 average = nodeId/2.0;
32 sensorData.sensorValue = sensorReads[0];
33 sendMsg.target=1;
34 sendMsg.sensorVal=sensorData.sensorValue;
```

2.2.2 Terra Implementation

A Céu-T program is compiled to a bytecode file that can then be disseminated to the network nodes, where it is interpreted by the VM-T, which implements the bytecode interpreter, the execution model, the code dissemination service, and some specific customized components.

In the next subsections we present the Céu-T compiler, the component-based VM-T architecture, and the bytecode dissemination algorithm. In the Appendix A.2 we present the basic operation process.

The Céu-T Compiler

The implementation of the Céu-T compiler is based on the Céu compiler implementation. The compiler was written em Lua programming language and uses the LPeg library (Ierusalimschy, 2009) for pattern-matching. From this base implementation we inherit all the static checking. The compiler checks scripts for non-deterministic memory accesses and tight loops (loops without *awaits*), and others properties, such as whether all possible block cancellations are correctly captured. Also, the compiling process uses the C preprocessor (cpp) to allow inclusion of header files, macro expansions, conditional compilation, and line control.

The main modifications for Céu-T are the types and the configuration block described in section 2.2.1 and the bytecode generation. Other modifications include 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 the Céu-T type system 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. We choose to use a stack-based architecture because of its smaller code size in comparison to register-based architecture (Gregg et al., 2005). Since memory is a limited resource, it is important to reduce the bytecode program size. In Terra all script variables are statically arranged in the program memory and the stack is used only for expression operations. Some assignment instructions access directly the memory variables and the push/pop instructions put and get values into/from the stack. All expression operations are evaluated using the stack. During code generation the compiler checks for code size optimization opportunities. Whenever possible, code generation prioritizes

accesses to memory instead of use of stack. Expressions with binary operations like sum or minus always need to use the stack.

Listing 2.2 shows an example of optimization for a simple assignment like v1 = v2;. Considering both as short type variables and with the memory address bellow 256 (i.e needing only 1-byte for address). In this example, the first part (lines 2–8) pushes to the stack two 16 bits addresses for each variable and, the last instruction, pops these addresses to copy the contents of one address to the other address. The second part (lines 12–14) uses only one instructions that does all work without using the stack. In this case using addresses of 8 bits. In this example, the optimization changes from seven bytes of non-optimized code to three bytes of optimized code

Listing 2.2: A code optimization example.

```
1 /*** Not optimized and using stack ***/
2 push &v2          : opcode
3                   : addr2Low
4                   : addr2High
5 push &v1          : opcode
6                   : addr1Low
7                   : addr1High
8 setshort          : opcode
9 total of 7 bytes of code + 2 stack word (8 bytes)
10
11 /*** Optimized ***/
12 setshort &v1, &v2 : opcode
13                   : addr2Low
14                   : addr1Low
15 total of 3 bytes
```

Another kind of optimization is to reduce the path to terminate hierarchical blocks. For example, in the follow code:

```
1     if x do
2         <do something>
3         if z do
4             <do something>
5             if w do
6                 <do something>
7             end
8         end
9     end
```

At the end of the most inner block, the compiler generates a instruction to jump to the end of the middle block, where the compiler also generates a instruction to jump to the end of the outer block. In this case the most inner

block may have a instruction to jump directly to the end of the most outer block.

VM-T architecture

The Terra virtual machine (VM-T) is composed by three modules as shown in Figure 2.5. The interfaces between modules or sub-modules are indicated by arrows.

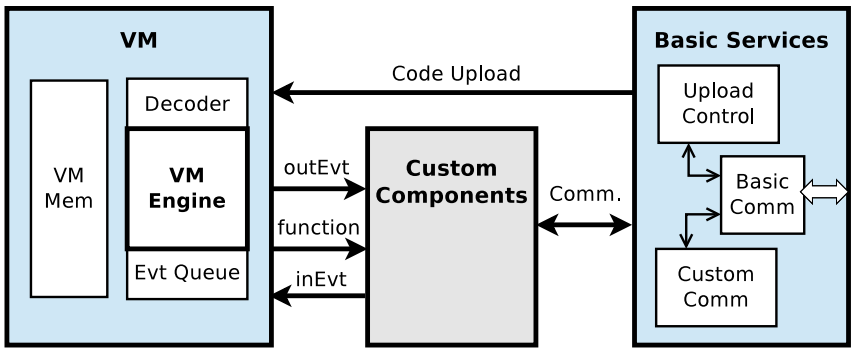


Figure 2.5: VM-T modules

The *VM* module is the main module. It provides an interface for receiving new application code from the *Basic Services* module (Code Upload interface) and three interfaces for customized events and functions (outEvt, function, and inEvt interfaces). The *Engine* submodule controls the execution of code interpreted by the *Decoder* submodule and handles external events received from the *Event Queue* submodule. As the VM-T is implemented using TinyOS, each task runs to completion in a single-threaded model, guaranteeing race-free conditions over application trails and embedded operations. The only exception are the interrupt-handlers, which must be isolated in the low-level functions. Terra uses a similar control for trail execution that Céu uses to maintain execution guarantees. Basically the application program is broken in execution trails, each trail has an address as entry point and an 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 runtime maintains a set of slots to execute entry points. When an event is received, the engine scans all slots to execute, one by one, all trails that were awaiting this event. Appendix A.1 presents the Céu-T code and the assembler code for this example.

The *Basic Services* module controls the communication primitives to give support to code dissemination (Code Upload interface) and to the custom components module interface (Comm. interface). The *Upload Control* submodule controls the dissemination protocol and loads code into VM program memory.

The *Custom Comm* submodule has a generic interface to support new communication protocols defined at the *Custom Components* module level. All communication protocols implemented in Terra aim to be operational with no intention of implementing the most optimized algorithms.

The *Custom Components* module implements specific flavors of Terra. The developer of new customization needs only to implement the custom events and functions inside this module and write the equivalent configuration file to be used by a Céu-T script. It is possible to start from a very basic customization of Terra to include the new events and functions.

Currently, an output event returns a void value. These events have one argument of any type, including void, a basic type address, a register, or a packet. Because this argument is passed to the VM-T interpreter as an instruction parameter, constants and variables are passed by value and registers and packets are passed by address. The custom component that implements the output events must handle correctly each argument.

Custom functions may have none or many arguments. Arguments can be basic types, basic type addresses, registers, or packets. 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 a basic type value by pushing it back to the stack. The use of the stack for the returned value is important to enable the use of functions inside expressions.

An input event may be defined to return a basic type value or an address. In all cases, the returned data is copied directly 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. Because this operation does not use the stack, input events can not be used inside expressions.

Bytecode dissemination algorithm

This 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 VM-T runtime includes a dissemination algorithm that floods code blocks into the network. Each block goes as a wave. The basestation starts the code dissemination process with a *newProgramVersion* message and next sends the bytecode blocks. Each node forwards each incoming 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 identify when it is a new program version.

A periodic timeout forces each node, if necessary, to request any missing block from its neighbors.

2.3 Terra Customizations

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 basic library of components that 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 the goal of this basic library 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 the next subsections we present three different flavors/customizations of Terra. The first customization, called TerraNet, is a very light customization using only simple radio communications and sensors operations. The second customization, called TerraGrp, uses the basic library of components, which includes the high-level abstractions for group management. The third customization, called TerraVolcano, is a special customization for the Volcano project (Tan et al., 2010; Tan et al., 2013) used in the evaluation of Terra.

2.3.1 TerraNet - Basic Operations

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 that application. TerraNet offers only basic communication components to send and receive messages within radio range. This allows the programmer to write Céu-T applications that uses a specific communication protocol. This is useful, for instance, to allow specific applications to decide how they will handle faults or even routing. The macro system can 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 VM components.

To support this line of experimentation, we designed a basic Terra configuration called *TerraNet*. This Terra customization mainly provides only very basic "send" and "receive" events and local sensors readings.

Communication

The SEND() command is a basic send command where only the nodes in the radio range may receive the message. If the field target is set to BROADCAST value, all nodes in the range will receive the message. On the other hand, if this field is set to a specific node identifier, only this node will receive the message (if the node is in the radio range). A SEND_DONE() event indicates that the send request was processed by the radio. The RECEIVE event returns a received message. A variation of the SEND() command is the SEND_ACK() command that requests an acknowledgement from the target mote. In this variation the SEND_DONE_ACK() event return a boolean value indicating the acknowledgement. Additionally, TerraNet implements a simple message queue to support message buffering needs.

Listing 2.3 presents a simple code example where the node 2 sends a counter value to its neighbor nodes at one hop radio range. Lines 1–4 define the message packet and creates the message variable. Lines 5–7 initialize the default fields of radio message. In the lines 12–17, the node 2 sends the message. In the line 20 the other nodes receive the message.

Listing 2.3: A simple TerraNet example.

```
1  pktype msg from radioMsg with
2      var byte count;
3  end
4  var msg dataMsg;
5  dataMsg.type=1;
6  dataMsg.source=getNodeId();
7  dataMsg.target=BROADCAST;
8
9  dataMsg.count = 0;
10
11 if getNodeId() == 2 then
12     loop do
13         await 10s;
14         dataMsg.count = dataMsg.count +1;
15         emit SEND(dataMsg);
16         await SEND_DONE;
17     end
18 else
19     loop do
20         dataMsg = await RECEIVE();
21         emit LEDS(dataMsg.count);
```

```
22      end
23  end
```

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 Céu language with the await <time> command. Our example (Listing 2.3) illustrated the use of leds and timers.

2.3.2 Terra Group - Basic Library

In order to determine the set of components that we should include in the basic Terra library, we considered, on the one hand, 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) and, on the other hand, some typical applications (Newton et al., 2007; Cervantes et al., 2008; Kothari et al., 2007). As a result of this work (Branco, 2011), we organized the needed functionality 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 the three first areas. The local operations are the same as defined for TerraNet in 2.3.1.

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. The output event SEND_GR can be used to send a message either to all nodes in a group or to a specific node. When a message is received by a node, it is signaled to the script code through a new REC_GR input event. In the 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 output event SEND_BS script is signaled.

To provide delivery guarantees (no message 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.

Grouping

Because WSN applications frequently involve large numbers of nodes, 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 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 component allows the program developer to implement several alternative group structures. To have different alternatives, the grouping component provides two identifiers that, when combined, define different clusters of nodes (groups/subgroups). By definition the first identifier (group level) may be used to associate a specific message data structure allowing to have different type of messages for each group instantiated. The subgroup parameter (second identifier) is used to define a kind of subgroup of nodes for the same group type As an example, in order to broadcast a message one can initialize all nodes with the same group identifier. In this case, the two identifiers are set with same value for all nodes and the maximum of hops can be used to define the reachable range of the group. Another example would be the creation of dynamic groups: the group type parameter is defined with

same value to all nodes and the subgroup parameter can be defined by the current node' state, for instance based on the last value read from a sensor. These two examples may coexist in the same application, simply by using two different parameter values for group type.

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 current leader. In the case when a leader has not yet been defined, a new election is started. The 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.

Listing 2.4 presents an example program that uses grouping and communication facilities. Line 2 invokes the system call grouplnit() 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 grouplnit() define constant group identifiers respectively grId for group type and subgrId for subgroup. The fourth argument defines the highest range in hops. The fifth argument initializes the "active" flag as TRUE. The next arguments define the "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 inside the application code. In Listing 2.4, the gr1 variable (defined in line 1) is used for that. The script can modify these values at any time.

Listing 2.4: Grouping and communication example in Terra.

```
1  var group_t gr1;
2  grouplnit(gr1,1,1,3,TRUE,OFF,0);
3
4  pktype msg from msgGR_t  with
5      var byte val;
6  end;
7  var msg countMsg;
8  countMsg.grId = gr1.grId;
9  countMsg.node = BROADCAST;
10 countMsg.msgId = 1;
11
12 if getNodeId() == 2 then
13     countMsg.val = 0;
14     emit LEDS(3);
15     loop do
16         countMsg.val = countMsg.val + 1;
17         emit SEND_GR(countMsg);
```

```
18          await 1s;
19      end
20  else
21      loop do
22          countMsg = await REC_GR;
23          emit LEDS(countMsg.val);
24      end
25  end
```

In the program of Listing 2.4, node 2 periodically sends a counter value to its neighbors (lines 13–19). Each neighboring node shows the three less significant bits of the received counter on its leds (lines 21–24). As discussed before, the system call grouplnit() (lines 1–2) makes each node join a group described in the structure assigned to gr1. A message group package type — msg — is defined in lines 4–6. In this case, we create a field value val to send the counter value. The message variable counMsg is created in line 7 and its default fields are initialized at lines 8–10. The countMsg.grId = gr1.grId associates this message with the group identifier grId=1. The send output event in line 17 sends the countMsg message structure to all neighbors that has defined a group with same grId=1. The await command in line 22 waits for a group message and updates its countMsg structure with the received data.

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 implementation of this component provides the following built-in functions: SUM (sum of values), AVG (average values), MAX (maximum) and MIN (minimal value). In addition, each aggregation operation is also associated with a relational operator ($>, <, <=, >=, ==, != $) and a reference value. Besides the end result, the aggregation operation also accumulates the number of partial values that evaluated as true in this criterion. Aggregation operations are performed by group of nodes, that is, one aggregate value is produced for

each group. The script starts an aggregation operation by emitting the output event AGGREG with the specific aggregation identifier as parameter. When the aggregation is completed, this is signaled 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 time.

The program in Listing 2.5 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 aggregInit(). This aggregation will be associated with the gr1 group (the second argument). The third and fourth arguments to aggregInit() 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 11 uses a predefined data structure type that will hold the result of the aggregation operation. In lines 15–16, 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 17, 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 19, 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 SEND_GR event, but in this case msgBS_t type does not have predefined field grId.

Listing 2.5: Aggregation and communication example in Terra.

```
1  var group_t gr1;
2  groupInit(gr1,1,0,2,TRUE,eACTIVE,0);
3  var aggreg_t agA;
4  aggregInit(agA,gr1,SID_TEMP,fAVG,opGTE,0);
5
6  pktype msg from msgBS_t  with
7      var ulong average;
8  end;
9  var msg dataMsg;
10 dataMsg.msgId=1;
11 var aggDone_t data;
12
13 loop do
14     await 10s;
15     if (getNodeId() == gr1.leader) then
16         emit AGGREG(agA);
```

```
17          data = await AGGREG_DONE;
18          dataMsg.average = data.value;
19          emit SEND_BS(dataMsg);
20      end
21  end
```

2.3.3 Terra Volcano - CPU Intensive Operation

Volcano is an application that uses WSN as a cheap alternative for traditional volcanic instrumentation. The application was built in nesC/TinyOS and is detailed by Tan[5] (Tan et al., 2010; Tan et al., 2013). The main idea is to reduce raw data transmission doing some in-network signal processing. In the laboratory version, that we had access, the application maintains real data in the node flash memory and emulates a seismic sensor that reads these data as streams.

In our TerraVolcano customization, we break the Volcano application into five functions: Mean, Seismic Energy, Energy Scale, Copy Buffer, and Detect. The Mean operation computes the intensity mean of valid values of the raw data from seismic sensor. The Seismic Energy operation computes the seismic energy considering the intensity mean. The Energy Scale operation finds the scale of the energy computed. The Copy Buffer operation fills a five stage buffer with data to be used in Detect operation. The Detect operation includes the Fast Fourier Transform (FFT) and the seismic detection algorithm. Additionally, this customization offers an interface to read seismic data as a stream and a storage interface to load the Gaussian data model used in the detection algorithm.

This kind of application needs a lot of memory to accommodate all data vectors. The original work uses the TelosB mote with 10kB of RAM and 48kB of ROM. In our case, combining the Terra Virtual Machine code with the Volcano components overflows the available ROM space of the TelosB. Our alternative was to exclude some basic functionalities from Terra to be able to add Volcano operations. In the evaluation, in section 3.6, we present an use case for Volcano application.

2.3.4 Terra memory usage

Traditional WSN platforms impose an architectural restriction where the microcontroller has, at least, two types of memories. The equivalent to

[5]We thank the authors for making the source code available.

the ROM (Read-Only Memory) where the machine code to be executed is written and the RAM (Random Access Memory) where the program variables, runtime controls, and the execution stack are stored. Using the virtual machine approach, we have to load and execute the VM-T runtime in ROM space and allocate part of the RAM memory to load the script bytecode and variables. Besides the VM-T runtime needs some RAM space for its execution. As we increase the embedded custom components, the use of ROM and RAM, by VM-T, is also increased. Consequently, the memory space for the Céu-T script decreases. Some hardware platforms have memory limitations that may restrict the use of specific configurations. Table 2.1 presents the Terra memory configuration for different hardware platforms.

Table 2.1: Terra memory usage

Customization	**Memory**	**MicaZ**	**Mica2**	**TelosB**
TerraNet	ROM	40.0k	37.3k	35.0k
	RAM	3.6k	3.5k	7.5k
TerraGrp	ROM	55.3k	52.4k	47.1k
	RAM	3.6k	3.5k	7.8k
TerraVolcano	ROM	—	—	45.2k
	RAM	—	—	8.4k

Units in bytes

The ROM utilization depends on the CPU type and the specific TinyOS component implementations for each hardware. The RAM value represents the memory used by variables in VM-T and in TinyOS, including the total memory allocated for the Céu-T script. This is not the full RAM size because we need to leave some memory for the C stack.

Table 2.2: Céu-T script memory size

Customization	**Platform**	**Script max size**
TerraNet	mica2/micaz	2,000
	telos	7,500
TerraGrp	mica2/micaz	768
	telosb	4,800
TerraVolcano	telosb	2,668

Units in bytes

When writing a Céu-T program, it is important to verify the amount of memory used. Table 2.2 shows how much of Céu-T script memory is left to the application programmer in each of the customizations we explored. For example, TerraNet on MicaZ has about 2,000 bytes for the Céu-T script program, but TerraGrp has only 800 bytes on the same platform. This

happens because the TerraGrp components use more RAM than the TerraNet components, consequently leaving little memory to the user script.

Because the radio of MicaZ and TelosB are fully compatible, it is possible to have a heterogeneous network using the same Terra customization. In this case, because the Terra interface is the same for all nodes, it is possible to run the same Céu-T script on all network nodes.

3
Programming evaluation

In this part of the evaluation, we built and evaluated different types of applications using different abstraction levels. For example, the script application can use a specific component that offers a ready-to-use complex routing protocol or can use another component that offers a set of basic communication operations, leaving to the application programmer to implement his own routing protocol. Another example is the use of pre-defined calculation components like a Fast Fourier Transform (FFT), instead of providing the programmer only with basic math functions.

We next recall the list of programming issues defined in Section 1.2.1. In the next section, we defined a set of metrics that were used to evaluate the role of Terra in resolving these issues.

A. Programming complexity
   - A.1. Sequential and event-driven programming
     - A1.i. Learning curve
     - A1.ii. Split phase
     - A1.iii. Global variables
   - A.2. Local starvation
   - A.3. Invalid pointers

B. Networking complexity
   - B.1. Radio operations
   - B.2. Communication protocols

Our main evaluation procedure doesn't cover three items from this list: learning curve, local starvation, and invalid pointers. In the end of this chapter we present our analysis and discussion for these three remaining items. Although most of the cost evaluation is left to the next chapter, we took the opportunity to measure the size of the codes used in the programming evaluation to identify the code dissemination cost.

The next sections detail the evaluation process, presenting the execution strategy, the experiment metrics, the test applications, and the execution of the evaluation.

3.1 Execution strategy and Metrics

In most of our tests, we built two versions of an application: a reactive one, using Terra and the Céu-T programming language, and an event-driven one, using TinyOS (Levis et al., 2004) and the nesC (Gay et al., 2003) programming language. We compared the applications built for the two environments and, in some cases, we compared different applications for the same environment. Table 3.1 contains the metrics we selected for our evaluation. This selection determined the data we gathered.

Table 3.1: Programming – evaluation metrics

Metric	**Description**
Program lines	Number of lines in a program.
Bytecode size	Script bytecode in bytes.
Machine code size	Machine code in bytes.
Code blocks	Number of code blocks to be disseminated.
Global variables	Perception of explicit and implicit global states variables.
Distributed system concerns	Problems found during programming and debugging.
Abstraction level	Positive and negative points using different abstraction levels.

The *program lines* is a traditional metric to measure program size and complexity. Although this metric considers blank lines and comments, it is important in counting the total effort to produce the code. The Céu-T script *bytecode size* and the TinyOS *machine code size* indicate the program size to be loaded. In our case, we are interested in the amount of *code blocks* that must be disseminated on the network for remote installation. The *global variables* are the use of global variables to maintain the program state. This is a key feature used in event-driven programming because it is not possible maintain the local state between two independent events. Protothreads (Dunkels et al., 2006) and Céu (Sant'Anna et al., 2013) also used similar metrics in its evaluation. In Protothreads, the authors focus on reducing the number of explicit state machines and events. In Céu, the authors focus on reducing the global variables. The *distributed system concerns* and the *abstraction level* are qualitative metrics to identify significant points to our evaluation. These points appear in our evaluation text as different items depending on each test variation.

3.2 Test applications

We define four test applications to analyze abstraction level and code complexity. These applications were specially selected so as to exercise different system/network execution models. To maintain a certain degree of impartiality we used WSN applications from the literature in two of four test applications. Also, all applications were tested on real motes. Table 3.2 shows the environments considered in our tests and Table 3.3 describes the application functionality for each test.

Table 3.2: Execution environments

Prog. model	Environment	Description
Event-driven	TinyOS	Low level code environment.
Reactive	TerraNet	Environment with low level abstraction.
	TerraGrp	Environment with high-level abstraction.
	TerraVolcano	Environment with an abstraction of intensive use of CPU.

Table 3.3: Applications

#	Application	Description
1	Multi-Hop monitoring & alarm	A monitoring and alarm application with multi-hop network topology. Requires routing protocol to send messages to base-station. (#1a. Using its own routing script and #1b. using the TinyOS CTP routing component)
2	Complex Grouping	A monitoring and alarm application for different spaces and with multi-hop network topology. Requires routing protocol to send messages to base-station and local group communication protocol.
3	Topology Control protocol	A topology control support that actively varies the radio transmission power to discover the lower energy consumption path.
4	Volcano Application	High processing application to monitor volcanos.

Application #1a, Multi-Hop monitoring & alarm, is a typical WSN application in which the programmer needs to deal with a simple routing protocol. It is a simple application, but exemplifies a network model very commonly used in WSN application. As alternative test application (#1b) we use the routing abstraction CTP (Gnawali et al., 2009) supplied by TinyOS. In this test it is possible to compare the implementation and the execution of a simple abstraction. Also it is possible to compare the same application using different abstraction levels.

Application #2, Complex Grouping, exercises a more complex programming pattern where the programmer deals with network subgroups, coordinator nodes, and also with a routing protocol. In this case, the grouping control must be implemented from scratch in the low abstraction level environment of TinyOS. This application is very important for understanding the impact of using a high-level abstraction.

Application #3 is a topology builder based on radio transmission power that was implemented by Auza in his master thesis (Auza, 2013; Auza et al., 2014). This application allows us to evaluate the use of Terra for writing more complex network protocols.

Application #4, Volcano, is a CPU-heavy application built in nesC/TinyOS to support in-network collaborative signal processing algorithms in the Volcano experiment as defined by Tan (Tan et al., 2010; Tan et al., 2013). This is an application that stresses the limitations of Terra as to program memory size and processing capacity. Also, it allows us to work at a very high abstraction level and to experiment with scripts that uses higher or lower-level constructs.

Table 3.4 shows each test application and the respective execution environment.

Table 3.4: Applications X Execution environments (Abstraction level)

Application	**TinyOS**	**Terra Net**	**Terra Grp**	**Terra Volcano**
Multi-Hop monit. & alarm	1a + 1b	1a	1b	
Complex Grouping	2[*]		2	
Topology Control	3	3		
Volcano	4			4

[*] - only pseudocode for TinyOS version.

3.3 App #1 - Multi-Hop monitoring & alarm

We implemented two variations of application #1. The first one (#1a) implements its own algorithm for message routing and the other one (#1b) uses CTP (Gnawali et al., 2009), a message routing protocol already implemented in the TinyOS. We compare these two applications using two programming models: event-driven and reactive. For the event-driven model, we built #1a and #1b applications using nesC/TinyOS. For the reactive model, we use two flavors of Terra, TerraNet in application #1a and TerraGrp in application #1b. While TerraNet doesn't include any support for message routing, TerraGrp includes the CTP routing abstraction. These applications allow us to examine

the use of different abstraction levels. At the end of this section we present the results of the Terra version applications running in a network of 14 real nodes.

3.3.1 App #1a - programming the routing algorithm

In this application, each node periodically sends its temperature to the central computer (via base-station node). Nodes also send an alarm message when the temperature value exceeds a predefined value. The alarm check period must be much smaller than the monitoring period, to allow a fast reaction. Communicating with the central computer requires support for message routing from any network node to the base-station node.

Our routing solution is based on a spanning tree in which the root node may be the base-station node or any node in the radio range of the base-station. The root node starts a flooding message that is repeated by all nodes on a best-effort basis. A parent node is defined by the first message received, and nodes must repeat only this first message. For simplicity, we are ignoring failures. During the monitoring operation, after the tree construction process, a node sends the reading or alarm messages to its parent node. All nodes must redirect the received messages to their parent node. The root node must redirect the received messages to the central computer via the base-station node. To have a minimal guarantee of message delivery, the send and receive operations must implement an acknowledgement protocol and avoid message duplication. The send operation must implement an output buffer to avoid loss of message caused by concurrency in the radio service.

Because this test is the first one in our text, we present some general considerations about comparing an application built with an event-driven model and with a reactive model. Listings 3.1 and 3.2 shows the pseudocodes for a simplified version of our test application. The event-driven programming model is represented here by the nesC language and the reactive programming model is represented by the Céu-T language. These pseudocodes don't contain the code for message queues, message retries, and duplicated-message checks.

In the nesC program version, as in traditional event-driven programs, the code is split into several nesC event procedures (callbacks or event handlers). In nesC we use call to request a command and event to define an event handler, while in Céu-T we use, respectively, emit and wait. To the nesC application code it doesn't matter where an event handler is positioned in the text. The idea is that an event procedure is always ready to be called and its execution condition is controlled by the user code, via global variables. This kind of combination, considering global variables and event procedures,

amplifies possible valid and invalid control combinations that the programmer must be aware of. This situation complicates program debugging and test cases creation, and also creates an error-prone environment. On the other side, Céu-T enables writing a more structured program in which the programmer may combine sequential and parallel structures. This approach also allows creating different operation stages. For example, it is possible to enable or disable an event handler depending of program flow.

Listing 3.1 presents the nesC pseudocode version. Although we differentiate the pseudocode in two parts, all events are defined for the same execution context. The Céu-T version presented in the Listing 3.2 has two explicit execution contexts. The first stage is executed before the second stage. This kind of separation avoids context mixing and is important to simplify the number of possible control combinations. Also, using parallel structures to separate concurrent contexts reduces possible conflicts in global variables.

Listing 3.1: The nesC pseudocode for the test application #1a.

```
1  hasParent = false
2  alarmMute = false
3
4  :first part - Build ad-hoc tree
5      event booted
6          start radio
7      event radio started
8          if root node then
9              broadcast discover message
10     event discover message
11         if hasParent is false then
12             hasParent = true
13             broadcast discover message
14             start data periodic timer
15             start alarm periodic timer
16             start monitoring periodic timer
17
18 :second part - monitoring functionality
19     event alarm timer
20         read sensor
21     event sensor done
22         if is an alarm and alarmMute is false
23             alarmMute=true
24             start mute timer
25             send alarm message to parent node
26     event mute timer
27         alarmMute = false
28     event data timer
29         send data message to parent node
30     event data message
```

```
31          send data message to parent node
32
33 : empty events must be declared
34     event discover message send done
35     event data message send done
36     event radio stopped
```

Listing 3.2: The Céu-T pseudocode for the test application #1a.

```
1 :first stage - Build ad-hoc tree
2   If is root node then
3     broadcast discover message
4   else
5     wait to receive a discover message
6     broadcast discover message
7   end
8
9 :second stage - monitoring functionality
10  do -- in parallel
11    periodic loop
12      read sensor
13  with
14    periodic loop
15      if is an alarm then
16        send alarm to parent node
17        wait mute time
18  with
19    periodic loop
20      send sensor to parent node
21  with
22    loop
23      wait sensor or alarm messages
24      send message to parent node
25   end
```

Another point worth discussing is related to the idea of split-phase operation. A split-phase operation is when we have an event reacting from an action triggered by a command. For example, a sensor reading must be requested and, when the sensor value is ready, an event is generated to the user application. Further example are the timer command with its associated fired event, and the send message command, with the sendDone event. In environments like nesC/TinyOS, this kind of operation multiplies the number of event handlers distributed in the application code. The full application #1a written in nesC code has 16 event definitions and 24 calls to commands. In the case of Céu-T, an action like sensor reading or send message uses an emit command to trigger the action and the await command to register the event

handler. The await command is a local blocking operation that blocks only the command trail. A typical sensor reading in Céu-T is written as:

```
...
emit REQ_TEMP;
val = await TEMP;
...
```

Timers are included explicitly in Céu-T, and do not require a split-phase operation. An example of timer of five seconds written in Céu-T is:

```
...
await 5s;
...
```

This Céu-T approach reduces the use of global variables because it leaves the split-phase commands contained in a unique local context. This simplification allows a reduced code and less complexity as presented in the pseudocode of Listing 3.2. The full code of application #1a written in Céu-T has 12 awaits and five pairs of emit/waits.

A third point is about commands and event interfaces. In nesC/TinyOS, the component interfaces tend to be more complex than the component interfaces of Terra. In Terra, we took care to encapsulate some complexities, giving simple and clear interfaces to the application script. In general, in nesC, the programmer needs to write more code than in Céu-T for the same operation. Table 3.5 shows the results for the Terra and TinyOS applications. We note that the total of lines for the nesC version is about 1.5 times the total of Céu-T version. In this application, because it implements a low-level algorithm with no need for any high-level component, the size of programs has similar magnitude.

Table 3.5: Quantitative metrics - for app #1a

Metric	**Terra**	**TinyOS**
Program lines	199	307
Bytecode size (bytes)	650	–
Machine code size (bytes)	40,000	15,566
Code blocks	28	649
Global variables	4	5

As regards bytecode size, because Terra already embeds the core of nesC/TinyOS and other several operations, the machine code size of the TinyOS application is about 24 times the bytecode size of Terra. Although the machine code of VM-T runtime is 3.13 times the size of the application written for TinyOS, we assume that in the Terra the VM-T runtime was previously loaded via wired interface and we need to disseminate by radio only the new

bytecode. (Specific details about Terra runtime memory usage is presented in Section 2.3.4.) The bytecode size in the case of Terra or the machine code size in the case of TinyOS defines the amount of data needed to disseminate a program in the network. The necessary number of machine code blocks is computed assuming that one single message can carry up to 24 bytes of code. In the case of Terra a message can also carry up to 24 bytes. These blocks are aligned with the Céu-T script memory and the program start may happen in the middle of a block. Because of that, one or two additional blocks may be needed to carry the complete bytecode. Assuming that the energy cost for the radio is related to the number of transmitted messages, we can use the number of blocks of codes to compare this cost. In this example, the cost of dissemination of the TinyOS machine code is also about 24 times the cost of Terra. This is not a new result, as similar comparison are described in Maté (Levis and Culler, 2002)

Analyzing the global variables in both versions, we identified four common variables: the last sensor reading, the parent node Id, the message sequencer Id, and the neighbor table. The difference is an additional global variable in the nesC version that avoids alarms occurring very close to each other. In the Céu-T version we resort to a parallel structure for a solution that doesn't depend on a global variable. A similar solution might be also possible in nesC code, but substituting a global timer for the global variable.

For this specific network algorithm, in the Céu-T version we identified two patterns that facilitate this kind of implementation. One pattern is the receive/send as found in the second block of:

```
if nodeID==ROOT then
    msg1.target = BROADCAST;
    msg1.type = FLOOD_MSG;
    emit SEND(msg1);
else
    msg2 = await RECEIVE(FLOOD_MSG);
    parentNode = msg2.source;
    emit SEND(msg2);
end
```

This pattern is very useful to broadcast messages in an ad-hoc network like a WSN. In this case, the root node starts the flooding process and any other node broadcasts only the first received message. Each node saves as its parent the message-source node for this first message. A variant of this pattern is as follows:

```
par do
    // Local data generation
    loop do
```

```
            await PERIOD s;
            msg1.target = parentNode;
            msg1.type = ROOT_MSG;
            emit SEND(msg1);
        end
    with
        // Routing mechanism
        loop do
            msg2 = await RECEIVE(ROOT_MSG);
            msg1.target = parentNode;
            emit SEND(msg2);
        end
    end
```

This variant shows how to construct a simple routing mechanism. The first parallel block generates and sends, periodically, some local data to its parent node. The second block awaits new messages to forward them to its parent node. The final result is that all data messages are routed to the root node. This second pattern is a naive solution. A more robust solution must consider message queues, message acknowledgements and retries, and duplicated-messages check. When two or more points in the code try to send a radio message at same time, this may corrupt the message data in the buffer of the radio. The message queue is used to insert output messages. The messages are recovered in only one point of code to be sent via radio avoiding conflicts. Acknowledgements are used when communicating with a specific target node. In this case the sender requests the confirmation from the target node. In case of absence of the confirmation, the sender may try to resend the pending message. Because the confirmation may be generated by the target node, but not received by the sender, this retried message may be duplicated in the target node. To avoid the processing of duplicated messages, the sender must stamp each message with a unique identifier and the target node must verify duplicated received identifiers.

3.3.2 App #1b - using TinyOS routing

Intentionally, this application has the same functionalities of the application in section 3.3.1, but it uses a pre-built TinyOS component for message routing called CTP (Gnawali et al., 2009).

Because we use a router component, it is not necessary to deal with message acknowledgements, retries, and duplicated messages. The application needs only activate sensor reading, alarm condition checking, and data sending via the CTP interface. TerraGrp already implements an internal message

queue, but in the TinyOS version we need to program the queue.

Listing 3.3 presents the pseudocode for the nesC application. The nesC code is very similar to code for the application without CTP, but it doesn't need the flooding process to build the routing tree. For the Terra version, as the code size is small, we present, excluding the initial variable declarations, the complete Céu-T code in Listing 3.4. In the Céu-T code, without flooding, we got a very concise program with three infinite loops in parallel. In this case, because all parallel blocks are infinite loops, the par statement doesn't need to be specified as an /or neither as an /and. The first loop (lines 5–9) reads the temperature value, while the second loop (lines 11–21) checks the conditions to send an alarm message. The third loop (lines 26–38) sends a data message every 60 seconds. In this case, inside the infinite loop we have a par/or statement showing how to implement the timeout pattern using the await FOREVER command (line 36).

Listing 3.3: The nesC pseudocode for the application #1b.

```
1 : first part - monitoring functionality
2     event booted
3         start radio
4     event radio started
5         if root node
6             setRoot
7         else
8             start data periodic timer
9             start alarm periodic timer
10        end
11    event alarm timers
12        read sensor
13    event sensor done
14        if is an alarm and alarmMute==false
15            alarmMute=true
16            start mute timer
17            send alarm message
18    event mute timer
19        alarmMute = false
20    event data timer
21        send data message to parent node
22
23 : empty events must be declared
24    event data message send done
25    event radio stopped
```

Listing 3.4: The Céu-T code for the application #1b.

```
1 par do
2 /**************************************************
```

```
3  * Monitoring temperature and alarm
4  ***************************************************/
5      loop do
6          emit REQ_TEMP();
7          gblTemp = await TEMP;
8          await 500ms;
9      end
10 with
11     loop do
12         if gblTemp > ALARM then
13             inc msgAlarm.d8[2];
14             msgAlarm.d16[0] = gblTemp;
15             emit SEND_BS(msgAlarm);
16             await SENDBS_DONE(ID_ALARM);
17             await 30s;
18         else
19             await 500ms;
20         end
21     end
22 with
23 /***************************************************
24  * Send temperature value
25  ***************************************************/
26     await 500ms; // Waits to read first sensor value
27     loop do
28         par/or do
29             await 60s;
30         with
31             msgData.d16[0] = gblTemp;
32             inc msgData.d8[2];
33             await (nodeId*500)ms;
34             emit SEND_BS(msgData);
35             await SENDBS_DONE(ID_DATA);
36             await FOREVER;
37         end
38     end
39 end
```

Tables 3.6 and 3.7 compare some values for the implementations with and without CTP. In the example of section 3.3.1, without the CTP component, the nesC code had about 1.5 times more lines than the Céu-T version. Now, considering the CTP component as an abstraction in both sides, the nesC code has about 4 times more lines than the Céu-T version.

If we compare the program size for the two Céu-T applications, we find that the version that implements its own routing has 3.16 times more lines than the version using CTP. We counted, excluding the initial variable declarations, the number of lines of commands and of flow-control statements. The result

Table 3.6: Terra: No CTP X CTP

Metric	**no CTP**	**CTP**
Program lines	199	63
Bytecode size (bytes)	650	189
Machine code size (bytes)	40,000	55,300
Code blocks	28	9

Table 3.7: TinyOS: No CTP X CTP

Metric	**no CTP**	**CTP**
Program lines	307	224
Machine code size (bytes)	15,566	21,548
Code blocks	649	898

was, respectively, 75 and 46 lines for the version without CTP and 17 and 16 lines for the version with CTP. This means that, in Terra, using a high-level abstraction not only reduces code size but also reduces the proportion of number of commands to the number of flow control statements.

If we compare both nesC versions, we find that the number of lines doesn't change as much as in the Terra version. The nesC version without CTP has about 1.4 times more lines than the version with CTP. This means that the use of the CTP component in TinyOS had low impact when we look at the full application code. On the other hand, the TinyOS implementation of the CTP increases the machine code size for Terra and TinyOS. Thus, the ratio between the machine code of the TinyOS version and the bytecode of the Terra version jumps to 114. This was only 24 in the version without CTP. The Terra version without CTP represents an increase of 3.4 times the bytecode size when compared to the version with CTP. The TinyOS version without CTP represents a decrease of 0.72 times the bytecode size when compared to the version with CTP. This difference, comparing the two version in TinyOS, is due to the fact that the TinyOS CTP implementation considers radio link quality and message buffering to offer much more guarantees than our nesC flooding algorithm and this reflects directly on bytecode size. Because the CTP is already embedded in the Terra runtime, and is sent along with the application in the case of TinyOS, this difference reflects in the dissemination cost. In this case the TinyOS version spends 100 times more energy than the Terra version.

The results for global variables are similar to those of the first test. As expected, the main difference is that now the code uses fewer globals.

3.3.3 Real motes experiment

This experiment uses a network of real motes to evaluate the execution of the two Terra versions – applications #1a and #1b. To identify the routing tree topology, we add in the data message of each application the parent of the node considered in the routing tree. Our experimental test uses the *Céu na Terra*[1] testbed, a testbed for WSN experiments built with the support of RNP.

We use a network with 14 MicaZ motes distributed under the desks of a classroom. The nodes are identified from 4 to 17. The output of our experiment is the topology tree formed for each application, #1a using its own routing tree algorithm and #1b using the embedded CTP component. We also register, for each node and cycle, the node messages received by the basestation during 5 minutes of test. Considering a data cycle of one minute, we can receive at least five messages for each node.

Node 8 is configured as the Terra basestation and does not execute the Céu-T script. This node is used only to interconnect the radio network with the USB serial interface of our conventional computer. In application #1a we defined, in the Céu-T script, node 9 as the root node and, in application #1b, the CTP component also uses node 8 (basestation) as the root node.

Figures 3.1 and 3.2 present the routing tree topology formed for the two applications. We colored in gray the nodes that were not discovered in the routing tree building process.

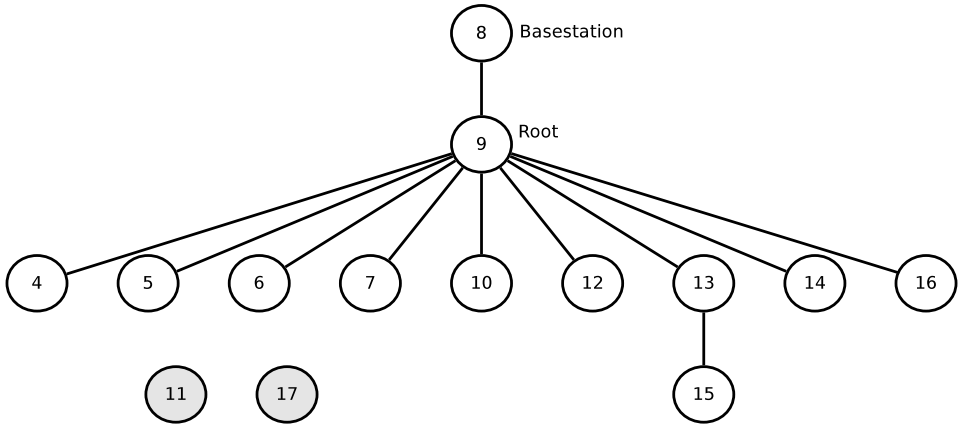


Figure 3.1: Tree topology formed for the application #1a

Table 3.8 presents, for Apps.#1a and #1b, the parent node for each node. We present only one data cycle because all cycles have the same values. We have empty values for node 8 because this node works as basestation and

[1]Ceú na Terra Testbed – http://ceunaterra.voip.ufrj.br/

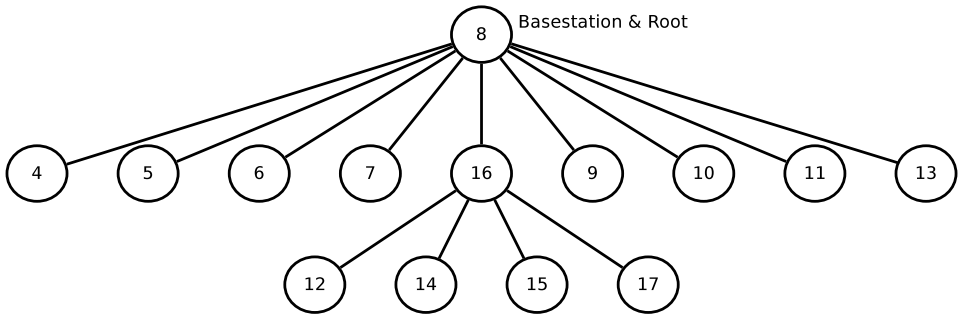


Figure 3.2: Tree topology formed for the application #1b

doesn't run the application. In application #1a, nodes 11 and 17 were left outside of the tree, probably because of the default radio transmission power used in Terra. In the case of application #1b, the CTP component has a more complex algorithm that considers the link quality to build the tree.

Table 3.8: Received parent node for each node – Apps.#1a and #1b

node Id	App.#1a	App.#1b
4	9	8
5	9	8
6	9	8
7	9	8
8		
9	8	8
10	9	8
11	-	8
12	9	16
13	9	8
14	9	16
15	13	16
16	9	8
17	-	16

3.4 App #2 - Complex Grouping

We next present an example of application with complex group requirements. Application #2 monitors the average temperature value for each floor in a building. A coordinator node must be elected for each floor. The criterion for election is battery voltage and the greater identifier is used to break ties. The coordinator node aggregates all values in its floor, computes the average value, and sends this information to the base-station node. The implementations use CTP for messaging routing.

To simplify our implementation, we establish a link between the node identifier and the floor number. We assume a maximum of 10 nodes in each floor and also that the tens of the node identifier represent its floor number. For example, nodes 21 and 25 are in the 2nd floor and nodes 42 and 47 are in the 4th floor.

We again compare the code written for an event-driven model with that of the Terra reactive model. Listing 3.5 presents the pseudocode for the nesC version of the monitoring application. This nesC version is based on some premises to simplify the implementation. We used a predefined leader for each group and, considering the radio range, a connected graph is formed by the nodes of same group. Also, the pseudocode for group messaging does not include message queue management, message acknowledgements, message retries, or checks for duplicated message.

Listing 3.5: nesC pseudocode for the application #2.

```
1  : first part - monitoring functionality
2      event booted
3          start radio
4      event radio started
5          if root node
6              setRoot
7          end
8          if nodeID%10 == 0
9              start data periodic timer
10         end
11     event data timer
12         msg.group=nodeID/10
13         msg.hops=0
14         total=0
15         count=0
16         start timeout timer
17         send request msg
18         read sensor
19     event request msg
20         if msg.group == nodeId/10 and msg.hops < MAX_HOPS
21             parent = msg.source
22             read sensor
23     event sensor done
24         if nodeID%10 == 0
25             total = total + svalue
26             count = count + 1
27         else
28             msg.target = parent
29             msg.value = svalue
30             send answer msg
31         end
```

```
32      event receive answer msg
33          if nodeID%10 == 0
34              total = total + msg.value
35              count = count + 1
36          else
37              msg.target = parent
38              send answer msg
39          end
40      event timeout timer
41          msg.value = total/count
42          msg.group=nodeID%10
43          send average msg
```

Listing 3.6 presents the complete Céu-T code for the monitoring application. Because TerraGrp provides a ready-made parametrized grouping algorithm with support for aggregation, the Céu-T code is very concise. In this case, programming is not the most important task: the user must understand how the grouping mechanism works to be able to set all parameters correctly. Variable floor, initialized in line 2, holds the floor of the node. This variable is used as parameter to the groupInit() function (line 7) and identifies the node group (subgroup parameter). The group type parameter has the same value for all nodes and is represented by the constant GRID. As the leader election algorithm is set as active by argument eACTIVE (line 7), the grouping component elects one node as the leader. This leader node is stored in the leader field of structure grFloor. An aggregation operation, using the created group, is defined by function aggregInit() in line 11. During the periodic loop, only leader nodes start the aggregation process (line 16). The emit AGGREG() command (line 19) starts the aggregation operation. The aggregation result, when completed, is returned in await AGGREG_DONE; in line 20. Then the node can send the results via the emit SEND_BS() command in line 24. Section 2.3.2 presented TerraGrp in details.

Listing 3.6: The Céu-T code for the application #2.

```
1  var ushort nodeId = getNodeId();
2  var ubyte floor = (nodeId/10)+1;
3  var ubyte  seqData=0;
4
5  var group_t grFloor;
6  //        (RegName, grtype,subgr,nhops,status, elFlag,leader)
7  groupInit(grFloor,   GRID,floor,    5,  TRUE,eACTIVE,0);
8
9  var aggreg_t agFloor;
10 //         (RegName, grName,sensorId,agOper,agComp,refVal)
11 aggregInit(agFloor,grFloor,SID_TEMP,  fAVG,  opGT,0);
12 var aggDone_t agResult;
```

```
13
14 loop do
15     par/and do
16         if nodeId == grFloor.leader then
17             await (floor*500)ms;
18             inc seqData;
19             emit AGGREG(agFloor);
20             agResult = await AGGREG_DONE;
21             msgData.d16[0] = agResult.value;
22             msgData.d8[2] = seqData;
23             msgData.d8[3] = agResult.count;
24             emit SEND_BS(msgData);
25             await SENDBS_DONE(ID_DATA);
26         end
27     with
28         await 10s;
29     end
30 end
```

Table 3.9 presents the quantitative values observed for this test. This table only shows the values for Terra, because we implemented the application in TerraGrp and only wrote the pseudocode for the TinyOS version. We considered this to be enough because the complexity of the nesC version is similar to that of multi-hop monitoring application #1a. Additionally, here we had to handle a bunch of parameters, like group identifier, leader node, and aggregation result. Because TerraGrp defines high-level abstractions for components of general use, the interface with these components are also rather complex. The interface parameters must represent, in a way, the several allowed operations modes. For example, a group definition needs parameters like group type, group id, max hop range, activated flag, and election flag. In this case we have to deal with the trade-off between the script size and the complexity of parametrization. Implementing equivalent script directly in nesC and using the TerraGrp components would also reduce the program lines in the nesC version. But, in this case, the user does not take advantage of the guarantees and facilities given by proposed model. Mainly the programming safeties and the remote reconfiguration.

The bytecode size of the application in Terra shows that using an embedded complex component, it is possible to have a relatively complex application with small code size. This Céu-T script needs only 9 bytecode blocks to have its full code disseminated. This is an example of bytecode that starts in the end of the first block, thus adding one block to the dissemination process. We included in Table 3.9 the total memory size used by the script. This total size includes the memory for the bytecode, the variables, the operations

Table 3.9: Quantitative metrics - for app #2

Metric	**Terra**
Program lines	43
Bytecode size (bytes)	174
Machine code size (bytes)	55,300
Code blocks	9
Total memory size (bytes)	316
Global variables	1

stack, and the runtime controls. In TerraGrp, we must take care with the memory limit, in this case 316 bytes. As presented in Section 2.3.4, in the worst case, TerraGrp in the MicaZ platform allows for only 800 bytes of script memory.

This Terra application uses only one global variable, field leader of the group control structure.

To exercise the reprogramming of applications we built a variant of application #2 where only well-lighted sensors participate in the computation. It was enough to set the field status of the group control structure. This field is a boolean flag that indicates if the group is activated or not in the respective node. This flag is computed from readings of luminosity sensor, as defined in Listing 3.7. This additional code increased the application in 13 lines and the number of radio messages to disseminate the application increased only in one. We need to disseminate all the 10 messages to replace the old application and to have the new version running in the network, using the same VM-T previously installed in the nodes.

Listing 3.7: The additional Céu-T code for luminosity control.

```
1  par do
2      // previous control
3  with
4      loop do
5          emit REQ_PHOTO;
6          var ushort photo = await PHOTO();
7          if photo > 10 then
8              grFloor.status=TRUE;
9          else
10             grFloor.status=FALSE;
11         end
12         await 30s;
13     end
14 end
```

3.5 App #3 - Topology Control Protocol

Compared to application #1a, application #3 introduces a more advanced routing protocol. In this application, we evaluate the flexibility of Terra to implement a complex network algorithm.

This is an application from the literature that implements a routing tree algorithm with energy saving for WSN as implemented by Auza in his master thesis (Auza, 2013; Auza et al., 2014). Auza's implementation is based on the theoretical algorithm defined by Chen and Rowe (Chen and Rowe, 2011). To build a routing topology, the algorithm adjusts the radio transmission power to the minimum that maintains the original network tree. The application works in two phases. First, it executes the DTNBOR (Determine the minimal Transmission power to reach each NeighBOR) where all nodes exchange radio messages, gradually reducing the transmission power, to discover the minimal value to reach all neighbors. After that, an application similar to #1a is executed, using the chosen transmission power, to build the routing tree. Here we evaluate the ability of the Céu-T to build network algorithms. In this case, because we do not use ready-made protocols, we use TerraNet which contains only low-level abstractions.

We begin focusing only on the implementation of the DTNBOR algorithm without building the routing tree. The DTNBOR algorithm implementation defines different time execution windows for each node. In the beginning of its execution window, a node sets its radio power to the maximum value and broadcasts a HELLO message to its neighbor nodes. The HELLO message includes the value of radio power in use. Each node, upon receiving a HELLO message, sets its radio power at the received value and sends an answer to the source node. This message exchange is repeated decreasing the radio power settings until the minimal value. As the radio power is decreased, the communication may fail depending on the nodes distance. When it receives an answer, the requester node updates a local table with the node identifier and the radio power level. At the end of this process, all nodes will have a neighborhood table with all neighbor nodes and the respective minimal radio power. The selected radio power for each node is the minimal stored value that reaches all neighbors. The implementation assumes that all nodes start together and plays with different timers to synchronize the message exchanges without any conflict. Because we use the node identifier to define the execution window, the global execution time depends on the highest node identifier.

We used the TinyOS version implemented by Auza (Auza, 2013; Auza

et al., 2014)[2]. We do not present the pseudocode of this implementation. Table 3.10 shows some quantitative data for the original code.

Figure 3.3 shows the block diagram for the Céu-T DTNBOR program. Inside the main block, in the left side of the diagram, we have the DTNBOR block in parallel with a time-out procedure. The time-out is configured to cancel the DTNBOR execution, and considers the execution time for all nodes. For its part, the DTNBOR block has two parallel blocks. One, in the role of active node, sends a request a message to all neighbors. The other, in the role of passive neighbor, answers any received requests. The request block is detailed on the right side of the diagram. This block starts with a command that holds the node in the passive role until the time windows (that is based on the node identifier) elapses. After this, the node repeats, for each radio power step, a broadcast command and a timed receive loop. The broadcast command sends the HELLO message. The receive loop waits for any ANSWER message until a time-out that breaks its loop to returns to the next radio power step loop.

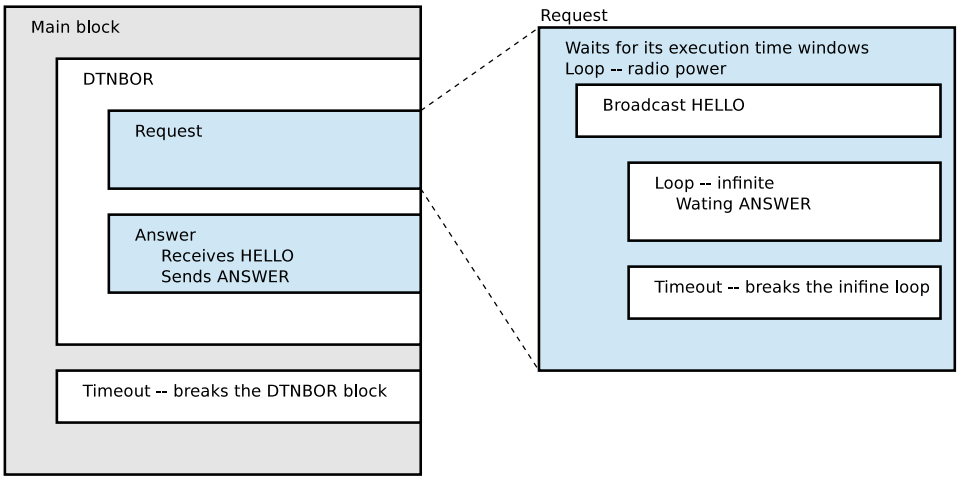


Figure 3.3: DTNBOR block diagram

Listing 3.8 presents the complete Céu-T code for this application. Lines 2–50 contains the DTNBOR block and line 52 the DTNBOR time-out. Inside the DTNBOR block, lines 3–36 contains the request block and lines 38–49 the answer block. For the details of request block, after the delay in line 3, we have the radio power step loop encompassing the broadcast HELLO message at lines 7–14 and the timed received loop at lines 16–34. This radio power step loop is repeated 8 times for different radio-transmission power levels and each received ANSWER message updates the local neighbor table. The answer block, that works in the passive mode, is represented by lines 39–49.

Listing 3.8: The Céu-T code for the application #3.

[2]We thank the author for making the source code available.

```
1  par/or do
2     par do
3        await (((nodeId-FIRST_ID)*T_CYCLE))ms;
4        loop i,8 do
5           var ubyte power = 7-i;
6           // send Hello
7           helloMsg.type=HELLO_ID;
8           helloMsg.source=nodeId;
9           helloMsg.target=BROADCAST;
10          helloMsg.power = power;
11          helloMsg.tp = 0;
12          setRFPower(power);
13          emit SEND(helloMsg);
14          await SEND_DONE();
15          // receive HelloAnswer
16          par/or do
17             loop do
18                respMsg = await RECEIVE(ANSWER_ID);
19                loop x, MAX_NBORS do
20                   if nbor.id[x]==respMsg.source then
21                      nbor.power[x] = respMsg.power;
22                      nbor.stat[x] = 1;
23                      break;
24                   else/if nbor.id[x]==0 then
25                      nbor.id[x] = respMsg.source;
26                      nbor.power[x] = respMsg.power;
27                      nbor.stat[x] = 1;
28                      break;
29                   end
30                end
31             end
32          with
33             await T_ANSWERS ms;
34          end
35       end
36       await FOREVER;
37    with
38       // Receive Hello and send answerHello
39       loop do
40          helloMsg = await RECEIVE(HELLO_ID);
41          respMsg.type=ANSWER_ID;
42          respMsg.target=helloMsg.source;
43          respMsg.source=nodeId;
44          respMsg.power = helloMsg.power;
45          await (nodeId*ANSWER_DELAY)ms;
46          setRFPower(respMsg.power);
47          emit SEND(respMsg);
48          await SEND_DONE();
49       end
```

```
50      end
51  with
52      await (T_TOTAL_DTNBOR) ms;
53  end
```

In WSNs network algorithms, in general, each node must implement the active and passive roles. The active role initiates message exchanges with each neighbor. In this case each neighbor remains passive. In the DTNBOR block the Céu-T parallel structure supports very well this role separation. Lines 39–49 contain a code example for this passive mode where a node answers any received HELLO message. The request block is similar to the pattern already identified in application #1a. Here the active node sends a request (line 13) and waits for a while to receive the answers from its neighbors (loop at lines 17–30). In WSNs, instead of waiting for answers from all nodes, we typically apply a timeout to receive answers only from reachable nodes and then continue the process. The main block also uses a timeout pattern to resume the execution.

Table 3.10 presents the quantitative data for the DTNBOR algorithm. We don't have the compiled result for TinyOS version because the DTNBOR algorithm doesn't compile without the main application for the routing tree. Later on, we discuss the complete application considering the compiled full application. The nesC version has about 3.7 times the number of lines of the Céu-T version. In this case we had a significant difference in the use of global variables. The Céu-T version has to maintain only the neighborhood table. The nesC version, besides the neighborhood table, uses three more globals to control the protocol flow.

Table 3.10: Quantitative metrics - for app #3 partial (Only DTNBOR)

Metric	**Terra**	**TinyOS**
Program lines	97	364
Bytecode size (bytes)	487	–
Machine code size (bytes)	40,000	–
Code blocks	21	–
Global variables	1	4

The implementations above do not include the code to build the routing tree and the code to monitor the sensor. We used the code from application #1a for these two missing operations. Table 3.11 presents the new values for the complete application.

In the complete application, the nesC version has about 2.2 times the number of lines compared to the Céu-T version. In the first test, the application #1a using its own routing code, the number of lines of nesC was about 1.5 times

Table 3.11: Quantitative metrics - for app #3 complete

Metric	**Terra**	**TinyOS**
Program lines	261	364
Bytecode size (bytes)	1,113	–
Machine code size (bytes)	40,000	15,852
Code blocks	47	661

that of the Céu-T version. In these examples, the same increase in complexity of the algorithm led to a greater increase in code size for nesC than to Céu-T. On the other hand, the Terra bytecode also grows with the application, and we must check whether it exceeds some memory limitation. From Table 2.1 in page 34, the minimum memory available for TerraNet is about 2,000 bytes. Application #3 in TerraNet has bytecode size of 1,113 bytes, but we have to check the total memory used by the application. This application uses 1,524 bytes, including bytecode, variables, Céu-T runtime controls, and operation stack.

To stress the memory limitation, we complement this test with a Terra implementation for the DTRNG (Determine transmission power using RNG-relative neighborhood graph) (Chen and Rowe, 2011), also implemented by Auza (Auza, 2013; Auza et al., 2014). Because this algorithm is more complex and also includes the DTNBOR algorithm, we expect it to use more memory than the first one. Like the DTNBOR implementation, our DTRNG implementation doesn't include the routing tree or sensor monitoring. As DTRNG assumes only direct node communication, we cannot use our basic routing tree algorithm, because it uses broadcast messages. Table 3.12 presents the quantitative data for the DTRNG implementation in Terra.

Table 3.12: Quantitative metrics - for DTRNG in Terra

Metric	**DTRNG**
Program lines	229
Bytecode size (bytes)	1,470
Machine code size (bytes)	40,000
Code blocks	62
Total memory size	1,979
Variables and runtime control memory size	489

The two last lines of the table show the total memory used by the application and the total memory reserved for the runtime control and variables. In this case we have 1,979 bytes of total used memory for the DTRNG algorithm. Yet we will need more memory for routing and sensor monitoring. The total used memory will exceed the 2,000 bytes of the TerraNet version for MicaZ

and Mica2, but we can still use the TelosB version. From Table 2.1, TerraNet for TelosB has about 7,500 bytes of memory available for the application.

Algorithm DTRNG may have problems to scale memory when the network grows: it stores the neighborhood table of each neighbor locally. In our case, we define that each node may have the maximum of 10 neighbors, and the full neighborhood table in our implementation needs 100 registers. In this case it uses about 25% of total used memory for flow control and variables. If we consider 20 neighbors, the total memory size jumps to 2,625 bytes and the total memory for runtime control and variables jumps to 43% of total used memory.

3.6 App #4 - Volcano Application

In this test application, we evaluate Terra using a CPU-intensive application with complex operations that tests the limit of CPU processing and program memory. This test uses the TerraVolcano customization as defined in Section 2.3.3.

The Volcano application[3] (Tan et al., 2010; Tan et al., 2013) is based on periodic reading of seismic data. The readings happens at 100Hz, i.e., 100 points per second. In a WSN, the network bandwidth is a bottleneck that bars the transfer of all data to a conventional computer. The challenge is to process these data against a complex computational model, using a WSN, in less than one second. The solution proposed by the authors is to relay to a conventional computer the heaviest computation and the decision about the nodes which should execute more processing. In the first round, each sensor will report an energy scale to the base station, which chooses a subset of sensors dynamically to execute further advanced signal processing. The process starts with the conventional computer, called Basestation, sending a START message to synchronize all network nodes. Each node, within a 1 second period, first reads the 100 points of data stream from seismic sensor, computes the energy scale, sends this result – as a ENERGY_SCALE message – to the Basestation, and waits for a SELECT message or the timeout for the next period. Using the received ENERGY_SCALE message, the Basestation selects the nodes that are to proceed with the computation, and sends them a SELECT message. On receiving this message, a node starts the decision algorithm, a high-processing computation based on Fast Fourier Transform (FFT), and sends the result, as a DECISION message, to the Basestation. The Basestation them computes the fusion algorithm to decide if it indicates a significant event.

[3]We thank the authors for making the source code available.

Our evaluation focuses only on the role of WSN nodes. The application includes two main algorithms, the Energy Scale and the Detection. The Detection algorithm has the heaviest processing. Although the focus of the Volcano work was the detailed measurement of performance, our goal is only to check whether we manage to keep processing time under 1 second with our approach.

The TerraVolcano customization, as defined in Section 2.3.3, embeds four functions for the Energy Scale and one function for the Detection. To execute our test we create a scenario where the SELECT message is always simulated, in that way all cycles execute the Energy Scale and the Detection operations. The difference are the seismic data of each cycle that determines different execution conditions.

In our test we compare three different program versions. Two versions were written for Terra, one using Céu-T only to glue all Volcano functions and the other implementing in Céu-T the three first functions of Energy Scale. We named them, respectively, Terra-A and Terra-B versions. The third version is the original code written for TinyOS, with the modification to generate internally the SELECT message. We included in the Céu-T versions the same measurement points of execution time used in the original version.

Listing 3.9 presents the Céu-T pseudocode. The program has an outer loop with cycles of at least 1 second (the real time depends of the total time to execute the Volcano operations). The program starts by initializing the FFT reserved memory (line 1) and reading the Gaussian model that was recorded in the data storage memory (line 2). The core part of the program is divided in the two main operations, the energy scale computation (lines 6–11) and the detection algorithm (lines 14–21). The VM-T customized Volcano functions are identified with (F1)–(F5) codes at the beginning of the respective lines.

Listing 3.9: Céu-T pseudocode for Volcano application

```
1  Initialize FFT reserved memory
2  Read the Gaussian Model from data storage memory.
3  loop for each decision cycle
4     par/and
5        ------------- Energy Scale -------------
6        Read next stream of seismic data
7        (F1) Compute mean from raw data -- intensityMean()
8        (F2) Compute seismic energy -- seismicEnergy()
9        (F3) Compute energy scale -- energyScale()
10       (F4) Copy data to buffer pool -- copyBufferPool()
11       Send ENERGY_SCALE message
12
13       ------------- Detect -------------
```

```
14        Wait simulating the SELECT message
15        loop for 5 pool data
16           if it has the minimal detect conditions
17              (F5) DECISION = call detect()
18              break the loop
19           end
20        end
21        Send DECISION message
22     with
23        timeout 1 second
24     end
25 end
```

The Terra-A version follows exactly the defined pseudocode calling the five Volcano functions. However, the Terra-B version replaces the calls to the first three functions with Céu-T code. Listing 3.10 presents the Céu-T code of the Terra-B version only for this part. The computation of the intensity mean of raw data is based on a loop that sums the 100 values, discarding invalid readings (lines 2–13). The seismic energy is computed (lines 16–27) by taking the average of squared values. Finally, the energy scale is computed (lines 30–37) as the integer value for the *log(10)* of the seismic energy.

Listing 3.10: Céu-T code for Terra-B – only the functions replacement part

```
1     (F1) Compute mean from raw data -- intensityMean()
2        valid_count=0;
3        intensity_mean = 0;
4        loop i, count do
5           var ulong val = sData.val[i];
6           if(val != INVALID_MEASUREMENT) then
7              intensity_mean = intensity_mean + val;
8              inc valid_count;
9           end
10       end
11       if valid_count > 0 then
12          intensity_mean = intensity_mean / valid_count;
13       end
14
15    (F2) Compute seismic energy -- seismicEnergy()
16       var ulong sum_sq=0;
17       seismic_energy = 0;
18       if valid_count > 0 then
19          loop i, count do
20             var ulong val = sData.val[i];
21             if(val != INVALID_MEASUREMENT) then
22                var long buf1 = (val - intensity_mean);
23                sum_sq = sum_sq + ( buf1 * buf1 );
24             end
```

```
25          end
26          seismic_energy = sum_sq / valid_count;
27      end
28
29  (F3) Compute energy scale -- energyScale()
30      var ulong base=10;
31      scale=8;
32      loop iScale, 8 do
33          if (seismic_energy < base) then
34              scale=iScale; break;
35          end
36          base = base * 10;
37      end
```

Table 3.13 presents the quantitative data for the three versions. Because this experiment embeds high-complexity operations, the number of lines of TinyOS version ranges from 6.8 to 8.5 times the size of the Terra versions. For the TinyOS version, we counted only the lines of the core code for Volcano. This does not consider the FFT code and the Seismic sensor code. Comparing the A and B versions of Terra, the replacement of the 3 functions by Céu-T code increased the number of lines by 26% and the bytecode size by 40%.

Table 3.13: Quantitative metrics - for app #4

Metric	**Terra-A**	**Terra-B**	**TinyOS Original**
Program lines	129	162	1099
Bytecode size (bytes)	667	937	–
Machine code size (bytes)	45,200	45,200	35,504
Code blocks	29	40	1,479
Total script memory (bytes)	2344	2636	–

Memory allocation in the Terra Volcano version was a challenge. Apart from the ROM, where we had to eliminate some basic sensors, such as temperature and luminosity, to create space for the Volcano components, we had to balance the amount of RAM memory used by the internal components with the memory used by the script. The main difficulty was aligning the byte stream from the seismic sensor to the Volcano functions as required for the 16-bit microcontroller architecture. This was specially hard in respect to some forced casting in the FFT data structure. This requires us to keep some internal data structures in the functions, diminishing the opportunity to break them into small pieces. This was the case of copyBufferPool() and detect() functions. In table 3.13, the script memory size shows an example of how tight it was to manage the RAM memory. We worked near the limit of 2,668 bytes

available for the Céu-T script in the Terra Volcano customization, as presented in table 2.2 in section 2.3.4.

We also used the Volcano application to evaluate Terra in high-processing condition. Similarly to the original work in Volcano for TinyOS, we measured, for each version, the execution time for different code sections. These sections are grouped as Mean, Energy+Scale, Copy Buffer, and Detect.

Figure 3.4 presents a graph with the values we obtained for the three versions – Terra-B, Terra-A, and Original in TinyOS. The durations of the four section are stacked and the three graphs have the same scale to facilitate the comparison. The Y axis has the duration, in milliseconds, of the execution time for the processing of each 1-second data stream in the X axis.

Although we ran the test during the first 300 seconds of data, we selected a sampling from seconds 200–219. This sampling range contains three different activity periods. The first period (seconds 200–208) is a full valid data stream with no seismic events that activated the Mean, Energy + Scale, and Copy Buffer operations. The second period (seconds 209–216) is a section combining seismic events that activated all operations, including the Detect operation. The third period (seconds 217–219) is a stream with small number of valid data which does not require so much processing.

The Terra-B version, implementing some functions in Céu-T, spends more time processing data than the other cases and almost reaches the one second limit. As this measurements refer to only part of the code, and do not cover the time needed to read the seismic sensor and to send messages, the total time, may, in fact, exceed one second. This graph also shows that the time of processing for the Terra-A version, that basically glues the embedded Volcano functions together, has the same order of magnitude than the original TinyOS version. The processing cost of the Detect operation is similar in the three versions and is significant even in the TinyOS version. Breaking the full Volcano application into different operations brings the possibility of configuring the script, not only using different parameters when calling the operations, but making it possible to replace or complement specific parts of the operation by script code.

Figure 3.5 presents a magnified view of the graph comparing the Terra-A version with the original TinyOS version. In this graph it is possible to identify the additional cost incurred by the script interpretation, including the flow control operations and the function calls. For example, it is possible to see the constant difference for the CopyBuffer operation and the additional processing for the Detect operation even outside the detection event.

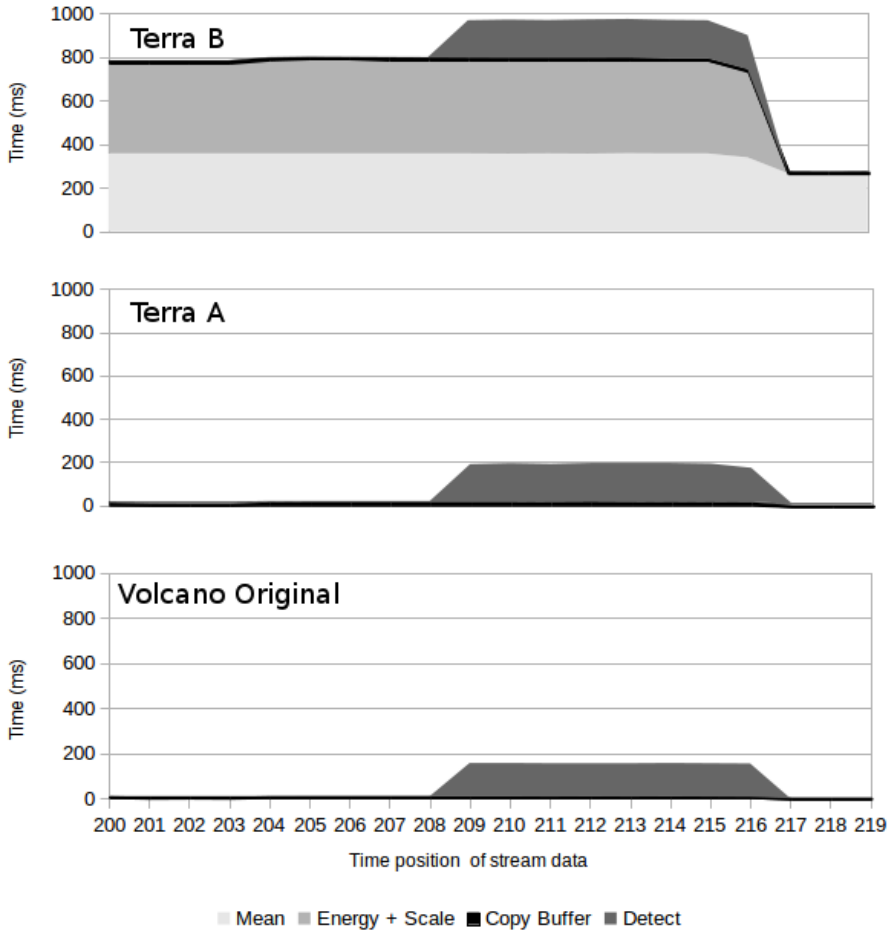


Figure 3.4: Comparative of execution time for the three versions – B, A, and Original

3.7 Items outside the programming evaluation procedure

In our programming evaluation procedure we did not consider three items: learning curve, local starvation, and invalid pointers. This section presents our analysis and discussion for these three items. The learning curve is evaluated based on our teaching experience in WSN programming. Terra solves local starvations and invalid pointers imposing some restrictions at system definition level.

3.7.1 Learning curve

We have been using Terra for teaching for at least three years. As part of this experience, we proposed the use of WSN programming as support to teach distributed system (Branco et al., 2013). Currently we use Terra in two courses at PUC-Rio, one of them a graduate course on Distributed Systems, and the

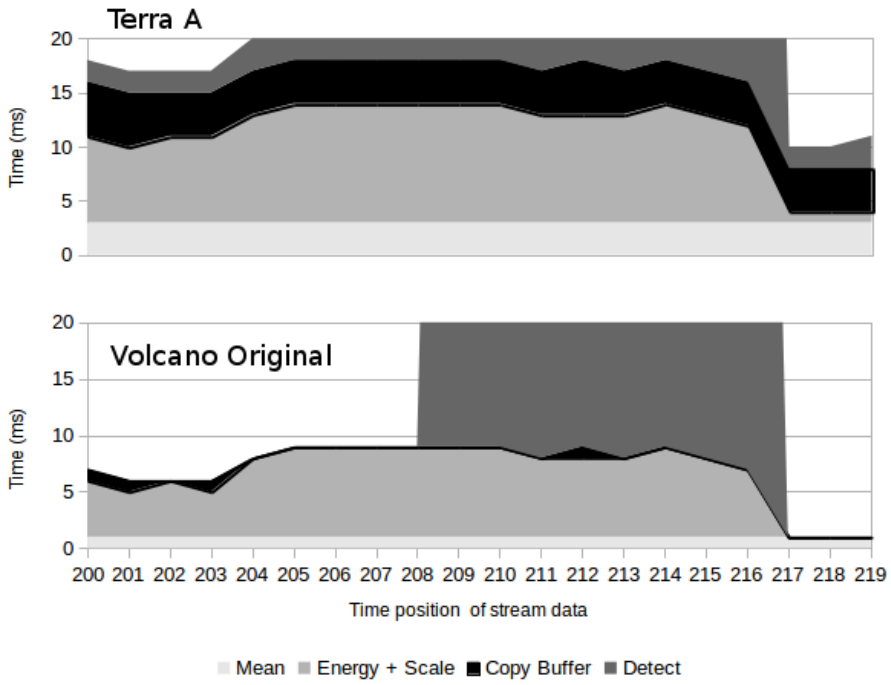


Figure 3.5: Magnified view of Terra-A vs Original version

other an undergraduate course on Reactive Systems. We started by spending about 10 hours (five classes) to introduce students to programming WSNs with nesC and TinyOS, and including some extra time for the implementation of RPC and of Probe/Echo (Andrews, 1991). After we started using Terra, we spend about four hours (two classes) teaching approximately the same material. The first class explores WSN basics, the Terra programming model, its resources, and some simple exercises. In a second class, the students do some exercises, including simple network routing. With that, most of the students are able to build, in the extra time, a network protocol similar to application #1. Besides the reduction in classroom time needed to achieve the same results, we observe, informally, that students are more motivated and have a better programming experience in Terra than they did before.

3.7.2 Local starvation

Generally, in WSN, starvation occurs with tight processing loops that don't leave the CPU free to react to other pending events. TinyOS, over which VM-T is implemented, has a simple scheduler to execute tasks posted in a queue. A task cannot be re-posted if it is pending in the queue. Because the system is single threaded, each task is executed to completion before starting the next task. Exceptions are the CPU interruptions that may execute during the execution of a task. All interrupt handlers in TinyOS must be as short as

possible and post tasks that complete the necessary work. In the developer's point of view, almost everything runs as a task. The scheduler guarantees starvation freedom only at task level, but a simple infinite loop inside a task will block the system.

In the Terra environment we have to worry about possible infinite loops at two programming levels. At one level are the Terra customized components and, at the second level, the user script application. At the first level, in nesC/TinyOS, we expect an experienced programmer to build and to test exhaustively all components before making them available to the application programmer. For the second level, we combine the guarantees in the original Céu language with a specific detail of the Terra execution model. The Terra implementation treats each Céu trail as a TinyOS task. (See Section 2.2.2 for more details). The Céu and Céu-T compilers include in their static analysis checks for infinite loops without await statements inside them. Thus, an infinite loop in Céu-T code is necessarily composed of different tasks. This guarantees that the scheduler will run all pending tasks.

3.7.3 Invalid pointers

Terra avoids invalid pointers in three different ways. First, Céu-T treats all memory addresses statically, both for code and for variables. Second, we implemented in Terra a simple type system that does not allow pointer variables. (See Section 2.2.1 for more details.) Terra also offers guarantees against out-of-bound array indexes. For constant indexes, the Terra compiler gives an out of bound error for invalid values. For variable indexes, the assembler opcode for array indexing carries the index max value and the Terra runtime checks it in execution time. In case of error, the operation is not executed and an ERROR event is generated with E_IDXOVF value.

3.8 Analysis

In this section we consolidate our analysis with a general view.

Although the reduction of global variables is not explicit in all cases, a Céu-T program tends to push globals to local procedures. This also tends to reduce the number of possible combination of global variables and events, keeping the reactive program logic clearer when compared to an equivalent event-driven version. Even when we need to use global variables, in general, these variables are strongly related to only few blocks of code.

When compared to two separate operations in different position in the code, the split phase operations using (emit+await) of Céu-T also reduce program complexity, allowing a more concise code. Terra's guarantees against local starvation and invalid pointers contribute to safer code.

Handling a basic radio operation in TinyOS requires some special knowledge and needs several lines of code in different parts of the nesC program. Besides handling the message data structure and calling the send command, the user must additionally configure some TinyOS components, define the used interfaces, and call some functions to build the message buffer. Terra simplifies this operation, because the user needs only to define the message data structure and call the send command.

The reactive model of Céu-T has some patterns of construction that are very useful for programming network protocols. These patterns, along with the points identified above, allow a network algorithm written in Céu-T to be very concise when compared to an equivalent one written in nesC/TinyOS. The low-level abstractions of Terra for send and receive operations already present an interface that is simpler to use than one provided by TinyOS. As we get higher up in the abstractions levels, we may also have simpler interfaces. But, in abstractions of general use like the grouping component, the interface takes a number of different arguments to allow the configuration of different operation modes. This variety of arguments make complex the use of the component because it transposes the operation complexity to the arguments of the interface. We have a trade-off between the program size simplification and the component parametrization. Sometimes it may be better restrict some internal configurations to reduce the parameterization degree and provide a more simple interface.

As regards error checking, the VM-T implementation captures the array-index overflow and generates a special ERROR input event to the user script. Division by zero and stack overflow are handled similarly.

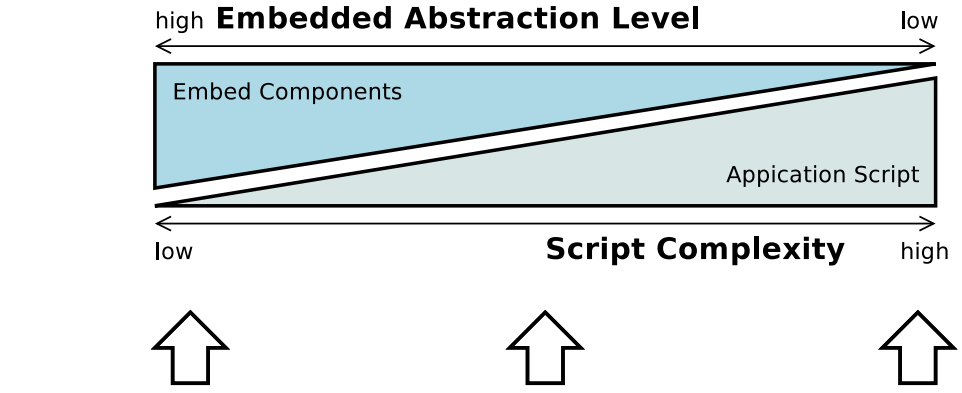
In general, the Terra system allows the VM-T expert programmer to build customized components for use by the application developer in a specific context. This opens the opportunity for high-level components with simple interfaces, in contrast to the relatively complex interface of our generic grouping components. These interface simplifications, added to the Céu-T language characteristics, allow a fast learning curve when compared to a programming environment like nesC/TinyOS.

Using high-level abstraction we obtained drastic reductions in bytecode size. A good side effect is the small amount of the messages necessary to reprogram the application running on the network. In general, the energy cost

for the dissemination of this code is not significant compared to the energy cost of a long running application. The remote reprogramming capacity, given by the use of virtual machine, is an advantage over traditional systems like TinyOS and Contiki. The developer easily reprograms the application and the network nodes use much less energy than needed to disseminate the full machine code via radio.

Although CPU-intensive processing is not common in WSN applications, we experimented with a Terra system configuration using a specialized high-processing application. When we embedded all high-processing functions in a customized component, the execution time of the Terra version was compatible with that of the original application written in TinyOS. On the other hand, implementing high-processing operations in Céu-T may not be adequate for some applications.

As expected, the code complexity decreases when we increase the abstraction level. Figure 3.6 shows the expected relation between complexity of scripts, complexity of components, and the roles of Application programmer versus Expert programmer. The border adjustment between code complexity and abstraction level may create a simple development environment for application programmers.



Abstraction Support	High Level	Intermediate Level	Low Level
Main role	application domain developer	Hybrid developer	distributed system expert
Embedded Components	complex	medium	simple
Script Complexity	Tends to be simple	Tends to medium	Tends to be complex

Figure 3.6: Behavior of abstraction complexity

4
Cost evaluation

In this chapter, we address the cost issues due the use of virtual machine architecture, as defined in Section 1.2.2.

The next two sections present the execution strategy, the test metrics and the test scenarios we used to measure CPU and memory overhead. The following sections describe our observations regarding the overhead imposed by VM-T and the time needed for code dissemination.

4.1
Execution strategy and Metrics

The test application was again compared in two different environments, one using nesC/TinyOS and other using the specific TerraNet customization.

Table 4.1 describes all metrics we chose to help us understand and compare resource usage in different test scenarios.

Table 4.1: Resource usage metrics

Metric	**Description**
Cycles/Time	Number of loop cycles in CPU/IO bound test.
CPU Active/Idle Cycles	Number of clock cycles in active and idle CPU modes.
Radio/CPU Energy	Amount of consumed energy.
Byte-code size	Program size in bytes/blocks to be disseminated via radio.

To obtain values for the first three metrics we ran the tests using the Avrora simulator (Titzer et al., 2005). Avrora can simulate a network with MicaZ nodes. It acts at machine code level and also emulates radio-chip operation.

4.2
Test scenarios

We chose three test scenarios to support our analysis. The first two scenarios are conditions of saturated execution and aim to make the VM overhead explicit. The third scenario represents a simple and typical application. Table 4.2 shows the two environments considered in our tests.

Table 4.2: Execution environments (Resource)

Environment	Description
nesC/TinyOS	Low level code environment.
TerraNet	Low level abstraction environment.

Table 4.3 describes the application used for each test scenario. Applications #1 and #2, respectively CPU-Bound and IO-Bound, are used to stress the system to reveal execution differences between nesC and Terra. These applications are used to identify performance impacts from the use of virtual machine. Application #3 is a simple read sensor loop like a regular monitoring application. We use this to show resource utilization in a normal operation condition. For all tests we compared similar applications running in nesC/TinyOS and in TerraNet.

Table 4.3: Applications

#	Application	Description
1	CPU-Bound	An intensive CPU loop without I/O events.
2	IO-Boud	An intensive I/O loop.
3	Normal operation	A simple monitoring application running in a normal operation condition without intensive use of CPU or IO.

4.3 Results

In this section, we evaluate VM-T from two different points of view. In Section 4.3.1, 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[1]. Next, in Section 4.3.2, we evaluate the time for code dissemination and its scalability for network growth.

4.3.1 VM overhead benchmarking

We use three 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.

[1]All program versions, including the Terra runtime, were compiled to MicaZ platform using the same radio transmission power (CC2420_DEF_RFPOWER=7).

The idea of this test is to stress the processing capacity of VM-T to the limit. In the second test, we measure the overhead of the VM bounded by an IO operation. In this case the application repeatedly reads data from a sensor in a closed loop. In the third 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 periodic loop.

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

In both systems, programs are coded with event-based loops. In Terra, because a tight loop is forbidden, we use a custom event to break the loop with an await command. In the corresponding Terra custom component, the return event is generated immediately from the request. In the nesC/TinyOS version, each iteration posts a task representing the following iteration.

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 and help us to understand the difference in the processing time.

Test scenario 1 - CPU-bound Application

Table 4.4 presents the results obtained with Avrora for our first scenario. Listings 4.1 and 4.2 show 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 (re)post of the task itself. A periodic timer sends the counter value to the basestation every 10 seconds. The message data are copied to the radio message buffer (lines 16–20) and the sending command (lines 21–23) is executed followed by a debug message (lines 24–26) specific to the TOSSIM version. In Terra we have a “par” with two sections. The first section controls the loop and increments the counter variable (lines 3–7) and the second section sends the counter value to the basestation every 10 seconds (lines 10–14). We use the CUSTOM event to act as dummy event, as it forces a returned value via event interface (lines 4–5).

[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.

Table 4.4: CPU-bound Test

Metric	Program Version		
	Terra(a)	**nesC**(b)	b/a
loop counter	597,511	11,735,607	19.64
active cycles	2,175,061,049	2,174,060,892	1.00
idle cycles	37,735,747	4,768	0.0

Listing 4.1: The code for CPU-bound experiment in TinyOS/nesC.

```
1  // CPU bound loop
2  task void incTask(){
3      counter++;
4      post incTask();
5  }
6
7  // Monitoring message
8  event void RadioControl.startDone(error_t error){
9      call sendTmr.startPeriodic(10000);
10     post incTask();
11 }
12 event void sendTmr.fired(){
13     error_t stat;
14     Msg.d16[0]++;
15     Msg.d32[0]=counter;
16     memcpy( call send.getPayload(
17                     &sendBuff,
18                      call send.maxPayloadLength()),
19             &Msg,
20              sizeof(sendBS_t));
21     stat = call send.send(  0xffff,
22                             &sendBuff,
23                              sizeof(sendBS_t));
24     if (stat != SUCCESS) {
25         dbg(APPNAME,"CM::send(): Send error\n");
26     }
27 }
```

Listing 4.2: The code for CPU-bound experiment in Terra.

```
1  par do
2      // CPU bound loop
3      loop do
4          emit REQ_CUSTOM;
5          await CUSTOM();
6          inc msg1.count;
7      end
8  with
9      // Monitoring message
```

```
10      loop do
11          await 10s;
12          inc msg1.seq;
13          emit SEND(msg1);
14      end
15  end
```

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 after 300 seconds. The TinyOS version ran 19.64 times the iterations executed by the Terra 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 600,692 and 11,735,309.)

We now estimate the number of cycles per instruction in VM-T. The main loop of our test script translates to six 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 6 to estimate the number of cycles per instruction. The result is 607 cycles, which is close to the value of 550 cycles reported for DVM (section 4.1 §2 of (Balani et al., 2006)) and not so far from the 400-cycles value obtained in the micro-benchmark of ASVM (section 4.5 §2 of (Levis et al., 2005)).

Test 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. Listings 4.3 and 4.4 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 (lines 2–5). 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. The remainder lines of program, that send the data message, are similar to the first scenario. 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 (lines 3–7). 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.

Listing 4.3: The code for IO-bound experiment in TinyOS/nesC.

```
// I/O bound loop
event void sensor.readDone(error_t e,uint16_t v){
    counter++;
    call sensor.read();
}

// Monitoring message
event void RadioControl.startDone(error_t e){
    call sendTmr.startPeriodic(10000);
    call sensor.read();
}
event void sendTmr.fired(){
    error_t stat;
    Msg.d16[0]++;
    Msg.d32[0]=counter;
    memcpy( call send.getPayload(
                    &sendBuff,
                     call send.maxPayloadLength()),
            &Msg,
             sizeof(sendBS_t));
    stat = call send.send(  0xffff,
                                &sendBuff,
                                 sizeof(sendBS_t));
    if (stat != SUCCESS) {
        dbg(APPNAME,"CM::send(): Send error\n");
    }
}
```

Listing 4.4: The code for IO-bound experiment in Terra.

```
par do
    // I/O bound loop
    loop do
        emit REQ_PHOTO();
        value=await PHOTO;
        inc msg1.count;
    end
with
    // Monitoring message
    loop do
        await 10s;
        inc msg1.seq;
        emit SEND(msg1);
    end
end
```

Table 4.5 presents the results for this scenario.

Table 4.5: IO-bound Test

Metric	Program Version		
	Terra(a)	**nesC**(b)	b/a
loop counter	27,269	29,999	1.10
active cycles	265,848,122	104,726,668	0.39
idle cycles	1,959,251,305	2,068,673,827	1.06

In this case, predictably, CPU active time was much less than in the first test scenario. CPU was idle around 88%-95% of the time. The nesC variant executed approximately 10% more iterations than the Terra variant. As regards CPU cycles, however, the Terra version needed around 2.5 times the cycles used by nesC. In Terra, CPU was active 11.95% of the time, while in nesC only 4.82%.

Direct execution on the MicaZ mote again produced results close to the simulator's: the value of the counter was 27,270 for the Terra version and 29,999 for the nesC one.

In Terra, approximately 91 iterations were executed per second. In ASVM, in a similar test using a mica mote (Hill and Culler, 2002), 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 order of magnitude of the direct execution over nesC/TinyOS.

Test scenario 3 - IO-timer application

In this test, the application repeatedly reads the sensor every 10 seconds, increments the loop value when the sensor returns a value, and sends this value via radio. Listings 4.5 and 4.6 show the code we used for this experiment. In the nesC version we have a periodic timer of 10 seconds to trigger the main loop. This main loop is a TinyOS event handler that contains two commands: the loop counter and the *call sensor.read()* call, which initiates a new sensor reading (lines 6–9). At this point, TinyOS places the CPU in *Idle* state. When the sensor interruption occurs, the program reads the sensor, sends the data message, and waits for next timer event (again in *Idle* state) (lines 11-24). In the Terra version, the loop is the main procedure for the VM, and contains a few commands: the instruction to wait a timer, the loop counter, and the instruction for requesting the sensor value (lines 3–5). After this request, the VM becomes idle awaiting new events, and again TinyOS puts the CPU in

Idle state. When the sensor generates an interruption, TinyOS creates a task to execute the event handler for the sensor, and this in turn generates an event for the VM. After receiving the sensor event and sending the radio message (lines 6–7), the VM posts a task to (re)initiate the main procedure.

Listing 4.5: The code for IO-timer experiment in TinyOS/nesC.

```
1 event void RadioControl.startDone(error_t e){
2     call sendTmr.startPeriodic(10000);
3     call sensor.read();
4 }
5
6 event void sendTmr.fired(){
7     counter++;
8     call sensor.read();
9 }
10
11 event void sensor.readDone(error_t result, uint16_t val){
12     error_t stat;
13     Msg.counter=counter;
14     memcpy( call send.getPayload(
15                 &sendBuff,
16                  call send.maxPayloadLength()),
17             &Msg,
18              sizeof(sendBS_t));
19     stat = call send.send(0xffff,
20                      &sendBuff,
21                       sizeof(sendBS_t));
22     if (stat != SUCCESS) {
23         dbg(APPNAME,"CM::send(): Send error\n");
24     }
25 }
```

Listing 4.6: The code for IO-Timer experiment in Terra.

```
1 msg1.count=0;
2 loop do
3     await 10s;
4     inc msg1.count;
5     emit REQ_PHOTO();
6     value=await PHOTO;
7     emit SEND(msg1);
8 end
```

Table 4.6 presents the results for this test scenario.

In this case, as expected, CPU active time is much lesser than in the two first scenarios. The CPU was idle around 99.6% of the time. The nesC variant and the Terra variant executed exactly the same number of iterations. As regards CPU cycles, however, the Terra version needed around 1.08 times

Table 4.6: IO-timer Test

Metric	Program Version		
	Terra(a)	**nesC**(b)	b/a
loop counter	30	30	1.00
active cycles	9,630,812	8,896,252	0.92
idle cycles	2,239,073,188	2,239,807,748	1.00

the cycles used by nesC. In Terra, the CPU was active 0.43% of the time, while in nesC 0.41%.

Direct execution on the MicaZ mote again produced similar results to the simulator's: the value of the counter was 30 for the Terra version and 30 for the nesC one.

The results for this third test 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 a periodic timer pattern.

Energy consumption analysis

Table 4.7 shows the values of energy consumption that are reported at the end of execution of all three test scenarios using the Avrora simulator. The energy values are shown in Joules and represent total consumption in Terra and in nesC. We analyse only the two major energy consumers, radio and CPU. For the radio, we separate the energy consumption in the receive and the transmit modes.

Table 4.7: Energy consumption results

		Total CPU Cycles	**Energy (in Joules)**		
			CPU	**Receive**	**Transmit**
CPU-bound	Terra	2,212,796,796	6.75	16.86	0.0018
	nesC	2,173,038,370	6.69	16.56	0.0014
IO-bound	Terra	2,225,099,427	3.48	16.96	0.0019
	nesC	2,172,629,320	3.14	16.56	0.0014
IO-timer	Terra	2,222,949,986	3.04	16.94	0.0018
	nesC	2,185,740,922	2.99	16.66	0.0014

As expected, the radio energy for the receive mode is a constant value of 0.0076 Joules per CPU Cycle. This means that energy spent in the receive mode was the same in all tests and that the use of virtual machine doesn't

impact this value. In general, applications must use some mechanism to reduce the energy utilization of the radio in receive mode, like the LPL-Low Power Listening (Polastre et al., 2004).

In the case of radio transmission energy, all three nesC executions have used 0.0014 Joules to transmit the same 30 radio messages. In Terra, considering the energy spent in the received mode and in the transmission, we had a small energy overhead incurred by the code dissemination protocol, but this is negligible in a long running application.

The difference in energy consumed by the CPU is due to the difference in the periods of activity. In the documentation of the Atmel microcontroller (Atmel, 2011), table *DC Characteristics* in pages 318/319 indicates that an active cycle consumes roughly 2.5 times the energy consumed by an idle cycle. The IO-bound test for Terra had 2.5 times the number of active cycles used by nesC. However, because the total number of active cycles still remains small in proportion to the number of idle cycles, energy consumption was only 11% higher. In the IO-timer test, Terra had only 1.09 times the number of active cycles used by nesC and the energy consumption was only 1.7% higher. This overhead would typically diminish, possibly to negligible rates, in real applications, in most of which the active/idle ratio is very small. Part of this overhead is due to the cost of code dissemination, and would also typically diminish in long running applications.

4.3.2 Bytecode dissemination time and scalability

Although, optimal code dissemination is not part of our research goals, we report in this Section some measurements of the dissemination time. To avoid radio messages collisions during the flooding process, the dissemination algorithm must include a delay after each packet is disseminated. But we would naturally like the 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. Our solution was to use the TinyOS simulator (TOSSIM) because 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 4.8 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 reaches up to 8 neighbor nodes. Only one node in the corner of the grid exchanges messages with the basestation. This configuration forces the use of the flooding mechanism. Figure 4.1 shows an example for the 7 x 7 grid with the radio range highlighted for nodes 11, 32, and 44. We also measured, for each node, the total time it took to load the program (local load time). Table 4.8 includes the minimum, maximum, and average for local load times in each scenario. All dissemination tests were done considering that all radios were switched on.

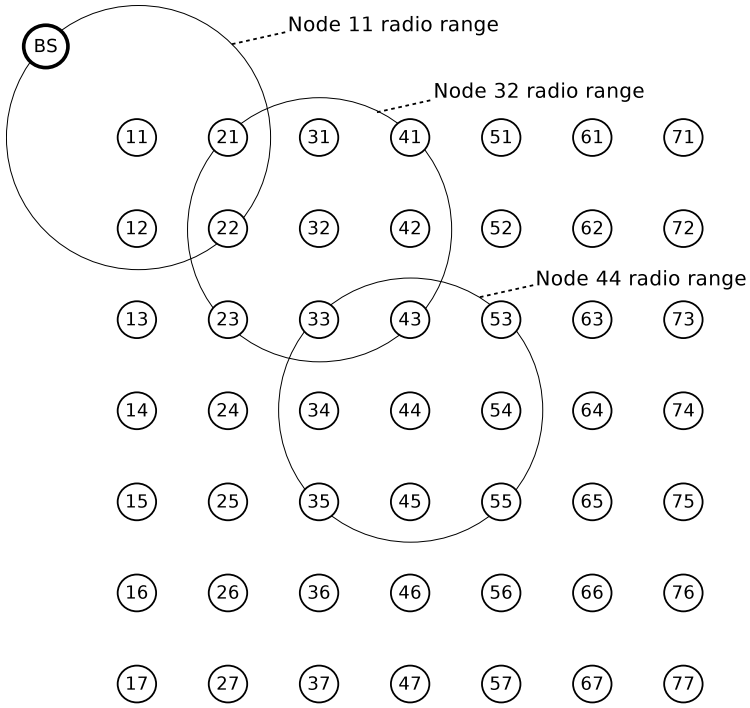


Figure 4.1: Simulated 7x7 grid - node 11, 32, and 44 with radio range highlighted.

Table 4.8: Dissemination time

Scenario	Total Nodes	Total Duration	Avg local load time	Min local load time	Max local load 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

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 conservative to minimize radio collision in dense networks. The VM-T customizer may choose different values for this parameter. This subject should be matter of further investigation and depends on network topology. The differences between minimum, maximum, and average local load times are of fractions of seconds. Considering the nature of WSN applications, in general these differences don't affect system operation. The combination of the step delay with the random send delay allows some nodes to receive more than one message in the same dissemination step. When this happens, the local load time drops to less than the standard 1.7ms.

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 an increment of 40 nodes (444%). These results indicate that the system scales well.

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 works that are related to each of these approaches and discuss how Terra relates to them.

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 and DAViM 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 other

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 a 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 basic operations and access to devices. A Regiment application is compiled to an intermediate language called *Token Machine*(TM). A TM segment propagates across 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

VM-T architecture combines small size with a model that is less restrictive than Maté's. Because Terra implements the concurrency model of Céu, it is possible to have several concurrent execution paths. Terra also enables up to 255 identifiers for each group of input events, output events, and functions. VM-T stack is defined at compile time and is limited only by memory space shared with the 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 Céu-T program it is possible to suspend the execution of

one program block and wait for an event without suspending all other 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. However, the compiler offers 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. Céu-T avoids tight loops which is not recommended but allowed by most of the related work. By itself, Céu-T doesn't give execution guarantees in intra-network operations. In Terra, these guarantees may be given by built-in runtime intra-network operations. Unlike Cosmos and Regiment, 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
Final remarks

Programming WSN system is a difficult task. The distributed system nature and the resource limitation turn it into a complex activity and error prone. Our research seeks how to simplify this programming activity and how to reduce typical errors. We formulated the following as research question for this thesis: *To what extent can a programming environment based on the combination of a reactive high-level scripting language with safety guarantees with a virtual machine that encapsulates customized components facilitate the task of programming WSNs, providing abstractions to simplify programming, reducing the possibility of errors, and allowing reprogramming?*

To investigate this idea we built Terra, a flexible system that uses a reactive language combined with a virtual machine which allows to embed new operations as components and facilitates remote distribution of scripts using low energy consumption. In this work we described the Terra implementation and its operation mode and evaluated Terra in different test scenarios. Our evaluation for programming used different abstraction levels for the functionalities available at the script level. Also we evaluate the Terra performance to identify the resource overhead incurred from the use of virtual machine.

To meet our evaluation requirements we built three different customizations of Terra — TerraNet, TerraGrp, and TerraVolcano. TerraNet represents our low-level abstraction environment. It includes only basic operations as sensor readings and simple radio operations. The TerraGrp represents our version with high-level abstractions. It implements a set of components for network operations over groupings of nodes. TerraVolcano is a high-level abstraction that uses a specialized implementation for the volcano experiment. This version combines heavy use of CPU with large data memory.

We evaluated Terra comparing the reactive programming model of Terra with the event-driven programming model of nesC/TinyOS. Applications in the test experiments ranged from low-level network algorithms up to high-level grouping operations or data processing. The evaluation produced a set of listing and some metrics regards to programming issues in WSNs.

In the performance evaluation we collected some metrics from saturated processing situation up to regular operation. Also we compared the Terra virtual machine operation with the TinyOS operation for an equivalent program.

6.1 Main findings

Our experiments showed that the Terra programming model, compared to an event-driven programming model, simplifies the programming and gives guarantees to a safer code.

An interesting discovery, in the programming evaluation, was related the use of global variables. We started from the idea that a program in Terra reduces the needs of global variables in the application. This reduction might contribute to reduce the application complexity and subsequently reduce the chance of programming errors. As well, we discovered that the structured programming patterns of Céu-T allows a more clear context separation. In this case, a global variable may be related to a few contexts in the programming, diminishing the program complexity and subsequently reducing the chance of programming errors.

Another point is the combination of the split phase operations using (emit+await) of Céu-T and the simplification in the operation interfaces, this also contribute to have a more concise code. Also, Terra's guarantees for race-free and against local starvation and invalid pointers contribute to safer code.

Considering the use of low-level abstractions in low-level networking algorithms, the simplification came from the combination of Céu-T facilities, in special the combination of some Céu-T programming patterns and global variables, with the simplification in the interface of low-level components. Programming low level networking algorithms has a small reduction in the program lines, because these kind of algorithms, in general, has a transactional model for message exchanges that hardly can be reduced. In our example of full transactional algorithm the reduction was 35% and in the case of hybrid algorithms the reductions reached 73%. In this test variant the major benefit of Terra is the opportunity, if desired, to remotely changing the network algorithm. This is useful in WSN systems in which the best network algorithm may not be completely determined beforehand and may need modifications during application life time.

Considering use of high-level abstractions for complex functionalities, we show that when we embed complex components this will reduce drastically the code size. An example of complex grouping operation compared to an example of a simple low-level network algorithm had a reduction of 78% in the line codes. But we also identify that creating general components may have more complexity in the interface with functionality abstractions. In this case, the script is more simple, but the use of component may be more complex and may generate difficulties in the code implementation. In some cases it may be

better to offer more specialized components, with fixed internal parameters, rather than generic components.

The performance issues evaluation shows that the virtual machine approach is viable for WSNs system. The additional costs are not so important in a long run application. However we observed a significant trade-off between the memory (RAM and ROM) requirement and the high-level abstraction customizations. A complex component tends to use more ROM and RAM and this limits, in special for small platforms, the available memory for the script application and may also overflow the ROM. In general this has low impact in the case of RAM, where the application script tends to be small, however someone may write a more complex script and reach the memory limit. Another attention point is about high-processing operations. Our experiment with Volcano shows that it is better to leave this operation embedded in a custom component instead of try to execute by script.

Our experiments show that several benefits of Terra come from the combination of two or three of the basic elements of the system: Céu-T language, Embedded VM, and Built in components. The reduction in the programming complexity came from the combination of the Céu-T reactive language with the use of embedded components. The verifications from the compiler combined with the VM implementation avoids local starvation and invalid pointers in a Terra application. The use of the VM approach combined with the high-level components allows very small bytecode size and low energy cost to reprogramming an application.

In general, we are very satisfied with the demonstrated results. The Terra approach showed that it is possible to simplify the WSN programming while reducing the chances of typical errors. Currently we have used Terra in undergraduate and graduate courses to teach concepts of WSN and distributed systems. From our experience, we identified that Terra allowed to reduce the learning curve compared to a low-level event-driven programming model.

Additionally, we exercise a bit more the reactive programming model of Céu. Applying it in different applications brings more confidence in its use.

6.2 Future work and related improvements

The work on Terra was born from our interest in WSN macroprogramming (network-wide programming) and from thinking that we needed node-level support before moving to the network level. We might now be able to investigate this issue using Terra as the base system for a new macroprogramming language. This new macroprogramming language may use Céu-T

as intermediate language or be compiled directly to the VM-T assembly code.

Another approach is to evaluate Terra's model in the world of IoT (Internet of Things). As WSN applications are one of the base of IoT, we may take advantage of reconfigurable characteristic of Terra to make "things" more adaptable. A more specific approach, also related to IoT, is to evaluate the Terra model for use in embedded system for different kinds of appliances. In this direction, we have already started to migrate the VM-T implementation to the Arduino[1] platform.

We believe that some future experiments and related improvements in Terra may give a better support to the areas of macroprogramming and IoT. One of them is to evaluate the Terra model to allow different roles in a heterogeneous network. In this case Terra will work as a homogeneous environment layer over heterogeneous devices, enabling the dissemination of a specific code by node. This is important for IoT experiments that connect different devices for specific applications like home automation, health-care monitoring, and industrial automation.

Finally, another important evaluation is about security. This Terra implementation relies on radio services from TinyOS, where messages are exchanged without any security support. An intruder may capture data and inject malicious data or scripts. A future experiment is to evaluate the impact, in memory size and processing, of embedding some security in Terra's communication layer.

[1] www.arduino.cc

7
Bibliography

ANDREWS, G. R. Paradigms for process interaction in distributed programs. **ACM Computing Surveys**, ACM, New York, NY, USA, vol. 23, no. 1, p. 49–90, mar. 1991. ISSN 0360-0300. Available from Internet: <http://doi.acm.org/10.1145/103162.103164>. 3.7.1

ASLAM, F. et al. Introducing TakaTuka: a Java virtual machine for motes. In **Proceedings of the 6th ACM conference on Embedded network sensor systems**. New York, NY, USA: ACM, 2008. (SenSys '08), p. 399–400. ISBN 978-1-59593-990-6. Available from Internet: <http://doi.acm.org/10.1145/1460412.1460472>. 5

ATMEL. **ATMEGA128**. 2467x–avr–06/11. ed. San Jose, CA, USA, 2011. Available from Internet: <http://www.atmel.com/Images/doc2467.pdf [accesed in june/2015]>. 4.3.1

AUZA, J. M. N. **Análise de Desempenho de Algoritmos de Eficiência Energética em RSSF**. Master thesis — Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Elétrica, 2013. 94p Text in Portuguese. 3.2, 3.5, 3.5

AUZA, J. N.; BRANCO, A.; MARCA, J. Boisson de. Experimental evaluation of energy efficient algorithms for WSN using variable transmission powers. In **Sensors (IBERSENSOR), 2014 IEEE 9th Ibero-American Congress on**. Washington, DC, USA: IEEE, 2014. p. 1–4. ISBN 978-1-4799-6835-0. 3.2, 3.5, 3.5

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**. New York, NY, USA: ACM, 2007. (EuroSys '07), p. 159–172. ISBN 978-1-59593-636-3. Available from Internet: <http://doi.acm.org/10.1145/1272996.1273014>. 1, 2.3.2, 5

BAKSHI, A. et al. 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**. Berkeley, CA, USA: USENIX Association, 2005. (EESR '05), p. 19–24. ISBN 1-931971-32-3. Available from Internet: <http://dl.acm.org/citation.cfm?id=1072530.1072535>. 2.3.2

BALANI, R. et al. Multi-level software reconfiguration for sensor networks. In **Proceedings of the 6th ACM & IEEE International Conference on Embedded Software**. New York, NY, USA: ACM, 2006. (EMSOFT '06), p. 112–121. ISBN 1-59593-542-8. Available from Internet: <http://doi.acm.org/10.1145/1176887.1176904>. 4.3.1, 5

BOUSSINOT, F.; SIMONE, R. de. The ESTEREL language. **Proceedings of the IEEE**, vol. 79, no. 9, p. 1293 –1304, sep 1991. ISSN 0018-9219. 2.1.2

BRANCO, A. **A WSN programming model with a dynamic reconfiguration support**. Master thesis — PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO - PUC-RIO, april 2011. Text in Portuguese. Available from Internet: <http://www.maxwell.lambda.ele.puc-rio.br/Busca_etds.php?strSecao=resultado&nrSeq=18309@2>. Cited jan 2015. 2.3.2

BRANCO, A. et al. Teaching concurrent and distributed computing – initiatives in Rio de Janeiro. In **Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International**. Washington, DC, USA: IEEE, 2013. p. 1318–1323. 3.7.1

BRANCO, A. et al. Terra: Flexibility and safety in Wireless Sensor Networks. **ACM Transactions on Sensor Networks**, ACM, New York, NY, USA, vol. 11, no. 4, p. 59:1–59:27, sep. 2015. ISSN 1550-4859. Available from Internet: <http://doi.acm.org/10.1145/2811267>. 2.2

BROUWERS, N.; LANGENDOEN, K.; CORKE, P. Darjeeling, a feature-rich VM for the resource poor. In **Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2009. (SenSys '09), p. 169–182. ISBN 978-1-60558-519-2. Available from Internet: <http://doi.acm.org/10.1145/1644038.1644056>. 5

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**. Washington, DC, USA: IEEE, 2008. p. 147–151. ISSN 0197-2618. 2.3.2

CHEN, X.; ROWE, N. Saving energy by adjusting transmission power in wireless sensor networks. In **Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE**. Washington, DC, USA: IEE, 2011. p. 1–5. ISSN 1930-529X. 3.5, 3.5

DUNKELS, A.; GRONVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In **Local Computer Networks,**

2004. 29th Annual IEEE International Conference on. Washington, DC, USA: IE, 2004. p. 455–462. ISSN 0742-1303. 1.2, 2.1.1, 2.1.2

DUNKELS, A. et al. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In **SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems**. New York, NY, USA: ACM, 2006. p. 29–42. ISBN 1-59593-343-3. 3.1, 5

GAY, D. et al. The nesC language: A holistic approach to networked embedded systems. ACM, New York, NY, USA, p. 1–11, 2003. 2.1, 2.1.1, 2.1.2, 3.1

GNAWALI, O. et al. Collection tree protocol. In **Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2009. (SenSys '09), p. 1–14. ISBN 978-1-60558-519-2. Available from Internet: <http://doi.acm.org/10.1145/1644038.1644040>. 3.2, 3.3, 3.3.2

GREGG, D. et al. The case for virtual register machines. **Science of Computer Programming**, vol. 57, no. 3, p. 319 – 338, 2005. ISSN 0167-6423. Advances in Interpreters, Virtual Machines and Emulators IVME'03. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167642305000389>. 2.2.2

HAN, C.-C. et al. A dynamic operating system for sensor nodes. In **Proceedings of the 3rd international Conference on Mobile Systems, Applications, and Services**. New York, NY, USA: ACM, 2005. (MobiSys '05), p. 163–176. ISBN 1-931971-31-5. Available from Internet: <http://doi.acm.org/10.1145/1067170.1067188>. 5

HARBAUM, T. **The NanoVM - Java for the AVR**. 2005. Available from Internet: <http://www.harbaum.org/till/nanovm/index.shtml>. 5

HILL, J. L.; CULLER, D. E. Mica: A wireless platform for deeply embedded networks. **IEEE Micro**, IEEE Computer Society Press, Los Alamitos, CA, USA, vol. 22, no. 6, p. 12–24, nov. 2002. ISSN 0272-1732. Available from Internet: <http://dx.doi.org/10.1109/MM.2002.1134340>. 4.3.1

IERUSALIMSCHY, R. A text pattern-matching tool based on parsing expression grammars. **Software – Practice & Experience**, John Wiley & Sons, Inc., New York, NY, USA, vol. 39, no. 3, p. 221–258, mar. 2009. ISSN 0038-0644. Available from Internet: <http://dx.doi.org/10.1002/spe.v39:3>. 2.2.2

KOSHY, J.; PANDEY, R. VMSTAR: synthesizing scalable runtime environments for sensor networks. In **Proceedings of the 3rd international conference on Embedded networked sensor systems**. New York, NY, USA: ACM,

2005. (SenSys '05), p. 243–254. ISBN 1-59593-054-X. Available from Internet: <http://doi.acm.org/10.1145/1098918.1098945>. 5

KOTHARI, N. et al. Reliable and efficient programming abstractions for wireless sensor networks. **PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation**, ACM, New York, NY, USA, p. 200–210, 2007. 1, 2.3.2

LEVIS, P. Experiences from a decade of TinyOS development. In **Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (OSDI'12), p. 207–220. ISBN 978-1-931971-96-6. Available from Internet: <http://dl.acm.org/citation.cfm?id=2387880.2387901>. 1.2.1

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**. New York, NY, USA: ACM, 2002. p. 85–95. ISBN 1-58113-574-2. 3.3.1, 5

LEVIS, P.; GAY, D.; CULLER, D. Active sensor networks. In **Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2**. Berkeley, CA, USA: USENIX Association, 2005. (NSDI'05), p. 343–356. Available from Internet: <http://dl.acm.org/citation.cfm?id=1251203.1251228>. 4.3.1, 4.3.1, 5

LEVIS, P. et al. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In **Proceedings of the 1st international conference on Embedded networked sensor systems**. New York, NY, USA: ACM, 2003. (SenSys '03), p. 126–137. ISBN 1-58113-707-9. Available from Internet: <http://doi.acm.org/10.1145/958491.958506>. 2.1.1

LEVIS, P. et al. TinyOS: An operating system for sensor networks. In **Ambient Intelligence**. [S.l.]: Springer Verlag, 2004. 1.2, 2.1, 2.1.1, 3.1

MADDEN, S. R. et al. TinyDB: an acquisitional query processing system for sensor networks. **ACM Transactions on Database Systems**, ACM, New York, NY, USA, vol. 30, no. 1, p. 122–173, 2005. ISSN 0362-5915. 2.3.2, 5

MICHIELS, S. et al. DAViM: a dynamically adaptable virtual machine for sensor networks. In **Proceedings of the international workshop on Middleware for sensor networks**. New York, NY, USA: ACM, 2006. (MidSens '06), p. 7–12. ISBN 1-59593-424-3. Available from Internet: <http://doi.acm.org/10.1145/1176866.1176868>. 5

MOTTOLA, L.; PICCO, G. P. Programming wireless sensor networks: Fundamental concepts and state of the art. **ACM Computing Surveys**, ACM, New York, NY, USA, vol. 43, no. 3, p. 19:1–19:51, apr. 2011. ISSN 0360-0300. Available from Internet: <http://doi.acm.org/10.1145/1922649.1922656>. 1

MUELLER, R.; ALONSO, G.; KOSSMANN, D. SwissQM: Next generation data processing in sensor networks. In **Third Biennial Conference on Innovative Data Systems Research**. [S.l.: s.n.], 2007. 5

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**. New York, NY, USA: ACM, 2007. p. 489–498. ISBN 978-1-59593-638-X. 2.3.2, 5

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**. New York, NY, USA: ACM, 2004. p. 78–87. 2.3.2, 5

OUSTERHOUT, J. Scripting: Higher-level programming for the 21st century. **IEEE Computer**, vol. 31, no. 3, p. 23–30, 1998. 1.1

POLASTRE, J.; HILL, J.; CULLER, D. Versatile low power media access for wireless sensor networks. In **Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2004. (SenSys '04), p. 95–107. ISBN 1-58113-879-2. Available from Internet: <http://doi.acm.org/10.1145/1031495.1031508>. 4.3.1

RIEDEL, T.; ARNOLD, A.; DECKER, C. Poster abstract: An OO approach to sensor programming. In **European conference on Wireless Sensor Networks (EWSN)**. [S.l.: s.n.], 2007. 5

SANT'ANNA, F. et al. Safe system-level concurrency on resource-constrained nodes. In **Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2013. (SenSys '13), p. 11:1–11:14. ISBN 978-1-4503-2027-6. 1.1, 2.1, 2.1.2, 2.1.2, 4, 3.1

TAN, R. et al. Quality-driven volcanic earthquake detection using wireless sensor networks. In **Real-Time Systems Symposium (RTSS), 2010 IEEE 31st**. Washington, DC, USA: IEEE, 2010. p. 271–280. ISSN 1052-8725. 2.3, 2.3.3, 3.2, 3.6

TAN, R. et al. Fusion-based volcanic earthquake detection and timing in wireless sensor networks. **ACM Transactions on Sensor Networks**, ACM, New York,

NY, USA, vol. 9, no. 2, p. 17:1–17:25, apr. 2013. ISSN 1550-4859. Available from Internet: <http://doi.acm.org/10.1145/2422966.2422974>. 2.3, 2.3.3, 3.2, 3.6

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**. Piscataway, NJ, USA: IEEE Press, 2005. (IPSN '05). ISBN 0-7803-9202-7. Available from Internet: <http://dl.acm.org/citation.cfm?id=1147685.1147768>. 4.1

WIKIPEDIA. **List of wireless sensor nodes — Wikipedia, The Free Encyclopedia**. May 2015. [Online; accessed 1-July-2015]. Available from Internet: <https://en.wikipedia.org/w/index.php?title=List_of_wireless_sensor_nodes&oldid=662572493>. 2.1.1

A
Terra – complementary informations

A.1
Execution model example

We build a very simple script to exemplify the Terra execution model. This script creates two variables, sets a value to the first variable, waits two seconds, and sets a value to the second variable. Listing A.1 presents the Céu-T script and Listing A.2 presents a commented assembler code for the simple example.

Listing A.1: A simple example in Céu-T of Terra execution control.

```
1 var byte a,b;
2 a=10;
3 await 2000ms;
4 b=20;
```

In Listing A.2, we can observe that the first eight bytes of memory was allocated for timer control (lines 3–10). The next three bytes was allocated for variables (lines 14–16). The execution starts in the first entry point and executes, in the same TinyOS task, up to the end opcode (lines 20–30).

The opcode clken_c sets the clock 0 with a timer of 2000 milliseconds and the address 30 as entry point for the end of timer. At this point the execution is suspended and it waits to the timer event. When the timer event identifies occurs, the execution is restarted from the stored address. In this case it goes to address 30 where starts the second entry point (line 34–37).

Listing A.2: A simple example in assembler code of Terra execution control.

```
1 -- Timer control space
2 Addr, Data addr
3 00008 0000 wClock 0
4 00009 0001
5 00010 0002
6 00011 0003
7 00012 0004
8 00013 0005
9 00014 0006
10 00015 0007
11
12 -- Vars space
```

```
13 Addr, Data addr
14 00016 0008 $ret:                  | internal use variable
15 00017 0009 a:                     | var byte a;
16 00018 0010 b:                     | var byte b;
17
18 -- First entry point
19 Addr, bytecode, mnemonic          | Ceu-T code
20 00019 c4 set_c byte 9 10          | a = 10
21 00020 09
22 00021 0a
23 00022 29 clken_c 0 2000 30        | await 2000ms;
24 00023 03
25 00024 00
26 00025 07
27 00026 d0
28 00027 00
29 00028 05
30 00029 01 end                      | end
31
32 -- Second entry point
33 Addr, bytecode, mnemonic          | Ceu-T code
34 00030 c4 set_c byte 10 20         | b = 20
35 00031 0a
36 00032 14
37 00033 01 end                      | end
```

A.2
Terra operation

After VM-T 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 of VM-T in the TOSSIM TinyOS Simulator or in emulators like AVRORA and COOJA.

In a typical use of Terra, the programmer writes a Céu-T program, compiles it, and uploads its bytecode to the network. The compilation process must include a specific Terra configuration file for the chosen virtual machine. The Céu-T program can only use events and functions defined in the configuration 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, which is divided in blocks, to all nodes. This is a basic flooding algorithm where all nodes forward each incoming message until all nodes are reached. Figure A.1

presents the load interface of Terra Tool.

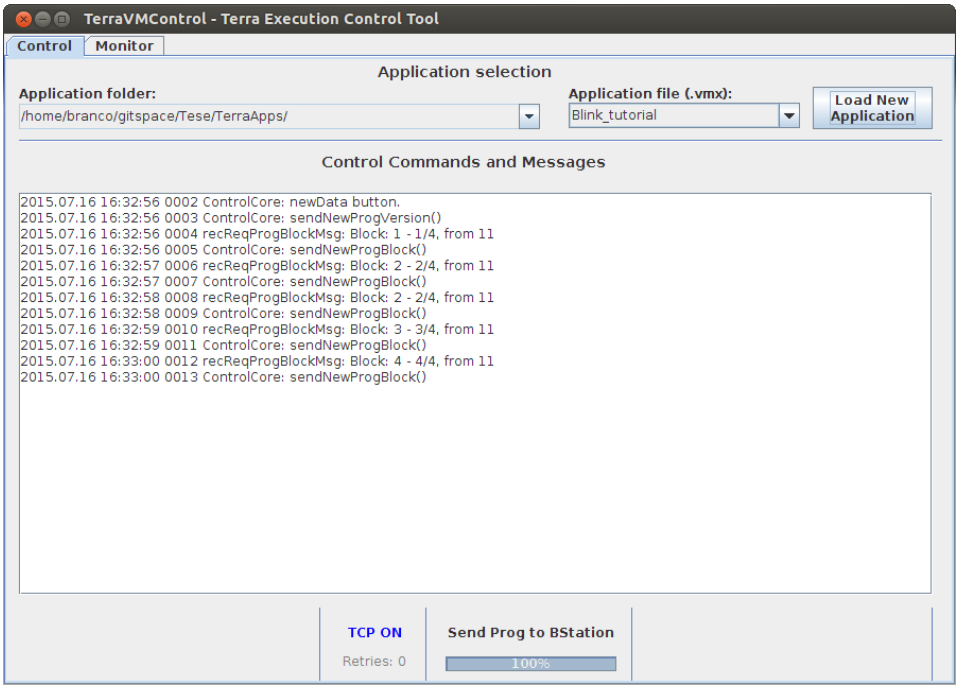


Figure A.1: Terra Tool - load bytecode interface

A.3 Integration between script and components

In Céu, the user program may indicate a list of external events and functions (written in C) that will be called. In the virtual machine approach, the script code should call only events and functions that have been previously embedded in the virtual machine over which it will execute. We have thus decided that, besides input and output events, the VM components would also provide functions in order to allow some interactions to occur in a more natural way. Typical examples of functions are getNodeId(), that returns the node identifier, and groupInit(), that sets parameters for a node to participate in a group of nodes.

System calls behave like normal function calls and are used to initialize and configure the components (e.g. groupInit()). The invocation of a system call is synchronous: control returns to the script when the system call finishes execution. The developer 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 components (e.g. emit REQ_TEMP();). Signaling 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. An event occurrence starts a new reaction in the script, awaking all trails awaiting that event (e.g, await TEMP).

The virtual machine developer must describe the custom data structures, external events and functions that the VM provides using the Céu-T syntax for configuration blocks. These descriptions and some definitions of constant values must be written in a configuration file to be included in the user application program. The customized virtual machine and the configuration file must be distributed together in order to ensure the correct execution of the user program. The Céu-T compiler can generate, without any modification, the bytecode of any scripts compatible to the new configuration.

Listing A.3 shows an example of a configuration file. The header of configuration block defines the name and the version of the customization. Also defines, for each compatible platform, the amount, in bytes, of RAM memory available to the application script. The body of configuration block defines the events and functions available in the customization. In this example we define two output events, one input event and two functions. Event definitions have always two types in its definition, the first is the returned value and the second is its argument. Function definitions have a first type that defines its returned value and may have a list of types of its arguments. All definitions must have, at the end of line, a unique number for each type of definition. This number is used to identify the respective operation inside the VM-T.

Listing A.3: A simple configuration block example.

```
1  config
2      name: TerraNet ,
3      code: 00.03.00 ,
4      {
5          telosb: 5808,
6          micaz : 2016,
7          mica2 : 2016,
8      }
9  do
10     output void REQ_TEMP         void        1;
11     output void SEND_SENSOR      radioMsg    2;
12
13     input  ushort TEMP           void        1;
14
15     function ubyte getNodeId()               1;
16     function ubyte queuePut(radioMsg)        2;
17 end
```

Figure A.4 list the definition file to be included into the VM custom module implementation. This definition is related to same definitions for the configuration presented in the Figure A.3.

Listing A.4: A include file related to a configuration block.

```
typedef nx_struct sensorMsg{
  nx_uint8_t  id;
  nx_uint16_t value;
} sensorMsg_t;

enum {
  O_REQ_TEMP     = 1;
  O_SEND_SENSOR = 2;

  I_TEMP         = 1;

  F_GETNODEID    = 1;
  F_QUEUEPUT     = 2;
};
```

Figure A.5 presents a partial implementation for the elements introduced in the Figure A.4. The two first functions are called from the VM decoder, the fist function procOutevt() dispatch any defined output events and the function callFunction() dispatch any defined customized functions. Lines 17–22 shows an example of a custom function that returns a value via stack. Lines 24–27 has a example of an external event call. Lines 29–34 presents an example how a input event is put in the queue of the VM engine.

Listing A.5: VM Customization – input/output events and functions.

```
1  // Output event dispatcher
2  command void VM.procOutEvt(uint8_t id, uint32_t value){
3    switch (id){
4      case O_REQ_TEMP: proc_req_temp(id,value);     break;
5      case O_SEND_SENSOR: proc_send_sensor(id,value); break;
6    }
7  }
8
9  // Function dispatcher
10 command void VM.callFunction(uint8_t id){
11   switch (id){
12     case F_GETNODEID: func_getNodeId(id); break;
13     case F_QUEUEPUT: func_queuePut(id);  break;
14   }
15 }
16
17 // Pushing a value to the stack
18 void  func_getNodeId(uint16_t id){
19   uint16_t stat;
20   stat = TOS_NODE_ID;
21   signal VM.push(stat);
22 }
23
24 // Calling a output event
25 void  proc_req_temp(uint16_t id, uint32_t value){
```

```
26    call S_TEMP.read();
27 }
28
29 // Queueing an input event + value
30 uint16_t lastTemp;
31 event void S_TEMP.readDone(error_t result, uint16_t val)
32   lastTemp = val;
33   signal VM.queueEvt(L_TEMP, 0, &lastTemp);
34 }
```