Journal of Systems Architecture 63 (2016) 16–32

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

A safe-by-design programming language for wireless sensor networks

Luís Lopes [a,*], Francisco Martins [b]

[a] CRACS/INESC-TEC & Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
[b] LASIGE & Faculdade de Ciências, Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal

ARTICLE INFO

Article history:
Received 29 November 2013
Revised 17 January 2016
Accepted 18 January 2016
Available online 28 January 2016

Keywords:
Programming language
Compiler
Virtual machine
Type safety
Wireless sensor network

ABSTRACT

Wireless sensor networks are notoriously difficult to program and debug. This fact not only stems from the nature of the hardware, but also from the current approaches for developing programming languages and runtime systems for these platforms. In particular, current systems do not place enough stress on providing formal descriptions of the language and its runtime system, and on proving static properties, like type-safety and soundness. In this paper, we present the design, specification, and implementation of a programming language and a runtime system for wireless sensor networks that are safe by design. We say this in the sense that we can statically detect a large set of would-be runtime errors, and that the runtime system will not incorrectly execute an application, once the latter is deployed. We have a full prototype implementation of the system that supports SunSPOT devices, the simulation tool VisualSense, and local computer networks for fast deployment and testing of applications. Development is supported by an IDE implemented on top of the Eclipse tool that embeds both the compiler and the virtual machine seamlessly, and is used to produce software releases.

1. Introduction

Wireless sensor networks (WSN) are one of the most challenging hardware platforms to program. They are gatherings of large numbers of small physical devices, commonly referred to as sensors or motes, capable of sensing the environment. The communication infra-structure is based on low-power wireless technologies and uses ad-hoc networking protocols [1]. The difficulty in programming WSN results from the unique characteristics of these platforms, especially when compared with other ad-hoc networks such as MANETs. The sensor devices are extremely limited in terms of hardware resources, namely CPU and memory, and energy, typically provided by batteries. Their deployment at remote locations makes physical access to the devices, e.g., for maintenance and debugging, in many cases difficult if not impossible, or simply not practical.

There are many proposals for programming languages for WSN providing the programmers with distinct levels of hardware and network awareness and distinct programming abstractions [2]. Given the aforementioned restrictions, programming languages for wireless sensor networks are often tightly coupled with the underlying operating system, which is typically very lightweight and modular [3–7]. At the very lowest level of programming, running on the bare hardware, we have languages such as Pushpin [8], and languages such as TinyScript and Mottle that use a thin abstraction layer for the hardware provided by a virtual machine [9,10]. Abstracting away from the hardware there are languages like the (ubiquitous) component-based language nesC [11] tightly coupled with its host operating system TinyOS [7]. Higher up in the abstraction level we find macroprogramming languages that allow programmers to abstract away, not only from devices, but also from the network infra-structure, by resorting to sophisticated compilers to automate code generation and deployment. They provide abstractions such as: streams, e.g., Regiment [12]; databases, e.g., TinyDB [13] and Cougar [14]; regions, e.g., Abstract Regions [15]; agents, e.g., Sensorware [16] and Agilla [17]; web-services, e.g., IrisNet [18].

Despite the diversity of proposals, applications for wireless sensor networks are difficult to debug and often produce runtime errors. The problem stems from the fact that most languages are built in a fairly ad-hoc way, typically by first identifying a set of adequate programming abstractions and implementing a compiler that maps the high-level syntax directly into native code or, more commonly, into an intermediate language representation, nesC code for example, or some form of bytecode. Macroprogramming languages are illustrative of this state of affairs. Regiment [12], for example, a strongly typed functional macroprogramming language, is compiled into a low-level token

* Corresponding author. Tel.: +351 960376714.
E-mail addresses: lblopes@dcc.fc.up.pt (L. Lopes), fmartins@di.fc.ul.pt (F. Martins).

http://dx.doi.org/10.1016/j.sysarc.2016.01.004
1383-7621/

machine language, which is then itself compiled into a nesC implementation of the runtime based on the distributed token machine model. The complex compilation scheme makes it rather difficult to establish a link between the semantics of the language and that of the corresponding runtime system, especially in the absence of a formal specification for the programming language and for the runtime system.

Runtime errors in sensor network applications can have multiple origins: (Type I) device malfunction or interference from the environment; (Type II) semantic errors in the application; (Type III) the runtime system does not preserve the language semantics; (Type IV) the compiler generates code that does not preserve the language semantics.

Errors of Type I are difficult or impossible to eliminate in most deployments. Type II errors can be controlled by imposing an adequate programming discipline, e.g., enforced by a type system, and by carefully testing the application before deployment. Type III and IV errors are far more subtle but very important, as they may undermine a deployment with seemingly unexplainable errors and result in significant extra costs. Type III errors can be eliminated by proving that the specification of the runtime preserves the semantics of the source language. This of course still leaves some margin for errors in the programming of the runtime, but these can be weeded out through conventional tests. Finally, Type IV errors can be eliminated by proving that the compiler generates code that preserves the semantics of the original program. This is usually called a certified compiler.

In short, errors of types II to IV can be eliminated by providing a formal specification for the programming language semantics and for the runtime semantics, and proving static properties that relate them, e.g., type-safety and soundness. Language type-safety ensures that well-typed programs do not give rise to runtime protocol errors. A compiler for a type-safe programming language can statically type-check code and identify would-be runtime protocol errors, before the application is deployed over the network. *This is possible since the full application, including the code to be run at the sink(s) and the code to be run at the nodes, is compiled as a unit, allowing for communication protocol errors to be prematurely detected.* This addresses errors of Type II. On the other hand, the soundness property ensures that the underlying runtime system preserves the semantics of the programming language. This is achieved by implementing the runtime system based on an abstract specification (e.g., a virtual machine) that can be proved to preserve the semantics of the programming language. This addresses errors of Type III. We do not address Type IV errors in this paper. This is the subject of current research.

To illustrate the design and implementation principles that we propose, we present the step by step development of Callas [19], a programming language for WSN. The language and its semantics are specified using a formal model, based on concurrency theory [20,21]. The runtime system for the language was specified in the form of a virtual machine, defined as a state transition system. Elsewhere we proven that the language is type-safe and that therefore well-typed programs do never produce a large set of runtime errors [22]. Moreover, we also proved that the runtime system preserves the semantics of the language, a property also known as soundness, and thus correctly executes Callas applications. In this paper we overview the design of the programming language and of the runtime system, and describe a full prototype implementation of this framework. The prototype includes a language compiler, a modular virtual machine that supports multiple hardware and software platforms, e.g., SunSPOT networks [23] and the VisualSense simulator [24] for deployment, and a development environment based on an Eclipse plugin that seamlessly embeds both the compiler and the runtime system and is used for software releases.

To our knowledge the use of process calculi to model and design languages for sensor networks is a novel approach. Previous work on process calculi for wireless systems is scarce and focuses on communication protocols. Prasad [25] established the first process calculus approach to modeling broadcast based systems. Later work by Ostrovský et al. [26] established the basis for a higher-order calculus for broadcasting systems. More recently, Mezzetti and Sangiorgi [27] discuss the use of process calculi to model wireless systems, again focusing on the details of the lower layers of the protocol stack (e.g., collision avoidance) and by establishing an operational semantics for the networks.

In the recent past the Internet of Things (IoT) gained a lot of attention both from the Academia and from the Industry. The IoT is a network of physical objects or "things" embedded with electronics, software, sensors, and network connectivity, which enables these objects to collect and exchange data. This paper focus on a more restricted scenario, that of wireless sensor networks. WSN aggregates a myriad of devices with similar hardware and software characteristics that autonomously collect, eventually process, and send data to gateways. From the IoT perspective, the data emanating from the gateway would be thought of as a single resource. IoT poses interesting challenges of its own, like, for instance, the interoperability between things. On the other hand, programming WSN is by itself difficult, error-prone, and correcting bugs can be difficult if not impossible after deployment in the field. It is this last problem that we tackle in this paper and propose a solution that involves a language that is demonstrably correct, thus significantly diminishing the sources of error for WSN applications.

The remainder of the paper is structured as follows. Section 2 presents the Callas language: its syntax, semantics, and briefly overviews the language safety results. Section 3 presents the Callas virtual machine: the bytecode format, the reduction rules, and briefly describes the soundness result. Sections 4 and 5 describe the prototype implementation that includes: the language compiler, the virtual machine with support for several hardware and software platforms, and a development environment based on the Eclipse tool [28]. Section 6 describes related work on programming languages and virtual machines for wireless sensor networks. Finally, Section 7 ends the paper with some conclusions and perspectives for future work.

2. The programming model

This section aims at describing Callas, a programming language for sensor networks that offers constructs to describe local computations, communications, code mobility, and code updates. The language is based on a calculus [19,22] with the goal of establishing a foundation for developing programming languages and runtime systems for sensor networks.

We start by presenting the language with a running example to illustrate the programming style of Callas (Section 2.1). Thereafter, we introduce an abstract core language (Section 2.2) suitable for defining its formal semantics (Section 2.3), of which we present only an excerpt to emphasize the foundations of Callas. In Section 2.4 we state informally a type safety result—the interested reader may refer to [22] for the details.

2.1. The Callas programming language

We introduce the Callas language by example, programming a device that periodically reads the ambient's temperature and sends it over the network, as presented in Listing 1. A Callas program is a sequence of *type declarations* followed by a code *module* that implements the type for the devices in a WSN, known as `Device`. Other declared modules arise as submodules of this top-level module. We adopt Python's line-oriented syntax, where